NEURAL NETWORKS PERFORMANCE AND STRUCTURE OPTIMIZATION USING GENETIC ALGORITHMS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Ariel Kopel

August 2012

Committee Membership

TITLE:	Neural Networks Performance and Structure Optimization Using Genetic Algorithms
AUTHOR:	Ariel Kopel
DATE SUBMITTED:	August 2012
COMMITTEE CHAIR:	Xiao-Hua (Helen) Yu, Professor
COMMITTEE MEMBER:	Bryan Mealy, Associate Professor
COMMITTEE MEMBER:	Tina Smilkstein, Assistant Professor

Abstract

Neural Networks Performance and Structure Optimization

Using Genetic Algorithms

By

Ariel Kopel

Artificial Neural networks have found many applications in various fields such as function approximation, time-series prediction, and adaptive control. The performance of a neural network depends on many factors, including the network structure, the selection of activation functions, the learning rate of the training algorithm, and initial synaptic weight values, etc.

Genetic algorithms are inspired by Charles Darwin's theory of natural selection ("survival of the fittest"). They are heuristic search techniques that are based on aspects of natural evolution, such as inheritance, mutation, selection, and crossover.

This research utilizes a genetic algorithm to optimize multi-layer feedforward neural network performance and structure. The goal is to minimize both the function of output errors and the number of connections of network. The algorithm is modeled in C++ and tested on several different data sets. Computer simulation results show that the proposed algorithm can successfully determine the appropriate network size for optimal performance. This research also includes studies of the effects of population size, crossover type, probability of bit mutation, and the error scaling factor.

Keywords: neural network, genetic algorithm, initial weights, optimize

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Neural networks and the genetic algorithm were developed based on phenomena found in nature. Both of them have been widely used to solve a variety of computationally intensive problems. When combined they yield advantages over many conventional approaches. The purpose of this research was to apply a genetic algorithm to determine the optimal dimensions of an arbitrary neural network.

1.1. Artificial Neural Networks

The concept of an artificial neural network is based on the human brain, which is made up of billions of neurons which are interconnected by synapses. Similarly, an artificial neural network is composed of many computational units which are also called neurons . The interconnections of the neurons dictate the characteristics of both a brain and a neural network. Neural networks have three major advantages: parallelism, the ability to learn, and the ability to generalize.

The human brain is highly parallel in nature since each neuron may communicate with several others simultaneously. The parallelism is ideal for complicated tasks such as pattern recognition, which would be much more difficult to accomplish in a serial fashion. Neural networks are arranged in a similar fashion, and are therefore ideal for pattern recognition and similar applications which can exploit parallelism.

Another major advantage of neural networks is their ability to learn by training. Training consists of supplying the neural network with many training samples, each of which consists of a set of inputs and the desired set of outputs. The back-propagation

algorithm is a popular method used to train neural networks. Through an iterative learning process the synaptic weights are modified and eventually the neural network is trained to produce outputs close to those desired.

Neural networks can be trained to solve different types of problems, such as pattern recognition, function approximation, control, filtering, and many others. Since only training samples are required, the actual relationship between inputs and outputs does not need to be known. This is an advantage for many applications, especially when the input/output relationship is extremely complex. Furthermore, neural networks are able to generalize, and produce accurate results for inputs which are not found in the training sample set.

Before a neural network can be trained, its size (how many neurons), topology (how they are connected), learning rate (the speed of the back-propagation algorithm) and several other parameters must be selected. Generally, more complex functions require larger neural networks consisting of more neurons and more synapses than those required for simpler functions. Two major problems can occur if the parameters are not selected correctly: unacceptable error and overfitting.

If a neural network is too small for a given application, it may never be able to learn the desired function and thus produce an unacceptably high error. An unacceptable error may also occur if the learning rate of the training algorithm is selected incorrectly. For example, the back-propagation algorithm can sometimes get stuck at local minima (as opposed to the desired absolute minimum) in error-space due to a poor choice of the learning rate. If a local minimum is reached, successive iterations of training will not reduce the error any further.

Finally, if a neural network is too large for a particular problem, it may learn the training samples too well and not be able to generalize to inputs outside the training set (known as overfitting). Selecting the appropriate neural network parameters is more of an art than a science and usually turns into a trial-and-error ordeal.

1.2. Genetic Algorithms

The genetic algorithm also has its roots in nature, and is based on Charles Darwin's theory of natural selection [24]. In Darwin's theory, individuals in a population of reproductive organisms inherit traits from their parents during each generation. Each individual's genome represents one's phenotype (i.e., the physical characteristics), and is made up of many genes (the actual genetic makeup). Over time, desirable traits become more common than undesirable ones since individuals with desirable traits are more likely to reproduce. The genetic algorithm follows natural selection quite closely.

In the genetic algorithm, each gene is usually represented as a binary number which is encoded to represent some phenotype. A population of individuals is initially created with all of their genotypes randomly selected, with each individual representing a potential solution to the problem at hand. After the initial population is created, it is sorted by fitness. The algorithm designer can choose how the fitness value is calculated, with a higher fitness value representing a better solution. During each generation, two individuals (the parents) are randomly selected to reproduce, with the more fit individuals more likely to be selected. The genotypes of the two parents are combined to create a new offspring in a process known as crossover.

During crossover, the offspring's genotype is created by combining the genes of its parents in a random fashion. After crossover, some of the bits in the offspring's genome may be flipped at random in the process of mutation, which also occurs in nature. Finally, the offspring's fitness value is calculated. If the offspring's fitness is better than the worst individual currently in the population, then the offspring replaces that individual. The population continually improves its overall fitness during each generation until finally the top individual is optimized to a satisfactory level.

With today's extremely fast computer processors, running genetic algorithms for hundreds of generations is possible in a reasonable amount of time (on the order of minutes, hours, or days). The genetic algorithm takes a portion of burden out of an engineer's hands by allowing the computer to try to solve the problem using a brute-force approach.

1.3. Applying the Genetic Algorithm to Neural Networks

The selection of neural network size, topology, and other parameters is a guess and check process. The relationship between the neural network parameters and the ability to minimize error and generalize is not known, usually requiring a guess and check approach. For this reason, genetic algorithms have been used to optimize neural networks in many different ways (discussed in section 2.3). The genetic algorithm developed in this research determines: initial synaptic weights, learning rate, number of hidden layers, number of neurons in each hidden layer, activation function and its parameters. The algorithm differs from previous ones in that it uses a significantly smaller genome and allows for a variable number of hidden layers and neurons.

1.4. Thesis Overview

This thesis is organized as follows. Chapter 2 outlines some necessary background information including a description of neural networks, the genetic algorithm, and an outline of relevant previous research. Detailed descriptions of the scaling factor based genetic algorithm, simulation methods, and results are found in chapter 3. The final conclusions and possible future research topics are discussed in chapter 4.

Chapter 2: Background Information

This chapter provides a basic explanation about how neural networks and genetic algorithms work, followed by a discussion of previous relevant research.

2.1. Neural Network Basics

Neural networks are easier to understand if they are broken down into their core components. This section explains the basics of neural networks [13]. First, the nature of the neuron is explored (the elementary unit of a neural network). Next, the different ways in which neurons can be connected are shown. Finally, neural network training (the way in which a neural network learns) is examined.

2.1.1. The Neuron

Neural networks are made from as few as one to as many as hundreds of elementary units called neurons. As shown in figure 2.1, each neuron is made up of the following: inputs, synaptic weights, a bias, a summing junction, a local induced field, an activation function, and a single output. Each of the following need to be examined in order to understand how a neuron produces an output from its arbitrary inputs:

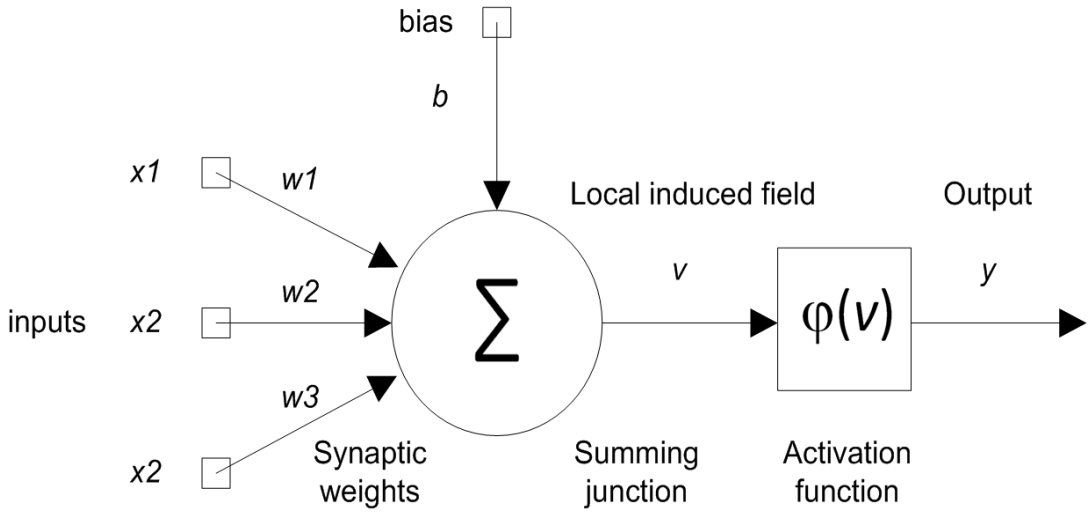

Figure 2.1 – The components of a neuron.

Inputs:

A neuron can have an arbitrary number of inputs (at least one). In figure 2.1, the three inputs are represented as *x1*, *x2*, and *x3*. Each neuron input can take on a positive, zero, or negative value. Since there is little restriction on the input values, neurons can be used in a broad range of applications. The outputs of several neurons can be connected to the inputs of another neuron, allowing the formation of multi-neuron networks (further explained in section 2.1.2).

Synaptic weights(s):

Each neuron input has a corresponding synaptic weight (or simply weight) which can also take on any positive, zero, or negative value. In figure 2.1, the three synaptic weights are *w1*, *w2*, and *w3*. The synaptic weights are used to scale the inputs as they enter the summing junction.

Bias, Summing Junction, and Induced Local Field:

The summing junction has at least two inputs (one for the bias and one for each of the neuron inputs). The bias (commonly represented as *b*) can be thought of as the weight for an input of unity. The summing junction therefore produces an output equal to the sum of the bias and all of the weighted inputs. The summing junction's output is called the induced local field, *v*, and is defined in equation 2.1 for the general case of neuron *k*.

$$v_k = b_k + \sum_{j=1}^{m_k} w_{kj} x_{kj} \tag{2.1}$$

where

v_k = induced local field of neuron k
b_k = bias of neuron k
m_k = number of inputs of neuron k
w_{kj} = weight of synapse j of neuron k
x_{kj} = input j of neuron k

Activation Function and Output:

As shown in figure 2.1, *v* is the input of the activation function φ(*v*). There are several different kinds of activation functions, each of which is beneficial in different situations. The commonality between all activation functions is that they produce the neuron output *y*, which is restricted to values with a maximum value of 1 and a minimum value of either 0 or -1 (depending on the application). There are three basic activation functions that range from 0 to 1: threshold, piecewise-linear, and sigmoid. Another commonly used function, known as the hyperbolic tangent, is a modified version of the sigmoid having a range of -1 to 1. Figure 2.2 shows the four activation functions graphically, while equations 2.2 – 2.5 explicitly define them.

Threshold function:
$$\varphi(v) = \begin{cases} 1 \text{ if } v \geq 0 \\ 0 \text{ if } v < 0 \end{cases} \tag{2.2}$$

Piecewise-linear function:
$$\varphi(v) = \begin{cases} 1, & v \geq \frac{1}{2} \\ v, & \frac{1}{2} > v > -\frac{1}{2} \\ 0, & v \leq -\frac{1}{2} \end{cases} \tag{2.3}$$

Sigmoid function:
$$\varphi(v) = \frac{1}{1+\exp(-av)} \tag{2.4}$$

Tanh function:
$$\varphi(v) = \tanh(av) \tag{2.5}$$

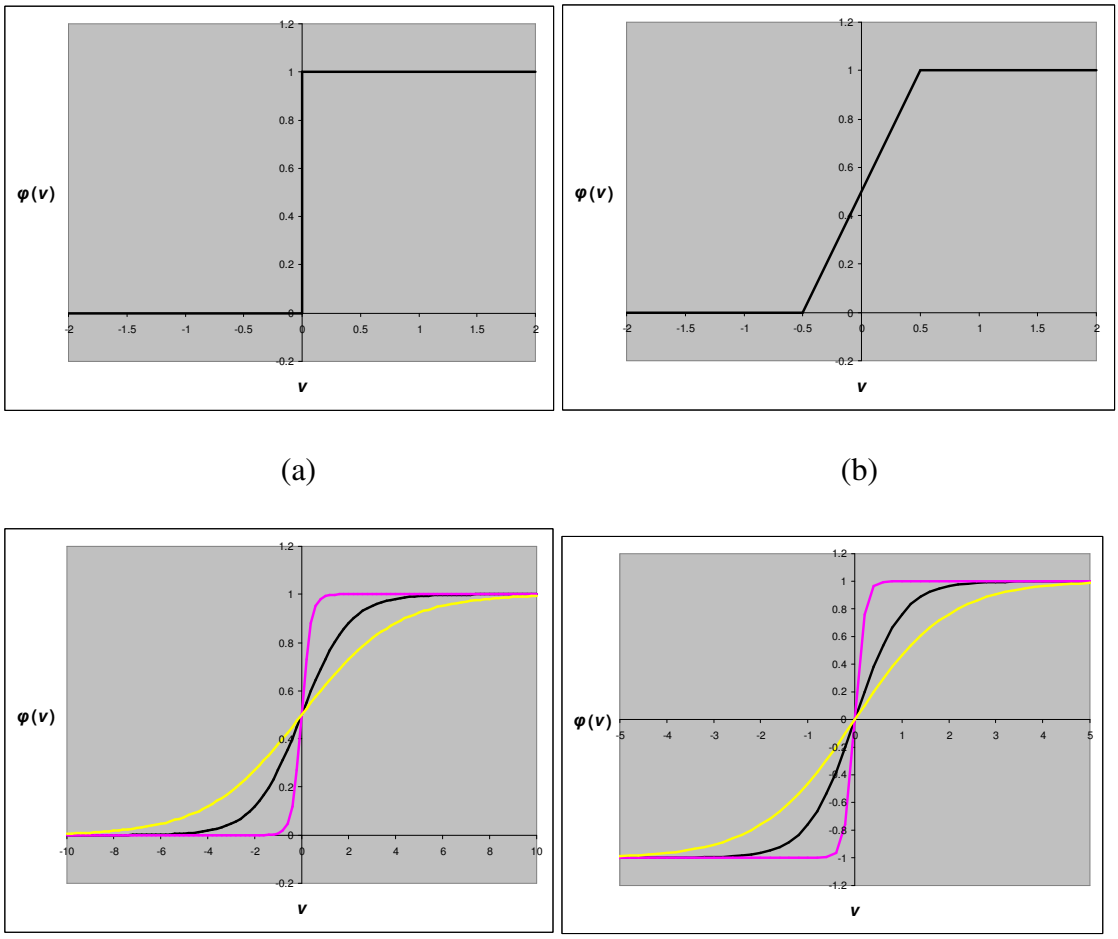

Figure 2.2 – Four commonly used activation functions. (a) Threshold, (b) Piecewise-linear, (c) Sigmoid, and (d) Tanh.

As seen in equations 2.4 and 2.5, the sigmoid and tanh functions include an *a* parameter, which can be modified to vary the steepness of the activation function. In figure 2.2, the sigmoid and tanh functions are shown with three different *a* values: 5, 1, and 0.2. As *a* is increased, the steepness of the activation function increases. Depending on the input value ranges, architecture, and application of the neural network, *a* can be modified to decrease the training time and error. The sigmoid and tanh functions allow neural networks solve non-linear problems, since both functions are themselves non-linear.

2.1.2. Network Architecture

As mentioned in section 2.1.1, the output of a neuron can be connected to the input of another neuron. In fact, one neuron's output can be connected to any number of other neurons' inputs, allowing numerous possible ways of combining neurons to form a neural network. Neural networks are commonly organized in a layered fashion, in which neurons are organized in the form of layers. There are three kinds of layers: input layer, hidden layers, and output layer.

The input layer is made up of the input nodes of the neural network. The output layer consists of neurons which produce the outputs of the network. All layers which do not produce outputs, but instead produce intermediate signals used as inputs to other neurons, are considered hidden layers. Figure 2.3 shows a simple neural network, which consists of an input layer and an output layer (circles represent neurons and arrows represent synapses). In this network, three neurons process the three inputs to produce three outputs.

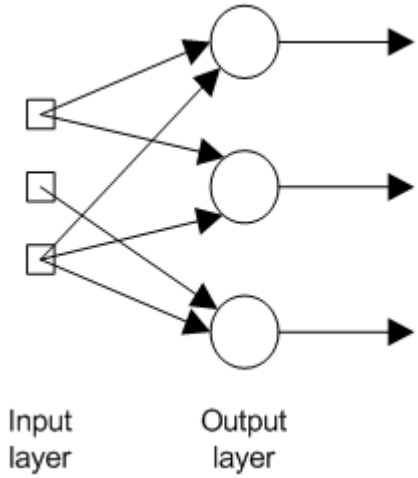

Figure 2.3 – A simple feedforward neural network.

Figure 2.4 shows a slightly more complex network. This network consists of all three layers: an input, hidden, and output layer. When designing the architecture of a neural network, there is no limit on the number of layers or the number of neurons within each of those layers. Some complex tasks require architectures that contain multiple hidden layers.

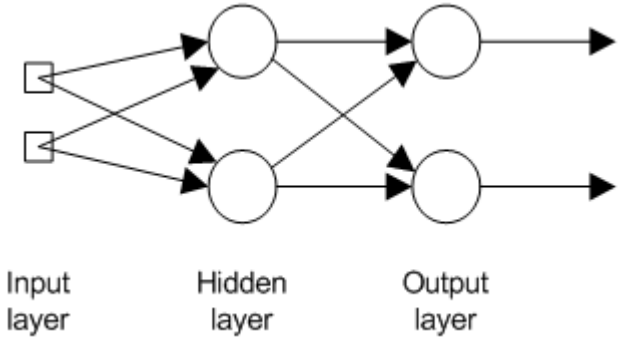

Figure 2.4 – A two layer fully-connected feedforward neural network.

There is another distinction between the two above networks, apart from the number of layers or neurons. The network in figure 2.4 is known as fully-connected, since each of the neurons at each layer are connected to all possible sources from the previous layer. The network in figure 2.3 is not fully-connected due to the lack of connections between the input nodes and each of the output neurons.

Both of the above networks are also known as feedforward networks. In a feed-forward network, all signals travel from the input nodes towards the output nodes. That is to say, the output of each neuron does not connect to neurons within its own layer or any previous layers. Figure 2.5 shows an example of a feedback (or recurrent) network. If a single feedback loop exists, such as the one between the output and hidden layers, a network is considered a feedback network.

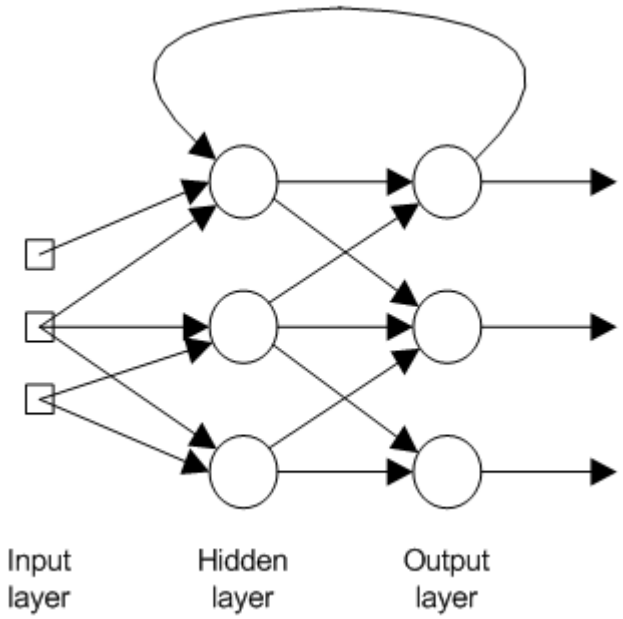

Figure 2.5 – A feedback network.

These three networks illustrate the endless number of possible architectures to choose from when creating a neural network. In general, more complex tasks require more hidden layers and more neurons within each layer.

2.1.3. Training

Neural networks can be trained in several different ways, the most common of which is called the back-propagation algorithm. The details of the back-propagation algorithm are discussed in section 2.1.4. The only information necessary for this section is that the back-propagation algorithm attempts to minimize error.

In neural network training, error represents the difference between the produced outputs and the desired outputs. In order to train a network, it must be trained with a set of training samples (training set). Each training sample in the training set is made up of a set of inputs and the desired set of outputs. The training set should be a representative collection of input/output samples (all possible samples if available). Table 2.1. contains the training set for the XOR function, in which all possible sets of inputs and desired outputs can be provided.

Table 2.1 – XOR function training set

Inputs	Output
00	0
01	1
10	1
11	0

Initially, all synaptic weights are chosen at random throughout a neural network. Back-propagation learning is an iterative process, which modifies the synaptic weights to minimize error in each training iteration. During each iteration, the input set (from the

training sample) is fed into the network. The produced outputs are compared to the expected outputs (from the training sample) and the error is computed. The error is used to modify the weights throughout the network in order to bring the outputs closer to their expected values.

Depending on the complexity of the application, a neural network requires many epochs of training before it produces an acceptably low error. An epoch consists of training a network with the entire training set. As seen in table 2.1, an epoch for the XOR function consists of four training samples. It is important to note that the order in which the samples are processed should be randomized. Otherwise, the neural network might favor some samples over others and perform poorly overall.

A set of samples, sometimes referred to as the evaluation set, is used to determine the mean squared error (MSE). If possible, the evaluation set should contain a different set of samples than the training set in order to test the network's ability to generalize. Generalization is the ability of a neural network to produce accurate results for inputs not found in the training set. The mean squared error for a single sample (ε_j) is calculated as shown in equation 2.6, where n is the number of outputs, d_i is desired output i, and o_i is the actual output i. The MSE for the entire evaluation set (E_{MSE}) is an average of the individual sample MSE values, as shown in equation 2.7, where m is the total number of samples in the evaluation set and j is the index.

$$\varepsilon_j = \frac{\frac{1}{2}\sum_{i=1}^{n}(d_i - o_i)^2}{n} \tag{2.6}$$

$$E_{MSE} = \frac{\sum_{j=1}^{m}\varepsilon_j}{m} \tag{2.7}$$

The total number of training epochs can vary, and usually relates with the maximum allowed E_{MSE}. The training continues until E_{MSE} has reached an acceptable value. However, networks can sometimes reach a local minimum in the error space (as opposed to the global minimum). In this case, the error cannot be lowered any further and the neural network is unacceptable. If a neural network has reached a local minimum, the only possible recourse is to restart the process with a new neural network that either has a different architecture, new randomly selected weights, or both.

2.1.4. Back-propagation Algorithm

The back-propagation algorithm is used to modify the synaptic weights throughout a neural network in order to minimize error. It is an iterative process, which modifies the network one training sample at a time. During each iteration the error signal travels backwards through the network, starting at the output neurons and ending at the input synapses. The derivation of the back-propagation algorithm is not shown. Instead, a few of the important equations are described so that the effects of the learning rate and activation function can be shown.

The correction for a weight is defined as

$$\Delta w_{j,i} = \eta \delta_j y_i \tag{2.8}$$

where η is the learning rate, $\Delta w_{j,i}$ is the change in weight connecting neuron *i* in layer L to neuron *j* in layer L+1, δ_j is the local gradient, and y_i is the output of neuron *i*. The learning rate affects how much a weight will change based on the error and can be chosen to be any real number. The local gradient is the error signal that travels backwards

through the network and is based on activation function, as well as whether the neuron in question is an output and non-output neuron. Only the correction functions for sigmoid and tanh activation functions are shown, as they are the only functions used in this thesis.

For output neurons using the sigmoid activation function

$$\delta_j = a(d_j - o_j)o_j(1-o_j) \tag{2.9}$$

where a is the sigmoid parameter defined in equation 2.4, d_j is desired output, and o_j is the actual output. The correction function for output neurons using the sigmoid activation function is therefore

$$\Delta w_{j,i} = a\eta(d_j - o_j)o_j(1-o_j)y_i \tag{2.10}$$

For non-output neurons using the sigmoid activation function

$$\delta_j = ay_j(1-y_j)\sum_k \delta_k w_{k,j} \tag{2.11}$$

where y_j is the output of neuron j, δ_k is the local gradient of neuron k in the next layer, and $w_{k,j}$ is the weight connecting neuron j with each of the neurons in the next layer. The correction function for non-output neurons using the sigmoid activation function is therefore

$$\Delta w_{j,i} = a\eta y_j(1-y_j)y_i\sum_k \delta_k w_{k,j} \tag{2.12}$$

For output neurons using the tanh activation function

$$\delta_j = a(d_j - o_j)(1 - o_j)(1 + o_j) \tag{2.13}$$

where a is the tanh slope defined in equation 2.5. The correction function for output neurons using the tanh activation function is therefore

$$\Delta w_{j,i} = a\eta(d_j - o_j)(1 - o_j)(1 + o_j)y_i \tag{2.14}$$

For non-output neurons using the tanh activation function

$$\delta_j = a(1 - y_j)(1 + y_j)\sum_k \delta_k w_{k,j} \tag{2.15}$$

The correction function for non-output neurons using the sigmoid activation function is therefore

$$\Delta w_{j,i} = a\eta(1 - y_j)(1 + y_j)y_i \sum_k \delta_k w_{k,j} \tag{2.16}$$

2.1.5. Neural Network Applications

Neural networks offer the most advantages when used in applications where large input/output pairs are available. The relationship between the input/output may be unknown or complex, which is ideal for neural networks since they learn the patterns within the data. A well-trained neural network can also generalize and accurately generate the outputs for inputs outside the training set. Neural networks have been used in a variety of fields due to their many advantages.

A common use of neural networks is in pattern recognition and classification due to their highly parallel structure. In [18], neural networks were used to recognize digits in a picture. In a more complex application, neural networks were used to perform facial recognition [28]. In the medical field, neural networks have been used in the diagnosis and prognosis of cancer [20, 25]. Neural networks have been used to classify data in high-energy physics experiments. Recently, neural networks were used to classify the measurement of the spin structure of deuterons [27]. In astronomy, neural networks have been applied to detect patterns in spatial image data mining [26]. Neural networks have also been used in many control systems applications, such as controlling a DC-DC converter [21].

2.2. The Genetic Algorithm

The genetic algorithm (GA) is based on Charles Darwin's theory of evolution and can be used to solve a wide variety of problems. In general, a GA is defined by an iterative process, comprised of six key stages: generating initial population, evaluation, ranking, selection, crossover, and mutation [13]. This iterative process resembles what occurs when organisms reproduce in nature. The first step necessary to understand the GA is to learn about the overall structure and terminology of the GA. The next step is to learn about each of the six stages, some of which require a more detailed explanation than others.

2.2.1. Structure and Terminology

Each genetic algorithm can differ, but all genetic algorithms have a few things in common. Each GA has a population certain size, which is made up of individuals. Each individual represents a potential solution to the problem the GA is trying to solve. For the remainder of chapter 2, a GA that is used to find the dimensions of a rectangle with the largest area possible (seven units by seven units max) will be examined. This might seem to be a trivial task (a seven by seven rectangle is the obvious answer), but it illustrates how the GA works and could be applied to a much more complex task.

As the problem dictates, each individual in the population represents a rectangle. The height and width will be used to define a rectangle; and each of them will be represented by a gene. A gene usually consists of a string of binary bits, the number of which varies for each application. For the sake of simplicity, assume that the width and height of the rectangle must be an integer between zero and seven. With this assumption, each gene will require three bits to represent the entire range. Table 2.2 shows an example of a genotype of an individual.

Table 2.2 – Possible genotype of an individual in max rectangle area GA.

Gene 1 (width)	Gene 2 (height)
010	101

While the genotype of an individual is made up of the values of its genes, the phenotype of an individual is the actual meaning the genes represent (in this case, the width and height of a rectangle). As seen in table 2.2, this individual's genes represent a width of four units and a height of five units, making a rectangle with an area of twenty units. With an understanding of genotypes and phenotypes, it is now possible to examine the general structure of a GA, as shown in figure 2.6.

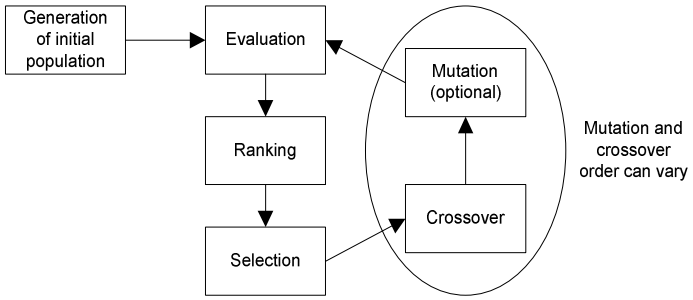

Figure 2.6 – The general structure of a GA.

Each of the six steps is explained in detail in the following sections, but a short description of the steps is in order. The first step involves randomly generating the initial population. Each individual in the population (plus the newly created individual starting in the second iteration) is given a fitness value in the evaluation step. All of the individuals are then ranked based on their fitness. In the selection step, two individual are selected to reproduce based on their rank. The two selected individuals' genotypes are used to produce a new individual in the crossover step. Finally, the new individual's genes may be mutated (bits are randomly flipped) and the next iteration begins.

2.2.2. Initial Population and Evaluation

The basis of the GA is that the population of a fixed size is improving with each iteration of the algorithm. An initial population of size *P* is created by generating *P* individuals with randomly selected genes. Expanding on the example of the rectangle GA, assume a population of size five. Table 2.3 shows a possible randomly generated initial population.

Table 2.3 – Possible initial population of max rectangle area GA.

Individual ID	Gene 1 (width)	Gene 2 (height)
A	011	101
B	111	011
C	101	100
D	010	010
E	010	011

After the initial population has been generated, the population must be evaluated. Evaluation is one of the most important steps in the GA process, since it defines what constitutes an individual as 'fit'. At the end of the evaluation, each individual will have a fitness value. The method for calculating the fitness of an individual is completely up to the GA designer, with the requirement that an individual with higher fitness represent a better solution than an individual with lower fitness. In this example, a rectangle with a large area represents a better solution than a rectangle with a small area. The fitness criterion for this example could be defined as shown in equation 2.17, where f_i, w_i, and h_i are the fitness, width, and height of individual *i*, respectively.

$$f_i = w_i * h_i \tag{2.17}$$

Using the above equation, table 2.4 shows the fitness values of all individuals in the initial population. The individuals with phenotypes representing larger areas have a higher fitness.

Table 2.4 – Initial population fitness for max rectangle area GA.

Individual ID	f_i
A	**15**
B	**21**
C	**20**
D	**4**
E	**6**

The evaluation step also occurs after a new individual has been generated, and is used to maintain the population at a constant size *P*. Like in the theory of evolution, the more fit individuals survive: the least fit individual in the population is replaced if the new individual has a higher fitness value. For example, a new individual with a fitness value of ten would replace individual D in the population. However, a new individual with a fitness value of three would not be allowed into the population. Due to this requirement, the population is constantly improving and represents a better set of possible solutions after each iteration.

2.2.3. Ranking and Selection

In nature, two parents' genes are used to construct their offspring's genotype. The offspring therefore exhibits similar traits and behaviors as the parents. Similarly in the GA, a new individual is created by combining two other individual's genes. The ranking and selection stages of the GA decide which two individuals are selected to reproduce.

Following the evaluation step, all of the individuals in the population are given a fitness rank (r_{fit_i}) based on their fitness values. Table 2.5 shows the ranking for this example. As mentioned earlier, a higher fitness value represents a better possible solution. At any time, the top ranked individual represents the best current solution. The GA can run for as many iterations as desired, usually until the top ranked individual meets some minimum threshold criteria.

Table 2.5 – Ranked initial population for max rectangle area GA.

r_{fit_i}	Individual ID	f_i
1	B	21
2	C	20
3	A	15
4	E	6
5	D	4

After the population is ranked, two individuals must be selected for reproduction in the selection stage. Since parents with high fitness are likely to produce individuals with high fitness, the selection method should favor higher ranked individuals for reproduction. There are two widely used selection methods: rank-based selection and fitness-based selection. In the fitness-based approach, the probability of being selected is

based on an individual's relative fitness in the population. The probability of individual *i* being selected is defined in equation 2.18, where *P* is the population size.

$$P_{fitness\_based\_selection}(i) = \frac{f_i}{\sum_{j=1}^{P} f_j} \tag{2.18}$$

Rank-based selection, as proposed in [22] and [23], is based on the rank instead of the fitness. The probability is based on the rank of each individual (r_{fit_i}). The individual with the highest fitness value is ranked 1, while the one with the smallest fitness value is ranked P. Equation 2.19 defines the probability of individual *i* being selected.

$$P_{rank\_based\_selection}(i) = \frac{P - r_{fit_i} + 1}{\sum_{j=1}^{P} j} \tag{2.19}$$

where P is the total number of individuals.

Table 2.6 compares the probability of each individual being selected using both selection methods. In this example, as the table shows, rank-based selection provides a bigger separation between the first and second ranked individuals when their fitness is close.

Table 2.6 – Probability of selection for initial population using both rank and fitness based selection.

r_{fit_i}	Individual ID	f_i	$P_{rank\_based\_selection}(i)$	$P_{fitness\_based\_selection}(i)$
1	B	21	**0.33**	**0.32**
2	C	20	**0.27**	**0.30**
3	A	15	**0.20**	**0.23**
4	E	6	**0.13**	**0.09**
5	D	4	**0.07**	**0.06**

2.2.4. Crossover and Mutation

There are two commonly used ways of generating new individuals in GA: mutation and crossover. As shown in Figure 2.6, the order and inclusion of mutation and crossover can be arbitrarily made by a GA designer. Mutation is the simpler of the two processes and is examined first.

In mutation, an arbitrary number of bits are flipped in an individual's genome. The probability of a bit flipping, the selection of which bits will be flipped, and the number of bits to be flipped are all parameters that a GA designer can adjust to produce a better solution. If crossover is not used, then new individuals are generated by mutating a single individual selected in the selection stage. Mutation can be applied randomly as an independent operator on any individual in the population, or just to the offspring produced with crossover. Table 2.7 shows one possible offspring F produced by mutating individual E in the example GA. As shown in the table, the new individual F has a fitness value double that of individual E's.

Table 2.7 – Possible offspring F produced by mutating individual E with the two flipped bits in bold.

Individual ID	Gene 1 (width)	Gene 2 (height)	f_i
E	010	011	6
F	**1**10	01**0**	12

The more complex crossover process is commonly used in combination with mutation. In crossover, the genes of two selected parents are combined to produce an offspring, similar to the reproduction of organisms in nature. There are several commonly

used forms of crossover: 1-point crossover, multi-point crossover, and uniform crossover [13].

Continuing with the rectangle example, assume that individuals B and C were selected for crossover in the selection stage. In 1-point crossover, the genome (which consists of gene 1 and gene 2) of both parent individuals is partitioned into two sections at an arbitrary point. In this example, the genome is split down the middle. The genome of the new individual is created by randomly selecting each of the two genome sections from the two parents. In general, the probability of selecting genes from a particular parent is 50%, but this can be adjusted by a GA designer. Assuming 50% is used, in other words, the first half of the genome has an equal probability of coming from either parent B or C; and the same with the second half.

Table 2.8 shows a possible new individual G created using the above 1-point crossover of parents B and C. The first half of the genome was selected from parent C and the second half of the genome was selected from parent B. In this example, the new individual G turned out to have a fitness of fifteen. Generally, the new individual G would undergo mutation following the crossover.

Table 2.8 – Possible offspring G produced by a 1-point crossover of parents B and C. The selected bits are in bold.

Individual ID	Genome		f_i
B	111	**011**	21
C	**101**	100	20
G	**101**	**011**	15

As the name implies, multiple points are selected to split up the genome in multi-point crossover. In uniform crossover, every single bit is selected individually. Uniform crossover is a multi-point crossover, with the maximum number of points of separation for a given genome. Tables 2.9 and 2.10 show one possible generation of individuals H and I, generated by multi-point and uniform crossover, respectively.

Table 2.9 – Possible offspring H produced by a multi-point (2) crossover of parents B and C. The selected bits are in bold.

Individual ID	Genome			f_i
B	**11**	10	11	21
C	10	**11**	**00**	20
H	**11**	**11**	**00**	28

Table 2.10 – Possible offspring I produced by a uniform crossover of parents B and C. The selected bits are in bold.

Individual ID	Genome						f_i
B	**1**	1	**1**	0	**1**	**1**	21
C	1	**0**	1	**1**	0	0	20
I	**1**	**0**	**1**	**1**	**1**	**1**	35

Offspring H inherited one third of its genome from parent B and two thirds from parent C, producing a fitness of 28. Offspring I inherited two thirds of its genome (i.e., 4 bits) from parent B and a third from parent C (i.e., 2 bits), producing a fitness of 35.

2.2.5. Genetic Algorithm Applications

The genetic algorithm is commonly used in applications where there isn't a clearly defined procedure for reaching a solution. Based on Charles Darwin's theory of evolution [24], the genetic algorithm solves problems by incrementally improving a set of possible solutions. After enough iterations, the set of possible solutions improves, with

the fittest member of the set representing the solution to the problem at hand. Like neural networks, the genetic algorithm has been applied in a wide variety of fields.

Scheduling has always been a difficult problem without a well defined procedure for solution. For this reason, the genetic algorithm is commonly used for scheduled tasks. In a major UK hospital, the genetic algorithm was used to schedule nurse work hours [31]. In the field of computing, the genetic algorithm has been used in task scheduling for multiprocessor systems [30].

In another computing application, internet packet routing, the genetic algorithm was used to set the weights in a cost function for determining optimal packet path [29]. In the field of medicine, the genetic algorithm was used to simulate HIV evolution [32]. The algorithm has also been applied in surface registration, which is a process used in 3D object recognition and 3D model reconstruction [33].

2.3. Optimizing Neural Network Performance Using Genetic Algorithm

Neural networks and the genetic algorithm are both powerful tools that are modeled after natural phenomena. Neural networks are modeled after the brain, which is highly parallel, and offers many advantages when solving pattern recognition and classification problems. The GA is based on the theory of evolution and survival of the fittest [24] and has been applied to solve many optimization problems.

Neural networks offer many advantages in a variety of applications, but are ineffective if they are not properly designed. There are many choices when designing a neural network (NN) but a poor selection of any one parameter can render the NN useless. There have been some attempts to generate formulas or rules for designing the

structure of a NN. In [6], a formula was developed to generate a range of the desired number of hidden neurons, *h*, based on the number of inputs and outputs of a NN:

$$h = \sqrt{n + m} + \ (1 \sim 10) \tag{2.20}$$

where n is the number of inputs, m is the number of outputs (square root result is rounded up). The formula produces a range, which is not particularly useful because several networks need to be trained and evaluated in order to find the best one. Another problem with this formula is that it assumes the number of inputs and outputs is directly correlated with the complexity of the problem, which is not always the case. Furthermore, this formula only applies to a network with one hidden layer.

Since there are no well-defined procedures for selecting the parameters of a NN for a given application, finding the best parameters can be a case of trial and error. There are many papers, like [1, 4, 8] for example, in which the authors arbitrarily choose the number of hidden layer neurons, activation function, and number of hidden layers. In [10], networks were trained with 3 to 12 hidden neurons and found that 9 was optimal for that specific problem. The GA had to be run 10 times, one for each of the network architectures.

Since selecting NN parameters is more of an art than a science, it is an ideal problem for the GA. The GA has been used in numerous different ways to select the architecture, prune, and train neural networks. In [27], a simple encoding scheme was used to optimize a multi-layer NN. The encoding scheme consisted of the number of neurons per layer, which is a key parameter of a neural network. Having too few neurons does not allow the neural network to reach an acceptably low error, while having too

many neurons limits the NN's ability to generalize. Another important design consideration is deciding how many connections should exist between network layers. In [18], a genetic algorithm was used to determine the ideal amount of connectivity in a feed-forward network. The three choices were 30%, 70%, or 100% (fully-connected).

In general, it is beneficial to minimize the size of a NN to decrease learning time and allow for better generalization. A common process known as pruning is applied to neural networks after they have already been trained. Pruning a NN involves removing any unnecessary weighted synapses. In [14], a GA was used to prune a trained network. The genome consisted of one bit for each of the synapses in the network, with a '1' represented keeping the synapse, while a '0' represented removing the synapse. Each individual in the population represented a version of the original trained network with some of the synapses pruned (the ones with a gene of '0'). The GA was performed to find a pruned version of the trained network that had acceptable error. Even though pruning reduces the size of a network, it requires a previously trained network. The algorithm developed in this research optimizes for size and error at the same time, finding a solution with minimum error and minimum number of neurons.

There are several different kinds of pruning algorithms that do not involve the GA. In [11], three pruning algorithms were compared and analyzed to create the Hybrid Sensitivity Analysis with Repruning (HSAR) algorithm. In this research, the weight decay, sensitivity analysis, and aforementioned GA based evolutionary pruning techniques were discussed. The weight decay method is based on the idea that smaller weights have less effect on the network's output and can be removed. A penalty function is used to force smaller weights towards zero. Sensitivity analysis works by removing

different weights and measuring the effect on the network's performance. The weights with the least effect on the network are then removed. Again, these approaches require two steps (training a network and then pruning it).

Another critical design decision, which is application-specific, is the selection of the activation function. Depending on the problem at hand, the selection of the correct activation function allows for faster learning and potentially a more accurate NN. In [20], a GA was used to determine which of several activation functions (linear, logsig, and tansig) were ideal for a breast cancer diagnosis application.

Another common use of GA is to find the optimal initial weights of back-propagation and other types of neural networks. As mentioned in [3], genetic algorithms are good for global optimization, while neural networks are good for local optimization. Using the combination of genetic algorithms to determine the initial weights and back propagation learning to further lower error takes advantage of both strengths and has been shown to avoid local minima in the error space of a given problem. Examining the specifics of the GA used in [1] shows the general way in which many other research papers use GA to determine initial weights.

In [1], this technique was used to train a NN to perform image restoration. The researchers used fitness based selection on a population of 100, with each gene representing one weight in the network that ranged from -1 to 1 as a floating point number. Dictated by the specifics of the problem, the structure of the neural network was fixed at nine input and one output node. The researchers arbitrarily chose five neurons for the only hidden layer in the network. To determine the fitness of an individual, the initial weights dictated by the genes are applied to a network which is trained using back

propagation learning for a fixed number of epochs. Individuals with lower error were designated with a higher fitness value. In [3, 9], this technique was used to train a sonar array azimuth control system and to monitor the wear of a cutting tool, respectively. In both cases, this approach was shown to produce better results that when using back-propagation exclusively.

In [7], the performance of a two back propagation neural networks were compared: one with GA optimized initial weights and one without. The number of input, hidden, and output neurons were fixed at 6, 25, and 4, respectively. Other parameters such as learning rate and activation functions were also fixed so that the only differences between the two were the initial weights. The authors set an error performance target and compared how long it took the two networks to reach it. In one of the runs, the GA optimized network reached the target in 77 epochs while the non-optimized network got stuck at a local minima after 347 epochs. This research showed that using a GA to optimize initial weights produces networks that train faster and escape local minima in error space. In [1, 3, 7, 9], each of the synaptic weights was encoded into the genome as a floating point number (at least 16 bits), making the genome very large. The algorithm developed in this research only encodes a random number seed, which decreases the search space by many orders of magnitude.

Determining the initial values using the GA has improved the performance of non-back propagation networks as well. In [2], a GA was used to initialize the weights of a Wavelet Neural Network (WNN) to diagnose faulty piston compressors. WNNs have an input layer, a hidden layer with the wavelet activation function, and an output layer. Instead of using back propagation learning, these networks use the gradient descent

learning algorithm. The structure of the network was fixed, with one gene for each weight and wavelet parameter. Using the GA was shown to produce lower error and escape local minima in the error space.

Neural networks with feedback loops have also been improved with GA generated initial weights. In [4], a GA was used to optimize the initial weights of an Elman Recurrent Network (ERN) to produce short-term load forecasting model for a power system. ERNs have an input layer, middle layer, continue layer, and output layer, as shown in Figure 2.7.

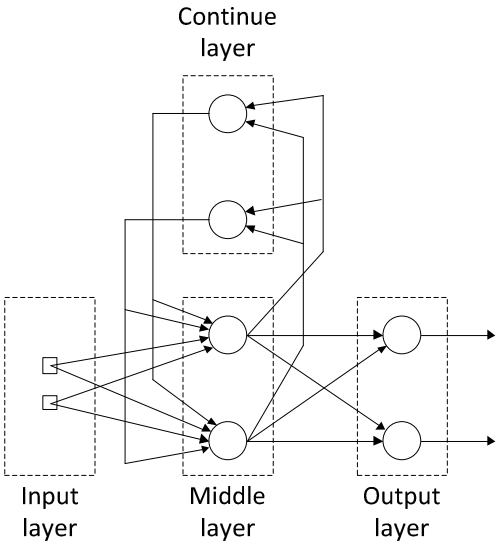

Figure 2.7 – The structure of an Elman Recurrent Network.

The continue layer delays the previous middle layer outputs and feeds them back into the middle layer neurons. This allows the network to be more dynamic and sensitive to historic data. The network had a fixed size of 12 input neurons, 4 output neurons, 4 continue neurons, and 19 hidden neurons. The GA was used, along with standard back-propagation, to optimize the initial weights of synapses between the input layer to middle

layer (12x19=228 synapses) and middle layer to output layer (19x4=76 synapses). The continue layer to middle layer weights changed according to ERN's dynamic feature. This research showed that using GA in combination with NN is more suitable for problems that have very large data sets. This algorithm's genome contained 304 genes for all of the synaptic weights, which would take 4864 bits to represent if 16-bit floating point numbers were used. That means the search space consisted of 2^4864, or 1.62e1464 possible individuals.

Genetic algorithms have also been used in the training process of neural networks, as an alternative to the back-propagation algorithm. In [16] and [19], genes represented encoded weight values, with one gene for each synapse in the neural network. It is shown in [13] that training a network using only the back-propagation algorithm takes more CPU cycles than training using only GA, but in the long run back-propagation will reach a more precise solution.

In [5], the Improved Genetic Algorithm (IGA) was used to train a NN and shown to be superior to using a simple genetic algorithm to find initial values of a back propagation neural network. Each weight was encoded using a real number instead of a binary number, which avoided lack of accuracy inherent in binary encoding. Crossover was only performed on a random number of genes instead of all of them, and mutation was performed on a random digit within a weight's real number. Since the genes weren't binary, the mutation performed a "reverse significance of 9" operation (for example 3 mutates to 6, 4 mutates to 5, and so on). The XOR problem was studied, and the IGA was shown to be both faster and produce lower error. Similar to [4], this algorithm requires a large genome since all the weights are encoded.

Previously, genetic algorithms were used to optimize a one layered network [15], which is too few to solve even moderately complex problems. Many other genetic algorithms were used to optimize neural networks with a set number of layers [1, 2, 4, 7, 9, 12, 17]. The problem with this approach is that the GA would need to be run once for each of the different number of hidden layers. In [12], the Variable String Genetic Algorithm was used to determine both the initial weights of a feed forward NN as well as the number of neurons in the hidden layer to classify infrared aerial images. Even though the number of layers was fixed (input, hidden, and output), adjusting the number of neurons allowed the GA to search through different sized networks.

In summary, the papers mentioned above studied genetic algorithms that were lacking in several ways:

1. They do not allow flexibility of the number of hidden layers and neurons.
2. They do not optimize for size.
3. They have very large genomes and therefore search spaces.

The GA developed in this thesis addresses all of these issues. It searches through networks with one through four hidden layers, with anywhere from one to eight neurons in each. Also, the developed algorithm optimizes for error and size concurrently. Finally, the developed algorithm drastically minimizes the search space by storing a random number generator seed instead of all synaptic weights for a total genome size of 24 bits.

Chapter 3: The Scaling Factor Based Genetic Algorithm

This chapter describes the Scaling Factor Based Genetic Algorithm, which is used to optimally select neural network parameters by minimizing both error and network size. This algorithm combines the advantages of several previously mentioned algorithms and adds new features to improve performance. The genetic algorithm has the following benefits:

1. Can optimize for either size, error, or a combination of both. Only [11 and 14] describe algorithms that minimize networks size by pruning, but they require already trained networks.
2. Allows optimization for a variable number of hidden layers and neurons per hidden layer. The number of hidden layers were fixed in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12]
3. Includes a seed number instead of all synaptic weights, which greatly reduces the search space and increases performance. None of the researched genetic algorithms use a seed number in their genome, and instead store one genome for each synaptic weight (either 32 or 64 bits each depending on what kind of floating point number is used) [1, 2, 3, 4, 7, 8, 9, 10, 12].
4. Uses an encoded learning rate and activation function slope genes instead of containing floating point numbers to reduce search space and increase performance. The learning rates and activation function slopes either chosen arbitrarily, possibly by trial and error, or stored as a 32 or 64 bit floating point number in the genome [1, 2, 3, 4, 7, 8, 9, 10, 12].

The proposed algorithm is described in the next section. The application examples that are used to test the algorithm are included in section 3.2. Finally, computer simulation results are presented and discussed in section 3.3.

3.1. Algorithm Definition

For simplicity, the genetic algorithm in this thesis only optimizes multi-layer feed-forward neural networks. In this section, each of components and procedures of the genetic algorithm are described in detail. First, the genome, which describes the relationship between each genotype and its phenotype, is explained. Secondly, the procedure for determining the fitness of each individual is described. Finally, the selection, crossover, and mutation procedures are examined.

3.1.1. Genome

The genome of the genetic algorithm is made up of twenty four bits which represented nine genes, as shown in table 3.1. Each gene is chosen to represent a different neural network parameter. Each gene's purpose is explained below.

Table 3.1 – Genome Layout

Gene number	1	2	3	4	5	6	7	8	9
Gene bits	xx	xxx	xxx	xxx	xxx	xxx	xxx	xxx	x

Gene 1 (number of hidden layers):

Gene 1 consisted of two bits and could therefore take on four different values. The two bits, read as a binary number, represent one less than the number of hidden layers in the neural network. Therefore, the genetic algorithm searches for neural networks having between one to four hidden layers. The option of zero hidden layers is removed because most problems are too complex for such a small network. Three bits would have allowed eight hidden layers, which is far more than the number necessary for the problems that were studied. For the sake of simplicity, the maximum of four hidden layers is chosen, which reduces the search space of the genetic algorithm and facilitates finding an optimal solution in fewer generations.

Genes 2 – 5 (number of neurons in hidden layer):

Since there are a maximum of four hidden layers, each of these four genes represents the number of neurons in the corresponding hidden layer. Gene 2 represents the number of hidden neurons in the first layer, gene 3 corresponds to the second layer, and so on. These genes, read as binary numbers, also represent one less than the actual number of hidden neurons in the corresponding layer (allowing anywhere between one and eight neurons per hidden layer). Zero neurons are not allowed since at least one neuron must exist in each hidden layer. The maximum number of hidden neurons is selected to be eight (i.e., 3 bits) in order to keep the size of the search space in a reasonable range.

Similar to other genetic algorithms [16, 19], the genome selected in this research has a hierarchical structure. Genes 2 through 5 are dependent on gene 1, which is at the top of the gene hierarchy as shown in figure 3.1. Genes 2 through 5 are used depending on the values of gene 1. For example, if gene 1 has a value of '01' (i.e., two hidden layers) then genes 2 and 3 would be meaningful, while genes 4 and 5 would not since only two hidden layers exist. Even if some of the genes are inactive, they still undergo crossover and mutation throughout the generations. Inactive genes currently in the population can be activated in new individuals due to a change in gene 1 caused by crossover and mutation.

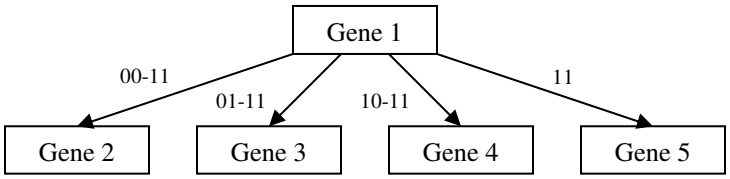

Figure 3.1 – Hierarchical genes. Arrows represent which values of gene 1 activate the corresponding gene.

Gene 6 (learning rate):

The learning rate, η, is an integral part of the back-propagation algorithm and therefore an important neural network parameter. If the learning rate is too small, then the back-propagation algorithm is more likely to reach the absolute minimum in error space but also much slower (requiring more epochs of execution). On the other hand, if the learning rate is too large, then the back-propagation algorithm will minimize error more quickly but is also more likely to get stuck in a local minimum in error space [5, 7, 9]. The genetic algorithm studied in this research allows for the optimal learning rate to be selected; one that allows both absolute minimization of error and fast learning.

Gene 6 is chosen to consist of three bits and represents an encoding of eight different learning rate values. The values are chosen to be 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1.0, and 2.0 to cover three orders of magnitude. The eight genotypes are listed with their corresponding phenotypes in table 3.2. Other genetic algorithms [15] use a floating point representation for the learning rate gene, which allows a more continuous set of values to be searched. The encoding scheme is chosen to limit the search space to only eight values which in turn shortens the number of generations required to find an adequate solution.

Table 3.2 – Gene 6 encoding for the learning rate.

Gene 6 value	000	001	010	011	100	101	110	111
Learning rate	0.01	0.02	0.05	0.1	0.2	0.5	1.0	2.0

Gene 7 (activation function slope):

Similar to the learning rate, the activation function slope is a critical parameter for Neural Network design and learning. Again, instead of using a continuous-value floating point number as the gene, a three-bit encoding is used to represent three orders of magnitude of the activation function slope. By using a three bit encoding instead of a sixteen or thirty two bit floating point representation, the search space is limited and the algorithm can reach a solution in less generations. The encoding of gene 7 is shown in table 3.3.

Table 3.3 – Gene 7 encoding for the activation function slope.

Gene 7 value	000	001	010	011	100	101	110	111
Activation function slope	0.01	0.02	0.05	0.1	0.2	0.5	1.0	2.0

Gene 8 (random number seed):

Neural network learning is related with the randomly selected initial values of all of the synaptic weights. Most genetic algorithms have stored the initial values of all of the initial synaptic weights in the genome as genes [14, 15, 16, 17, 18, 19, 20]. Gene 8 is included for several reasons instead of including one gene for each synaptic weight.

Most neural networks are implemented in software, and the initial synaptic weights are therefore selected randomly using a software based random number generator (which can be seeded by an integer number to produce different random numbers). Using one gene for the seed of the random number generator instead of a set of initial weights can significantly minimize the search space. In order to store all of the synaptic weights,

the genome would have needed to be hundreds of bits longer and the genetic algorithm performance would have suffered greatly.

Also, since the genetic algorithm is chosen to search through an arbitrarily sized neural network, many of the synaptic weight genes would have been unused. For example, the gene for a synaptic weight in the third hidden layer would not be used if the genetic algorithm individual represented a two layer hidden network. In this thesis, the random number seed gene is chosen to be three bits long, representing eight different random number generator seeds.

Gene 9 (using sigmoid):

When designing a neural network, there are several different activation functions that can be used for every neuron in the network. For this thesis, the tanh and sigmoid activation functions are studied. Previous research [20] used one gene to represent which activation function would be used for each neuron. This means one gene was required for each neuron in the neural network, which could reach hundreds of neurons. Gene 9 is chosen to be a one bit value to represent which activation function ('0' for tanh and '1' for sigmoid) would be used by all of the neurons in the network.

3.1.2. Fitness

An important part of the genetic algorithm is the generation of new individuals by combining the genomes of two "fit" parents. In order to determine which individuals are more "fit" than others, a formula for calculating fitness is defined. The purpose of the genetic algorithm studied in this research is to find parameters that minimize both the error and size of a neural network. At the end of every generation, all of the individuals in

the population are ranked based on their E_{MSE} and size (number of synapses), with each individual given an error rank (r_{err}) and a size rank (r_{size}). The reverse error and reverse size rank are found using r_{err} and r_{size}. For example, in a population of fifty, the individual with the lowest E_{MSE} would have an error rank of one and a reverse error rank of fifty. After the reverse error ranks and reverse size ranks are determined for each individual, all of the fitness values are calculated using the following equations:

$$f_i = (rr_{err_i} * s_{err}) + (rr_{size_i} * s_{size}) \tag{3.1}$$

where

f_i = fitness of individual i

rr_{err_i} = reverse error rank for individual i

rr_{size_i} = reverse size rank for individual i

s_{err} = error scaling factor

s_{size} = size scaling factor

P = population size

and

$$s_{err} + s_{size} = 1 \tag{3.2}$$

$$rr_{err_i} = P + 1 - r_{err_i} \tag{3.3}$$

$$rr_{size_i} = P + 1 - r_{size_i} \tag{3.4}$$

As the equation 3.1 shows, f_i depends on both the error and size of each individual. Depending on the desired solution of the genetic algorithm, different scaling factors are chosen. For example, if minimizing E_{MSE} is more important than minimizing the size of the neural network, s_{err} is chosen to be greater than s_{size}. The effects of s_{err} and s_{size} are discussed in section 3.3.1.

Finally, f_i values are used to assign the individuals with a fitness rank (r_{fit_i}). A fitness rank of '1' representing the most "fit" individual. Table 3.4 shows an example calculation of the fitness and fitness ranks of all individuals in a population of three. As shown in the table, individual A achieves the highest fitness rank, while individual B achieves the lowest fitness rank.

Table 3.4 – Example calculation of fitness rank using s_{err} = 0.6 and s_{size} = 0.4 for a population of 3.

ID	E_{MSE}	rr_{err_i}	Number of connections	rr_{size_i}	f_i	r_{fit_i}
A	0.03	**2**	5	**3**	(2)(0.6)+(3)(0.4)=2.4	**1**
B	0.01	**3**	10	**1**	(3)(0.6)+(1)(0.4)=2.2	**2**
C	0.05	**1**	6	**2**	(1)(0.6)+(2)(0.4)=1.4	**3**

3.1.3. Selection

In order to promote the reproduction of "fit" individuals, the probability of being selected depends on the individual's fitness rank. As suggested in previous research [22, 23], the rank-based selection method is found to perform better than the previously used fitness-based selection. For this reason, rank-based selection is used in the genetic algorithm studied in this thesis. Rank-based selection probability is defined in equation 2.19.

3.1.4. Crossover

After two individuals are selected in each generation, their genomes are crossed over to produce a new individual. Two crossover methods are studied in this research: gene-level and bit-level crossover. In gene-level crossover (also known as multi-point crossover) the genomes of both parents are crossed over on a gene by gene basis with a 50% chance of each parent's gene being selected. This means that each of the new individual's genes has a 50% chance of coming from either parent. An example of a gene-level crossover is shown in table 3.5.

Table 3.5 – Example of gene-level crossover. Bold genes represent those selected with a 50% chance from each parent.

Gene number	1	2	3	4	5	6	7	8	9
Parent 1 genome	**01**	**110**	101	**000**	010	101	110	**111**	1
Parent 2 genome	00	110	**010**	001	**110**	**111**	**101**	001	**0**
Child genome	01	110	010	000	110	111	101	111	0

In bit-level crossover (also known as uniform crossover), the genome of the new individuals is composed bit-by-bit, with a 50% chance of each bit coming from each of the parents. An example of bit-level crossover is shown in table 3.6 using the same parents as in table 3.5. The gene-level and bit-level crossover methods are compared and analyzed in this research.

Table 3.6 – Example of bit-level crossover. Bold bits represent those selected with a 50% chance from each parent.

Gene number	1	2	3	4	5	6	7	8	9
Parent 1 genome	**0**1	1**1**0	**10**1	00**0**	**0**1**0**	**1**01	**110**	1**11**	1
Parent 2 genome	0**0**	**11**0	01**0**	**00**1	1**1**0	1**11**	101	**0**01	**0**
Child genome	00	110	100	000	010	111	110	011	0

3.1.5. Mutation

Individuals generated by the genetic algorithm undergo mutation like organisms do in nature. During every generation, after two parents cross over, the genome of the new individual is mutated on a bit-by-bit basis. Mutation allows individuals to be generated that contain new genotypes that may potentially be better than any found in the current population. In the genetic algorithm studied in this thesis, each bit of the new individual's genome has the same probability of being flipped. This probability is set as a parameter of the genetic algorithm, as P_{bit_mutate}, and its effects are studied in the thesis.

3.2. Simulation Methods

In this section, the methods of simulating the genetic algorithm are described. The development environment used to run simulations is described. Later, the training and evaluation of individuals in the genetic algorithm population is described. The description of the five different test functions that are used to train the neural networks is also provided. Finally, the flow chart of the simulation program is outlined.

3.2.1. Development Environment

Neural networks are structured systems that are made up of neurons and synapses. Simulating a neural network naturally fit the object oriented programming model. The simulation software in this research is developed in Microsoft Visual C++ .Net using the C++ programming language. Three C++ classes are created to represent a neural network: Synapse, Neuron, and FullyConnectedNN.

The Synapse class represents one synaptic weight, and stores the following: the neurons that the synapse connected, the weight of the synapse, the learning rate, and the correction produced by back-propagation learning. The Neuron class represents one neuron, and stores the following: a list of all input Synapse objects connected to the neuron, the type of activation function used, and the activation function parameters. The FullyConnectedNN class represents a fully connected neural network, and stores the following: a collection of all of the Neuron objects including in which layer they resided, the number of inputs, and the number of outputs in the neural network.

3.2.2. Training and Evaluation

In order to train and evaluate the genetic algorithm individuals, training and evaluation sets are required for each of the five test functions studied. Simulations are driven using binary training files. A training file contains a header that includes the parameters of the test function: the number of inputs, outputs, training samples, and evaluation samples. Following the header, the file contains all of the training samples and then all of the evaluation samples. Each training or evaluation sample consists of an array of the sample inputs and an array of the desired outputs, all stored as 32-bit floating point numbers.

During the generation of each genetic algorithm individual, a neural network is created using the phenotype generated from the individual's genome. The neural network is then trained using the training samples found in one of the five test function training files. Standard on-line back-propagation learning is performed, with weights being modified after every sample. Training takes place for 2000 epochs, with each sample

being selected at random from the training set to allow for better generalization. After training is completed, the individual's E_{MSE} is determined by using the evaluation set found at the end of the training file. The difference between the actual and desired output found in the evaluation samples is used to determine E_{MSE}. When possible, the evaluation sets are different from the training sets to test the neural network's ability to generalize.

3.2.3. Test Functions

Five test functions are used to study the genetic algorithm developed in this research: the XOR function, the Sine function, and the DC-DC converter controller, BUPA Liver Disorders classification, and Iris plant classification. A description of each of these functions is provided in this section.

XOR function:

The XOR function has been regularly used to test neural network performance [5]. The XOR function is a digital operator consisting of two inputs and one output, with the truth table displayed in table 3.7. All four possible training samples are included in the training set and the evaluation set.

Table 3.7 – XOR truth table.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Sine function:

The Sine function has also been regularly used in neural network research since it has a continuous input and output relationship yet it is still relatively simple. One period of a 1-unit amplitude, 1-Hz Sine wave is studied in this thesis. The Sine function has one input and one output. For the training set, 100 samples of the Sine wave are used. The evaluation set is made up of 30 samples of the same Sine wave, which consists of different input/output pairs and therefore tested the generalization capabilities of the neural network in question. Both the training and evaluation sets are shown in figure 3.2.

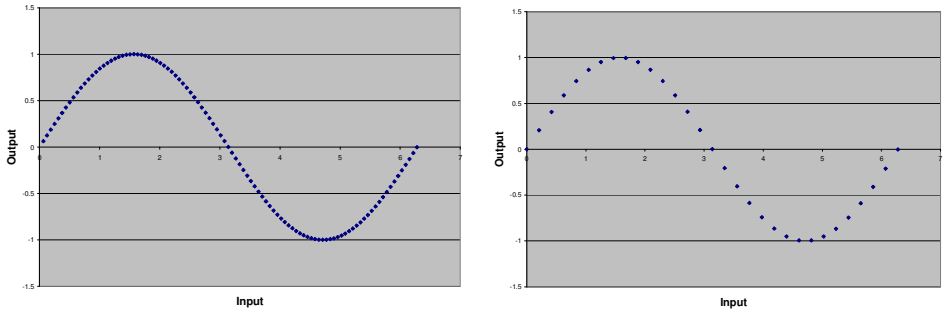

Figure 3.2 – Training (left) and evaluation (right) sets for the 1-unit amplitude, 1-Hz Sine wave.

DC-DC converter controller:

In order to test the genetic algorithm's ability to handle more complex tasks, the phase-shifted full-bridge DC-DC converter controller studied in previous research [21] is used. The controller's task is to maintain a constant output voltage as different loads are placed on the converter's terminals. This is accomplished by the controller varying the pulse-width modulation duty cycle appropriately. The controller has three inputs: load current in amps, input voltage in volts, and the output voltage differential in volts. The output of the controller is the pulse-width modulation duty cycle, which translates to the output voltage in this type of converter. The data collected in previous research [21] is

used to train and evaluate the individuals in the genetic algorithm. The training set consists of 1001 three-input and one-output samples. Similarly, the evaluation set consists of 1001 more of these samples which are different from the training set.

BUPA Liver Disorder Classification:

This data was provided by BUPA Medical Research Ltd. The data consists of 345 samples, each of which contain six inputs and two outputs. The first five inputs are blood test scores which are thought to be sensitive to liver disorders caused by excessive alcohol consumption. The sixth input is the number of half-pint alcoholic beverages the patient drank every day. The samples are separated into two sets of patients, which are represented by a '1' value in the two outputs. Outputs of '1' and '0' correspond to the first group, while outputs of '0' and '1' represent the second group. The training set and evaluation set both consist of all 345 samples.

Iris Plant Classification:

This data contains 150 total samples, each representing an Iris plant, made up of 50 samples of each of the three Iris plant types. Each sample consists of four inputs and three outputs. The inputs are the sepal length, sepal width, petal length, and petal width (all in cm). The three outputs represent three types of Iris plants: Setosa, Versicolour, and Virginica. Only one output can be '1' at a time, while the remainder are '0's. A '1' in the first output represents Setosa, while a '1' in the second output represents Versicoulour, and a '1' in the third output represents Virginica.

3.2.4. Program Flow

The simulation software allows neural networks of different sizes to be created and trained given a training file. The software also allows for the creation of training files that are later used to train the neural networks. Once a neural network is trained, its synaptic weights can be saved to a weights file and later loaded to skip the training process. The flowchart for the simulation of the genetic algorithm is shown in figure 3.3.

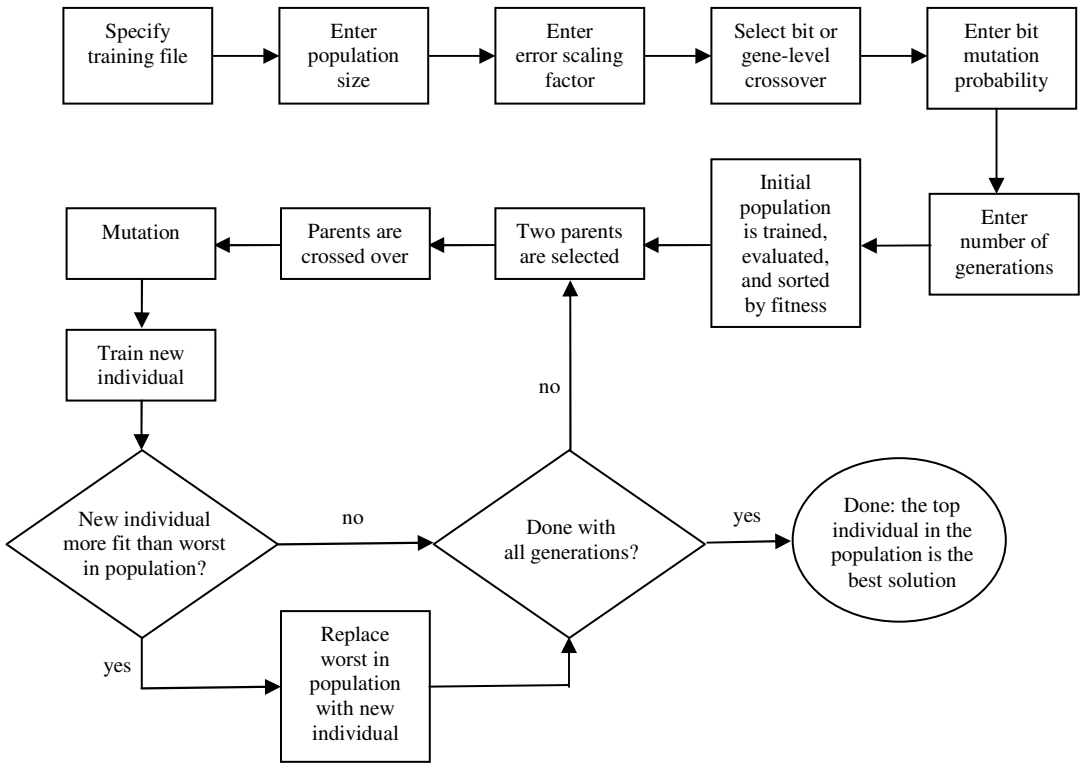

Figure 3.3 – Flowchart of genetic algorithm simulation software.

3.3. Results and Discussions

The genetic algorithm studied in this research has four parameters:

1. Error scaling factor (s_{err}) – ranges between 0 and 1 and used to determine the fitness of an individual.
2. Crossover method – either gene-level (50% chance of each gene coming from each parent) or bit-level crossover (50% chance of each bit coming from each parent) is used to create the new individual's genome.
3. Population size (P) – the size of the population which is maintained throughout the generations.
4. Bit mutation probability (P_{bit_mutate}) – the probability of one bit flipping during the mutation stage of the genetic algorithm.

The effect of each parameter on the E_{MSE} (calculated using equation 2.7) and number of connections is analyzed. This is accomplished by varying a parameter through a range of values while keeping all other parameters constant. For each of the values of parameter *i* (within the decided range), the genetic algorithm produces a solution in the form of the most fit individual at the end of the simulation. For example, if parameter *i* takes on ten different values, there would be ten corresponding solutions, each of which has an E_{MSE} and number of connections. These ten pairs of E_{MSE} values and number of connections are plotted against parameter *i* to show their relationship.

The simulations are executed for each of the five test functions. All of the simulations run for 50 generations and each individual is trained using the back-propagation algorithm for 2000 epochs to determine E_{MSE}. Standard on-line back-propagation learning is performed, with weights being modified after every sample. The samples are also chosen in a random order to improve generalization. The population was chosen to be in the range of 20-100, which was found to be ideal in [1] and similar to the population used in other research [5, 8, 10, 12]. P_{bit_mutate} was chosen to be 0.02, similar

to [4] for all simulations where it remained fixed. s_{err} was chosen to be 0.7 when it remained fixed, to give preferences to error over network size since it is generally more important. The values of the parameter at which E_{MSE} and the number of connections were at their minimum and maximum are recorded.

3.3.1 Effects of Error Scaling Factor

To test the effects of the error scaling factor, (s_{err}), the simulation software executes using a population of 50, bit-level crossover, and P_{bit_mutate} of 0.02. s_{err} is varied between 0 to 1 in increments of 0.1 for all five test functions.

XOR approximation:

Figure 3.4 and table 3.8 show the effects of s_{err} on E_{MSE} and number of connections for the XOR function. As expected, increasing s_{err} lowered E_{MSE} and increases the number of connections. The E_{MSE} goes through three phases. The high phase has an E_{MSE} three orders of magnitude larger than the low phase. The transition phase lasts from s_{err} of 0.4 to 0.5. The number of connections rises exponentially with s_{err}.

Table 3.8 – Min and max values of E_{MSE} and num Connections based on s_{err} for XOR approximation using bit-level crossover.

	s_{err}	E_{MSE}	Num Connections
Min E_{MSE}	1	0.000007	65
Max E_{MSE}	0.1	0.044798	5
Min num connections	0	0.042059	5
Max num connections	1	0.000007	65

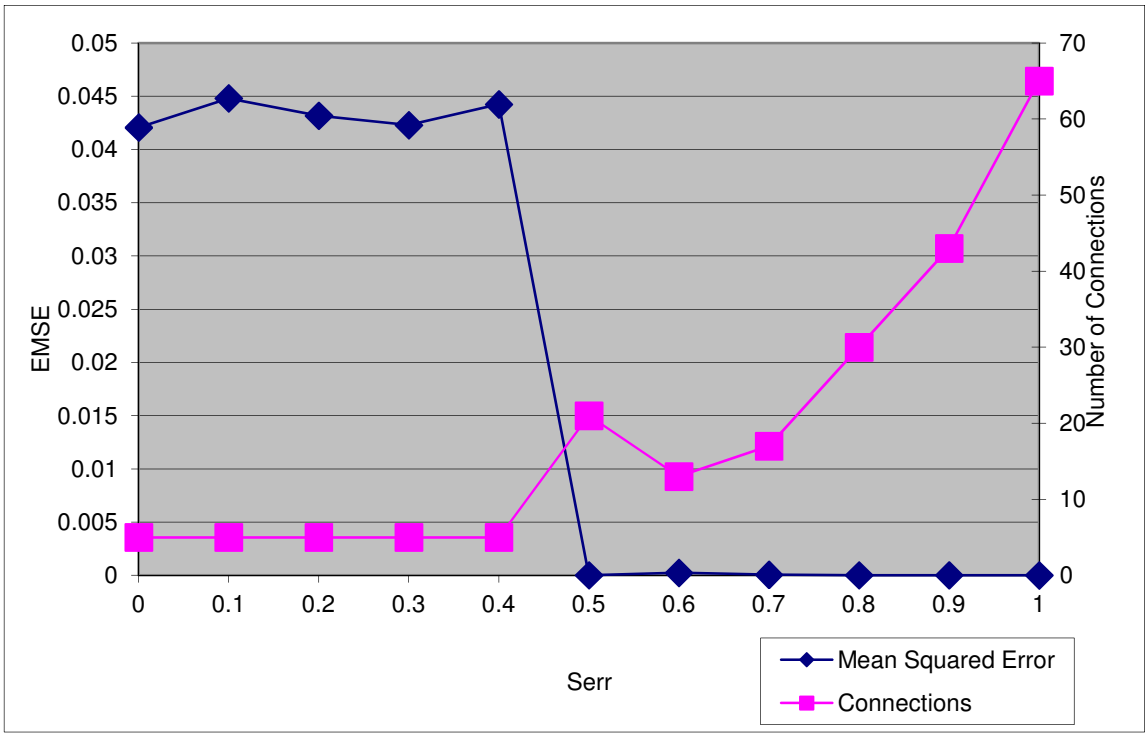

Figure 3.4 – Effects of s_{err} on E_{MSE} and number of connections for XOR approximation using bit-level crossover.

Sine approximation:

Table 3.9 and figure 3.5 show the effects of s_{err} on the E_{MSE} and number of connections for the Sine function. The data suggests, as expected, that increasing s_{err} lowers the E_{MSE} and increases the number of connections. The E_{MSE} goes through three phases: high, transition, and low. The high phase occurs between s_{err} values of 0 to 0.4, with an E_{MSE} around 0.01. The transition phase occurs from between s_{err} values of 0.4 to 0.6 in which the E_{MSE} decreases by two orders of magnitude. The low phase occurs when s_{err} is above 0.6, with an E_{MSE} around 0.0001.

Table 3.9 – Min and max values of E_{MSE} and num Connections based on s_{err} for Sine approximation using bit-level crossover.

	s_{err}	E_{MSE}	Num Connections
Min E_{MSE}	1	0.000058	22
Max E_{MSE}	0.1	0.009636	4
Min num connections	0	0.009573	4
Max num connections	0.8	0.000067	25

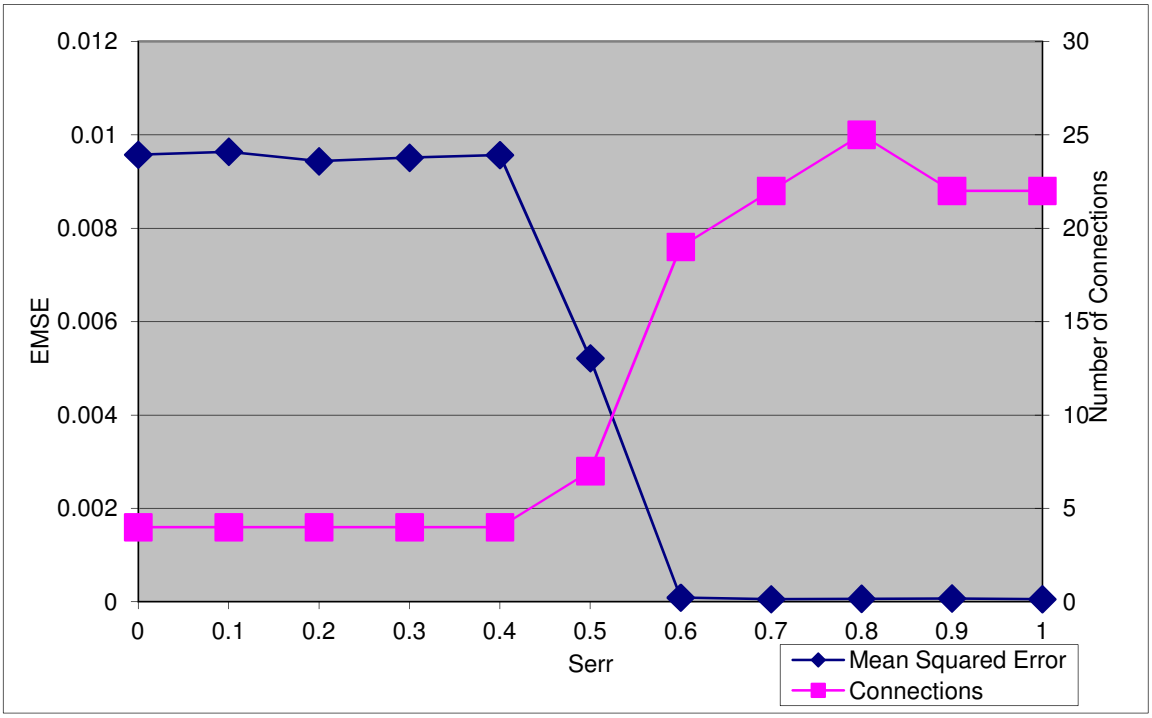

Figure 3.5 – Effects of s_{err} on E_{MSE} and number of connections for Sine approximation using bit-level crossover.

The number of connections has the inverse behavior (low, transition, then high phase). The high phase of the number of connections has around six times more connections than the low phase.

DC-DC converter controller:

Figure 3.6 and table 3.10 show the effects of s_{err} on the E_{MSE} and number of connections for the DC-DC converter controller application. The data suggests, as expected, that increasing s_{err} lowers the E_{MSE} and increases the number of connections. E_{MSE} declined sharply as s_{err} increased from 0 to 0.2 and remained relatively stable afterwards. The number of connections follows a generally increasing pattern, especially at the maximum value of s_{err}.

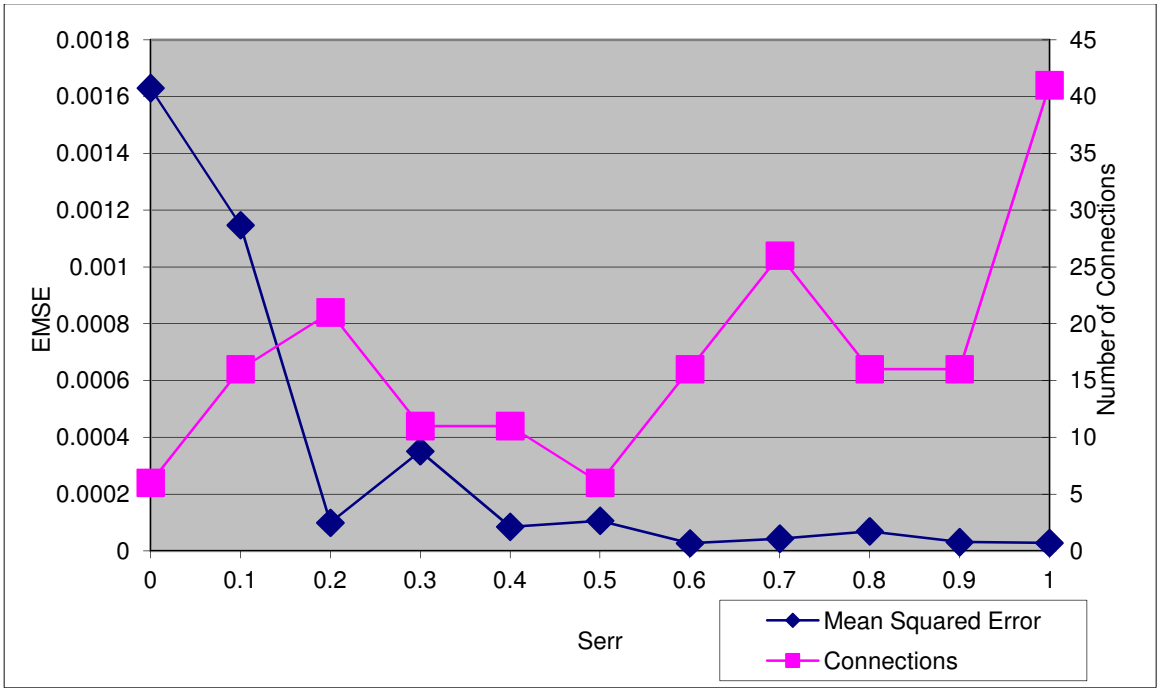

Figure 3.6 – Effects of s_{err} on E_{MSE} and number of connections for the DC-DC converter controller using bit-level crossover.

Table 3.10 – Min and max values of E_{MSE} and num Connections based on s_{err} for the DC-DC converter controller using bit-level crossover.

	s_{err}	E_{MSE}	Num Connections
Min E_{MSE}	0.6	0.000027	16
Max E_{MSE}	0	0.001630	6
Min num connections	0	0.001630	6
Max num connections	1	0.000028	41

BUPA Liver Disorder Classification:

Figure 3.7 and table 3.11 show the effects of s_{err} on the E_{MSE} and number of connections for the BUPA Liver Disorder Classification application. The data suggests, as expected, that increasing s_{err} lowers the E_{MSE} and increases the number of connections. E_{MSE} rose and declined sharply as s_{err} increased from 0 to 0.2 and declined steadily afterwards. The number of connections steadily increased as s_{err} increased.

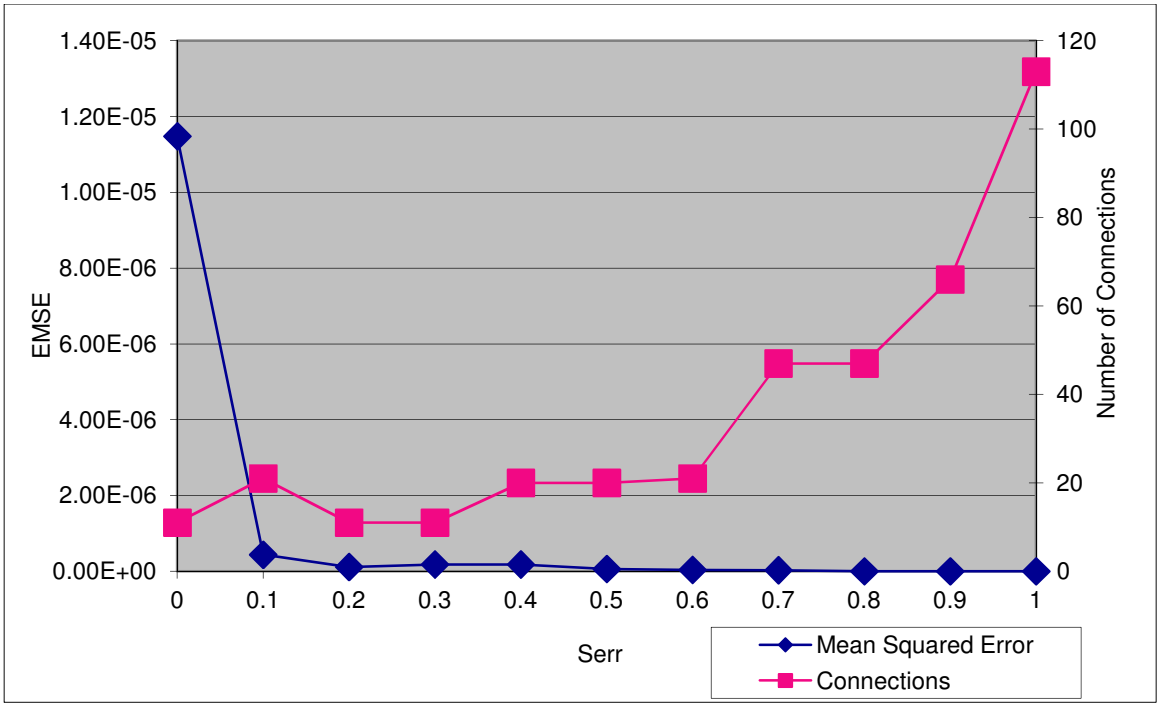

Figure 3.7 – Effects of s_{err} on E_{MSE} and number of connections for the BUPA Liver Disorder Classification using bit-level crossover.

Table 3.11 – Min and max values of E_{MSE} and num Connections based on s_{err} for the BUPA Liver Disorder Classification using bit-level crossover.

	s_{err}	E_{MSE}	Num Connections
Min E_{MSE}	0.8	6.61E-17	47
Max E_{MSE}	0.1	4.38E-07	21
Min num connections	0	9.04E-08	11
Max num connections	1	1.34E-11	113

Iris Plant Classification:

Figure 3.8 and table 3.12 show the effects of s_{err} on the E_{MSE} and number of connections for the Iris Plant Classification application. The data suggests that increasing s_{err} lowers the E_{MSE} and increases the number of connections. E_{MSE} declined sharply as s_{err} increased past 0.1. The number of connections trended upwards as s_{err} increased.

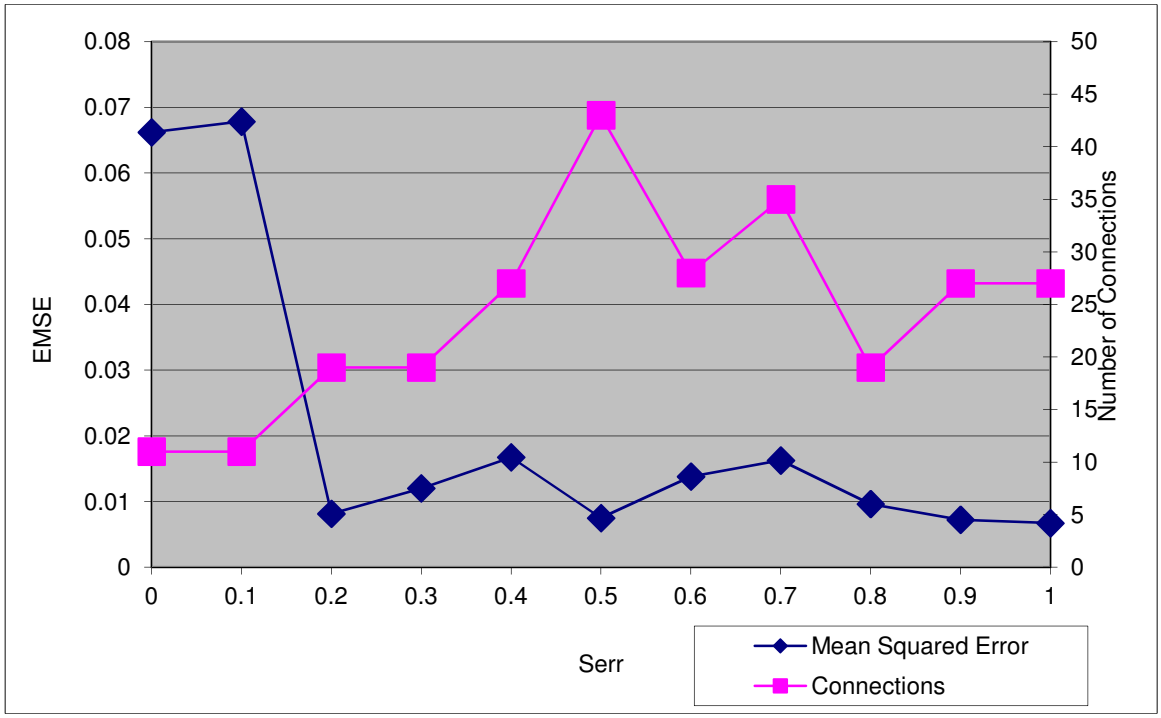

Figure 3.8 – Effects of s_{err} on E_{MSE} and number of connections for the Iris Plant Classification using bit-level crossover.

Table 3.12 – Min and max values of E_{MSE} and num Connections based on s_{err} for the Iris Plant Classification using bit-level crossover.

	s_{err}	E_{MSE}	Num Connections
Min E_{MSE}	1	0.006714	27
Max E_{MSE}	0.1	0.067845	11
Min num connections	0	0.066205	11
Max num connections	0.5	0.007494	43

3.3.2 Gene-level vs. Bit-level Crossover

The simulation in section 3.3.1 is repeated using gene-level crossover instead of bit-level crossover. The population remains at 50, P_{bit_mutate} remains at 0.02, and s_{err} varies between 0 to 1 in increments of 0.1 for all five test functions. To compare the results, the difference in E_{MSE} and number of connections between the two crossover types is displayed in the graphs. The difference in E_{MSE} , $E_{MSE_{diff}}$, is defined as

$$E_{MSE_{diff}} = E_{MSE_{bit}} - E_{MSE_{gene}} \tag{3.5}$$

where

$E_{MSE_{bit}}$ is the E_{MSE} of the bit-level crossover sample

$E_{MSE_{gene}}$ is the E_{MSE} of the gene-level crossover sample

The difference in the number of connections, C_{diff}, is defined as

$$C_{diff} = C_{bit} - C_{gene} \tag{3.6}$$

where

C_{bit} is the number of connections of the bit-level crossover sample

C_{gene} is the number of connections of the gene-level crossover sample

XOR approximation:

Table 3.13 and figure 3.9 show the effects of s_{err} on E_{MSE} and number of connections for the XOR application using gene-level crossover instead of bit-level crossover. E_{MSE} has high, transition, and low phases as with bit-level crossover, although the transition phase is half as long and starts at an s_{err} value of 0.2. The number of connections increased linearly except for the sharp increase at an s_{err} value of 1, as opposed to the three phase trend using bit-level crossover.

Table 3.13 – Min and max values of E_{MSE} and num Connections based on s_{err} for XOR approximation using gene-level crossover.

	s_{err}	E_{MSE}	Num Connections
Min E_{MSE}	1	0.000003	69
Max E_{MSE}	0	0.049888	5
Min num connections	0	0.049888	5
Max num connections	1	0.000003	69

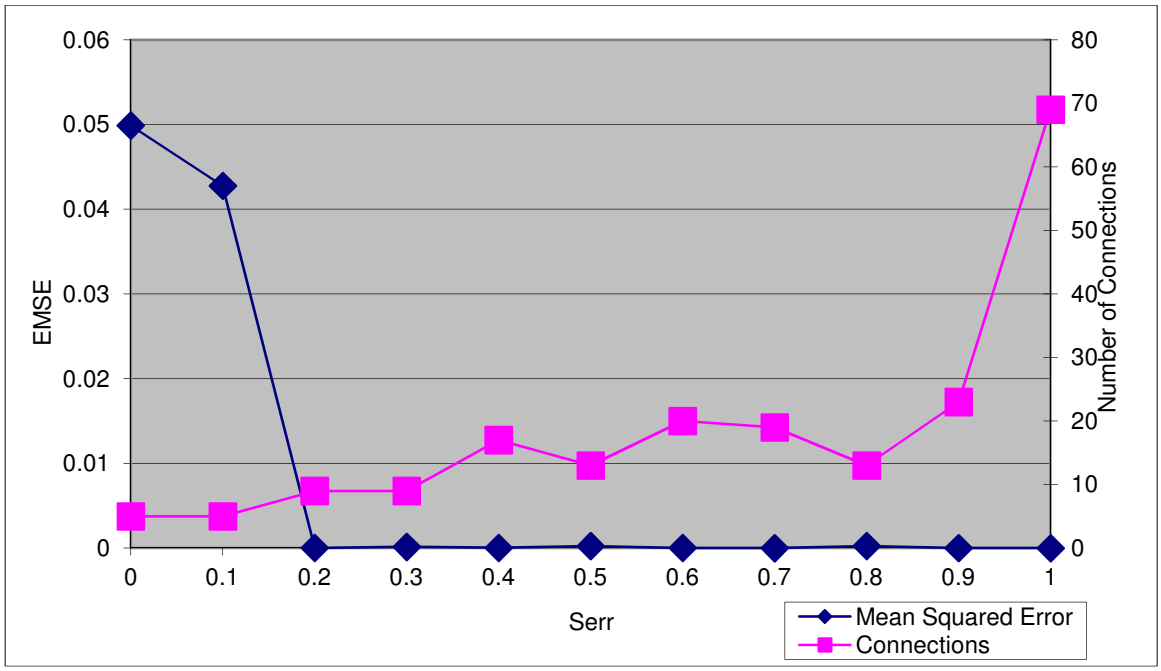

Figure 3.9 – Effects of s_{err} on E_{MSE} and number of connections for XOR approximation using gene-level crossover.

Figure 3.10 shows the difference in E_{MSE} and number of connections between gene-level crossover and bit-level crossover for the XOR approximation. Positive numbers on the graph represent an advantage in gene-level crossover, while negative numbers represent an advantage in bit-level crossover.

E_{MSE} using gene-level crossover for the XOR function approximation is superior, especially for an s_{err} value between 0.2 and 0.4. The number of connections is lower using gene-level crossover for high values of s_{err}.

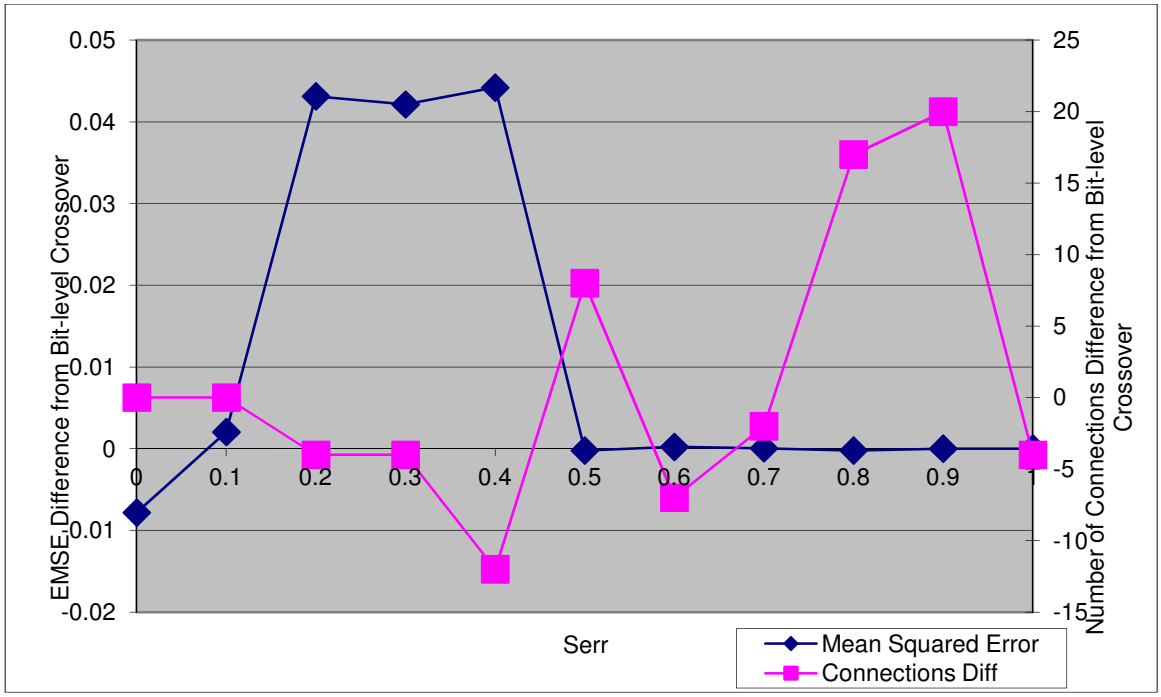

Figure 3.10 – Difference in E_{MSE} and number of connections using gene-level crossover instead of bit-level crossover for XOR approximation.

Sine approximation:

Figure 3.11 and table 3.14 show the effects of s_{err} on E_{MSE} and number of connections for the Sine application using gene-level crossover instead of bit-level crossover. The data suggests, as with bit-level crossover, that increasing s_{err} lowers E_{MSE} and increases the number of connections. E_{MSE} follows the same high, transition, and low phases as with bit-level crossover, although the transition phase is half as long and started at an s_{err} value of 0.3. The number of connections increases exponentially instead of following the three phase pattern.

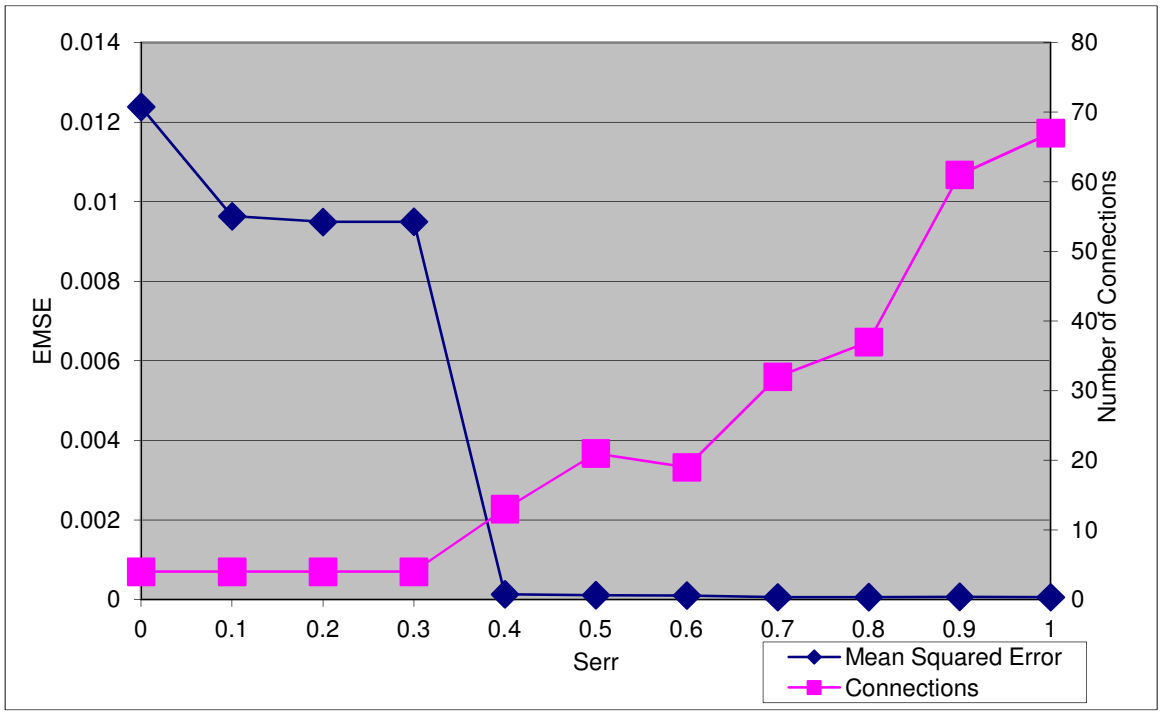

Figure 3.11 – Effects of s_{err} on E_{MSE} and number of connections for Sine approximation using gene-level crossover.

Table 3.14 – Min and max values of E_{MSE} and num Connections based on s_{err} for Sine approximation using gene-level crossover.

	s_{err}	E_{MSE}	Num Connections
Min E_{MSE}	1	0.000058	67
Max E_{MSE}	0	0.012383	4
Min num connections	0	0.012383	4
Max num connections	1	0.000058	67

Figure 3.12 shows the difference in E_{MSE} and number of connections between gene-level crossover and bit-level crossover for the Sine approximation. Positive numbers on the graph represent an advantage in gene-level crossover.

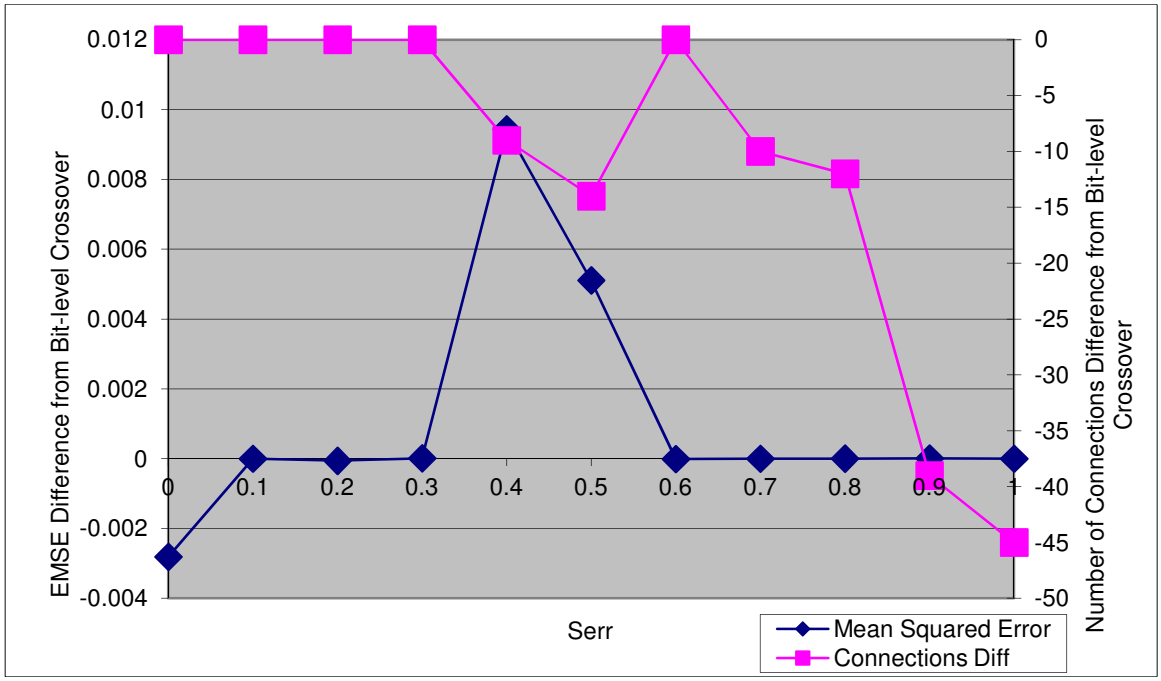

Figure 3.12 – Difference in E_{MSE} and number of connections using gene-level crossover instead of bit-level crossover for Sine approximation.

For the Sine function approximation, E_{MSE} is similar for both crossover methods, except gene-level crossover is superior in the mid-range (0.4 and 0.5) of s_{err}. The number of connections is significantly higher for gene-level crossover for an s_{err} value higher than 0.6. Neither crossover method has a clear advantage over the other.

DC-DC converter controller:

Table 3.15 and figure 3.13 show the effects of s_{err} on E_{MSE} and number of connections for the DC-DC converter controller application using gene-level crossover. The data suggests, as with bit-level crossover, that increasing s_{err} lowers E_{MSE} and increases the number of connections. The number of connections increases exponentially, while E_{MSE} is relatively stable after the sharp decrease that occurs at an s_{err} value of 0.1.

Table 3.15 – Min and max values of E_{MSE} and num Connections based on s_{err} for DC-DC converter controller using gene-level crossover.

	s_{err}	E_{MSE}	Num Connections
Min E_{MSE}	0.5	0.000029	31
Max E_{MSE}	0	0.000475	6
Min num connections	0	0.000475	6
Max num connections	1	0.000029	105

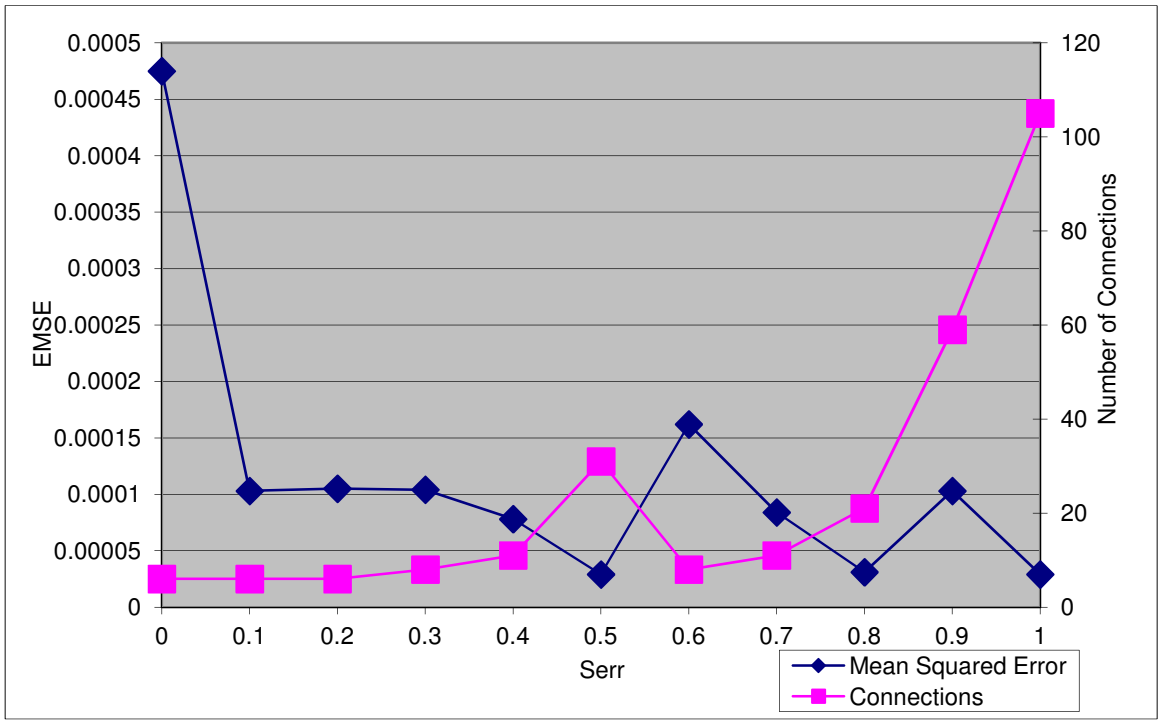

Figure 3.13 – Effects of s_{err} on E_{MSE} and number of connections for DC-DC converter controller using gene-level crossover.

Figure 3.14 shows the difference in E_{MSE} and number of connections between gene-level crossover and bit-level crossover for the DC-DC converter controller. Positive numbers on the graph represent a lower E_{MSE} and lower number of connections, signifying that the gene-level crossover is superior. Negative numbers represent the opposite, that the bit-level crossover is superior.

For the DC-DC converter controller application, E_{MSE} is lower using gene-level crossover for low values of s_{err} but is not significantly different for the remainder of the s_{err} range. The number of connections is significantly higher using gene-level crossover for an s_{err} value higher than 0.8.

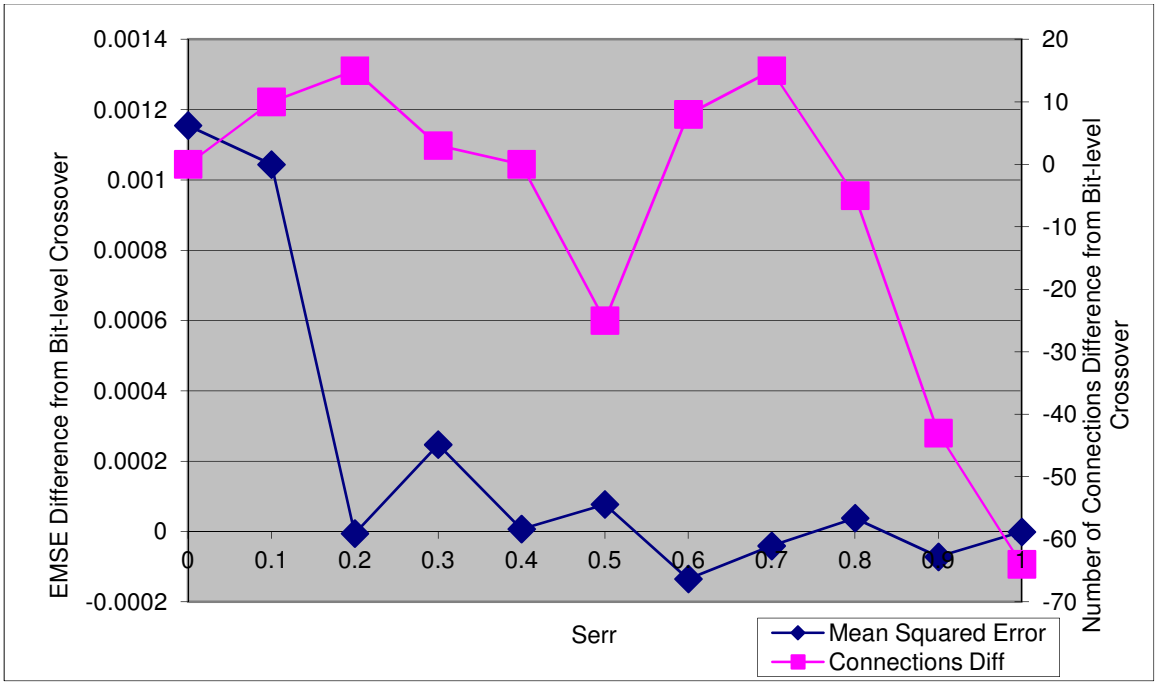

Figure 3.14 – Difference in E_{MSE} and number of connections using gene-level crossover instead of bit-level crossover for DC-DC converter controller.

BUPA Liver Disorder Classification:

Figure 3.15 and table 3.16 show the effects of s_{err} on E_{MSE} and number of connections for the BUPA Liver Disorder Classification application using gene-level crossover instead of bit-level crossover. The data suggests that increasing s_{err} lowers E_{MSE} and increases the number of connections. The change in E_{MSE} increases when s_{err} increases to 0.3, then plateaus until 0.6, then decreases sharply as s_{err} increases. The number of connections increases steadily with s_{err}.

Table 3.16 – Min and max values of E_{MSE} and num Connections based on s_{err} for BUPA Liver Disorder Classification using gene-level crossover.

	s_{err}	E_{MSE}	Num Connections
Min E_{MSE}	0.9	6.61E-17	47
Max E_{MSE}	0.3	1.98E-07	20
Min num connections	0	4.53E-08	11
Max num connections	1	9.70E-16	47

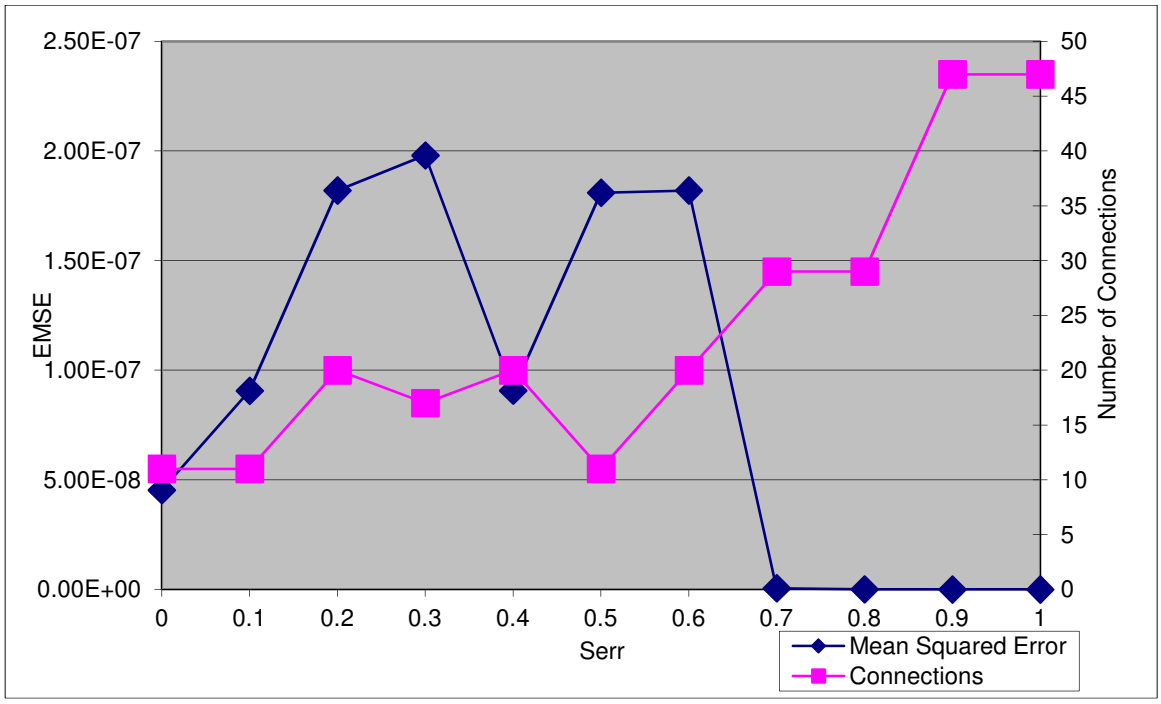

Figure 3.15 – Effects of s_{err} on E_{MSE} and number of connections for BUPA Liver Disorder Classification using gene-level crossover.

Figure 3.16 shows the difference in E_{MSE} and number of connections between gene-level crossover and bit-level crossover for the BUPA Liver Disorder Classification application. Positive numbers on the graph represent a lower E_{MSE} and lower number of connections, signifying that the gene-level crossover is superior. Negative numbers represent the opposite, that the bit-level crossover is superior.

For the BUPA Liver Disorder Classification application, E_{MSE} is lower using gene-level crossover for the s_{err} range of 0-0.1, and fluctuates for the remainder of the range. The number of connections is significantly lower using gene-level crossover for an s_{err} value higher than 0.6. From the s_{err} range of 0.6-0.9, gene-level crossover produces around 20 fewer connections and 66 fewer connections when s_{err} is at its maximum.

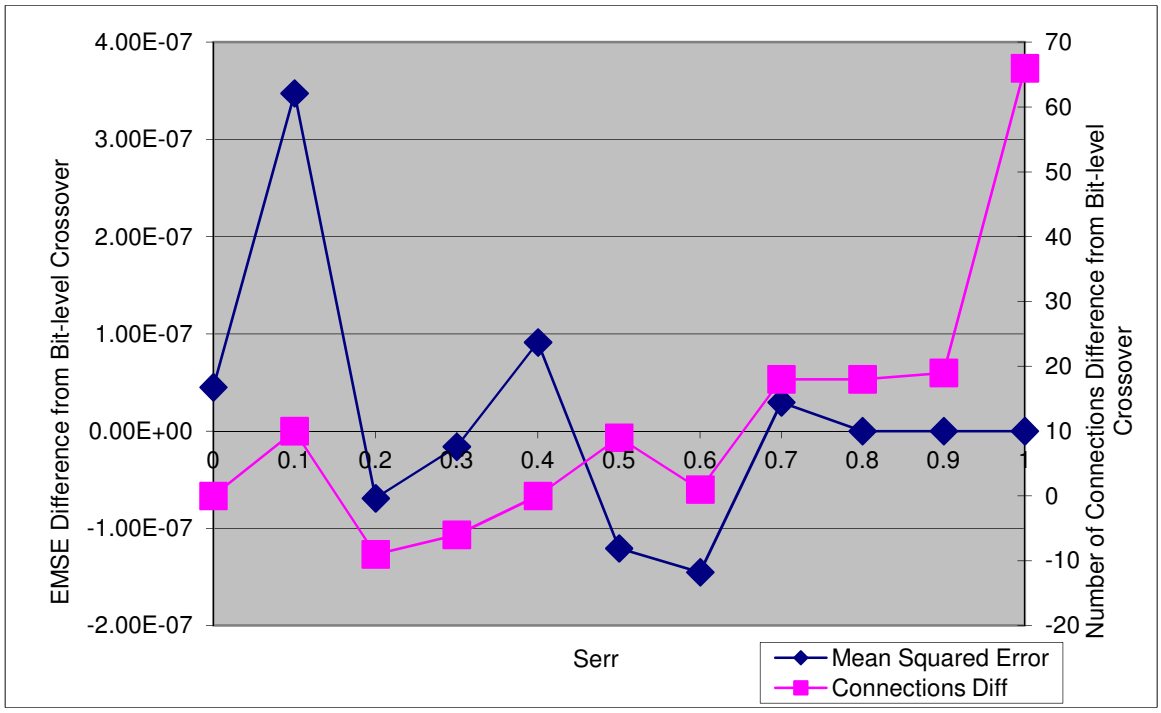

Figure 3.16 – Difference in E_{MSE} and number of connections using gene-level crossover instead of bit-level crossover for BUPA Liver Disorder Classification.

Iris Plant Classification:

Figure 3.17 and table 3.17 show the effects of s_{err} on E_{MSE} and number of connections for the Iris Plant Classification application using gene-level crossover instead of bit-level crossover. In a departure from the other applications, increasing s_{err} did not affect E_{MSE} or the number of connections significantly until s_{err} reached the range of 0.9-1. E_{MSE} decreased gradually until s_{err} reached 0.7, then showed a sharp increase followed by a sharp decrease. Regardless of the trend, E_{MSE} at maximum s_{err} when was less than half the E_{MSE} at minimum s_{err}. The number of connections did not show significant change as s_{err} increased, except when s_{err} increased from 0.9 to 1, where the number of connections increased from 19 to 91.

Table 3.17 – Min and max values of E_{MSE} and num Connections based on s_{err} for Iris Plant Classification using gene-level crossover.

	s_{err}	E_{MSE}	Num Connections
Min E_{MSE}	1	0.004046	91
Max E_{MSE}	0.9	0.017416	19
Min num connections	0	0.009755	19
Max num connections	1	0.004046	91

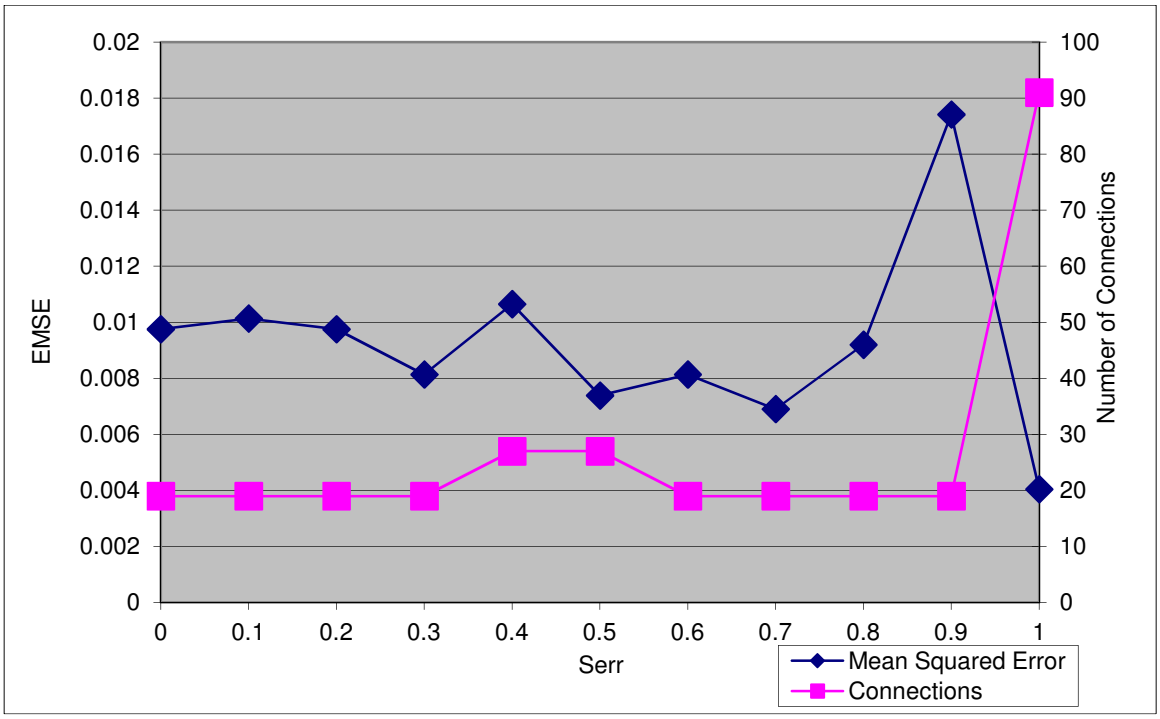

Figure 3.17 – Effects of s_{err} on E_{MSE} and number of connections for Iris Plan Classification using gene-level crossover.

Figure 3.18 shows the difference in E_{MSE} and number of connections between gene-level crossover and bit-level crossover for the Iris Plant Classification application. Positive numbers on the graph represent a lower E_{MSE} and lower number of connections, signifying that the gene-level crossover is superior.

For the Iris Plant Classification application, E_{MSE} is lower using gene-level crossover for the s_{err} range of 0-0.1, and does not seem to be correlated to the type of crossover otherwise (similar to that of the BUPA Liver Disorder Classification application). The number of connections fluctuated between +10 and -10 throughout the range of s_{err}, other than when s_{err} reached a value of 1, at which point gene-level crossover was inferior to bit-level crossover by 64 connections.

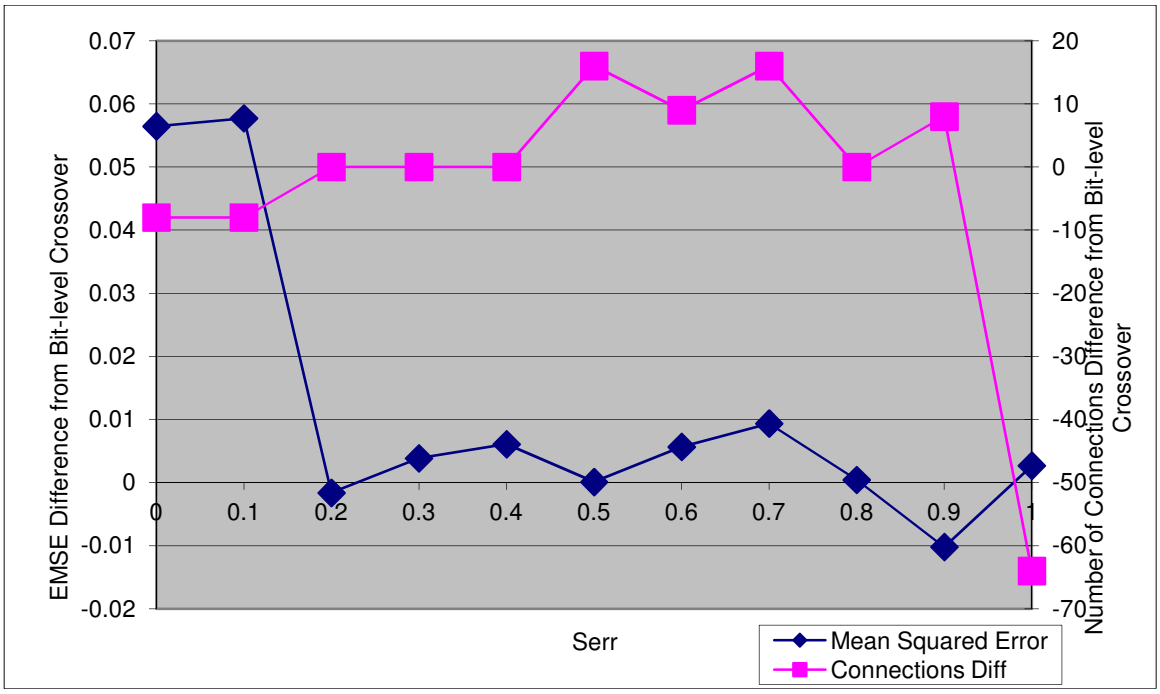

Figure 3.18 – Difference in E_{MSE} and number of connections using gene-level crossover instead of bit-level crossover for Iris Plant Classification.

3.3.3 Effects of Population Size

To test the effects of the population size (P), the simulation software executes using an s_{err} of 0.7, gene-level crossover, and P_{bit_mutate} of 0.02. Population size varies between 10 to 100 in increments of ten for all five test functions.

XOR approximation:

Figure 3.19 and table 3.18 show the effects of population size on E_{MSE} and number of connections for the XOR function approximation. Neither E_{MSE} nor the number of connections correlates with the population size. E_{MSE} fluctuates throughout the range of population sizes, while the number of connections stays relatively constant.

Table 3.18 – Min and max values of E_{MSE} and num Connections based on population size for XOR approximation.

	P	E_{MSE}	Num Connections
Min E_{MSE}	10	0.000017	15
Max E_{MSE}	50	0.000696	17
Min num connections	90	0.000151	9
Max num connections	30	0.000024	30

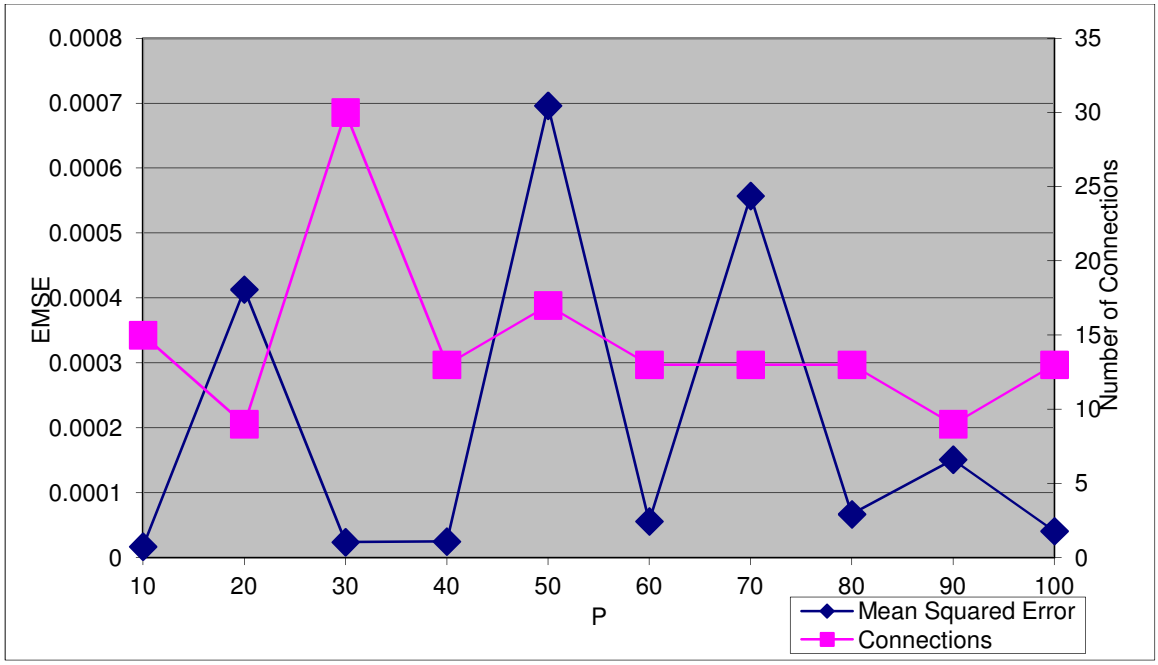

Figure 3.19 – Effects of population size on E_{MSE} and number of connections for XOR approximation.

Sine approximation:

Figure 3.20 and table 3.19 show the effects of population size on E_{MSE} and number of connections for the Sine function approximation. Similar to the XOR approximation, E_{MSE} doesn't correlate with population size, and varies by one order of magnitude as seen by the two spikes in figure 3.20. The number of connections shows a decreasing trend as the population size increased.

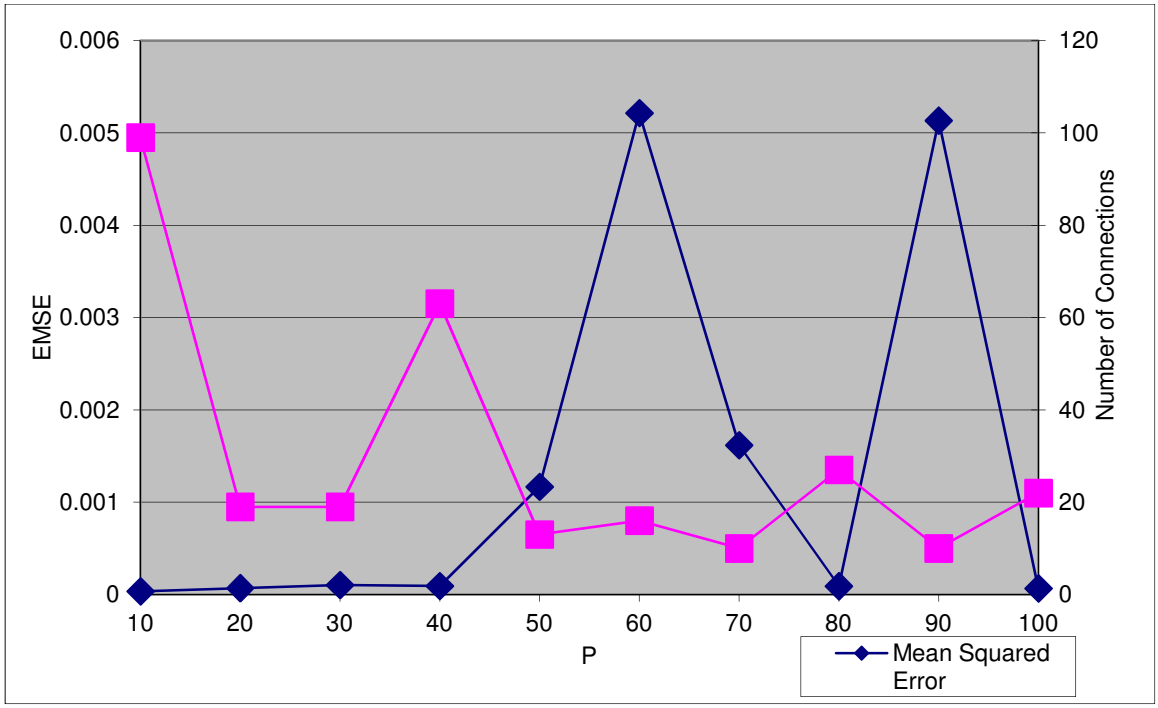

Figure 3.20 – Effects of population size on E_{MSE} and number of connections for Sine approximation.

Table 3.19 – Min and max values of E_{MSE} and num Connections based on population size for Sine approximation.

	P	E_{MSE}	Num Connections
Min E_{MSE}	10	0.000034	99
Max E_{MSE}	60	0.005215	16
Min num connections	70	0.001617	10
Max num connections	10	0.000034	99

DC-DC converter controller:

Figure 3.21 and table 3.20 show the effects of population size on E_{MSE} and number of connections for the DC-DC converter controller. E_{MSE} does not have a correlation with population size, as it fluctuates throughout the range of population sizes, varying within one order of magnitude. The number of connections shows a slight decreasing trend as the population size increases.

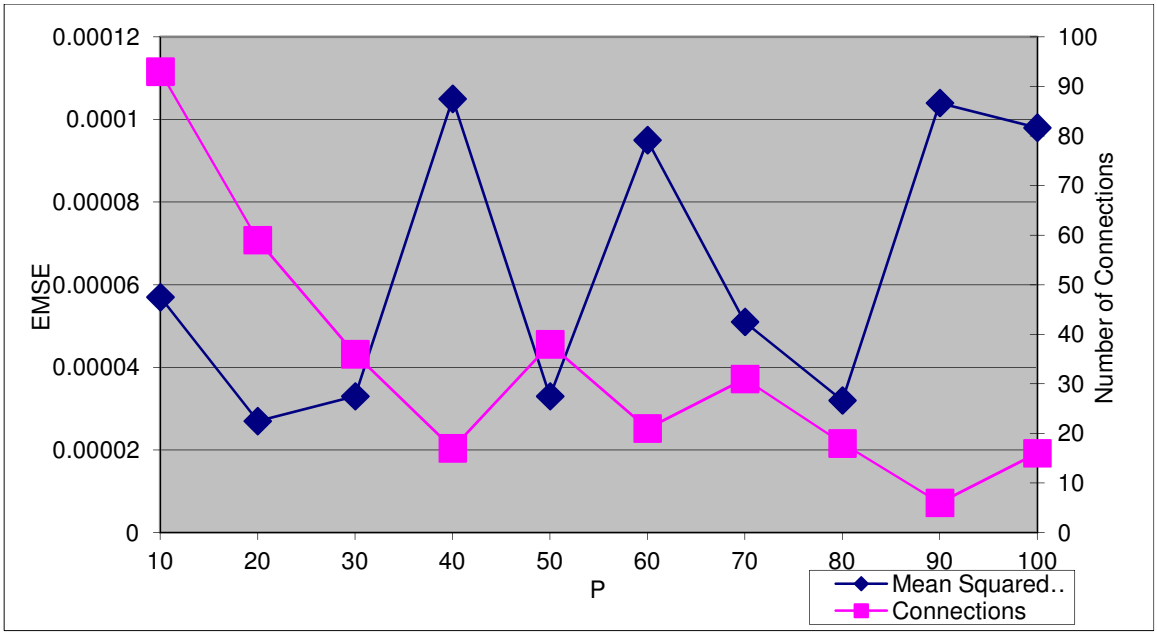

Figure 3.21 – Effects of population size on E_{MSE} and number of connections for DC-DC converter controller.

Table 3.20 – Min and max values of E_{MSE} and num Connections based on population size for DC-DC converter controller.

	P	E_{MSE}	Num Connections
Min E_{MSE}	20	0.000027	59
Max E_{MSE}	40	0.000105	17
Min num connections	90	0.000104	6
Max num connections	10	0.000057	93

BUPA Liver Disorder Classification:

Figure 3.22 and table 3.21 show the effects of population size on E_{MSE} and number of connections for the BUPA Liver Disorder Classification application. E_{MSE} does not have a correlation with population size, as it fluctuates throughout the range of population sizes, varying by five orders of magnitude. The number also shows no correlation with population size, fluctuating between 11 and 47 connections.

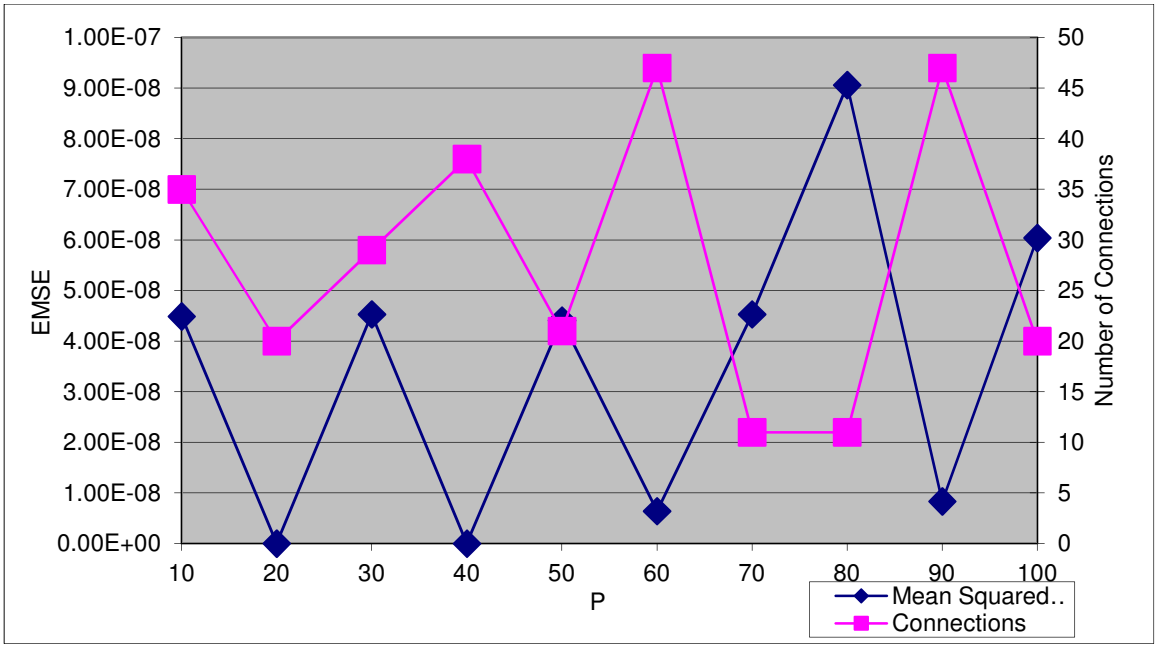

Figure 3.22 – Effects of population size on E_{MSE} and number of connections for BUPA Liver Disorder Classification.

Table 3.21 – Min and max values of E_{MSE} and num Connections based on population size for BUPA Liver Disorder Classification.

	P	E_{MSE}	Num Connections
Min E_{MSE}	40	1.37E-13	38
Max E_{MSE}	80	9.06E-08	11
Min num connections	80	9.06E-08	11
Max num connections	60	6.40E-09	47

Iris Plant Classification:

Figure 3.23 and table 3.22 show the effects of population size on E_{MSE} and number of connections for the Iris Plant Classification application. E_{MSE} increases as population size increases. The number of connections shows the opposite correlation and decreases as the population size increases.

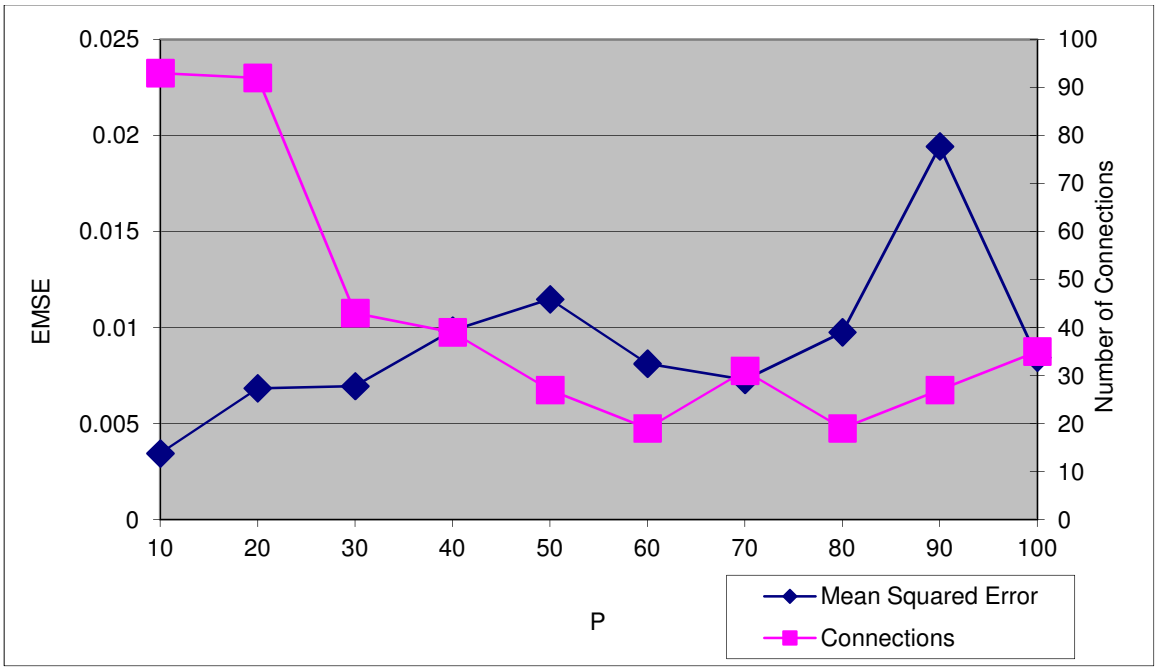

Figure 3.23 – Effects of population size on E_{MSE} and number of connections for Iris Plant Classification.

Table 3.22 – Min and max values of E_{MSE} and num Connections based on population size for Iris Plant Classification.

	P	E_{MSE}	Num Connections
Min E_{MSE}	10	0.003445	93
Max E_{MSE}	90	0.019429	27
Min num connections	60	0.008116	19
Max num connections	10	0.003445	93

3.3.4 Effects of Bit Mutation Probability

To test the effects of P_{bit_mutate}, the simulation software runs using an s_{err} of 0.7, gene-level crossover, and a population of 50. P_{bit_mutate} varies between 0.01 to 0.1 in increments of 0.01 for all five test functions.

XOR approximation:

Table 3.23 and figure 3.24 show the effects of P_{bit_mutate} on E_{MSE} and number of connections for the XOR function approximation. E_{MSE} fluctuates throughout the range of P_{bit_mutate}. The number of connections peaks to a value of 54 in the middle of the P_{bit_mutate} range while being at least twice as low for the high and low values of P_{bit_mutate}.

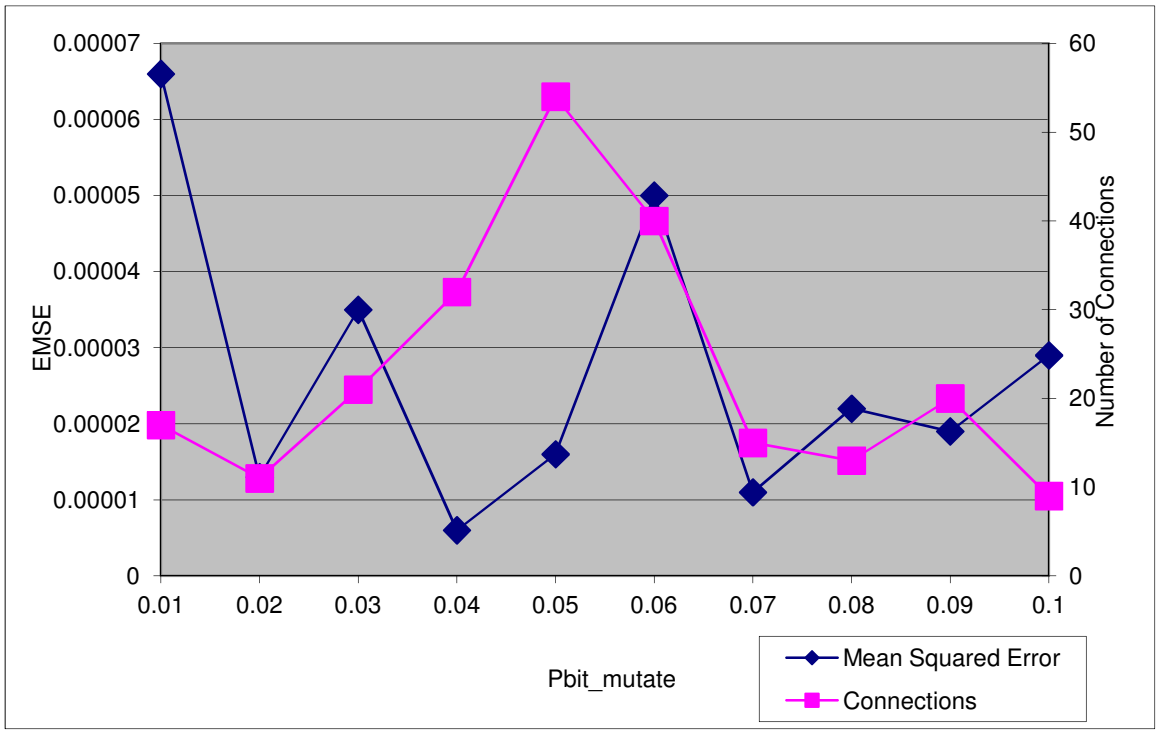

Figure 3.24 – Effects of P_{bit_mutate} on E_{MSE} and number of connections for XOR approximation.

Table 3.23 – Min and max values of E_{MSE} and num Connections based on P_{bit_mutate} for XOR approximation.

	P_{bit_mutate}	E_{MSE}	Num Connections
Min E_{MSE}	0.04	0.000006	32
Max E_{MSE}	0.01	0.000066	17
Min num connections	0.10	0.000029	9
Max num connections	0.05	0.000016	54

Sine approximation:

Figure 3.25 and table 3.24 show the effects of P_{bit_mutate} on E_{MSE} and number of connections for the Sine function approximation. E_{MSE} has a slight positive correlation with P_{bit_mutate}. The number of connections has a slight negative correlation with P_{bit_mutate}.

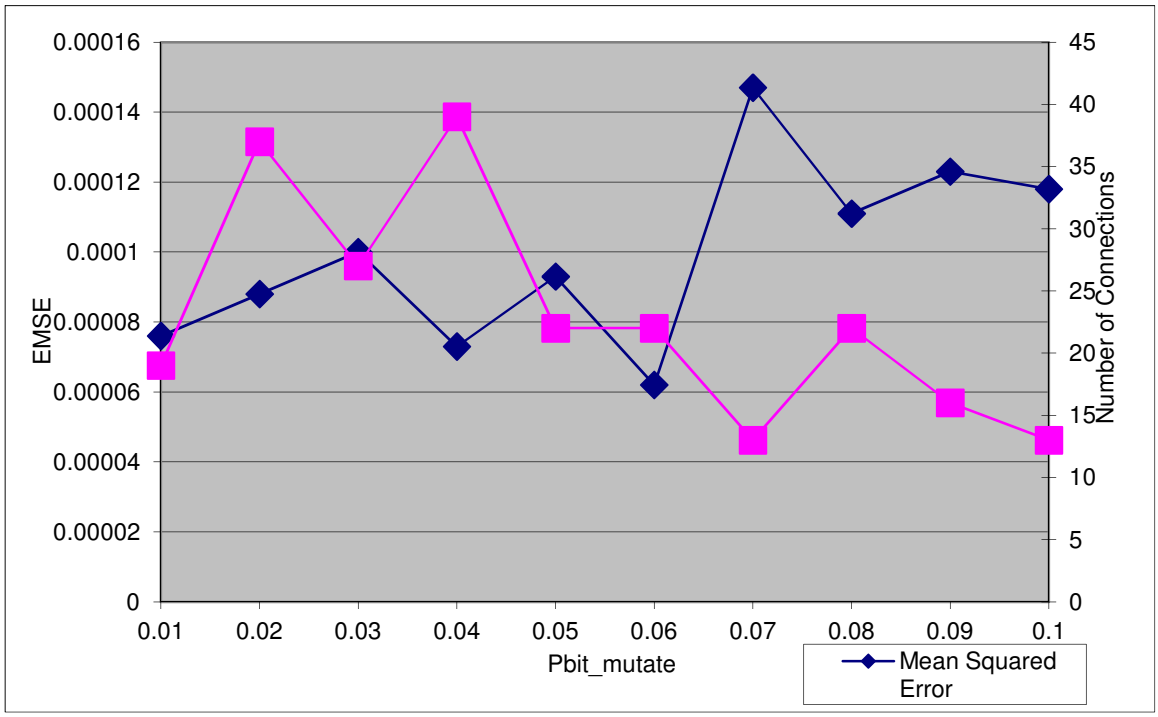

Figure 3.25 – Effects of P_{bit_mutate} on E_{MSE} and number of connections for Sine approximation.

Table 3.24 – Min and max values of E_{MSE} and num Connections based on P_{bit_mutate} for Sine approximation.

	P_{bit_mutate}	E_{MSE}	Num Connections
Min E_{MSE}	0.06	0.000062	22
Max E_{MSE}	0.07	0.000147	13
Min num connections	0.10	0.000118	13
Max num connections	0.04	0.000073	39

DC-DC converter controller:

Figure 3.26 and table 3.25 show the effects of P_{bit_mutate} on E_{MSE} and number of connections for the DC-DC converter controller. E_{MSE} rises through the range of 0.01 to 0.03, then declines and flattened out, not showing a strong correlation with P_{bit_mutate}.

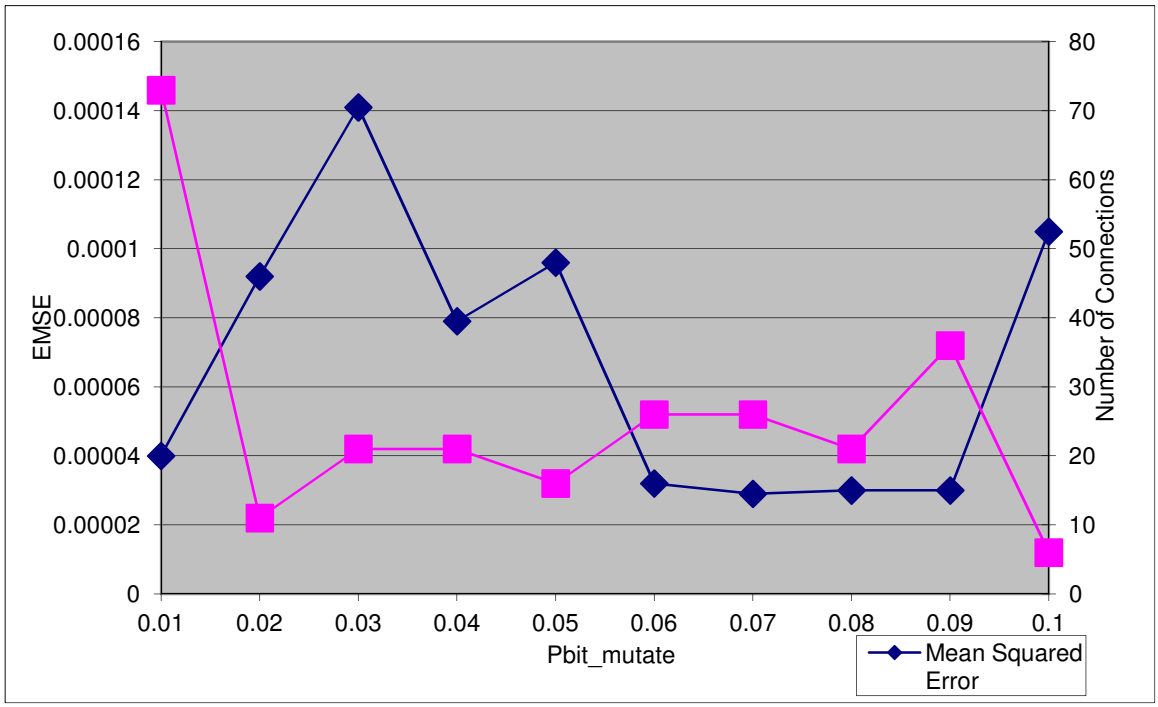

Figure 3.26 – Effects of P_{bit_mutate} on E_{MSE} and number of connections for DC-DC converter controller.

Table 3.25 – Min and max values of E_{MSE} and num Connections based on P_{bit_mutate} for DC-DC converter controller.

	P_{bit_mutate}	E_{MSE}	Num Connections
Min E_{MSE}	0.07	0.000029	26
Max E_{MSE}	0.03	0.000141	21
Min num connections	0.10	0.000105	6
Max num connections	0.01	0.000040	73

Other than the spikes at 0.01 and 0.10, the number of connections rises from 10 to 40 as P_{bit_mutate} increases, showing a slight positive correlation.

BUPA Liver Disorder Classification:

Figure 3.27 and table 3.26 show the effects of P_{bit_mutate} on E_{MSE} and number of connections for the BUPA Liver Disorder Classification application. Both E_{MSE} does not seem to have a correlation with P_{bit_mutate}. The number of connections has a slight negative correlation with P_{bit_mutate}.

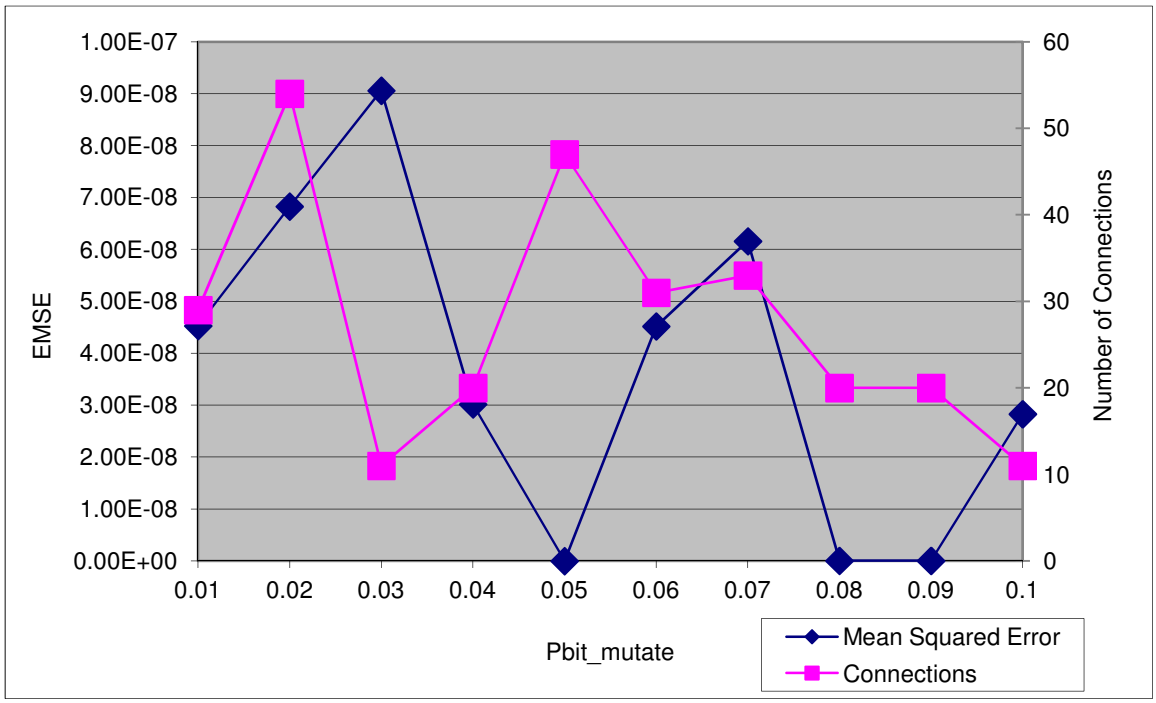

Figure 3.27 – Effects of P_{bit_mutate} on E_{MSE} and number of connections for BUPA Liver Disorder Classification.

Table 3.26 – Min and max values of E_{MSE} and num Connections based on P_{bit_mutate} for BUPA Liver Disorder Classification.

	P_{bit_mutate}	E_{MSE}	Num Connections
Min E_{MSE}	0.05	7.01E-17	47
Max E_{MSE}	0.03	9.06E-08	11
Min num connections	0.03	9.06E-08	11
Max num connections	0.02	6.83E-08	54

Iris Plant Classification:

Figure 3.28 and table 3.27 show the effects of P_{bit_mutate} on E_{MSE} and number of connections for the Iris Plant Classification application. Neither E_{MSE} or the number of connections seem to have a correlation with P_{bit_mutate}.

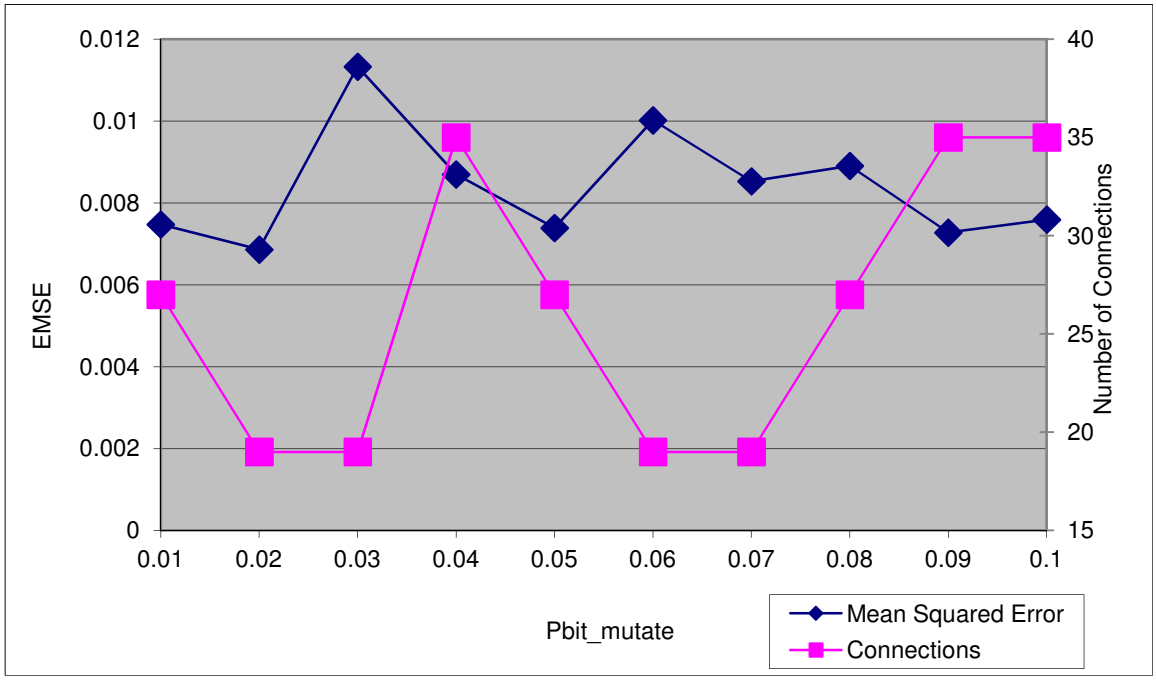

Figure 3.28 – Effects of P_{bit_mutate} on E_{MSE} and number of connections for Iris Plant Classification.

Table 3.27 – Min and max values of E_{MSE} and num Connections based on P_{bit_mutate} for Iris Plant Classification.

	P_{bit_mutate}	E_{MSE}	Num Connections
Min E_{MSE}	0.02	0.006863	19
Max E_{MSE}	0.03	0.011328	19
Min num connections	0.02	0.006863	19
Max num connections	0.04	0.008695	35

3.3.5 Comparison with another Algorithm

In [6], a formula was developed to generate a range for the ideal number of hidden layer neurons in a single hidden layer neural network (see formula 2.20). The results generated during simulations in this thesis are compared to that formula. First, the ideal number of hidden layer neurons is computed for the five test functions in table 3.28.

Table 3.28 – Ideal number of hidden layer neurons for all test functions based on formula 2.20.

	Inputs	Outputs	Ideal number of hidden neurons
XOR	2	1	(2+1)^0.5 + (1~10) = 4~13
Sine	1	1	(1+1)^0.5 + (1~10) = 3~12
DC-DC converter	3	1	(3+1)^0.5 + (1~10) = 3~12
BUPA liver disorder	6	2	(6+2)^0.5 + (1~10) = 4~13
Iris plant	4	3	(4+3)^0.5 + (1~10) = 4~13

Tables A.3, B.3, C.3, D.3, and E.3 were examined to compare the number of hidden layer neurons in the simulation solutions with the calculated ideal range. The population parameter table was chosen because of a constant s_{err} of 0.7 and since the population does not seem to affect error or size of the solution. The comparison is displayed in table 3.29. The results show that all single hidden layer solutions found by the genetic algorithm during simulations are either within or below the calculated ideal range.

Table 3.29 – Comparing number of hidden neurons in GA generated solutions with calculated ideal range.

	Number of single layer solutions	Number of solutions within range	Number of solutions below range
XOR	8	1	7
Sine	6	6	0
DC-DC converter	4	3	1
BUPA liver disorder	8	3	5
Iris plant	5	1	4

3.3.6 Simulation Algorithm Complexity

By analyzing the simulation code, the time complexity of each of the simulation steps is described. These components include: initial population generation, selection, crossover, mutation, evaluation and training.

Since initial population generation requires iterating over all individuals, it has a complexity of *O(n)*, or linear, as population size increases. Selection involves generating two random numbers, and therefore has a complexity of *O(1)*, or constant time. The crossover step involves iterating over each gene or bit in the two parents' genomes. Since the genome size is fixed at 24 bits, the complexity is also *O(1)*, or constant time. Similarly, mutation is *O(1)* as it iterates over all of the bits in the fixed size genome. The evaluation step involves several bubble sort operations, in order to rank the new individual based on error and size. The bubble sort, and therefore the evaluation step, has a complexity of $O(n^2)$.

Training consumes the majority of the processing time when running simulations. During training, the input signal propagates through the network to generate an output, followed by the error signal propagating backwards through the network to generate weight correction values. During each incremental training step, the code iterates over all

synaptic connections twice, and therefore has a complexity of $O(n)$, or linear, as number of synapses increases. Figure 3.29 illustrates the relationship between the number of connections in a network and training time. These samples represent training different sized networks for the DC-DC converter application for 2000 epochs.

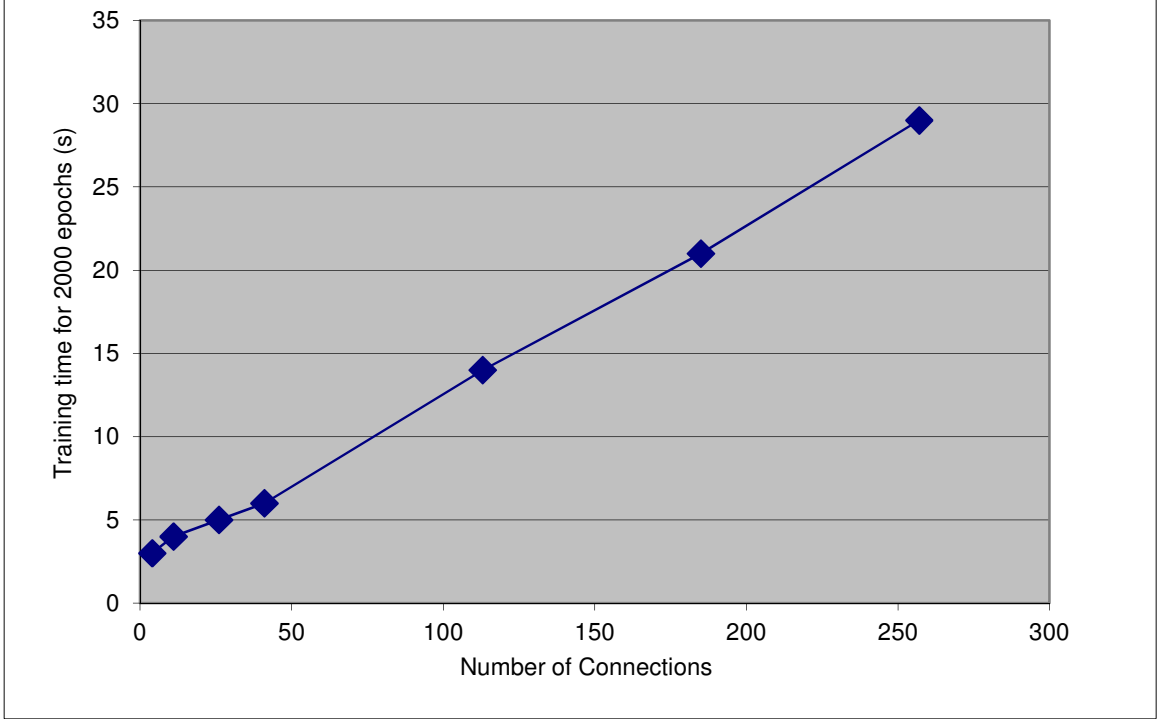

Figure 3.29 – Effects of network size on training time.

Chapter 4: Conclusions and Future Work

The scaling factor based genetic algorithm has been successfully applied to the five test functions: XOR approximation, Sine approximation, the DC-DC converter controller, BUPA Liver Disorder Classification, and Iris Plant Classification applications. The genetic algorithm generated neural networks with architectures that produced low E_{MSE} with few synaptic connections. The scaling factor based genetic algorithm has several advantages over previously used neural network related genetic algorithms.

The scaling factor based genetic algorithm allows for optimization based on both E_{MSE} and number of connections. The algorithm optimizes neural networks with anywhere from one to four hidden layers and up to eight neurons in each layer. The genome is much simpler than previous algorithm genomes because it only stores a random seed number instead of the starting values of all synaptic weights.

There have been a plethora of neural network related genetic algorithms to choose from over the years. The scaling factor based genetic algorithm offers another option that is relatively simple, allowing for straightforward implementation and fast performance. However, future research could be done to modify and enhance the algorithm, such as applying this algorithm to determine the architecture of recurrent neural networks and networks that are not fully connected. The algorithm can also be applied to new classes of problems.

Furthermore, the genome can be modified to improve performance or solution quality. Allowing for different activation functions and slopes for each neuron or each layer could also be studied, which might lower training time and improve E_{MSE}. The

individual genes can be enlarged to increase the search space. The number of allowed hidden layers and neurons per layer could be increased to search through larger neural networks. The number of possible learning rates, activation slopes, and random number generator seeds can also be increased.

List of References

[1] Q. Xiao, W. Shi, X. Xian and X. Yan, "An image restoration method based on genetic algorithm BP neural network," *Proceedings of the 7th World Congress on Intelligent Control and Automation*, pp. 7653-7656, 2008

[2] L. Jinru, L. Yibing and Y. Keguo, "Fault diagnosis of piston compressor based on Wavelet Neural Network and Genetic Algorithm," *Proceedings of the 7th World Congress on Intelligent Control and Automation*, pp. 6006-6010, 2008

[3] Y. Du and Y. Li, "Sonar array azimuth control system based on genetic neural network," *Proceedings of the 7th World Congress on Intelligent Control and Automation*, pp. 6123-6127, 2008

[4] W. Wu, W. Guozhi, Z. Yuanmin and W. Hongling, "Genetic Algorithm Optimizing Neural Network for Short-Term Load Forecasting," *International Forum on Information Technology and Applications*, pp. 583-585, 2009

[5] Z. Chen, "Optimization of Neural Network Based on Improved Genetic Algorithm," *International Conference on Computational Intelligence and Software Engineering,* pp.1-3, 2009

[6] X. Chen, X. Xu, Y. Wang, and X. Zhong, “Found on Artificial Neural Network and Genetic Algorithm Design the ASSEL Roll Profile,” *Proceedings of the International Conference on Computer Science and Software Engineering*, Vol. 01, pp. 40-44, 2008

[7] C. Tang, Y. He and L. Yuan, “A Fault Diagnosis Method of Switch Current Based on Genetic Algorithm to Optimize the BP Neural Network,” *International Conference on Electric and Electronics,* Vol. 99, pp. 943-950, 2011

[8] S. Zeng, J. Li and L. Cui, “Cell Status Diagnosis for the Aluminum Production on BP Neural Network with Genetic Algorithm,” *Communications in Computer and Information Science,* Vol. 175, pp. 146-152, 2011

[9] S. Nie and B. Ye, "The Application of BP Neural Network Model of DNA-Based Genetic Algorithm to Monitor Cutting Tool Wear," *International Conference on Measuring Technology and Mechatronics Automation*, pp. 338-341, 2009

[10] W. Yinghua and X. Chang, "Using Genetic Artificial Neural Network to Model Dam Monitoring Data," *Second International Conference on Computer Modeling and Simulation,* pp. 3-7, 2010

[11] D. Sabo, “A Modified Iterative Pruning Algorithm for Neural Network Dimension Analysis”, A Thesis for California Polytechnic State University, October,
2007.

[12] X. Fu, P.E.R. Dale and S. Zhang, “Evolving Neural Network Using Variable String Genetic Algorithms (VGA) for Color Infrared Aerial Image Classification,” *Chinese Geographical Science*, Vol. 18(2), pp. 162-170, 2008

[13] P. Koehn, “Combining Genetic Algorithms and Neural Networks: The Encoding Problem,” University of Tennessee, Knoxville, 1994

[14] D. Whitley, T. Starkweather and C. Bogart, “Genetic Algorithms and Neural Networks: Optimizing Connections and Connectivity,” *Parallel Computing*, Vol. 14, pp. 347-361, 1990

[15] P. W. Munro, "Genetic Search for Optimal Representation in Neural Networks," *Proceedings of the International Joint Conference on Neural Networks and Genetic Algorithms*, pp. 675-682, 1993

[16] D. Dasgupta and D. R. McGregor, "Designing Application-Specific Neural Networks using the Structured Genetic Algorithm," *Proceedings of International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pp. 87-96, 1992

[17] J. M. Bishop and M. J. Bushnell, "Genetic Optimization of Neural Network Architectures for Colour Recipe Prediction," *Proceedings of the International Joint Conference on Neural Networks and Genetic Algorithms*, pp. 719-725, 1993

[18] S. A. Harp and T. Samad, "Genetic Synthesis of Neural Network Architecture," *Handbook of Genetic Algorithms*, pp. 202-221, 1991

[19] G. G. Yen and H. Lu, "Hierarchical Genetic Algorithm Based Neural Network Design," *IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks*, pp. 168-175, 2000

[20] V. Bevilacqua, G. Mastronardi, F. Menolascina, P. Pannarale and A. Pedone, "A Novel Multi-Objective Genetic Algorithm Approach to Artificial Neural Network Topology Optimisation: The Breast Cancer Classification Problem," *International Joint Conference on Neural Networks*, pp. 1958-1965, 2006

[21] W. Li, "A Neural Network Controller for A Class of Phase-Shifted Full-Bridge DC-DC Converter," California Polytechnic University – San Luis Obispo, CA, 2006

[22] D. Whitley, “The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best,” *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 116-121, 1989

[23] D. W. White, “GANNet: A Genetic Algorithm for Searching Topology and Weight Spaces in Neural Network Design,” University of Maryland, 1993

[24] C. Darwin, *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*, London, England: John Murray, 1859

[25] P. J. Lisboa and A. F.G. Taktak, “The Use of Artificial Neural Networks in Decision Support in Cancer: A Systematic Review,” Liverpool John Moores University, UK, 2005

[26] O. Parsons and G. A. Carpenter, “ARTMAP Neural Networks for Information Fusion and Data Mining: Map Production and Target Recognition Methodologies,” Boston University, MA, 2002

[27] R. Sulej, K. Zaremba, K. Kurek and R. Rondio, “Application of the Neural Networks in Events Classification in the Measurement of the Spin Structure of the Deuteron,” Warsaw University of Technology, Poland, 2007

[28] J. E. Meng, W. Chen and W. Shiqian, “High-speed Face Recognition Based on Discrete Cosine Transform and RBF Neural Networks,” *IEEE Transactions on Neural Networks*, Volume 16, pp. 679-691, 2005

[29] L.S. Buriol, M. G.C. Resende, C. C. Ribeiro and M. Thorup, "A Hybrid Genetic Algorithm for the Weight Setting Problem in OSPF/IS-IS Routing," *Wiley Periodicals, NETWORKS*, Volume 46, pp. 36-56, 2005

[30] A. S. Wu, H. Yu, S. Jin, K. Lin and G. Schiavone, "An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems* Volume 15, pp. 824-834, 2004

[31] U. Aickelin and K. A. Dowsland, "An Indirect Genetic Algorithm for a Nurse-Scheduling Problem," *Computers & Operations Research*, Volume 31, pp. 761-778, 2004

[32] G. Bocharov, N. J. Ford, J. Edwards, T. Breinig, S. Wain-Hobson and A. Meyerhans, "A Genetic-Algorithm Approach to Simulating Human Immunodeficiency Virus Evolution Reveals the Strong Impact of Multiply Infected Cells and Recombination," *Journal of General Virology*, Volume 86, pp. 3109-3118, 2005

[33] C. K. Chow, H. T. Tsui and T. Lee, "Surface Registration Using a Dynamic Genetic Algorithm," *The Journal of the Pattern Recognition Society*, Volume 37, pp. 105-117, 2003

Appendix A – XOR Approximation Data

Table A.1. – Genomes of most fit individuals based on scaling factor using bit-level crossover for XOR approximation.

s_{err}	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Fitness	50	47.8	47.6	46.4	47.2	46	41.4	44.9	47	46.3	50
r_{err}	30	23	13	13	8	3	8	4	2	1	1
E_{MSE}	0.042059	0.044798	0.043165	0.042285	0.044240	0.000014	0.000244	0.000061	0.000014	0.000013	0.000007
r_{size}	1	1	1	1	1	7	12	11	12	38	24
Number of connections	5	5	5	5	5	21	13	17	30	43	65
Number of layers	1	1	1	1	1	1	1	1	2	2	2
Number of neurons L0	1	1	1	1	1	5	3	4	3	4	8
Number of neurons L1	5	8	1	2	1	5	5	3	4	5	4
Number of neurons L2	7	7	8	4	1	2	3	4	8	6	8
Number of neurons L3	5	2	8	8	2	1	3	8	6	8	3
Learning rate	3	5	3	5	4	7	6	5	5	6	5
Slope	7	4	7	5	6	5	6	5	6	5	6
Random number seed	2	5	5	4	7	5	5	3	7	4	1
Using sigmoid	0	0	1	0	0	0	1	0	0	0	0

Table A.2. – Genomes of most fit individuals based on scaling factor using gene-level crossover for XOR approximation.

s_{err}	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Fitness	50	48.3	50	45.8	45.2	40	45.4	49.1	46.6	46.2	50
r_{err}	26	18	1	8	7	16	4	1	4	3	1
E_{MSE}	0.049888	0.042765	0.000031	0.000157	0.000041	0.000244	0.000023	0.000006	0.000249	0.000015	0.000003
r_{size}	1	1	1	4	5	6	8	4	6	21	33
Number of connections	5	5	9	9	17	13	20	19	13	23	69
Number of layers	1	1	1	1	1	1	2	2	1	2	4
Number of neurons L0	1	1	2	2	4	3	3	2	3	2	8
Number of neurons L1	6	7	4	6	4	8	2	3	8	4	2
Number of neurons L2	2	7	7	7	6	7	4	7	4	3	6
Number of neurons L3	2	2	3	1	2	8	3	7	5	3	1
Learning rate	0	2	4	6	2	6	7	7	6	6	4
Slope	6	6	6	5	7	6	7	7	6	7	7
Random number seed	7	7	4	1	4	5	3	5	2	1	4
Using sigmoid	0	0	0	0	0	1	1	1	1	1	0

Table A.3. – Genomes of most fit individuals based on population size for XOR approximation.

P	10	20	30	40	50	60	70	80	90	100
Fitness	9.3	17.9	24.4	38.8	47	54.4	66.2	74.3	83.9	91.2
r_{err}	2	4	6	1	4	6	6	4	8	11
E_{MSE}	0.000017	0.000413	0.000024	0.000025	0.000696	0.000056	0.000557	0.000067	0.000151	0.000041
r_{size}	1	1	8	5	4	8	2	13	5	7
Number of connections	15	9	30	13	17	13	13	13	9	13
Number of layers	2	1	2	1	1	1	1	1	1	1
Number of neurons L0	2	2	3	3	4	3	3	3	2	3
Number of neurons L1	2	5	4	6	8	4	1	7	4	6
Number of neurons L2	1	5	6	3	6	2	8	1	1	4
Number of neurons L3	3	3	5	7	2	8	2	7	5	6
Learning rate	6	4	6	7	6	3	6	5	7	4
Slope	5	7	5	7	4	7	4	5	4	6
Random number seed	2	3	5	7	3	7	3	1	6	7
Using sigmoid	0	1	0	1	0	0	0	0	0	0

Table A.4. – Genomes of most fit individuals based on bit mutation rate for XOR approximation.

P_{bit_mutate}	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.10
Fitness	41.3	47.5	47.5	44.3	41.2	40.7	46	47.6	45	48.6
r_{err}	13	2	2	1	2	4	2	4	3	3
E_{MSE}	0.000066	0.000013	0.000035	0.000006	0.000016	0.00005	0.000011	0.000022	0.000019	0.000029
r_{size}	2	7	7	20	28	25	12	2	13	1
Number of connections	17	11	21	32	54	40	15	13	20	9
Number of layers	1	2	1	3	4	2	2	1	2	1
Number of neurons L0	4	2	5	3	3	7	2	3	3	2
Number of neurons L1	6	1	6	4	3	2	2	5	2	4
Number of neurons L2	1	5	8	1	3	4	6	2	6	2
Number of neurons L3	6	6	1	2	4	7	3	3	5	2
Learning rate	5	7	6	4	4	3	7	7	3	2
Slope	6	5	5	7	6	6	5	7	7	7
Random number seed	2	6	7	1	1	7	7	2	3	2
Using sigmoid	0	0	0	0	0	0	0	1	0	0

Appendix B – Sine Approximation Data

Table B.1. – Genomes of most fit individuals based on scaling factor using bit-level crossover for Sine approximation.

s_{err}	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Fitness	50	48.6	43.4	44.3	43.6	40	44	40.6	42.6	47.5	50
r_{err}	13	15	34	20	17	16	3	8	5	3	1
E_{MSE}	0.009573	0.009636	0.009433	0.009509	0.009566	0.005217	0.000092	0.000058	0.000067	0.000075	0.000058
r_{size}	1	1	1	1	1	6	13	16	22	8	24
Number of connections	4	4	4	4	4	7	19	22	25	22	22
Number of layers	1	1	1	1	1	1	1	1	1	1	1
Number of neurons L0	1	1	1	1	1	2	6	7	8	7	7
Number of neurons L1	4	2	2	4	1	7	7	2	6	1	4
Number of neurons L2	4	4	2	8	7	2	7	4	8	3	8
Number of neurons L3	6	3	6	4	1	5	1	3	7	8	6
Learning rate	2	7	7	1	5	7	6	0	7	1	0
Slope	5	4	3	6	4	3	4	7	4	7	7
Random number seed	4	0	5	2	0	2	0	4	4	2	4
Using sigmoid	0	0	0	0	0	0	0	0	0	0	0

Table B.2. – Genomes of most fit individuals based on scaling factor using gene-level crossover for Sine approximation.

s_{err}	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Fitness	50	47.6	45	41.9	41.8	43	42	42.8	43.6	47	50
r_{err}	37	25	26	28	11	3	5	1	3	1	1
E_{MSE}	0.012383	0.009636	0.009491	0.009498	0.000132	0.000109	0.000102	0.000059	0.000063	0.000068	0.000058
r_{size}	1	1	1	1	8	13	15	25	25	31	47
Number of connections	4	4	4	4	13	21	19	32	37	61	67
Number of layers	1	1	1	1	1	2	1	2	2	2	2
Number of neurons L0	1	1	1	1	4	2	6	3	8	6	5
Number of neurons L1	3	6	2	5	8	4	1	5	2	6	8
Number of neurons L2	1	5	3	3	6	1	6	5	7	3	5
Number of neurons L3	1	5	7	1	3	7	7	6	3	2	4
Learning rate	3	7	0	6	3	7	7	2	3	0	3
Slope	7	4	7	3	5	4	4	6	5	7	5
Random number seed	6	0	6	2	2	4	3	7	1	3	2
Using sigmoid	0	0	0	0	0	0	0	0	0	0	0

Table B.3. – Genomes of most fit individuals based on population size for Sine approximation.

P	10	20	30	40	50	60	70	80	90	100
Fitness	9.1	19.7	23.2	31.9	44.6	51	58.4	68.3	82.9	91.1
r_{err}	1	1	6	1	4	7	15	7	9	3
E_{MSE}	0.000034	0.000070	0.000103	0.000092	0.001167	0.005215	0.001617	0.000091	0.005133	0.000067
r_{size}	4	2	12	28	12	17	7	26	6	26
Number of connections	99	19	19	63	13	16	10	27	10	22
Number of layers	3	1	1	3	2	1	1	2	1	1
Number of neurons L0	3	6	6	4	2	5	3	3	3	7
Number of neurons L1	8	3	8	6	2	2	4	4	2	4
Number of neurons L2	6	7	6	3	7	5	1	3	6	6
Number of neurons L3	8	1	8	2	7	8	3	3	2	7
Learning rate	2	5	3	2	2	2	5	2	7	1
Slope	6	5	6	5	6	5	3	5	3	7
Random number seed	2	6	2	2	1	2	5	7	6	4
Using sigmoid	0	0	0	0	0	0	0	0	0	0

Table B.4. – Genomes of most fit individuals based on bit mutation rate for Sine approximation.

P_{bit_mutate}	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.10
Fitness	44.7	43.4	42.9	40.8	42.3	43.1	46.8	40.7	43.2	47.2
r_{err}	6	4	3	3	3	4	3	4	3	2
E_{MSE}	0.000076	0.000088	0.000100	0.000073	0.000093	0.000062	0.000147	0.000111	0.000123	0.000118
r_{size}	7	16	20	27	22	17	7	25	19	8
Number of connections	19	37	27	39	22	22	13	22	16	13
Number of layers	1	2	2	2	1	1	1	2	1	1
Number of neurons L0	6	2	3	5	7	7	4	3	5	4
Number of neurons L1	1	8	4	4	5	5	8	3	4	4
Number of neurons L2	2	6	2	5	8	5	3	6	6	1
Number of neurons L3	1	8	7	5	4	1	7	4	4	8
Learning rate	4	7	0	5	2	4	7	2	6	5
Slope	5	3	6	5	6	5	3	5	4	5
Random number seed	4	4	7	6	2	4	7	2	7	3
Using sigmoid	0	0	0	0	0	0	0	0	0	0

Appendix C – DC-DC Converter Controller Data

Table C.1. – Genomes of most fit individuals based on scaling factor using bit-level crossover for DC-DC converter controller.

s_{err}	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Fitness	50	46.7	44.8	45.3	46.4	46	48.8	45	47.2	48.7	50
r_{err}	39	34	7	12	7	9	1	6	2	2	1
E_{MSE}	0.001630	0.001147	0.000099	0.000351	0.000085	0.000106	0.000027	0.000043	0.000069	0.000031	0.000028
r_{size}	1	1	6	3	3	1	4	6	11	5	29
Number of connections	6	16	21	11	11	6	16	26	16	16	41
Number of layers	1	1	1	1	1	1	1	1	1	1	2
Number of neurons L0	1	3	4	2	2	1	3	5	3	3	4
Number of neurons L1	4	6	3	3	2	5	5	4	5	5	4
Number of neurons L2	7	7	5	3	3	6	2	5	5	5	2
Number of neurons L3	6	7	4	4	3	3	5	7	8	6	1
Learning rate	7	5	0	0	6	6	7	3	6	6	7
Slope	0	2	5	5	3	3	3	6	5	4	4
Random number seed	7	7	5	6	7	4	0	7	0	2	3
Using sigmoid	1	1	0	1	1	1	1	1	1	1	1

Table C.2. – Genomes of most fit individuals based on scaling factor using gene-level crossover for DC-DC converter controller.

s_{err}	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Fitness	50	48.2	48.4	46.9	47.2	46.5	41.6	44.4	48.4	46.8	50
r_{err}	21	19	9	9	8	3	13	9	2	1	1
E_{MSE}	0.000475	0.000103	0.000105	0.000104	0.000078	0.000029	0.000162	0.000084	0.000031	0.000103	0.000029
r_{size}	1	1	1	2	1	6	4	1	5	33	42
Number of connections	6	6	6	8	11	31	8	11	21	59	105
Number of layers	1	1	1	2	1	1	2	1	1	4	4
Number of neurons L0	1	1	1	1	2	6	1	2	4	2	3
Number of neurons L1	5	6	3	1	5	7	1	6	1	7	5
Number of neurons L2	1	3	5	4	1	7	6	8	4	3	5
Number of neurons L3	1	4	3	1	3	6	8	5	7	1	6
Learning rate	4	5	5	5	6	4	7	7	7	3	5
Slope	3	4	3	2	4	5	3	4	3	5	3
Random number seed	2	4	0	7	7	6	2	5	3	0	7
Using sigmoid	1	1	1	0	1	1	1	1	1	0	0

Table C.3. – Genomes of most fit individuals based on population size for DC-DC converter controller.

P	10	20	30	40	50	60	70	80	90	100
Fitness	10	17.3	26.7	35.4	43.3	54	64.3	78.3	84.4	95.2
r_{err}	1	1	1	5	2	7	4	3	9	7
E_{MSE}	0.000057	0.000027	0.000033	0.000105	0.000033	0.000095	0.000051	0.000032	0.000104	0.000098
r_{size}	1	10	12	7	21	7	13	2	1	3
Number of connections	93	59	36	17	38	21	31	18	6	16
Number of layers	2	2	1	2	2	2	1	2	1	1
Number of neurons L0	8	4	7	2	7	2	6	3	1	3
Number of neurons L1	6	7	6	2	1	3	7	1	1	8
Number of neurons L2	8	3	3	1	1	5	1	5	8	6
Number of neurons L3	5	7	7	8	1	1	5	5	3	2
Learning rate	3	7	7	2	6	7	4	5	7	1
Slope	5	2	3	4	2	1	4	2	2	5
Random number seed	1	3	1	4	4	0	6	0	6	7
Using sigmoid	1	0	1	0	0	0	1	0	1	1

Table C.4. – Genomes of most fit individuals based on bit mutation rate for DC-DC converter controller.

P_{bit_mutate}	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.10
Fitness	39.3	45.1	45.9	45	44.1	46	46.4	48.5	43.3	47.2
r_{err}	3	8	6	6	9	2	1	1	2	5
E_{MSE}	0.000040	0.000092	0.000141	0.000079	0.000096	0.000032	0.000029	0.000030	0.000030	0.000105
r_{size}	32	1	3	6	2	12	13	6	21	1
Number of connections	73	11	21	21	16	26	26	21	36	6
Number of layers	3	1	1	1	1	1	1	1	1	1
Number of neurons L0	7	2	4	4	3	5	5	4	7	1
Number of neurons L1	3	2	2	8	2	3	3	8	3	5
Number of neurons L2	4	8	7	4	1	2	1	8	4	6
Number of neurons L3	5	8	2	3	5	5	4	5	6	2
Learning rate	3	6	1	5	4	6	7	5	5	5
Slope	3	3	4	4	3	4	3	4	4	3
Random number seed	6	6	4	5	5	0	1	3	4	5
Using sigmoid	0	1	0	1	0	1	1	1	1	1

Appendix D – BUPA Liver Disorders Classification Data

Table D.1. – Genomes of most fit individuals based on scaling factor using bit-level crossover for BUPA Liver Disorders Classification.

s_{err}	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Fitness	50	48.1	49.4	47	45.4	46.5	45.6	45.9	48.2	46.3	50
r_{err}	33	11	4	11	11	4	3	3	1	2	1
E_{MSE}	1.15E-05	4.38E-07	1.13E-07	1.82E-07	1.82E-07	6.04E-08	3.69E-08	3.02E-08	6.61E-17	8.95E-12	1.34E-11
r_{size}	1	2	1	1	2	5	9	10	10	29	33
Number of connections	11	21	11	11	20	20	21	47	47	66	113
Number of layers	1	2	1	1	1	1	2	3	1	3	3
Number of neurons L0	1	2	1	1	2	2	1	3	5	3	6
Number of neurons L1	8	1	1	2	3	2	3	3	2	2	3
Number of neurons L2	6	6	4	1	7	1	3	2	1	7	8
Number of neurons L3	7	4	7	8	5	7	1	4	5	1	7
Learning rate	2	2	4	5	5	6	6	7	2	6	5
Slope	4	6	6	7	7	5	5	5	6	5	5
Random number seed	6	0	0	0	7	6	7	3	4	3	0
Using sigmoid	0	0	0	1	1	0	0	0	0	0	0

Table D.2. – Genomes of most fit individuals based on scaling factor using gene-level crossover for BUPA Liver Disorders Classification.

s_{err}	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Fitness	50	49.6	47.6	46.1	46.6	43.5	44.8	45.8	48.2	48.3	50
r_{err}	1	5	9	14	5	14	9	7	3	1	1
E_{MSE}	4.53E-08	9.06E-08	1.82E-07	1.98E-07	9.07E-08	1.81E-07	1.82E-07	4.93E-10	2.82E-11	6.61E-17	9.70E-16
r_{size}	1	1	2	1	4	1	2	1	2	18	34
Number of connections	11	11	20	17	20	11	20	29	29	47	47
Number of layers	1	1	1	2	1	1	1	1	1	1	1
Number of neurons L0	1	1	2	1	2	1	2	3	3	5	5
Number of neurons L1	3	3	2	2	4	4	8	2	2	2	1
Number of neurons L2	8	2	8	2	3	7	7	3	7	3	2
Number of neurons L3	3	5	2	8	4	8	8	4	6	8	3
Learning rate	7	6	5	7	6	5	5	1	3	4	4
Slope	5	7	7	4	7	5	7	6	6	6	6
Random number seed	2	5	3	2	4	2	2	1	3	4	5
Using sigmoid	0	1	1	0	1	0	1	0	0	0	0

Table D.3. – Genomes of most fit individuals based on population size for BUPA Liver Disorder Classification.

P	10	20	30	40	50	60	70	80	90	100
Fitness	10	17.6	29.3	39.1	46.6	54.3	64.4	73.7	81	96.6
r_{err}	1	1	2	1	5	1	9	10	4	5
E_{MSE}	4.49E-08	3.57E-12	4.53E-08	1.37E-13	4.41E-08	6.40E-09	4.53E-08	9.06E-08	8.32E-09	6.04E-08
r_{size}	1	9	1	4	3	20	1	1	24	3
Number of connections	35	20	29	38	21	47	11	11	47	20
Number of layers	3	1	1	1	2	1	1	1	1	1
Number of neurons L0	2	2	3	4	1	5	1	1	5	2
Number of neurons L1	4	8	2	3	3	7	8	5	8	8
Number of neurons L2	1	8	3	4	5	5	4	4	6	7
Number of neurons L3	1	1	8	7	3	2	2	3	1	8
Learning rate	7	1	7	0	7	6	7	6	1	6
Slope	7	7	7	6	7	5	5	7	5	5
Random number seed	0	1	2	0	2	4	5	4	7	5
Using sigmoid	1	0	1	0	1	0	0	1	0	0

Table D.4. – Genomes of most fit individuals based on bit mutation rate for BUPA Liver Disorders Classification.

P_{bit_mutate}	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.10
Fitness	43.5	42.7	47.2	45.7	45.8	47.9	47	50	48.2	46.5
r_{err}	9	8	5	5	1	4	4	1	1	6
E_{MSE}	4.53E-08	6.83E-08	9.06E-08	3.02E-08	7.01E-17	4.52E-08	6.16E-08	2.27E-11	2.27E-11	2.83E-08
r_{size}	4	9	1	6	15	1	4	1	7	1
Number of connections	29	54	11	20	47	31	33	20	20	11
Number of layers	1	4	1	1	1	3	2	1	1	1
Number of neurons L0	3	1	1	2	5	2	1	2	2	1
Number of neurons L1	7	4	6	1	8	3	6	4	3	7
Number of neurons L2	4	1	1	8	6	1	6	5	7	5
Number of neurons L3	7	8	4	4	5	8	4	1	2	4
Learning rate	7	7	6	7	3	7	3	0	2	4
Slope	7	4	7	5	6	7	6	7	7	7
Random number seed	6	0	4	4	7	7	6	7	7	5
Using sigmoid	1	0	1	0	0	1	0	0	0	0

Appendix E – Iris Plant Classification Data

Table E.1. – Genomes of most fit individuals based on scaling factor using bit-level crossover for Iris Plant Classification.

s_{err}	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Fitness	50	48	49.2	49.1	42.8	47	42.6	46.2	47.2	46.3	50
r_{err}	20	21	1	4	16	5	8	3	3	4	1
E_{MSE}	0.066205	0.067845	0.008138	0.011998	0.016737	0.007494	0.013795	0.01626	0.00963	0.00722	0.006714
r_{size}	1	1	2	1	3	3	9	9	7	11	5
Number of connections	11	11	19	19	27	43	28	35	19	27	27
Number of layers	1	1	1	1	1	1	2	1	1	1	1
Number of neurons L0	1	1	2	2	3	5	1	4	2	3	3
Number of neurons L1	8	8	6	7	7	3	4	6	5	6	3
Number of neurons L2	6	3	4	2	1	4	4	8	2	6	6
Number of neurons L3	4	4	5	3	5	5	5	2	8	4	2
Learning rate	6	4	7	7	7	7	4	4	7	7	7
Slope	3	4	3	2	4	3	3	2	3	3	4
Random number seed	1	3	0	6	1	3	3	2	1	0	4
Using sigmoid	1	1	0	1	1	1	0	0	0	1	1

Table E.2. – Genomes of most fit individuals based on scaling factor using gene-level crossover for Iris Plant Classification.

s_{err}	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
Fitness	50	48.9	48	48.1	43.8	42.5	44.4	49.7	44.8	46.9	50
r_{err}	17	12	7	5	9	6	7	1	5	4	1
E_{MSE}	0.009755	0.010137	0.009755	0.008138	0.01065	0.00739	0.008138	0.006905	0.009204	0.017416	0.004046
r_{size}	1	1	2	2	6	11	6	2	11	5	40
Number of connections	19	19	19	19	27	27	19	19	19	19	91
Number of layers	1	1	1	1	1	1	1	1	1	1	2
Number of neurons L0	2	2	2	2	3	3	2	2	2	2	5
Number of neurons L1	5	4	4	4	8	4	3	8	6	8	7
Number of neurons L2	2	4	2	8	2	7	8	2	2	5	7
Number of neurons L3	4	8	8	2	1	4	3	3	8	3	4
Learning rate	5	7	5	7	7	7	7	7	6	6	7
Slope	4	2	4	3	2	3	3	4	3	1	4
Random number seed	1	5	1	0	7	2	0	2	6	7	4
Using sigmoid	1	0	1	0	0	1	0	1	0	0	1

Table E.3. – Genomes of most fit individuals based on population size for Iris Plant Classification.

P	10	20	30	40	50	60	70	80	90	100
Fitness	7.9	15.1	25.1	33.6	43.2	57.3	66.3	77.6	79.2	91
r_{err}	1	2	5	5	9	4	2	4	13	4
E_{MSE}	0.003445	0.00684	0.006949	0.009844	0.011475	0.008116	0.007278	0.009751	0.019429	0.008435
r_{size}	8	15	8	13	5	3	11	2	9	24
Number of connections	93	92	43	39	27	19	31	19	27	35
Number of layers	2	4	2	2	1	1	2	1	1	1
Number of neurons L0	6	5	2	3	3	2	2	2	3	4
Number of neurons L1	6	4	5	3	2	6	3	7	1	1
Number of neurons L2	1	2	5	3	4	8	4	8	3	1
Number of neurons L3	8	5	2	3	8	7	5	7	7	7
Learning rate	7	7	6	4	7	5	5	4	5	7
Slope	4	2	4	3	2	4	4	4	1	4
Random number seed	5	6	3	3	0	3	6	2	2	7
Using sigmoid	1	0	1	0	1	1	1	1	0	1

Table E.4. – Genomes of most fit individuals based on bit mutation rate for Iris Plant Classification.

P_{bit_mutate}	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09	0.10
Fitness	47	49.4	48.7	46.2	45.4	44.8	48.3	47.7	47.2	44
r_{err}	1	1	2	3	2	5	3	3	2	4
E_{MSE}	0.007475	0.006863	0.011328	0.008695	0.007389	0.01002	0.008529	0.008905	0.007277	0.007593
r_{size}	11	3	3	9	14	9	2	4	8	14
Number of connections	27	19	19	35	27	19	19	27	35	35
Number of layers	1	1	1	1	1	1	1	1	1	1
Number of neurons L0	3	2	2	4	3	2	2	3	4	4
Number of neurons L1	1	2	5	4	1	3	3	4	4	2
Number of neurons L2	8	6	2	2	2	8	2	4	5	3
Number of neurons L3	5	6	7	6	7	5	6	1	8	5
Learning rate	5	7	5	6	5	7	6	6	7	5
Slope	4	4	3	3	5	2	4	3	4	4
Random number seed	2	0	3	0	3	2	3	2	5	2
Using sigmoid	1	1	0	1	1	0	0	0	1	1

Appendix F – Simulator Source Code

F.1. NNSimulation.h

```
/*
Ariel Kopel
Thesis Project
Advisor: Dr. Helen Yu
Cal Poly State University - San Luis Obispo

NNSimulation.h - Defines constants and functions used for NN simulation.
*/

#ifndef NNSIMULATION_H
#define NNSIMULATION_H

#include <fstream>

/*** Constants ***/

// Maximum number of synapses per neuron and therefore also the max number
// of inputs in a fully connected NN.
#define MAX_INPUTS (MAX_HID_NEURONS + 1)

// Maximum number of outputs of the system.
#define MAX_OUTPUTS 20

// The maximum number of hidden neurons in the neural network.
#define MAX_HID_NEURONS x1s(NUM_BITS_numHidNeurons)+1

// The maximum number of layers in a fully connected NN
#define MAX_HIDDEN_LAYERS x1s(NUM_BITS_numHidLayers)+1

// The min and max number individuals to be used in the GA.
#define MAX_POPULATION_SIZE 100
#define MIN_POPULATION_SIZE 5

// The number of epochs of training used to evaluate the new child.
#define TRAINING_CYCLES_IN_GENERATION 2000

// The number of bits in each of the genes of the GAIndividual.
#define NUM_BITS_numHidLayers 2
#define NUM_BITS_n 3
#define NUM_BITS_slope 3
#define NUM_BITS_randomNumSeed 3
#define NUM_BITS_numHidNeurons 3

/*** Macros ***/
```

```
// Calculates the integer which is made up of x binary 1's.
#define x1s(x) (0x1 << x)-1

/*** Structure Definitions ***/

// TrainingFileHeader structure defines an input file and is the
// first data in the file. Following this header, the file contains
// training sets consisting of an array of doubles of inputs followed
// by an array of doubles of outputs. There are numSamples sets in the file
// and the input/output double arrays are of length numInputs and numOutputs.
// The first numTrainingSamples of them are used for training, while all of
them
// are used for evaluation.
struct TrainingFileHeader {
    int numInputs;
    int numOutputs;
    int numTotalSamples;
    int numTrainingSamples;
}typedef TrainingFileHeader;

// WeightFileHeader structure defines a weight file and is the first data
// in the file. Following the header, the remainder of the files contains
// doubles, each of which corrosponds to the weight of a synapse in the
system.
// The weights are placed in the file in the same order as the inputVector[]
// of each Neuron in the network. The weights of each Neuron of hidden layer 0
// will be first, then the weights of each Neuron in layer 1 and so on
// until all of the weights of all of the Neurons have been written into the
file.
// The members have the same names as the corrosponding members in the
// FullyConnectedNN class.
struct WeightFileHeader {
    int numHidNeurons[MAX_HIDDEN_LAYERS]; // Number of hidden neurons in each
layer.
    int numInputs; // Number of inputs.
    int numOutputs; // Number of outputs.
    int numHidLayers;    // Number of hidden layers
}typedef WeightFileHeader;

// GAIndividual defines the genes (data members) of one individual in the
// genetic algorithm population and other fitness related items.
struct GAIndividual {
    int numHidLayersOneLess;    // One less than the number of hidden layers
    int numHidNeurons[MAX_HIDDEN_LAYERS]; // One less than the number
        // of hidden neurons in each layer (A value of 15 means 16 hidden
neurons).
    int n; // An encoded learning rate.
    int slope; // An encoded tanh or sigmoid slope.
    int randomNumSeed; // Random number generator seed.
    char usingSigmoid; // Set to 1 for sigmoid, 0 for tanh
    int numConnections; // The number of synapses and biases in the NN.
    double errorRMS; // RMS of error after TRAINING_CYCLES_IN_GENERATION
                     // epochs of training.
    int connectionsReverseRank; // Reverse rank based on numConnections.
```

```
    int errorReverseRank; // Reverse rank based on errorRMS.
    double fitness; // Fitness based on both ranks.
}typedef GAIndividual;

/*** Function Prototypes ***/

// Writes header packet to output file. Returns false on success, true
// on failure.
static bool WriteHeaderToOutputFile();

// Reads header packet from input file. Returns false on success, true
// on failure, which could be due to the input file containing a system
// layout different from SYSTEM_LAYOUT. header will be populated with
// the file header.
static bool ReadHeaderFromInput();

// Writes one packet to outputFile which contains the input and output
// vectors as double arrays. Returns true on failure, false on success.
static bool WriteOnePacket();

// Reads one packet from inputFile which contains the input and desired
// output vectors as double arrays. print dictates whether or not to display
// the inputs to the screen when inputting from a file. Returns true on
// failure, false on success.
static bool ReadOnePacket(bool print, bool fromUser);

// Seeks to the next randomly selected training sample in inputFile.
// Returns true on failure, false on success.
static bool SeekToNextRandPacket();

// Deallocates all allocated memory and closese files.
static void CleanUp();

// Returns a random integer between 0 and maxInt.
static int GetRandInt(int maxInt);

// Calculates the RMS error for NN and displays the results if display is
true.
static double CalcRMSError(bool print);

// Trains NN for numEpochs
static void TrainForXEpochs(int numEpochs, bool print);

// Generates an XOR training file containing 4 training samples and
// 4 evaluation samples.
static int GenerateXORTrainingFile();

// Generates a 130 sample training file containing one cycle of a sin wave.
// 100 samples are training samples and 30 samples are for evaluation.
static int GenerateSinTrainingFile();

// Initializes GAPopulation by randomly creating all the individuals and
ranking
// them based on error RMS and number of connections.
```

```
static void InitializeGAPopulation();

// Determines the errorReverseRank, connectionsReverseRank, and rank for all
individuals
// in the GAPopulation. Finally, GAPopulation is sorted by fitness rank.
static void DetermineFitnessAndSort(bool firstTime);

// Performs a crossover between the two selected parents from GAPopulation.
// The crossover can either be on the gene level or bit level.
static void CrossOver(int parent1, int parent2, bool useBitXOver);

// Allocate NN based on index into GAPopulation.
static void AllocateNN(int i);

// Determine and set numConnections for individual i in GAPopulation.
static void SetNumConnections(int i);

// Mutates one individual with a 1/oneInX chance of mutation for each bit.
static void Mutate(int idx, int oneInX);

// Display either detailed or overview description of GAPopulation.
static void DisplayGAPopulation(bool detailed);

// Generates an input file based on file "DCController.csv" which has on each
line
// the three inputs and one output of the DC Power controller NN.
DCController.csv
// is expected to have 2002 samples with 1001 for training and 1001 for
evaluation.
// The program exits at the end of this function call.
static void GenerateDCControllerInputFile();

#endif
```

F.2. NNSimulation.cpp

```
/*
Ariel Kopel
Thesis Project
Advisor: Dr. Helen Yu
Cal Poly State University - San Luis Obispo

NNSimulation.cpp - Simulates a feedforward neural network, which can
be trained using the error correction back propagation algorithm.
*/

#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include "Neuron.h"
#include "FullyConnectedNN.h"
#include "NNSimulation.h"
#include <math.h>
#include <time.h>
using namespace std;

/*** Global Variables ***/

// Stores the header packet of the input file, which dictates the layout
// of the system.
TrainingFileHeader header;

// FILE pointers to the input and output files.
static FILE *inputFile = NULL;
static FILE *outputFile = NULL;
static FILE *csvFile = NULL;
static FILE *weightFile = NULL;

// Input, desired output, and actual output vectors of the NN.
static double inputs[MAX_INPUTS];
static double desiredOutputs[MAX_OUTPUTS];
static double actualOutputs[MAX_OUTPUTS];

// The fully connected Neural Network to be used in the simulation.
static FullyConnectedNN *NN;

// User constraints.
static bool training;

// Defines the 3 bit encoding of the learning rate N.
static double decodeN[] = {0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1.0, 2.0};

// Defines the 3 bit encoding of the sigmoid or tanh slope.
static double decodeSlope[] = {0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1.0, 2.0};
```

```
// Contains the population of individuals used in the GA plus the new child
// created each generation.
static GAIndividual GAPopulation[MAX_POPULATION_SIZE+1];

// Stores the highest probability slice for each individual based on fitness.
static int highestProbSlice[MAX_POPULATION_SIZE];

// Scaling factors for determining fitness of GA individuals.
static double SCALING_FACTOR_ERR_RMS;
static double SCALING_FACTOR_NUM_CONNECTIONS;

// Number of individuals in the GA population.
static int POPULATION_SIZE;

// There will be a 1 in MUTATION_RATE chance of a bit flipping.
static int MUTATION_RATE;

// Main function.
int main (int argc, char *argv[]) {
    int selection, numGenerations; // user input.
    string filename;
    double n = 0, a = 0, b = 0;
    bool usingSigmoid = false, useGA = false, useBitXOver = false;
       char useBitXOverSelect;
    int numHidNeuronsMinOne[MAX_HIDDEN_LAYERS]; // Number of hidden neurons in
each layer
                                                 // minus one.
    int numHidLayers; // Number of hidden layers
       int i, j;
    int maxProbSlice; // Max probability "slice" number in selection of
                      // individuals from GAPopulation.
    int parent1, parent2; // Store indecies of the two parents in one
generation.

    // Set precision to fixed and 6.
    cout << setprecision(6) << setiosflags( ios::left );

    // User input mode?
    cout << "1. Define desired function and create training file."
         << endl;
    cout << "2. Train Sigmoid based NN using training file." << endl;
    cout << "3. Train tanh based NN using training file." << endl;
    cout << "4. Use GA to determine optimal NN." << endl;
    cout << "Enter selection: ";
    cin >> selection;

    // Clear '\n'.
    getline(cin,filename);

    // Seed random number generator.
    srand((unsigned)time(NULL));

    // Decide mode based on user input.
    switch (selection) {
```

```
    // User chose to create an training file.
    case 1:
        training = false;
        break;

    // Train a sigmoid based NN using a training file and view error RMS
    // throughout the training.
    case 2:
        training = true;
                usingSigmoid = true;
        useGA = false;
        break;

    // Train a tanh based NN using a training file and view error RMS
    // throughout the training.
         case 3:
        training = true;
                usingSigmoid = false;
        useGA = false;
        break;

    // Use GA to determine optimal NN.
    case 4:
        training = true;
        useGA = true;
        break;

    // Invalid input.
    default:
        cout << "ERROR: Invalid selection." << endl;
        exit(1);
}

// If in user input mode, ask for output file name.
if (!training) {
    cout << "Enter output file name: ";
    getline(cin,filename);

    // Attempt to open output file.
    if (NULL == (outputFile = fopen(filename.c_str(),"wb"))) {
        cout << "ERROR: could not open output file." << endl;
        exit(1);
    }
}

// Otherwise, get input file name.
else {
    cout << "Enter input file name: ";
    getline(cin,filename);

    // Attempt to open input file.
    if (NULL == (inputFile = fopen(filename.c_str(),"rb"))) {
        cout << "ERROR: could not open input file." << endl;
        exit(1);
```

```
        }
    }

    // Determine the system layout of the NN by reading the header of the
input
    // file.
    if (ReadHeaderFromInput())
        exit(1);

    // Attempt to write header to output file if not training.
    if (!training && WriteHeaderToOutputFile())
        exit(1);

    // If the user chose to train the NN with a specific architecture.
      if (training && !useGA) {
            // Prompt user for the number of hidden layers.
            do {
                  cout << "Enter the number of hidden layers (0 - " <<
MAX_HIDDEN_LAYERS
                        << "): ";
                  cin >> numHidLayers;
            } while (numHidLayers < 0 || numHidLayers > MAX_HIDDEN_LAYERS);

            // Prompt user for the number of neurons in each hidden layer
            for (i = 0; i < numHidLayers; ) {
                  cout << "Number of hidden neurons in layer " << i << " (1
- "
                        << MAX_HID_NEURONS << "): ";
                  cin >> numHidNeuronsMinOne[i];

           // Decrement numHidNeuronsMinOne[i] since it should be one less
than
           // what the user requested.
           numHidNeuronsMinOne[i]--;

                  // If the number of hidden neurons is valid, then get the
number
                  // for the next layer, otherwise repeat the current layer.
                  if (numHidNeuronsMinOne[i] >= 0 && numHidNeuronsMinOne[i]
< MAX_HID_NEURONS)
                        i++;
            }

            // Prompt for learning rate.
            cout << "Learning rate (n): ";
            cin >> n;

            // Prompt for sigmoid parameters.
            if (usingSigmoid) {
                  // Prompt for sigmoid function slope
                  cout << "Sigmoid function slope (a): ";
                  cin >> a;

                  // Instantiate sigmoid based NN.
```

```
                    NN = new FullyConnectedNN(header.numInputs,
header.numOutputs, inputs,
                                                                n, a, b, true,
numHidLayers, numHidNeuronsMinOne);
              }

              // Otherwise, prompt for tanh parameters.
              else {
                    // Prompt for tanh amplitude
                    cout << "tanh amplitude (a): ";
                    cin >> a;

                    // Prompt for tanh slope
                    cout << "tanh slope (b): ";
                    cin >> b;

                    // Instantiate sigmoid based NN.
                    NN = new FullyConnectedNN(header.numInputs,
header.numOutputs, inputs,
                                                                n, a, b, false,
numHidLayers, numHidNeuronsMinOne);
              }
       }

       // Otherwise, if the user chose to run the GA to determine the optimal
       // neural network architecture.
       else if (training && useGA) {
         // Get POPULATION_SIZE from user.
         do {
             cout << "Enter size of population (" << MIN_POPULATION_SIZE
                 << " - " << MAX_POPULATION_SIZE << "): ";
             cin >> POPULATION_SIZE;
         } while (POPULATION_SIZE < MIN_POPULATION_SIZE ||
                  POPULATION_SIZE > MAX_POPULATION_SIZE);

         // Get scaling factors from user.
         do {
             cout << "Error RMS and number of connections are used to determine
fitness."
                  << endl << "What fraction of fitness should error RMS account
for (0-1): ";
             cin >> SCALING_FACTOR_ERR_RMS;
         } while (SCALING_FACTOR_ERR_RMS < 0 || SCALING_FACTOR_ERR_RMS > 1.0);
         SCALING_FACTOR_NUM_CONNECTIONS = 1.0 - SCALING_FACTOR_ERR_RMS;

         // Get crossover method.
              cout << "Use bit-level or gene-level crossover ('b' for bit): ";
              cin >> useBitXOverSelect;
         getline(cin,filename);
              useBitXOver = useBitXOverSelect == 'b' || useBitXOverSelect ==
'B';

         // Get MUTATION_RATE.
         do {
```

```
            cout << "Enter mutation rate (1 in X chance of a bit flipping): ";
            cin >> MUTATION_RATE;
        } while (MUTATION_RATE < 1);

        // First, initialize the GAPopulation.
        InitializeGAPopulation();

        // In order to determine which two individuals are selected from the
        // population for crossover, each individual is given a certain
percentage
        // probability of being selected. This percentage is higher for
        // individuals with higher fitness values. For example, if there are
        // only 3 individuals in the population, the highest fitness will have
        // 3 "slices" in the fitness spinning wheel. The 2nd highest fitness
individual
        // will have 2 "slices", and the least fit will have only 1 "slice".
        // Therefor there are 3+2+1 slices total, with the probability of
selection
        // being 3/6, 2/6, and 1/6 for the three individuals. This can be
applied
        // to any sized population and will be calculated in the following
loop.
        // When a random slice number is generated, the highest ranked
        // individual will be selected if 0-(POPULATION_SIZE-1) are generated.
        // Therefore highestProbSlice is set to POPULATION_SIZE-1.
        maxProbSlice = POPULATION_SIZE - 1;
        highestProbSlice[0] = maxProbSlice;
        for (i = 1; i < POPULATION_SIZE; i++) {
            // For each individual of rank x, add the reverse rank number of
slices.
            maxProbSlice += POPULATION_SIZE - i;

            // For any following individual i to be selected, the slice must
be within
            // POPULATION_SIZE-1 slices after individuals (i-1)'s
highestProbSlice.
            highestProbSlice[i] = maxProbSlice;

            // Example for a 3 individual population:
            // GAPopulation[0].highestProbSlice == 2 (selected for slices 0-2,
3/6 prob)
            // GAPopulation[1].highestProbSlice == 4 (selected for slices 3-4,
2/6 prob)
            // GAPopulation[2].highestProbSlice == 5 (selected for slice 5,
1/6 prob)
        }

        // The following loop will execute until the user decides the GA
        // should stop.
        do {
            // Display current ranks and prompt user for what to do.
            DisplayGAPopulation(false);
            cout << endl
                 << "1. Quit." << endl
```

```
                 << "2. Display population individual genes." << endl
                 << "3. Run GA for X generations." << endl
                 << "4. Select individual for training." << endl
                 << "5. Create CSV file of population data." << endl
                 << "Enter selection: ";
            cin >> selection;
            getline(cin,filename);

            switch (selection) {
                // Quit.
                case 1:
                    exit(0);

                // Display population individual genes.
                case 2:
                    DisplayGAPopulation(true);
                    break;

                // Run GA for X generations.
                case 3:
                    // Ask user how many generations of the GA should be run.
                    cout << endl << "Enter number of generations of the GA to
run: ";
                    cin >> numGenerations;

                    // Each iteration represents one generation of the GA.
                    for (i = 0; i < numGenerations; i++) {
                        // Select two individuals and store their indecies in
parent1
                        // and parent2. First, randomly generate one slice
number
                        // ranging from 0-maxProbSlice into parent1.
                        srand((unsigned)time(NULL));
                        parent1 = GetRandInt(maxProbSlice);

                        // Determine which individual is selected using the
parent1 slice.
                        for (j = 0; j < POPULATION_SIZE; j++) {
                            // If parent1 slice is lower than or the same as
the
                            // j'th individual's  highestProbSlice, then it is
the
                            // one to be selected. In this case, set parent1
to
                            // j, which is the j'th index.
                            if (parent1 <= highestProbSlice[j]) {
                                parent1 = j;
                                break;
                            }
                        }

                        // Generate parent2 and determine which individual is
selected using
```

```
                        // the parent2 slice. The following loop will continue
until parent2
                        // is different from parent1.
                        do {
                            parent2 = GetRandInt(maxProbSlice);
                            for (j = 0; j < POPULATION_SIZE; j++) {
                                // If parent2 slice is lower than or the same
as the j'th
                                // individual's highestProbSlice, then it is
the one to
                                // be selected. In this case, set parent2 to
j, which
                                // is the j'th index.
                                if (parent2 <= highestProbSlice[j]) {
                                    parent2 = j;
                                    break;
                                }
                            }
                        } while (parent1 == parent2);

                        // Now that the two parents were selected, cross them
over.
                        CrossOver(parent1, parent2, useBitXOver);

                        // Mutate the child with 1/MUTATION_RATE chance of
mutation
                        // for each bit.
                        Mutate(POPULATION_SIZE, MUTATION_RATE);

                        // Now, allocate NN based on the child.
                        AllocateNN(POPULATION_SIZE);

                        // Determine the number of connections in the NN.
                        SetNumConnections(POPULATION_SIZE);

                        // Train the child and determine errorRMS.
                        cout << "\r%0       complete for generation " << i+1
<< " of "
                             << numGenerations << "...";
                        TrainForXEpochs(TRAINING_CYCLES_IN_GENERATION, true);

                        // Get the error RMS of the child.
                        GAPopulation[POPULATION_SIZE].errorRMS =
CalcRMSError(false);

                        // Finally, deallocate the NN memory.
                        delete NN;

                        // Now, evaluate the entire population.
                        DetermineFitnessAndSort(false);
                    }

                    // Display new rank.
```

```
                    cout << "\r
\r";
                    break;

                // Select individual for training, nothing needs to be done
here.
                case 4:
                    break;

                // Create CSV file of population data.
                case 5:
                    // Get csv filename from user.
                    cout << "Enter csv file name: ";
                    getline(cin,filename);

                    // Attempt to open csv file.
                    if (NULL == (csvFile = fopen(filename.c_str(),"w"))) {
                        cout << "ERROR: could not open csv file." << endl;
                        break;
                    }

                    // Print column headers.
                    fprintf(csvFile, "ID,FITNESS,ERR_RMS_RANK,ERR_RMS,CON
RANK, CON,");
                    fprintf(csvFile, "NUM LAYERS,");
                    for (j = 0; j < MAX_HIDDEN_LAYERS; j++)
                        fprintf(csvFile, "L%d,", j);
                    fprintf(csvFile, "N,SLOPE,SEED,USING_SIGMOID\n");

                    // Print all data for each individual.
                    for (i = 0; i < POPULATION_SIZE; i++) {
                        fprintf(csvFile, "%d,", i);
                        fprintf(csvFile, "%lf,", GAPopulation[i].fitness);
                        fprintf(csvFile, "%d,",
                                POPULATION_SIZE -
GAPopulation[i].errorReverseRank + 1);
                        fprintf(csvFile, "%lf,", GAPopulation[i].errorRMS);
                        fprintf(csvFile, "%d,",
                                POPULATION_SIZE -
GAPopulation[i].connectionsReverseRank + 1);
                        fprintf(csvFile, "%d,",
GAPopulation[i].numConnections);
                        fprintf(csvFile, "%d,",
GAPopulation[i].numHidLayersOneLess + 1);
                        for (j = 0; j < MAX_HIDDEN_LAYERS; j++)
                            fprintf(csvFile, "%d,",
GAPopulation[i].numHidNeurons[j]+1);
                        fprintf(csvFile, "%d,", GAPopulation[i].n);
                        fprintf(csvFile, "%d,", GAPopulation[i].slope);
                        fprintf(csvFile, "%d,",
GAPopulation[i].randomNumSeed);
                        fprintf(csvFile, "%d\n",
GAPopulation[i].usingSigmoid);
                    }
```

```
                // Close file.
                if (csvFile != NULL)
                    fclose(csvFile);

                break;

            default:
                cout << "Invalid selection." << endl;
                break;
        }
    } while (selection != 4);

    // Prompt user for which NN to use.
    do {
        cout << "Enter ID of NN to use: ";
        cin >> i;
    } while (i < 0 || i >= POPULATION_SIZE);

    // Instantiate the NN that the user chose.
    AllocateNN(i);
    cout << "Neural Network created." << endl;
   }

// The following loop executes for all cases other than training.
// An iteration of the loop represents one sample, whether it is
// provided by the user or retrieved from an input file.
if (!training) {
    for (int sample = 0; sample < header.numTotalSamples; sample++) {
        // Read the next packet from the inputFile
        cout << endl;
        if (ReadOnePacket(true, true)) {
            exit(1);
        }

        // Prompt user for outputs.
        for (int i = 0; i < header.numOutputs; i++) {
            cout << "Output " << i << ": ";
            cin >> desiredOutputs[i];
        }

        // Attempt to write vectors to outputFile only in userInput mode.
        if (WriteOnePacket()) {
            exit(1);
        }
    }

    // Rewrite header to output file with correct number of samples.
    if (WriteHeaderToOutputFile())
        exit(1);
}

// Otherwise, the user chose to train a NN.
else {
```

```
        // The following loop will execute until the user decides to quit.
        while (true) {
            // Prompt the user for how many epochs of training to perform.
            cout << "\r" << "          " << endl
                 << "1. Quit." << endl
                 << "2. Show synaptic weights." << endl
                 << "3. Generate outputs for one set of inputs." << endl
                 << "4. Train NN and display error RMS." << endl
                 << "5. Create CSV file using all samples in training file."
<< endl
                 << "6. Save synaptic weights to weight file." << endl
                 << "7. Load synaptic weights from weight file." << endl
                 << "Enter selection: ";
            cin >> selection;
            getline(cin,filename);

            // Quit.
            if (selection == 1)
                break;

            // Show weights.
            else if (selection == 2) {
                NN->PrintWeights();
                continue;
            }

            // To generate one set of outputs, get the inputs from the user.
            else if (selection == 3) {
                // First, read the inputs from the user.
                cout << endl;
                ReadOnePacket(true, true);

                // Calculate the set of outputs.
                NN->CalcOutput();

                // Display all outputs.
                for (int i = 0; i < header.numOutputs; i++)
                    cout << "Output " << i << ": "
                         << setw(11) << NN->GetOutput(i) << endl;

                continue;
            }

            // To train NN, prompt user for the number of epochs of training.
            else if (selection == 4) {
                cout << "Enter number of epochs of training: ";
                cin >> selection;
            }

            // Generate a CSV file using all samples in training file.
            else if (selection == 5) {
                // Get csv filename from user.
                cout << "Enter csv file name: ";
                getline(cin,filename);
```

```
                // Attempt to open csv file.
                if (NULL == (csvFile = fopen(filename.c_str(),"w"))) {
                    cout << "ERROR: could not open csv file." << endl;
                    continue;
                }

                // Seek to the first evaluation sample in inputFile.
                fseek(inputFile, sizeof(TrainingFileHeader) +
                      header.numTrainingSamples * sizeof(double) *
                      (header.numInputs + header.numOutputs),
                      SEEK_SET);

                // Run through all of the evaluation samples in the input
file.
                for (int sample = header.numTrainingSamples;
                     sample < header.numTotalSamples; sample++) {
                    // Get the inputs and desired outputs (one sample).
                    if (ReadOnePacket(false,false))
                        break;

                    // Calculate NN outputs based on inputs.
                    NN->CalcOutput();

                    // Print the inputs to the file one at a time, seperated
by a comma.
                    for (int i = 0; i < header.numInputs; i++)
                        fprintf(csvFile, "%lf,", inputs[i]);

                    // Seperate the inputs and generated outputs by an empty
column.
                    fprintf(csvFile, ",");

                    // Print the outputs to the file one at a time, seperated
by a comma.
                    for (int i = 0; i < header.numOutputs; i++)
                        fprintf(csvFile, "%lf,", NN->GetOutput(i));

                    // Seperate the generated and desired outputs by an empty
column.
                    fprintf(csvFile, ",");

                    // Print the desired outputs to the file one at a time,
seperated
                    // by a comma.
                    for (int i = 0; i < header.numOutputs; i++)
                        fprintf(csvFile, "%lf,", desiredOutputs[i]);

                    // Complete the line.
                    fprintf(csvFile, "\n");
                }

                // Close file.
                if (csvFile != NULL)
```

```
                fclose(csvFile);

            cout << "File created, close excel to continue..." << endl;
            system(filename.c_str());
            continue;
        }

        // Save synaptic weights to a weight file.
        else if (selection == 6) {
            // Get weight filename from user.
            cout << "Enter weight file name: ";
            getline(cin,filename);

            // Attempt to open weight file.
            if (NULL == (weightFile = fopen(filename.c_str(),"wb"))) {
                cout << "ERROR: could not open weight file." << endl;
                continue;
            }

            // Attempt to save weights to weightFile.
            if (NN->SaveWeightsToFile(weightFile)) {
                cout << "ERROR: Failed to save weights to weight file." <<
endl;
            }

            // If the weights were successfully written, inform user.
            else {
                cout << "File created." << endl;
            }

            // Close file.
            if (weightFile != NULL)
                fclose(weightFile);

            continue;
        }

        // Load synaptic weights from a weight file.
        else if (selection == 7) {
            // Get weight filename from user.
            cout << "Enter weight file name: ";
            getline(cin,filename);

            // Attempt to open weight file.
            if (NULL == (weightFile = fopen(filename.c_str(),"rb"))) {
                cout << "ERROR: could not open weight file." << endl;
                continue;
            }

            // Attempt to load weights to weightFile.
            if (NN->LoadWeightsFromFile(weightFile)) {
                cout << "ERROR: Failed to load weights from weight file."
<< endl;
            }
```

```
                // If the weights were successfully written, inform user.
                else {
                    cout << "Weights loaded." << endl;
                }

                // Close file.
                if (weightFile != NULL)
                    fclose(weightFile);

                continue;
            }

            // Otherwise, the selection was invalid.
            else {
                cout << "Invalid selection" << endl;
                continue;
            }

            // Execute selection epochs of training if user selected option 4.
            TrainForXEpochs(selection, true);

            // Also, display the error RMS.
            CalcRMSError(true);
        }
    }

    // Cleanup.
    CleanUp();

    // Done.
    exit(0);
}

// Writes header packet to output file. Returns false on success, true
// on failure.
static bool WriteHeaderToOutputFile() {
    // If outputFile is NULL then failure.
    if (outputFile == NULL)
        return true;

    // Seek to beginning of output file.
    fseek(outputFile,0,SEEK_SET);

    // Write the structure into the file.
    if (fwrite(&header,1,sizeof(TrainingFileHeader),outputFile)
        != sizeof(TrainingFileHeader))
        return true;

    // Reset offset to end of file.
    fseek(outputFile,0,SEEK_END);

    // Succees.
    return false;
```

```
}

// Reads header packet from input file and sets system layout. Returns false
on
// success, true on failure.
static bool ReadHeaderFromInput() {
    // In user input mode, get all the arameters from user.
    if (!training) {
        cout << "Number of inputs: ";
        cin >> header.numInputs;
        cout << "Number of outputs: ";
        cin >> header.numOutputs;
        cout << "Number of training samples: ";
        cin >> header.numTrainingSamples;
        cout << "Total number of samples: ";
        cin >> header.numTotalSamples;
    }

    // Otherwise, get it from the input file.
    else {
        // Otherwise, if inputFile or header are NULL then failure.
        if (inputFile == NULL) {
            cout << "ERROR: Input file not supplied." << endl;
            return true;
        }

        // Attempt to read the file header, return true on failure.
        if (fread(&header,1,sizeof(TrainingFileHeader),inputFile)
            != sizeof(TrainingFileHeader)) {
            cout << "ERROR: File header read failed." << endl;
            return true;
        }
    }

    // Sanity check on all members of the file header.
    if (header.numInputs <= 0 || header.numOutputs <= 0 ||
        header.numTrainingSamples <= 0 || header.numTotalSamples <= 0 ||
        header.numTotalSamples < header.numTrainingSamples) {
        cout << "ERROR: NN layout contains non-positive parameters." << endl;
        return true;
    }

    // Success.
    return false;
}

// Writes one packet to outputFile which contains the input and output
// vectors as double arrays.
static bool WriteOnePacket() {
    // If outputFile is NULL then failure.
    if (outputFile == NULL) {
        cout << "File has not yet been opened, packet could not be written."
<< endl;
        return true;
```

```
    }

    // Write the inputs array into the file.
    if (fwrite(inputs,sizeof(double),header.numInputs,outputFile)
        != header.numInputs) {
        cout << "ERROR: failed to write input vector to output file." << endl;
        return true;
    }

    // Write the output array into the file.
    if (fwrite(desiredOutputs,sizeof(double),header.numOutputs,outputFile)
        != header.numOutputs) {
        cout << "ERROR: failed to write output vector to output file." <<
endl;
        return true;
    }

    // Succees.
    return false;
}

// Reads one packet from inputFile which contains the input and desired
// output vectors as double arrays. print dictates whether or not to display
// the inputs to the screen when inputting from a file.
static bool ReadOnePacket(bool print, bool fromUser) {
    // Get inputs from user if fromUser is true.
    if (fromUser) {
        // Otherwise, get all inputs from user.
        for (int i = 0; i < header.numInputs; i++) {
            cout << "Input " << i << ": ";
            cin >> inputs[i];
        }
    }

    // Otherwise, get a packet from the input file.
    else {
        // If inputFile or header are NULL then failure.
        if (inputFile == NULL) {
            cout << "ERROR: Input file not supplied before trying to read one
packet."
                 << endl;
            return true;
        }

        // Otherwise, attempt to read in the input vector.
        // Write the inputs array into the file.
        if (fread(inputs,sizeof(double),header.numInputs,inputFile)
            != header.numInputs) {
            // Display error message if not eof.
            if (!feof(inputFile))
                cout << "ERROR: failed to read input vector from input file."
<< endl;

            // Failure.
```

```
            return true;
        }

        // Write the output array into the file.
        if (fread(desiredOutputs,sizeof(double),header.numOutputs,inputFile)
            != header.numOutputs) {
            // Display error message if not eof.
            if (!feof(inputFile))
                cout << "ERROR: failed to read output vector from input file."
<< endl;

            // Failure.
            return true;
        }

        // Display inputs that were retrieved from input file.
        for (int i = 0; print && i < header.numInputs; i++) {
            cout << "Input " << i << ": " << inputs[i] << endl;
        }
    }

    // Success.
    return false;
}

// Seeks to the next randomly selected training sample in inputFile.
// Returns true on failure, false on success.
static bool SeekToNextRandPacket() {
    int sampleNum; // randomly selected sample number.
    long seekTo; // stores the location of sample sampleNum in inputFile.

    // If inputFile or header are NULL then failure.
    if (inputFile == NULL) {
        cout << "ERROR: Input file not supplied before trying to seek to next"
             << "random packet." << endl;
        return true;
    }

    // Randomly select one sample of the header.numSamples.
    sampleNum = GetRandInt(header.numTrainingSamples);

    // Calculate seekTo, which is found by first seeking to the end of the
FileHeader
    // in inputFile. Sample number sampleNum packets away, and each packet has
size
    // (in bytes): sizeof(double) * (header.numInputs + header.numOutputs)
    seekTo = sizeof(TrainingFileHeader) +
             sampleNum * sizeof(double) * (header.numInputs +
header.numOutputs);

    // Seek to the sample.
    fseek(inputFile, seekTo, SEEK_SET);

    // Success.
```

```
    return false;
}

// Deallocates all allocated memory and closese files.
static void CleanUp() {
    // Deallocate NN memory.
    if (NN != NULL)
        delete NN;

    // Close outputFile if it was opened.
    if (outputFile != NULL)
        fclose(outputFile);

    // Close inputFile if it was opened.
    if (inputFile != NULL)
        fclose(inputFile);

    // Close csvFile if it was opened.
    if (csvFile != NULL)
        fclose(csvFile);
}

// Returns a random integer between 0 and maxInt.
static int GetRandInt(int maxInt) {
    return (int)((double)rand() / (RAND_MAX + 1) * (maxInt + 1));
}

// Calculates the RMS error for NN and displays the results if display is
true.
static double CalcRMSError(bool print) {
    double errorRMS = 0;

    // Calculate the RMS error using all of the samples in the input file.
    // Seek to the first evaluation sample in inputFile and reset errorRMS.
    fseek(inputFile, sizeof(TrainingFileHeader) +
          header.numTrainingSamples * sizeof(double) * (header.numInputs +
header.numOutputs),
          SEEK_SET);
    cout << "\r";

    // Run through all of the samples in the input file.
    for (int sample = header.numTrainingSamples; sample <
header.numTotalSamples; sample++) {
        // Get the inputs and desired outputs (one sample).
        if (ReadOnePacket(print,false))
            break;

        // Calculate NN outputs based on inputs.
        NN->CalcOutput();

        // Get all outputs.
        for (int i = 0; i < header.numOutputs; i++) {
            actualOutputs[i] = NN->GetOutput(i);
```

```
            // Display output.
            if (print) {
                cout << "Output " << i << ": ";
                cout << setw(11) << actualOutputs[i];
                cout << " Error " << i << ": " << desiredOutputs[i]-
actualOutputs[i];
                cout << endl;
            }

            // Error RMS sum is incremented by x^2 where x is the error
            // of the current sample.
            errorRMS += (desiredOutputs[i]-actualOutputs[i]) *
                        (desiredOutputs[i]-actualOutputs[i]);
        }

    }

    // Print the error RMS.
    if (print)
        cout << "RMS of error: " << errorRMS / 2 / header.numTotalSamples <<
endl;

    // Return the RMS error.
    return (errorRMS / 2 / header.numTotalSamples);
}

// Trains NN for numEpochs
static void TrainForXEpochs(int numEpochs, bool print) {
    for (int i = 0; i < numEpochs; i++) {
        // Display % complete.
        if (print)
            cout << "\r%" << (double)i/(double)numEpochs*100 << "   ";

        // Train the NN as many times as there are training samples in the
input file.
        for (int sample = 0; sample < header.numTrainingSamples; sample++) {
            // Seek to a random training sample in inputFile.
            if (SeekToNextRandPacket())
                break;

            // Get the inputs and desired outputs (one sample).
            if (ReadOnePacket(false,false))
                break;

            // Calculate NN outputs based on inputs.
            NN->CalcOutput();

            // Train using current sample.
            NN->BackPropLearn(desiredOutputs);
        }
    }

    // Clear the current line.
    cout << "\r               \r";
```

```
}

// Generates an XOR training file containing 4 training samples and
// 4 evaluation samples.
static int GenerateXORTrainingFile() {
    // Generate a 100-point sin wave.
    if (NULL == (outputFile = fopen("xor.bin","wb"))) {
        cout << "ERROR: could not open output file." << endl;
        return 1;
    }

    // Write header to output file.
    header.numInputs = 2;
    header.numOutputs = 1;
    header.numTrainingSamples = 4;
    header.numTotalSamples = 8;
    if (WriteHeaderToOutputFile()) {
        fclose(outputFile);
        return 1;
    }

    // Write the 4 XOR samples into the file twice.
    for (int i = 0; i < 2; i++) {
        // 0 ^ 0 = 0
        inputs[0] = 0;
        inputs[1] = 0;
        desiredOutputs[0] = 0;
        if (WriteOnePacket()) {
            fclose(outputFile);
            return 1;
        }

        // 0 ^ 1 = 1
        inputs[0] = 0;
        inputs[1] = 1;
        desiredOutputs[0] = 1;
        if (WriteOnePacket()) {
            fclose(outputFile);
            return 1;
        }

        // 1 ^ 0 = 1
        inputs[0] = 1;
        inputs[1] = 0;
        desiredOutputs[0] = 1;
        if (WriteOnePacket()) {
            fclose(outputFile);
            return 1;
        }

        // 1 ^ 1 = 0
        inputs[0] = 1;
        inputs[1] = 1;
        desiredOutputs[0] = 0;
```

```
        if (WriteOnePacket()) {
            fclose(outputFile);
            return 1;
        }
    }

    // Close file on success.
    fclose(outputFile);
    return 0;
}

// Generates a 130 sample training file containing one cycle of a sin wave.
// 100 samples are training samples and 30 samples are for evaluation.
static int GenerateSinTrainingFile() {
    // Generate a 100-point sin wave.
    if (NULL == (outputFile = fopen("sin.bin","wb"))) {
        cout << "ERROR: could not open output file." << endl;
        return 1;
    }

    // Write header to output file.
    header.numInputs = 1;
    header.numOutputs = 1;
    header.numTrainingSamples = 100;
    header.numTotalSamples = 131;
    if (WriteHeaderToOutputFile()) {
        fclose(outputFile);
        return 1;
    }

    // Write 100 training samples.
    for (int i = 0; i < 100; i++) {
        inputs[0] = i*3.1415926538/50;
        desiredOutputs[0] = sin(inputs[0]);
        if (WriteOnePacket()) {
            fclose(outputFile);
            return 1;
        }
    }

    // Write 30 evaluation samples.
    for (int i = 0; i < 31; i++) {
        inputs[0] = i*3.1415926538/15;
        desiredOutputs[0] = sin(inputs[0]);
        if (WriteOnePacket()) {
            fclose(outputFile);
            return 1;
        }
    }

    // Close file on success.
    fclose(outputFile);
    return 0;
}
```

```
// Initializes GAPopulation by randomly creating all the individuals and
ranking
// them based on error RMS and number of connections.
static void InitializeGAPopulation() {
    // First, the population of POPULATION_SIZE individuals must be
    // randomly created. The random number generator is seeded based on the
time.
    for (int i = 0; i < POPULATION_SIZE; i++) {
        GAPopulation[i].numHidLayersOneLess =
GetRandInt(x1s(NUM_BITS_numHidLayers));
        GAPopulation[i].n = GetRandInt(x1s(NUM_BITS_n));
        GAPopulation[i].slope = GetRandInt(x1s(NUM_BITS_slope));
        GAPopulation[i].randomNumSeed =
GetRandInt(x1s(NUM_BITS_randomNumSeed));
        GAPopulation[i].usingSigmoid = GetRandInt(1);

        // Generate a random number of neurons in each of the layers.
        for (int j = 0; j < MAX_HIDDEN_LAYERS; j++) {
            GAPopulation[i].numHidNeurons[j] =
GetRandInt(x1s(NUM_BITS_numHidNeurons));
        }

        // Set numConnections.
        SetNumConnections(i);
    }

    // Now, the error RMS is found for each of the individuals in the
population.
    cout << "Training initial population..." << endl;
    for (i = 0; i < POPULATION_SIZE; i++) {
        // Allocate individual i.
        AllocateNN(i);

        // Now, train NN for TRAINING_CYCLES_IN_GENERATION epochs.
        cout << "\r%0       training individual " << i+1 << " of " <<
POPULATION_SIZE
             << "...";
        TrainForXEpochs(TRAINING_CYCLES_IN_GENERATION, true);

        // Get the error RMS of the individual.
        GAPopulation[i].errorRMS = CalcRMSError(false);

        // Finally, deallocate the NN memory.
        delete NN;
    }

    // Now, GAPopulation must be sorted by fitness level.
    cout << "\rTraining complete.                                    " << endl;
    DetermineFitnessAndSort(true);
}

// Determines the errorReverseRank, connectionsReverseRank, and rank for all
individuals
```

```
// in the GAPopulation. Finally, GAPopulation is sorted by fitness rank.
static void DetermineFitnessAndSort(bool firstTime) {
    int i, j;
    GAIndividual temp;

    // The last element of GAPopulation stores the child for of the current
generation.
    // If this is not the firstTime, then the child is included in the
errorRMS
    // bubble sort.
    int includeChild = firstTime ? 0 : 1;

    // GAPopulation must be sorted three times using the bubble sort
algorithm.
    // GAPopulation is first sorted by errorRMS to determine errorReverseRank
    // with the lowest errorRMS at index 0. If this is not the firstTime, then
    // the child will be included in the sort.
    for (i = (POPULATION_SIZE-1+includeChild); i >= 0; i--) {
        for (j = 1; j <= i; j++) {
            if (GAPopulation[j-1].errorRMS > GAPopulation[j].errorRMS) {
                temp = GAPopulation[j-1];
                GAPopulation[j-1] = GAPopulation[j];
                GAPopulation[j] = temp;
            }
        }
    }

    // errorReverseRank is set for the first POPULATION_SIZE individuals.
    // The lowest ranked errorRMS individual is kicked out of the population.
    for (i = 0; i < POPULATION_SIZE; i++) {
        GAPopulation[i].errorReverseRank = POPULATION_SIZE - i;
    }

    // GAPopulation is then sorted by numConnections to determine
connectionsReverseRank
    // with the lowest numConnections at index 0. When comparing two
individuals that
    // have the same number of connections, errorRMS is used as the deciding
factor.
    for (i = POPULATION_SIZE-1; i >= 0; i--) {
        for (j = 1; j <= i; j++) {
            if (GAPopulation[j-1].numConnections >
GAPopulation[j].numConnections ||
                (GAPopulation[j-1].numConnections ==
GAPopulation[j].numConnections &&
                 GAPopulation[j-1].errorReverseRank <
GAPopulation[j].errorReverseRank)) {
                temp = GAPopulation[j-1];
                GAPopulation[j-1] = GAPopulation[j];
                GAPopulation[j] = temp;
            }
        }
    }
```

```
    // connectionsReverseRank is set for all individuals that were sorted.
    // Also, fitness is calculated because both connectionsReverseRank and
errorReverseRank
    // are known.
    for (i = 0; i < POPULATION_SIZE; i++) {
        GAPopulation[i].connectionsReverseRank = POPULATION_SIZE - i;
        GAPopulation[i].fitness = GAPopulation[i].errorReverseRank *
SCALING_FACTOR_ERR_RMS +
            GAPopulation[i].connectionsReverseRank *
SCALING_FACTOR_NUM_CONNECTIONS;
    }

    // Finally, GAPopulation is sorted by overall fitness level with the
highest fitness
    // ending up at index 0.
    for (i = POPULATION_SIZE-1; i >= 0; i--) {
        for (j = 1; j <= i; j++) {
            if (GAPopulation[j-1].fitness < GAPopulation[j].fitness) {
                temp = GAPopulation[j-1];
                GAPopulation[j-1] = GAPopulation[j];
                GAPopulation[j] = temp;
            }
        }
    }
}

// Performs a crossover between the two selected parents from GAPopulation.
// The crossover can either be on the gene level or bit level.
static void CrossOver(int parent1, int parent2, bool useBitXOver) {
    int i, j;
    int chosenParent; // The index of the chosen parent for a specific bit
    int bitMask; // Used to get and set specific bits

    // Sanity check on the indecies.
    if (parent1 < 0 || parent1 > POPULATION_SIZE-1 ||
        parent2 < 0 || parent2 > POPULATION_SIZE-1)
    {
        cout << "ERROR: parent indecies are invalid." << endl;
        exit(1);
    }

    // If the user chose a bit-level crossover.
    if (useBitXOver) {
        // The new child will be created in GAPopulation[POPULATION_SIZE].
        // Each bit of each gene has a 50/50 chance of being selected from
eith parent1
        // or parent2. Each of the following loops represent the crossover of
        // all of the bits in each of the genes. bitMask will be used to get
        // the bit from the chosenParent and set the child's bit.
        // The first gene to be crossed over is numHidLayersOneLess.
        bitMask = 0x1;
        GAPopulation[POPULATION_SIZE].numHidLayersOneLess = 0;
        for (i = 0; i < NUM_BITS_numHidLayers; i++) {
            chosenParent = GetRandInt(1) ? parent1 : parent2;
```

```
            GAPopulation[POPULATION_SIZE].numHidLayersOneLess |=
                bitMask & GAPopulation[chosenParent].numHidLayersOneLess;
            bitMask = bitMask << 1;
        }

        // Next, n is crossed over.
        bitMask = 0x1;
        GAPopulation[POPULATION_SIZE].n = 0;
        for (i = 0; i < NUM_BITS_n; i++) {
            chosenParent = GetRandInt(1) ? parent1 : parent2;
            GAPopulation[POPULATION_SIZE].n |=
                bitMask & GAPopulation[chosenParent].n;
            bitMask = bitMask << 1;
        }

        // Next, slope is crossed over.
        bitMask = 0x1;
        GAPopulation[POPULATION_SIZE].slope = 0;
        for (i = 0; i < NUM_BITS_slope; i++) {
            chosenParent = GetRandInt(1) ? parent1 : parent2;
            GAPopulation[POPULATION_SIZE].slope |=
                bitMask & GAPopulation[chosenParent].slope;
            bitMask = bitMask << 1;
        }

        // Next, randomNumSeed is crossed over.
        bitMask = 0x1;
        GAPopulation[POPULATION_SIZE].randomNumSeed = 0;
        for (i = 0; i < NUM_BITS_randomNumSeed; i++) {
            chosenParent = GetRandInt(1) ? parent1 : parent2;
            GAPopulation[POPULATION_SIZE].randomNumSeed |=
                bitMask & GAPopulation[chosenParent].randomNumSeed;
            bitMask = bitMask << 1;
        }

        // Finally, each element in numHidNeurons is crossed over.
        for (j = 0; j < MAX_HIDDEN_LAYERS; j++) {
            bitMask = 0x1;
            GAPopulation[POPULATION_SIZE].numHidNeurons[j] = 0;
            for (i = 0; i < NUM_BITS_numHidNeurons; i++) {
                chosenParent = GetRandInt(1) ? parent1 : parent2;
                GAPopulation[POPULATION_SIZE].numHidNeurons[j] |=
                    bitMask & GAPopulation[chosenParent].numHidNeurons[j];
                bitMask = bitMask << 1;
            }
        }
    }

    // Otherwise, perform a gene-level crossover.
    else {
        // First, cross over numHidLayersOneLess
        chosenParent = GetRandInt(1) ? parent1 : parent2;
        GAPopulation[POPULATION_SIZE].numHidLayersOneLess =
            GAPopulation[chosenParent].numHidLayersOneLess;
```

```
        // Next, n is crossed over.
        chosenParent = GetRandInt(1) ? parent1 : parent2;
        GAPopulation[POPULATION_SIZE].n = GAPopulation[chosenParent].n;

        // Next, slope is crossed over.
        chosenParent = GetRandInt(1) ? parent1 : parent2;
        GAPopulation[POPULATION_SIZE].slope =
GAPopulation[chosenParent].slope;

        // Next, randomNumSeed is crossed over.
        chosenParent = GetRandInt(1) ? parent1 : parent2;
        GAPopulation[POPULATION_SIZE].randomNumSeed =
            GAPopulation[chosenParent].randomNumSeed;

        // Finally, each element in numHidNeurons is crossed over.
        for (j = 0; j < MAX_HIDDEN_LAYERS; j++) {
            chosenParent = GetRandInt(1) ? parent1 : parent2;
            GAPopulation[POPULATION_SIZE].numHidNeurons[j] =
                GAPopulation[chosenParent].numHidNeurons[j];
        }
    }

    // Since usingSigmoid is only 1 bit, it is corossed over the same
regardless of
    // useBitXOver.
    GAPopulation[POPULATION_SIZE].usingSigmoid = 0;
    chosenParent = GetRandInt(1) ? parent1 : parent2;
    GAPopulation[POPULATION_SIZE].usingSigmoid =
        GAPopulation[chosenParent].usingSigmoid;
}

// Allocate NN based on index into GAPopulation.
static void AllocateNN(int i) {
    // Sanity check on i.
    if (i < 0 || i > POPULATION_SIZE) {
        cout << "ERROR: invalid index into GAPopulation in AllocateNN()." <<
endl;
        exit(1);
    }

    // First seed the random number generator.
    srand(GAPopulation[i].randomNumSeed);

    // Allocate NN depending on the type of activation function. First is
    // the sigmoid activation function.
    if (GAPopulation[i].usingSigmoid) {
        NN = new FullyConnectedNN(header.numInputs, header.numOutputs, inputs,
                                  decodeN[GAPopulation[i].n],
                                  decodeSlope[GAPopulation[i].slope], 0.0,
true,
                                  GAPopulation[i].numHidLayersOneLess+1,
                                  GAPopulation[i].numHidNeurons);
    }
```

```
    // Otherwise, using tanh.
    else {
        NN = new FullyConnectedNN(header.numInputs, header.numOutputs, inputs,
                                  decodeN[GAPopulation[i].n],
                                  1.0, decodeSlope[GAPopulation[i].slope],
false,
                                  GAPopulation[i].numHidLayersOneLess+1,
                                  GAPopulation[i].numHidNeurons);
    }
}

// Determine and set numConnections for individual i in GAPopulation.
static void SetNumConnections(int i) {
    // Sanity check on i.
    if (i < 0 || i > POPULATION_SIZE) {
        cout << "ERROR: invalid index into GAPopulation in
SetNumConnections()." << endl;
        exit(1);
    }

    // Reset numConnections to 0.
    GAPopulation[i].numConnections = 0;

    // The following loop increments numConnections for each layer.
    for (int j = 0; j <= GAPopulation[i].numHidLayersOneLess; j++) {
        // numConnections is incremented by the number of neurons at the
current
        // layer multiplied by the number of neurons in the previous layer +
        // one bias for each of the current neurons.
        // If this is the first hidden layer, then use header.numInputs for
the
        // previous layer.
        if (j == 0) {
            GAPopulation[i].numConnections +=
(GAPopulation[i].numHidNeurons[0]+1) +
                header.numInputs * (GAPopulation[i].numHidNeurons[0]+1);
        }

        // Otherwise, this is not the first hidden layer.
        else {
            GAPopulation[i].numConnections +=
(GAPopulation[i].numHidNeurons[j]+1) +
                (GAPopulation[i].numHidNeurons[j-1]+1) *
(GAPopulation[i].numHidNeurons[j]+1);
        }
    }

    // Increment by the number of connections between the output
    // neurons and the last hidden layer neurons.
    GAPopulation[i].numConnections += header.numOutputs + header.numOutputs *

(GAPopulation[i].numHidNeurons[GAPopulation[i].numHidLayersOneLess]+1);
}
```

```
// Mutates one individual with a 1/oneInX chance of mutation for each bit.
static void Mutate(int idx, int oneInX) {
    int i, j;
    int bitMask; // Used to mutate bits

    // Sanity check on i.
    if (idx < 0 || idx > POPULATION_SIZE) {
        cout << "ERROR: invalid index into GAPopulation in Mutate()." << endl;
        exit(1);
    }

    // Mutate numHidLayers.
    bitMask = 0x1;
    for (i = 0; i < NUM_BITS_numHidLayers; i++) {
        // There is a 1/oneInX chace of the bit being flipped.
        GAPopulation[idx].numHidLayersOneLess ^=
            bitMask & (GetRandInt(oneInX-1) == 0 ? 0xffffffff : 0);
        bitMask = bitMask << 1;
    }

    // Mutate n.
    bitMask = 0x1;
    for (i = 0; i < NUM_BITS_n; i++) {
        // There is a 1/oneInX chace of the bit being flipped.
        GAPopulation[idx].n ^=
            bitMask & (GetRandInt(oneInX-1) == 0 ? 0xffffffff : 0);
        bitMask = bitMask << 1;
    }

    // Mutate slope.
    bitMask = 0x1;
    for (i = 0; i < NUM_BITS_slope; i++) {
        // There is a 1/oneInX chace of the bit being flipped.
        GAPopulation[idx].slope ^=
            bitMask & (GetRandInt(oneInX-1) == 0 ? 0xffffffff : 0);
        bitMask = bitMask << 1;
    }

    // Mutate randomNumSeed.
    bitMask = 0x1;
    for (i = 0; i < NUM_BITS_randomNumSeed; i++) {
        // There is a 1/oneInX chace of the bit being flipped.
        GAPopulation[idx].randomNumSeed ^=
            bitMask & (GetRandInt(oneInX-1) == 0 ? 0xffffffff : 0);
        bitMask = bitMask << 1;
    }

    // Mutate usingSigmoid.
    GAPopulation[idx].usingSigmoid ^= (GetRandInt(oneInX-1) == 0 ? 1 : 0);

    // Mutate each element in numHidNeurons.
    for (j = 0; j < MAX_HIDDEN_LAYERS; j++) {
        bitMask = 0x1;
```

```
        for (i = 0; i < NUM_BITS_numHidNeurons; i++) {
            // There is a 1/oneInX chace of the bit being flipped.
            GAPopulation[idx].numHidNeurons[i] ^=
                bitMask & (GetRandInt(oneInX-1) == 0 ? 0xffffffff : 0);
            bitMask = bitMask << 1;
        }
    }
}

// Display either detailed or overview description of GAPopulation.
static void DisplayGAPopulation(bool detailed) {
    int i, j;

    // Display all genes for every individual in population.
    if (detailed) {
        // Display header.
        cout << endl
             << " ID | NUM LAYERS | ";
        for (j = 0; j < MAX_HIDDEN_LAYERS; j++)
            cout << "L" << j << " | ";

        cout << "N | SLOPE | SEED | USING_SIGMOID" << endl
             << "-------------------"
             << "-----------------------------";
        for (j = 0; j < MAX_HIDDEN_LAYERS; j++)
            cout << "------";
        cout << endl;

        // Display genes.
        for (i = 0; i < POPULATION_SIZE; i++) {
            cout << " "
                << setw(2) << i << "   "
                << setw(10) << GAPopulation[i].numHidLayersOneLess + 1 << "
";

            for (j = 0; j < MAX_HIDDEN_LAYERS; j++)
                cout << setw(2) << GAPopulation[i].numHidNeurons[j] + 1 << "
";

            cout << GAPopulation[i].n << "   "
                 << setw(5) << GAPopulation[i].slope << "   "
                 << setw(4) << GAPopulation[i].randomNumSeed << "   "
                 << (int)GAPopulation[i].usingSigmoid << endl;
        }
    }

    // Otherwise, display only fitness and rank.
    else {
        // Display header.
        cout << endl
            << " ID |  FITNESS  | ERR_RMS RANK |    ERR_RMS    "
            << "| CON RANK | CON" << endl
            << "-----------------------------------------------"
            << "----------------" << endl;
```

```
        // Display genes.
        for (i = 0; i < POPULATION_SIZE; i++) {
            cout << " "
                << setw(2) << i << "   "
                << setw(9) << GAPopulation[i].fitness << "   " << setw(12)
                << POPULATION_SIZE + 1 - GAPopulation[i].errorReverseRank
                << "   "
                << setw(13) << GAPopulation[i].errorRMS << "   " << setw(8)
                << POPULATION_SIZE + 1 -
GAPopulation[i].connectionsReverseRank
                << "   " << GAPopulation[i].numConnections << endl;
        }
    }
}

// Generates an input file based on file "DCController.csv" which has on each
line
// the three inputs and one output of the DC Power controller NN.
DCController.csv
// is expected to have 2002 samples with 1001 for training and 1001 for
evaluation.
// The program exits at the end of this function call.
static void GenerateDCControllerInputFile() {
    FILE *csvInput;
    int retval;

    // Attempt to open DCController.csv as the input file.
    if (NULL == (csvInput = fopen("DCController.csv","r"))) {
        cout << "ERROR: could not open input file." << endl;
        exit(1);
    }

    // Attempt to open DCController.bin as the training file.
    if (NULL == (outputFile = fopen("DCController.bin","wb"))) {
        cout << "ERROR: could not open output file." << endl;
        exit(1);
    }

    // Set all DCheader members and write the header to the output file.
    header.numInputs = 3;
    header.numOutputs = 1;
    header.numTotalSamples = 2002;
    header.numTrainingSamples = 1001;
    WriteHeaderToOutputFile();

    // Each iteration of the following loop will input one training sample
    // from csvInput and write it to training Output. It is assumed that
    // DCController.csv contains 2002 lines.
    for (int i = 0; i < 2002; i++) {
        retval = fscanf(csvInput, "%lf, %lf, %lf, %lf\n", &inputs[0],
                        &inputs[1], &inputs[2], &desiredOutputs[0]);
        WriteOnePacket();
    }
```

```
    // Close both files.
    fclose(csvInput);
    fclose(outputFile);

    // Report success and exit.
    cout << "DCController.bin was successfully created." << endl;
    exit(0);
}
```

F.3. FullyConnectedNN.h

```
/*
Ariel Kopel
Thesis Project
Advisor: Dr. Helen Yu
Cal Poly State University - San Luis Obispo

FullyConnectedNN.h - Defines the FullyConnectedNN class which can have an
arbitraty number of layers and neurons at each layer.
*/

#ifndef FULLYCONNECTEDNN_H
#define FULLYCONNECTEDNN_H

#include "NNSimulation.h"
#include "Neuron.h"

class FullyConnectedNN {
public:
    // Constructor
    FullyConnectedNN(int numInputs, int numOutputs, double *inputs,
                                    double n, double a, double b, bool
usingSigmoid,
                                    int numHidLayers, int *numHidNeuronsMinOne);

    // Destructor
    ~FullyConnectedNN();

    // Calculates all outputs of the NN.
    void CalcOutput();

    // Return output x of the NN.
    double GetOutput(int x);

    // Prints all of the synaptic weights of every neuron.
    void PrintWeights();

    // Executes back propagation learning for the current sample.
    void BackPropLearn(double *desiredOutputs);

    // Stores all synaptic weights in outFile. Returns true on failure,
    // false on success.
    bool SaveWeightsToFile(FILE *outFile);

    // Loads synaptic weight file. Returns true on failure, false on success.
    bool LoadWeightsFromFile(FILE *fromFile);

private:
    // Matrix of all hidden neurons, with maximum of MAX_HID_NEURONS in each
    // of the MAX_HIDDEN_LAYERS layers.
    Neuron *hidNeurons[MAX_HIDDEN_LAYERS][MAX_HID_NEURONS];
```

```
    // Array of output neurons in NN
    Neuron *outNeurons[MAX_OUTPUTS];

    // Number of hidden neurons in each layer.
    int numHidNeurons[MAX_HIDDEN_LAYERS];

    // Number of inputs.
    int numInputs;

    // Number of outputs.
    int numOutputs;

    // Number of hidden layers
    int numHidLayers;
};

#endif
```

F.4. FullyConnectedNN.cpp

```
/*
Ariel Kopel
Thesis Project
Advisor: Dr. Helen Yu
Cal Poly State University - San Luis Obispo

FullyConnectedNN.cpp - Implements functions of the FullyConnectedNN class,
as defined in FullyConnectedNN.h.
*/

#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include "FullyConnectedNN.h"
#include "Neuron.h"
using namespace std;

// Constructor
// IMPORTANT: the pointer numHidNeuronsMinOne passed into the constructor is
// expected to contain integers which are 1 less than the number of hidden
// neurons (0 means 1 hidden neuron).
FullyConnectedNN::FullyConnectedNN(int numInputs, int numOutputs, double
*inputs,
                                                        double n, double a,
double b, bool usingSigmoid,
                                                        int numHidLayers, int
*numHidNeuronsMinOne) {
    int i, j, k;
    double weight;
    string filename;

    // Exit if the number of inputs or outputs exceeds the max allowed.
    if (numOutputs > MAX_OUTPUTS ||
        numInputs > MAX_INPUTS) {
        cout << "Max number of inputs or output neurons surpassed." << endl;
        exit(1);
    }

    // Exit if the number of inputs or outputs is non-positive.
    if (numOutputs <= 0 ||
        numInputs <= 0) {
        cout << "Number of inputs and outputs must be positive." << endl;
        exit(1);
    }

    // Set number of inputs and outputs.
    this->numInputs = numInputs;
    this->numOutputs = numOutputs;
```

```
      // Exit if the number of hidden layers is invalid.
      if (numHidLayers < 0 || numHidLayers > MAX_HIDDEN_LAYERS) {
       cout << "Number of hidden layers must be between  0 and "
                    << MAX_HIDDEN_LAYERS << "." << endl;
       exit(1);
      }

      // Otherwise, set numHidLayers.
      this->numHidLayers = numHidLayers;

      // Check the number of neurons in each hidden layer
      for (i = 0; i < numHidLayers; i++) {
            // If the number of hidden neurons is valid, then exit.
       // IMPORTANT: the pointer numHidNeuronsMinOne passed into the
constructor is
       // expected to contain integers which are 1 less than the number of
hidden
       // neurons (0 means 1 hidden neuron).
            if (numHidNeuronsMinOne[i] < 0 || numHidNeuronsMinOne[i] >=
MAX_HID_NEURONS) {
                  cout << "Number of hidden neurons per layer must be
between 1 and "
                         << MAX_HID_NEURONS << "." << endl;
                  exit(1);
            }

            // Otherwise, set numHidNeurons to one more than the
numHidNeuronsMinOne.
            this->numHidNeurons[i] = numHidNeuronsMinOne[i]+1;
      }

    // If there are no hidden layers, then the inputs will be connected to the
    // outputNeurons.
    if (numHidLayers == 0) {
        // Allocate output Neurons and connect them to the inputs.
        // In the following loop:
        // i - current output neuron
        // j - current input
        for (i = 0; i < numOutputs; i++) {
            // Allocate memory for a sigmoid based neuron.
                  if (usingSigmoid)
                        outNeurons[i] = new Neuron(n, a);

                  // Otherwise, allocate a tanh based neuron.
                  else
                        outNeurons[i] = new Neuron(n, a, b);

            // Connect each input to the current neuron.
            for (j = 0; j < numInputs; j++) {
                // Randomly assign weight for the current synapse somwhere
between
                // -1 and 1.
                weight = (double)rand() / (RAND_MAX + 1) * 2 - 1;
```

```
                // Connect input j to output neuron i.
                if (outNeurons[i]->AddSynapse(NULL,weight,n,inputs+j)) {
                    cout << "ERROR: Too many Synapses added to Neuron" <<
endl;
                    exit(1);
                }
            }
        }

        // No more neurons need to be allocated since the inputs have been
        // connected to the output neurons for this 0 hidden layer network.
        return;
    }

    // Otherwise, the inputs will be connected to the first layer of hidden
neurons.
    else {
        // Allocate hidden layer 0 Neurons and connect them to the inputs.
        // In the following loop:
        // i - current hidden layer 0 neuron
        // j - current input
        for (i = 0; i < numHidNeurons[0]; i++) {
             // Allocate memory for a sigmoid based neuron.
                if (usingSigmoid)
                       hidNeurons[0][i] = new Neuron(n, a);

                // Otherwise, allocate a tanh based neuron.
                else
                       hidNeurons[0][i] = new Neuron(n, a, b);

            // Connect each input to the current neuron.
            for (j = 0; j < numInputs; j++) {
                // Randomly assign weight for the current synapse somwhere
between
                // -1 and 1.
                weight = (double)rand() / (RAND_MAX + 1) * 2 - 1;

                // Connect input j to hidden layer 0 neuron i.
                if (hidNeurons[0][i]->AddSynapse(NULL,weight,n,inputs+j)) {
                    cout << "ERROR: Too many Synapses added to Neuron" <<
endl;
                    exit(1);
                }
            }
        }
    }

    // Allocate each layer of hidden neurons in the network and inter connect
    // them. Each neuron in hidden layer i (i>0) will be connected to each
    // neuron in hidden layer i-1. If the NN has hidden layers, then the first
    // hidden layer has already been connected to the inputs. This loop will
not
    // execute if the NN has 0 or 1 hidden layers. In the following loop:
    // i - current hidden layer
```

```
    // j - current neuron in layer i
    // k - current neuron in layer i-1 (or input for i == 0)
    for (i = 1; i < numHidLayers; i++) {
        for (j = 0; j < numHidNeurons[i]; j++) {
             // Allocate memory for a sigmoid based neuron.
                    if (usingSigmoid)
                           hidNeurons[i][j] = new Neuron(n, a);

                    // Otherwise, allocate a tanh based neuron.
                    else
                           hidNeurons[i][j] = new Neuron(n, a, b);

            // Connect all neurons in layer i-1 to the current neuron.
            for (k = 0; k < numHidNeurons[i-1]; k++) {
                // Randomly assign weight for the current synapse somwhere
between
                // -1 and 1.
                weight = (double)rand() / (RAND_MAX + 1) * 2 - 1;

                // Connect neuron k (layer i-1) to neuron j (layer i)
                if (hidNeurons[i][j]->AddSynapse(hidNeurons[i-1][k],
                                                 weight,n,NULL)) {
                    cout << "ERROR: Too many Synapses added to Neuron" <<
endl;
                    exit(1);
                }
            }
        }
    }

    // Allocate output Neurons and connect them to the last layer of hidden
neurons.
    // If no hidden layers exist, the constructor would have already returned.
    // In the following loop:
    // i - current output neuron
    // j - current hidden neuron in last hidden layer (numHidLayers-1)
    for (i = 0; i < numOutputs; i++) {
             // Allocate memory for a sigmoid based neuron.
             if (usingSigmoid)
                    outNeurons[i] = new Neuron(n, a);

             // Otherwise, allocate a tanh based neuron.
             else
                    outNeurons[i] = new Neuron(n, a, b);

        // Connect each input to the current neuron.
        for (j = 0; j < numHidNeurons[numHidLayers-1]; j++) {
            // Randomly assign weight for the current synapse somwhere between
            // -1 and 1.
            weight = (double)rand() / (RAND_MAX + 1) * 2 - 1;

            // The last layer of hidden neurons is layer numHidLayers-1
            if (outNeurons[i]->AddSynapse(hidNeurons[numHidLayers-1][j],
                                          weight,n,NULL)) {
```

```
                cout << "ERROR: Too many Synapses added to Neuron" << endl;
                exit(1);
            }
        }
    }
}

// Destructor.
FullyConnectedNN::~FullyConnectedNN() {
    int i, j;

    // Free every hidden neuron j in every layer i.
    for (i = 0; i < numHidLayers; i++)
        for (j = 0; j < numHidNeurons[i]; j++)
            delete hidNeurons[i][j];

    // Free every output neuron i.
    for (i = 0; i < numOutputs; i++)
        delete outNeurons[i];
}

// Calculates all outputs of the NN.
void FullyConnectedNN::CalcOutput() {
    int i, j;

    // Free every hidden neuron j in every layer i.
    for (i = 0; i < numHidLayers; i++)
        for (j = 0; j < numHidNeurons[i]; j++)
            hidNeurons[i][j]->CalcOutput();

    // Now calculate the outputs of every output neuron since the hidden
    // neuron outputs are now known.
    for (i = 0; i < numOutputs; i++)
        outNeurons[i]->CalcOutput();
}

// Return output x of the NN, if x is out of bounds then a 0 is returned.
double FullyConnectedNN::GetOutput(int x) {
    if (x < 0 || x >= numOutputs)
        return 0;

    return outNeurons[x]->GetOutput();
}

// Prints all of the synaptic wieghts of every neuron.
void FullyConnectedNN::PrintWeights() {
    int i, j, k;
    string buf;

    cout << endl << "Press ENTER to see each layer:" << endl;

    // Print weights of every synapse k of every neuron j in every layer i.
    // Free every hidden neuron k in every layer i.
    for (i = 0; i < numHidLayers; i++) {
```

```
        cout << endl << "***Layer " << i << " Hidden Neurons***";
        getline(cin,buf);

        // Traverse every neuron j in layer i.
        for (j = 0; j < numHidNeurons[i]; j++) {
            cout << endl << "Hidden Neuron " << j << endl;
            cout << setw(17) << "Bias:" << setw(11)
                 << hidNeurons[i][j]->GetSynapseWeight(0)
                 << "Correction: " << hidNeurons[i][j]->GetSynapseCorrect(0)
                 << endl;

            // Print every synapse weight k of current neuron j.
            for (k = 1; k < hidNeurons[i][j]->GetNumSynapses(); k++) {
                cout << "Input " << k - 1 << " Weight:  " << setw(11)
                    << hidNeurons[i][j]->GetSynapseWeight(k) << "Correction: "
                    << hidNeurons[i][j]->GetSynapseCorrect(k) << endl;
            }
        }
    }

    // Print each synapse j of each output neuron i.
    cout << endl << "***Output Neurons***";
    getline(cin,buf);
    for (i = 0; i < numOutputs; i++) {
        cout << endl << "Output Neuron " << i << endl;
        cout << setw(17) << "Bias:" << setw(11) << outNeurons[i]-
>GetSynapseWeight(0)
             << "Correction: " << outNeurons[i]->GetSynapseCorrect(0) << endl;

        for (j = 1; j < outNeurons[i]->GetNumSynapses(); j++) {
            cout << "Hidden " << j - 1 << " Weight: " << setw(11)
                 << outNeurons[i]->GetSynapseWeight(j) << "Correction: "
                 << outNeurons[i]->GetSynapseCorrect(j) << endl;
        }
    }
}

// Executes back propagation learning for the current sample.
void FullyConnectedNN::BackPropLearn(double *desiredOutputs) {
    int i, j, k;
    double weightedLocGradSum; // Used to determine corrections for hidden
neurons.

    // Calculate the corrections for the synapses of the output Neuron i.
    for (i = 0; i < numOutputs; i++)
        // The error is passed to CalcCorrections().
        outNeurons[i]->CalcCorrections(desiredOutputs[i]-GetOutput(i));

    // Calculate the corrections for every synapse of every neuron j
    // in every layer i. The loop starts at the last hidden layer and
    // each iteration calculates the correction for successively earlier
layers.
    // In the following loop:
    // i - current hidden layer
```

```
    // j - current hidden neuron in layer i
    // k - current output neuron if layer i is the last hidden layer,
otherwise:
    // k - current hidden neuron in layer i+1
    for (i = numHidLayers-1; i >= 0; i--) {
        for (j = 0; j < numHidNeurons[i]; j++) {
            // Reset weightedLocGradSum to 0.
            weightedLocGradSum = 0;

            // For the last hidden layer, increment weightedLocGradSum using
the error
            // signal of every output neuron k connected to current neuron j
in layer i.
            if (i == numHidLayers-1) {
                // Calculate weightedLocGradSum, which is made up of the sum
of
                // the weighted local gradients for each synapse connecting
the hidden
                // neuron j in layer i to output neuron k.
                for (k = 0; k < numOutputs; k++)
                    // 1 is added to j because the first synapse of Neuron k
is its bias.
                    // Synapse j+1 of Neuron k is connected to the output of
neuron j.
                    weightedLocGradSum += outNeurons[k]-
>GetSynapseBackSignal(j+1);
            }

            // Otherwise, increment weightedLocGradSum using the error signal
of
            // every neuron k in layer i + 1.
            else {
                // Calculate weightedLocGradSum, which is made up of the sum
of
                // the weighted local gradients for each synapse connecting
the hidden
                // neuron j in layer i to hidden neuron k in layer i+1.
                for (k = 0; k < numHidNeurons[i+1]; k++)
                    // 1 is added to j because the first synapse of Neuron k
is its bias.
                    // Synapse j+1 of Neuron k is connected to the output of
neuron j.
                    weightedLocGradSum += hidNeurons[i+1][k]-
>GetSynapseBackSignal(j+1);
            }

            // The weightedLocGradSum is passed to CalcCorrections() of
current
            // neuron j in layer i.
            hidNeurons[i][j]->CalcCorrections(weightedLocGradSum);
        }
    }

    // Now apply all of the corrections to every output Neuron i.
```

```
    for (i = 0; i < numOutputs; i++)
        outNeurons[i]->ApplyCorrections();

    // Now apply all of the corrections to every hidden neuron j in every
layer i.
    for (i = 0; i < numHidLayers; i++)
        for (j = 0; j < numHidNeurons[i]; j++)
            hidNeurons[i][j]->ApplyCorrections();
}

// Stores all synaptic weights in outFile. Returns true on failure,
// false on success.
bool FullyConnectedNN::SaveWeightsToFile(FILE *outFile) {
    WeightFileHeader header; // Written to outFile.
    int i, j, k;
    double weight;

    // Failure if outFile has not been opened.
    if (outFile == NULL) {
        cout << "ERROR: File has not been opened before attempting to write "
             << "weights to it." << endl;
        return true;
    }

    // Populate header.
    for (i = 0; i < MAX_HIDDEN_LAYERS; i++)
        header.numHidNeurons[i] = numHidNeurons[i];

    header.numInputs = numInputs;
    header.numOutputs = numOutputs;
    header.numHidLayers = numHidLayers;

    // Attempt to write header to outFile.
    if (fwrite(&header,sizeof(WeightFileHeader),1,outFile) != 1) {
        cout << "ERROR: Failed to write header to output weight file." <<
endl;
        return true;
    }

    // The following loop attempts to write all synaptic weights k of all
hidden
    // neurons j in every layer i.
    for (i = 0; i < numHidLayers; i++) {
        for (j = 0; j < numHidNeurons[i]; j++) {
            for (k = 0; k < hidNeurons[i][j]->GetNumSynapses(); k++) {
                // Get weight of current synapse.
                weight = hidNeurons[i][j]->GetSynapseWeight(k);

                // Attempt to write weight to outFile.
                if (fwrite(&weight,sizeof(double),1,outFile) != 1) {
                    cout << "ERROR: failed to write weight to output weight
file." << endl;
                    return true;
                }
```

```
            }
        }
    }

    // Now write all synaptic weights j of all output neurons i.
    for (i = 0; i < numOutputs; i++) {
        for (j = 0; j < outNeurons[i]->GetNumSynapses(); j++) {
            // Get weight of current synapse.
            weight = outNeurons[i]->GetSynapseWeight(j);

            // Attempt to write weight to outFile.
            if (fwrite(&weight,sizeof(double),1,outFile) != 1) {
                cout << "ERROR: failed to write weight to output weight file."
<< endl;
                return true;
            }
        }
    }

    // Success.
    return false;
}

// Loads synaptic weight file. Returns true on failure, false on success.
bool FullyConnectedNN::LoadWeightsFromFile(FILE *fromFile) {
    WeightFileHeader header; // Stores header of fromFile.
    int i, j, k;
    double weight;

    // Failure if fromFile has not been opened.
    if (fromFile == NULL) {
        cout << "ERROR: File has not been opened before attempting to read "
             << "weights from it. Weights have not been changed." << endl;
        return true;
    }

    // Seek to begining of file.
    fseek(fromFile,0,SEEK_SET);

    // Attempt to read header from fromFile.
    if (fread(&header,sizeof(WeightFileHeader),1,fromFile) != 1) {
        // Display error message if not eof.
        if (!feof(fromFile))
            cout << "ERROR: failed to read header from weight file. "
                 << "Weights have not been changed." << endl;

        // Failure.
        return true;
    }

    // Now, compare every member of header to every member of this
FullyConnectedNN
    // object. If any differences occur, then the weight file is incompatible.
    if (header.numInputs != numInputs || header.numOutputs != numOutputs ||
```

```
        header.numHidLayers != numHidLayers ) {
            cout << "ERROR: Weight file is not compatible with current Neural
Network. "
                 << "Weights have not been changed." << endl;
            return true;
    }

    // Compare every element in numHidNeurons.
    for (i = 0; i < MAX_HIDDEN_LAYERS; i++) {
        if (header.numHidNeurons[i] != numHidNeurons[i]) {
            cout << "ERROR: Weight file is not compatible with current Neural
Network. "
                 << "Weights have not been changed." << endl;
            return true;
        }
    }

    // The weight file was deemed to be compatible. The following loop
attempts to
    // set all synaptic weights k of all hidden neurons j in every layer i
using
    // the values read from fromFile.
    for (i = 0; i < numHidLayers; i++) {
        for (j = 0; j < numHidNeurons[i]; j++) {
            for (k = 0; k < hidNeurons[i][j]->GetNumSynapses(); ) {
                // Attempt to get weight of current synapse from file.
                if (fread(&weight,sizeof(double),1,fromFile) != 1) {
                    // Display error message if not eof.
                    if (!feof(fromFile))
                        cout << "ERROR: failed to read weight from weight
file. "
                             << "Weights values might have been corrupted." <<
endl;

                    // Failure.
                    return true;
                }

                // Set new synapse weight.
                hidNeurons[i][j]->SetSynapseWeight(k++,weight);
            }
        }
    }

    // Now write all synaptic weights j of all output neurons i.
    for (i = 0; i < numOutputs; i++) {
        for (j = 0; j < outNeurons[i]->GetNumSynapses(); j++) {
            // Attempt to get weight of current synapse from file.
            if (fread(&weight,sizeof(double),1,fromFile) != 1) {
                // Display error message if not eof.
                if (!feof(fromFile))
                    cout << "ERROR: failed to read weight from weight file. "
                         << "Weights values might have been corrupted." <<
endl;
```

```
                // Failure.
                return true;
            }

            // Set new synapse weight.
            outNeurons[i]->SetSynapseWeight(j,weight);
        }
    }

    // Success.
    return false;
}
```

F.5. Neuron.h

```
/*
Ariel Kopel
Thesis Project
Advisor: Dr. Helen Yu
Cal Poly State University - San Luis Obispo

Neuron.h - Defines the Neuron and Neuron::Synapse classes, used to construct a
neural network.
*/

#ifndef NEURON_H
#define NEURON_H

#include "NNSimulation.h"

// A Neuron object consists of a vector of input synapses including the bias.
// A Neuron generates an output using the sigmoid function applied to the
// weighted sum of all input synapses.
class Neuron {
public:

    // Constructor for a sigmoid based neuron.
    Neuron(double n, double a);

    // Constructor for a tanh based neuron.
    Neuron(double n, double a, double b);

    // Destructor, responsible for deallocating all Synapse objects associated
    // with this Neuron object.
    ~Neuron();

    // Allocates a new Synapse object and adds it to inputVector.
    bool AddSynapse(Neuron *from, double weight, double n, double *input);

    // Returns ouptut.
    double GetOutput();

    // Calculate output which is made up of the weighted sum of all
    // synapse inputs applied to the sigmoid function.
    void CalcOutput();

    // Returns numSynapses.
    int GetNumSynapses();

       // Sets the weight of synapse x. If x is out of bounds, returns true.
    // Otherwise, returns false.
    bool SetSynapseWeight(int x, double newWeight);

    // Returns the weight of synapse x. If x is out of bounds, 0 is returned.
    double GetSynapseWeight(int x);
```

```
    // Returns the correction of synapse x. If x is out of bounds, 0 is
returned.
    double GetSynapseCorrect(int x);

    // Returns the backSignal of synapse x. If x is out of bounds, 0 is
returned.
    double GetSynapseBackSignal(int x);

    // Calculates corrections and backSignal for each Synapse.
    void CalcCorrections(double ej);

    // Applies the calculated corrections to all Synapses.
    void ApplyCorrections();

    // Each synapse object represents a weighted synapse in the neural
network.
    class Synapse {
    public:
        // Constructor, initializing all data members.
        Synapse(Neuron *from, double weight, double n, double *input);

        // Update correction and backSignal.
        void SetCorrection(double locGradient);

          // Sets weight.
        void SetWeight(double newWeight);

        // Applies to correction to weight.
        void ApplyCorrection();

        // Returns output of synapse (weight * input or
        // weight * from->GetOutput()).
        double GetOutput();

        // Returns *input or from->GetOutput().
        double GetInput();

        // Returns weight.
        double GetWeight();

        // Returns correction.
        double GetCorrection();

        // Returns backSignal().
        double GetBackSignal();

    private:
        Neuron *from;       // Input neuron.
        double weight;      // Weight of synapse.
        double correction;  // Correction calculated in one training step.
        double n;           // Learning rate.
        double backSignal;  // Back propagation signal sent to from Neuron.
        double *input;      // Pointer to the input, which is modified
elsewhere,
```

```
                            // and used if this synapse is an input synapse.
    };

private:
    Synapse *inputVector[MAX_INPUTS]; // The input synapses
                                      // connected to the neuron.
    int numSynapses; // Number of synapses connected to neuron.
    double output; // Contains current output.
    double a; // sigmoid activation function slope, or tanh amplitude.
       double b; // tanh slope.
       bool sigmoidActivation; // true if using sigmoid activation function,
                                               // false if using tanh.
};

#endif
```

F.6. Neuron.cpp

```
/*
Ariel Kopel
Thesis Project
Advisor: Dr. Helen Yu
Cal Poly State University - San Luis Obispo

Neuron.cpp - Implements functions of the Neuron and Neuron::Synapse classes,
as defined in Neuron.h.
*/

#include <iostream>
#include <fstream>
#include "Neuron.h"
#include <math.h>

using namespace std;

// Constant input of +1, to be used by the biases of each Neuron, represented
// by the first Synapse in inputVector of each Neuron.
static double one = 1;

/*** Neuron Functions ***/

// Constructor for a sigmoid based neuron.
Neuron::Neuron(double n, double a) {
    double bias;

    // Set a.
    this->a = a;

       // Using sigmoid activation function.
       sigmoidActivation = true;

    // Initialize the number of synapses to 0.
    numSynapses = 0;

    // Inititalize output to 0
    output = 0;

    // Randomize the bias for this neuron between -1 and 1.
    bias = (double)rand() / (RAND_MAX + 1) * 2 - 1;

    // Add the bias Synapse, with a random bias value.
    if (AddSynapse(NULL, bias, n, &one))
        cout << "WARNING: Adding bias synapse failed." << endl;
}

// Constructor for a tanh based neuron.
Neuron::Neuron(double n, double a, double b) {
    double bias;
```

```
    // Set a and b.
    this->a = a;
      this->b = b;

      // Using tanh activation function.
      sigmoidActivation = false;

    // Initialize the number of synapses to 0.
    numSynapses = 0;

    // Inititalize output to 0
    output = 0;

    // Randomize the bias for this neuron between -1 and 1.
    bias = (double)rand() / (RAND_MAX + 1) * 2 - 1;

    // Add the bias Synapse, with a random bias value.
    if (AddSynapse(NULL, bias, n, &one))
        cout << "WARNING: Adding bias synapse failed." << endl;
}

// Destructor, responsible for deallocating all Synapse objects associated
// with this Neuron object.
Neuron::~Neuron() {
    // Deallocate each of the Synapses in inputVector.
    for (int i = 0; i < numSynapses; i++)
        delete inputVector[i];
}

// Allocates a new Synapse object and adds it to inputVector. Returns
// false on success, true otherwise.
bool Neuron::AddSynapse(Neuron *from, double weight, double n, double *input)
{
    // If there is no more room for a new synapse, return false.
    if (numSynapses >= MAX_INPUTS)
        return true;

    // Otherwise, allocate a new Synapse object.
    inputVector[numSynapses++] = new Synapse(from, weight, n, input);

    // Success, increment the number of synapses.
    return false;
}

// Calculate output which is made up of the weighted sum of all synapse
// inputs applied to the sigmoid function.
void Neuron::CalcOutput() {
    double weightedSum = 0; // Weighted sum of inputVector.

    // Calculate the weighted sum of inputVector.
    for (int i = 0; i < numSynapses; i++)
        weightedSum += inputVector[i]->GetOutput();

    // If using a sigmoid activation function, then calcualte
```

```
	// output using the sigmoid function.
	if (sigmoidActivation)
		output = 1 / (1 + exp((-1)*a*weightedSum));

	// Otherwise, use tanh to calculate the output.
	else
		output = a * tanh(b * weightedSum);
}

// Returns output.
double Neuron::GetOutput() {
    return output;
}

// Returns numSynapses.
int Neuron::GetNumSynapses() {
    return numSynapses;
}

// Sets the weight of synapse x. If x is out of bounds, returns true.
// Otherwise, returns false.
bool Neuron::SetSynapseWeight(int x, double newWeight) {
    // Return true if x is out of bounds.
    if (x < 0 || x > numSynapses)
        return true;

    // Set the weight of synapse x.
    inputVector[x]->SetWeight(newWeight);

    // Success.
    return false;
}

// Returns the weight of synapse x. If x is out of bounds, 0 is returned.
double Neuron::GetSynapseWeight(int x) {
    if (x < 0 || x > numSynapses)
        return 0;

    return inputVector[x]->GetWeight();
}

// Returns the correction of synapse x. If x is out of bounds, 0 is returned.
double Neuron::GetSynapseCorrect(int x) {
    if (x < 0 || x > numSynapses)
        return 0;

    return inputVector[x]->GetCorrection();
}

// Returns the backSignal of synapse x. If x is out of bounds, 0 is returned.
double Neuron::GetSynapseBackSignal(int x) {
    if (x < 0 || x > numSynapses)
        return 0;
```

```
    return inputVector[x]->GetBackSignal();
}

// Calculates corrections and backSignal for each Synapse.
// If the Neuron is an output neuron, then ej is the error in the Neuron
// output, or difference between the desired output and the generated
// output. If the Neuron is hidden, then ej is the sum of weighted
// local gradients of all synapses connecting this Neuron to output Neurons.
void Neuron::CalcCorrections(double ej) {
    double locGradient; // Passed to SetCorrection() of each synapse.
    int i;

       // If using the sigmoid activation function:
       if (sigmoidActivation)
             // locGradient is defined as follows Neuron j:
             // locGradient = A*ej(n)*yj(n)*[1-yj(n)]
             // where yj(n) is the Neuron output, A is a constant defining
             // the sigmoid activation function, and ej is passed in and
defined above.
             locGradient = a*ej*output*(1-output);

       // Otherwise, using the tanh activation function.
       else
             // locGradient is defined as follows Neuron j:
             // locGradient = B/A*ej(n)*[A+yj(n)]*[A-yj(n)]
             // where yj(n) is the Neuron output, A is the amplitude of the
tanh,
             // B is the slope, and ej is passed in and defined above.
             locGradient = b/a*ej*(a+output)*(a-output);

    // SetCorrection() for each synapse using locGradient.
    for (i = 0; i < numSynapses; i++)
        inputVector[i]->SetCorrection(locGradient);
}

// Applies the calculated corrections to all Synapses.
void Neuron::ApplyCorrections() {
    // Apply the correction to each Synapse.
    for (int i = 0; i < numSynapses; i++)
        inputVector[i]->ApplyCorrection();
}

/*** Neuron::Synapse Functions ***/

// Constructor, initializing all data members.
Neuron::Synapse::Synapse(Neuron *from, double weight, double n, double *input)
{
    // Initialize all members to the values passed to the constructor.
    this->from = from;
    this->weight = weight;
    this->n = n;
    this->input = input;
    this->correction = 0;
    this->backSignal = 0;
```

```
    // If both input and from are NULL, then warn.
    if (input == NULL && from == NULL)
        cout << "WARNING: Both input and from are NULL in Synapse." << endl;
}

// Update correction and backSignal.
void Neuron::Synapse::SetCorrection(double locGradient) {
    // correction is defined as follows for Neuron j:
    // n*xi(n)*locGradient
    // where n is the learning rate and xi(n) is the input of the synapse.
    this->correction = n*GetInput()*locGradient;

    // The signal sent to the from Neuron is the weighted local gradient.
    this->backSignal = locGradient * weight;
}


// Sets weight.
void Neuron::Synapse::SetWeight(double newWeight) {
    weight = newWeight;
    correction = 0;
    backSignal = 0;
}

// Applies to correction to weight.
void Neuron::Synapse::ApplyCorrection() {
    weight += correction;
}

// Returns *input or from->GetOutput() depending on input.
double Neuron::Synapse::GetInput() {
    // If the synapse is an input synapse, return *input.
    if (input)
        return *input;

    // Otherwise, use from->GetOutput() to calculate input.
    else
        return from->GetOutput();
}

// Returns output of synapse (weight * input or weight * from->GetOutput())
double Neuron::Synapse::GetOutput() {
    return weight * GetInput();
}

// Returns weight.
double Neuron::Synapse::GetWeight() {
    return weight;
}

// Returns correction.
double Neuron::Synapse::GetCorrection() {
    return correction;
```

```
}

// Returns backSignal.
double Neuron::Synapse::GetBackSignal() {
    return backSignal;
}
```