A Comparison of Neural Network Training Methods for Character Recognition

Author: Brian Clow

Advisor: Dr. Tony White, Department of Computer Science

April 11, 2003
Abstract

Neural networks can be applied to many problems in computer science that are difficult to solve by traditional means. They consist of a network of small, interconnected processing units. Each unit is connected to other units via a weighted link, and each unit functions very simply. The network as a whole will display complex behaviour. The particular problem of choice here is character recognition – given a set of characters, the neural network must correctly determine which letter of the alphabet these characters represent. The network will be trained to recognize characters via three different methods – back-propagation of error, particle swarm optimization, and genetic algorithms. The first method is the original training method for multi-layer neural networks, and is a gradient descent algorithm. The other two methods are both evolutionary methods. They have the potential to avoid certain problems associated with gradient descent methods, such as the problem of local minima.

Back-propagation was found to be superior in terms of both training time and accuracy. However, the two evolutionary methods showed great promise. One version of a genetic algorithm achieved comparable performance on smaller datasets. The high computational cost of evolutionary methods seemed to be the major barrier – population sizes had to be kept small and training took a long time. Much improvement was made in the performance of these methods – they have much potential as good training methods, given a bit more processing power.
# Table of Contents

1. Introduction .......................... 1  
2. Neural Networks ...................... 3  
   2.1. Overview .......................... 3  
   2.2. Implementation .................. 7  
3. Data .................................. 9  
   3.1. Overview .......................... 9  
   3.2. Dataset Creation .................. 9  
   3.3. Final Dataset Information ...... 13  
4. Backpropagation Training ............ 16  
   4.1. Overview .......................... 16  
   4.2. Implementation .................. 19  
   4.3. Testing ........................... 19  
   4.4. Conclusion ...................... 21  
5. Fitness Functions for Evolutionary Methods 22  
   5.1. Overview .......................... 22  
   5.2. Class Fitness .................... 23  
   5.3. Mean Squared Error Fitness ...... 23  
   5.4. Sampled Fitness .................. 24  
   5.5. Combination Fitness .............. 25  
   5.6. Conclusion ...................... 26  
   6.1. Overview .......................... 28  
   6.2. Implementation .................. 30  
   6.3. Notes on Parameter Optimization 30  
   6.4. Network to Chromosome Encoding 31  
      6.4.1. Overview ..................... 31  
      6.4.2. Feature Detector Encoding ...... 32  
      6.4.3. Neuronal Encoding ............ 32  
      6.4.4. Permutation Problem .......... 32  
      6.4.5. Testing ....................... 33  
   6.5. Selection Methods ............... 34  
      6.5.1. Overview ..................... 34  
      6.5.2. Rank Selection ............... 35  
         6.5.2.1. Overview .................. 35  
         6.5.2.2. Parameter Testing .......... 36  
      6.5.3. Tournament Selection ......... 37  
         6.5.3.1. Overview .................. 37  
         6.5.3.2. Pick Best Member Probability ........ 37  
         6.5.3.3. Tournament Size .......... 39  
   6.6. Genetic Operators ............... 40  
      6.6.1. Overview ..................... 40  
      6.6.2. Crossover ..................... 40  
         6.6.2.1. Overview .................. 40  
         6.6.2.2. Crossover Probability ...... 41  
      6.6.3. Mutation ..................... 43  
         6.6.3.1. Overview .................. 43  
         6.6.3.2. Mutation Rate .............. 43  
   6.7. Genetic Algorithm Paradigms .... 46  
      6.7.1. Overview ..................... 46  
      6.7.2. Non-Elitist ................... 46  
      6.7.3. Elitist ....................... 47
6.7.4. Steady State 47
6.8. Final Testing 47
  6.8.1. Overview 47
  6.8.2. Testing 48
6.9. Conclusion 52
7. Particle Swarm Optimization 53
  7.1. Overview 53
  7.2. Implementation 55
  7.3. Maximum Velocity Decay 56
  7.4. Fitness Functions 57
  7.5. Testing 58
    7.5.1. Test Set 1 59
    7.5.2. Test Set 2 61
  7.6. Conclusion 62
8. Generalization 64
  8.1. Overview 64
  8.2. Implementation 65
  8.3. Testing 66
  8.4. Conclusion 69
9. Scaleability 71
  9.1. Overview 72
  9.2. Testing 72
  9.3. Conclusion 75
10. Conclusions and Future Directions 77
11. References
12. Appendix A – CD Contents
List of Figures

Figure 1: Neural Network Sizes
Figure 2: Back-propagation Learning Rates
Figure 3: Rank Selection max Parameter
Figure 4: Tournament Selection Pick Best Member Probability
Figure 5: Tournament Selection Tournament Size
Figure 6: Elitist Crossover Rate
Figure 7: Non-Eltitist Crossover Rate
Figure 8: Elitist Mutation Rate
Figure 9: Steady State Mutation Rate
Figure 10: Non-Elitist Mutation Rate
Figure 11: Elitist Encoding/Selection
Figure 12: Steady State Encoding/Selection
Figure 13: Non-Elitist Encoding/Selection
Figure 14: Final Genetic Algorithm Parameter Optimization
Figure 15: 1.0 Maximum Velocity, [-1.0,1.0] Starting Velocity
Figure 16: 0.5 Maximum Velocity, [-0.5,0.5] Starting Velocity
Figure 17: 0.5 Maximum Velocity, [-0.125,0.125] Starting Velocity bFin
Figure 18: Final Particle Swarm Design
Figure 19: Generalization Dataset 1
Figure 20: Generalization Dataset 2
Figure 21: Generalization Dataset 3
Figure 22: Generalization Dataset 4
Figure 23: Scaleability Dataset 1
Figure 24: Scaleability Dataset 2
Figure 25: Scaleability Dataset 3
Figure 26: Scaleability Dataset 4
List of Tables

Table 1: Custom .dat File Header
Table 2: Training/Testing Data Sets
Table 3: Paradigm Tests
Table 4: Final Genetic Algorithm Implementation Candidates
Table 5: Particle Swarm Velocity Decay Functions
Table 6: Generalization Tests
Table 7: Scaleability Datasets

List of Equations

Equation 1: Neuron Input
Equation 2: Sigmoid Function
Equation 3: Mean Squared Error for Back-propagation
Equation 4: Derivative of Sigmoid Function
Equation 5: Output Layer Error Gradient
Equation 6: Hidden Layer Error
Equation 7: Hidden Layer Error Gradient
Equation 8: Mean Square Error for Fitness
Equation 9: Expected Value for Rank Selection
Equation 10: Particle Swarm Velocity Update
Chapter 1: Introduction

Neural networks are often used to solve problems that computers generally find difficult. Computers are very good at tasks that humans are poor at – those that involve lots of number crunching, or brute force searches. However, they are notoriously poor at things that humans find simple, such as moving around in a dynamic environment, or recognizing simple patterns. Attempts have been made to program computers to do such tasks for many years, and neural networks have emerged as an effective means to do so.

A neural network isn’t programmed to solve a problem – instead, it learns to solve a problem after seeing many examples of this problem. It must be trained to solve the problem correctly. The traditional method is back-propagation of error through the network, but there are many other methods. Evolutionary computing is of particular interest because it is also good at finding solutions to problems that may not have a well-defined algorithm to solve. A neural network may benefit from the application of such methods to its’ training. Two alternative training methods were tested and compared to back-propagation: particle swarm optimization, and genetic algorithms. Each of these methods, as well as back-propagation training, is described separately in its’ own chapter.

The problem chosen to compare the training methods is pattern recognition, in the form of classification of characters. This is closely related to optical character recognition. However, OCR has several complexities (such as determining where characters begin and end) that would make the comparison of different training methods difficult. As such, training data will be prepared in a format that is easy for a neural network to read. Traditional character recognition methods work by methods such as
template matching, where letters are compared against a template letter, and they are classified according to which template they match most closely. If a letter is far from the given template, it is unlikely to be recognized correctly. Neural networks attempt to avoid this problem. Given many training examples of characters, and their correct classifications, a neural network will learn to recognize features of each letter. Since it recognizes the features, it will be able to recognize characters that are unlike those it has seen previously, but that still share the same features. This is the main benefit of neural networks – they adapt easily to new and previously unseen data.

The objectives of this project are to investigate the performance of the two evolutionary methods as compared to the performance of the traditional back-propagation. If their performance is found to be poor, attempts will be made to increase their performance such that they are comparable to back-propagation. Literature on the subject suggests that both particle swarm optimization and genetic algorithms have been used successfully to train neural networks for optical character recognition. However, most of the literature deals with very limited networks – a small number of characters of a small size. For these training methods to be practical, they must be scaleable to a large number of characters of variable sizes. The scaleability of all training methods will be compared, along with the ability to generalize to previously unseen data.
Chapter 2: Neural Networks

Section 2.1: Overview

Neural networks are inspired by the biological structure of the brain. The brain consists of many simple cells, called neurons. These cells are interconnected via dendrites and axons – the place where the axon of one cell connects to the dendrites of another cell is called a synapse. Each neuron is connected to many other neurons. Signals are propagated from neuron to neuron via a complex electrochemical process. Signals are received from other neurons, raising or lowering the electric potential of the cell. When the electric potential reaches a certain threshold, an electrical pulse, called an “action potential”, is sent along the axon. The pulse reaches a synapse that connects the axon to other neurons, raising or lowering their respective electric potential. The synapses that lower the potential of the receiving neuron are called inhibitory, and those that increase it are called excitatory. The connections between neurons are fluid and changeable – the strength of a connection will change depending on how often and how intensely the connection is stimulated. Neurons may form new connections with other neurons, or may break the connections that currently exist. This is the basis of learning in the human brain – after repeated exposure to a stimulus, the connections between neurons will change and adapt to recognize and react to this stimulus.

Neural networks are intended to mimic the functioning of the human brain. They consist of many small interconnected processing units referred to as “neurons”. These neurons are organized into separate layers. Neural networks first originated with two-layer networks, known as perceptrons. Such networks can only learn linearly separable functions, and are of limited use. Much work has been done with multi-layer networks,
but the most common implementation consists of three layers of neurons – the input layer, hidden layer, and output layer. Each neuron in each layer is connected to every neuron in the next layer. That is, every neuron in the input layer is connected to every neuron in the hidden layer, and every neuron in the hidden layer is connected to every neuron in the output layer. Each neuron maintains a list of incoming links, and a list of outgoing links. Each incoming connection has a numeric weight associated with it. These weights are the primary means of learning and storage in neural networks, and learning takes place by changing the weights of the connections. A positive weight represents an excitatory neuron, while a negative weight represents an inhibitory neuron.

In this implementation, the structure of the network was fixed – no new links could be added, and none could be removed. However, a connection weight may be set to 0, which essentially means that it doesn’t exist.

To process information in a neural network, each input layer neuron is set to a certain value according to the input. In the case of character recognition, each input layer neuron represents a pixel in the character, and will be set to 0 if the pixel is off or 1 if the pixel is on. Different problems will require different encodings of the input onto the input layer neurons. The output of each input layer neuron is sent along each connection to the hidden layer neurons. Each hidden layer neuron will compute the weighted sum of all incoming links. The input of each hidden layer neuron is as follows, where \( a_i \) represents the incoming information from each input layer unit, and \( W_i \) represents the weight of each incoming link.

\[
\text{input} = \sum W_i \times a_i \tag{Equation 1}
\]
Each hidden layer neuron will compute its’ output value from the weighted sum of its’ inputs. There are different ways of doing so – many ways rely on an activation threshold, where the neuron will output a 1 if the weighted input is above a certain threshold value. However, the sigmoid function is also commonly used, and was used for this implementation. The sigmoid function will output a value from the range of 0 to 1, and is shown in equation 2.

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$  \hspace{1cm} \textbf{Equation 2} 

The weighted input is used as the value x in the sigmoid function. This gives a smooth curve around the activation threshold (show diagram). Thus, the output of each unit will always be between 0 and 1. After the inputs have been propagated to the hidden layer, they must be further propagated to the output layer. The exact same process is used as for propagating to the hidden layer – each output layer neuron will compute the weighted sum of all incoming links, and will apply the sigmoid function to this to obtain its’ output value. The type of neural network just described is called a “feed-forward network” – all data is propagated through the network in a purely forward direction, from the input layer neurons to the outputs. There are other types of networks, including recurrent networks that contain backwards connections and possible cycles. These were not investigated in this report, but much work has been done researching them.

When a neural net is used for classification problems such as character recognition, each output will represent a possible classification of the input. In this case,
there are 26 separate output neurons – one for each letter of the alphabet. The output with the highest value is considered the classification of the input data. Thus, if the second output was the highest of all outputs after propagation, the network would have classified the input as a “B”.

The correct classification of inputs by a forward propagation requires that the weights be appropriately set. However, the network starts out with the weights initially randomized, so the network will incorrectly classify the input most of the time. A training algorithm must be applied to the network to modify the weights such that they classify data correctly. Networks are trained via supervised learning – a large amount of data is presented to the network, and the output of the network is compared to the correct output for each corresponding input. If the network does not classify the data correctly, the amount of error can be calculated and the weights can be modified accordingly. A network must be shown a large amount of data repeatedly to find the correct weights for the problem at hand. This is similar to the working of the human brain – a child will not learn to recognize the letter “A” after seeing just one example, but given many different examples and some time, it will learn to recognize many different “A”s.

The strength of neural network lies in their ability to generalize – they will correctly recognize data that they have never been shown before. This occurs because some error is allowed in the outputs of the network – although the highest output is considered the classification, other outputs may still be relatively high. The imperfection of the network allows it to generalize to previously unseen data – after it has completed training, it will correctly classify other inputs.
The structure of each network is still a topic of much active research. The number of hidden layer neurons is the primary structural concern – there is no formula or method that will reliably tell how many hidden layer neurons are needed for a particular problem. Finding the correct number is a matter of experience and experimentation. If too many hidden layer neurons are used, there will be too many weights, and the network will be subject to overfitting. The network will learn to recognize the training data exactly, with very little error. This lack of error means that the network will be unable to generalize to unseen data. If too few hidden layer neurons are used, there will not be enough weights to even learn to recognize the training data.

There are many different methods available for training a neural network. The original method used for training is called back-propagation of error. The error is assessed at the outputs, and each weight is considered responsible for a certain proportion of the error. The weights are modified according to this, and the error is propagated back to the previous layer of connection weights. The main purpose of this project is to compare back-propagation against two evolutionary training methods – genetic algorithms and particle swarm optimization. Each training method has been given a separate section in this report, and can be referred to in the table of contents.

Once trained, a neural network acts as a kind of “black box”. The workings of it are well understood, but the exact reasons for it achieving success cannot be determined. That is, the actual values of the weights of the network do not give any meaningful information when examined. Nothing can be said about the workings of a particular network – it works because the weights were appropriately modified.
Section 2.2: Implementation

The neural network implementation is contained in two files: “neuron.h” and “NeuralNet.h”. The class NeuralNet is the actual network – it maintains three separate arrays of Neurons, one for each layer, as well as storing the number of neurons and the size of character the network was designed to read. It contains various functions to read/write the network from/to files, and to set up the weights and inputs. It contains functions to perform the forward and backwards propagations, which are accomplished by calling local functions for each of the Neurons. It contains functions for encoding the network weights into chromosomes and particles. It also contains functions for determining the classification of the network after it has been run.

Each Neuron maintains a list of incoming links, a weight for each link, and a list of outgoing links. It can store both the input value (the weighted sum of all inputs) and the output value (a result of the activation function). When a Neuron’s activation function is called, it will find the input value, and determine the output value via the sigmoid function. Functions for performing a backpropagation are also contained in the Neuron.

To perform a forward propagation, the inputs of the network are first set via a function of the NeuralNet class. Each hidden layer Neuron is activated followed by each output layer Neuron. The network classification is then determined using the outputs of the output layer. The mean squared error can also be determined via a function in NeuralNet, which is used by all three training methods.
Chapter 3: Data

3.1: Overview

In order to use neural networks, data must be available that is used to train and test the network. The more data available, the better – the network will be able to generalize to previously unseen data with a greater accuracy. A large amount of training data will need to be provided.

If insufficient training data is provided, a neural network will only be able to recognize the data that it has already seen, as it will simply memorize each training example. When many different examples of each training item are shown, the network will learn to classify based on features shared by examples of the same classification. This is one of the main benefits of a neural network implementation, so enough data had to be created in a usable format to train the networks.

3.2: Dataset Creation

Each character is represented by an array of pixels, each of which is either on or off. The original intention was to create characters sets by hand, but this proved to be impractical due to the large amount of training data required. Instead, standard windows fonts were used to create the character sets that were used for training and testing. There are an almost unlimited number of fonts available, and each letter can be different sizes. This means that there will be sufficient training data available. These fonts are typically stored as .TTF or .FNT file format, but either format contains excessive information considering that only the values of the pixels in a simple character are needed. In this native format, characters are not all the same width. For a neural network to process the
input, all characters must contain the same number of pixels so that the number of input
units in the network remains constant.

As such, a different method of creating character sets from font files had to be
developed. All uppercase letters from the fonts were displayed on a screen using a word
processing program, and a screenshot was taken in .bmp format. From this .bmp file, the
area of the screen containing the uppercase letters was cut into another image, and this
image was saved as a greyscale .raw file. These files use a single byte for each pixel in
the image, and are very easy to read and manipulate.

Several different ways of displaying the fonts for the screenshot were used. A
few were discarded because they didn’t display letters of a uniform size, or because of
limited size difference in the letters. The final method used was to create a table in Word
with cells of a fixed size. It was possible to take a screenshot and convert the entire table
to a .raw file. This required little manual work, and enabled fonts to be converted
quickly.

Once a .raw file was created, it had to be converted into the custom file format
that was used for datasets in this project. A program was written to do so, called
“rawconverter”. It can be found on the CD as detailed in appendix A. The final version
reads in a table that is 26 columns wide and 4 rows high, and outputs 4 files in my custom
.dat format – one file for each row in the table. The final table can also be found on the
CD.

All character sets are stored in a custom .dat text file. Text files were used rather
than binary files so that visual inspection of the files was easy. The format is simple.
There is a small header, shown in table 1.
<table>
<thead>
<tr>
<th>Line</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of characters</td>
</tr>
<tr>
<td>2</td>
<td>Number of possible classifications</td>
</tr>
<tr>
<td>3</td>
<td>Width of each letter</td>
</tr>
<tr>
<td>4</td>
<td>Height of each letter</td>
</tr>
</tbody>
</table>

**Table 1: .DAT File Header**

The main data of the file continues on the next line. Each character in the set is listed as a series of 0s and 1s, indicating if a pixel is on or off. Each row of the character takes up one row in the file, so it is possible to visually see the shape of each letter. After each individual letter, the appropriate classification is listed on a separate line – the classification can range from 0 (“A”) to 25 (“B”). The rawconverter program converts .raw files to this file format.

Many different character sets were created via this method, and they were all a standard size – 21x28 pixels each. However, the letters within each 21x28 box are often much smaller than this, particularly in the case of smaller font sizes. A smaller character size greatly improves the performance of neural networks, so as many blank rows and columns as possible need to be removed. In order to accomplish them, a program was created called “whitespace”, which can be found on the CD as listed in appendix A. It finds the dimensions of the largest letter in the character set, and then reduces all characters to this size. Since all characters must be the same size, the size of the largest letter is the smallest size the character set can be reduced to. Character sets will eventually be merged together to create data sets. To do this, all character sets must have equal dimensions. To accomplish this, the program will examine all .dat files present in the directory where it is run, and will reduce all character sets to the size of the largest
letter amongst them all. When this occurs, characters will be positioned in the top left corner of the box they are contained in – any blank rows will be to the right and below the characters.

Now that character sets of a uniform, minimum size exist, they must be combined to create datasets. A dataset consists of two files – a training .dat file, and a testing .dat file. The training file will be used to train the network, and the testing file will be presented as unseen data for testing purposes. To accomplish this, a program called “ttmaker” (train and test Maker) was created. It simply combines all .dat files in the current directory into two files called train.dat and test.dat. For each .dat file in the directory, the program performs a stochastic test to determine which dataset to add it to. This is specified on the command line as a percentage – if 70% is specified on the command line, each character set has a 70% chance of becoming a member of the training dataset. If it does not become training data, it is automatically put into the testing data file. These final training and testing data files use the exact same file format as the character sets – it is possible, though slow, to combine character set files using a text editor.

All character sets created can be found on the CD as specified in appendix A, in the original .dat format. These were used to create different sized datasets composed of different characters. The final datasets used for training/testing are discussed in section 3.3. Many different datasets were created and tested – it was found that the bigger the training dataset, the better the networks generalized to testing data. Originally, 4 different font sizes for each font were used – 10, 12, 14, and 16 pt fonts. The 16-point fonts were far too big (21x28) to make the networks practical for the evolutionary
training methods. They could be used only in the final scaleability testing in chapter 9. 8-point fonts were added later to provide a set of smaller characters. Thus there were 5 different character sets for each font, although the 8-point fonts were of a different dimension. A total of 25 different fonts were used. It would have been possible to use many more, but the letters used needed to be reasonable similar. Using very dissimilar fonts (cursive fonts, for example) would have increased the generalization ability, but would have required much more training data. Neural networks cannot learn from single examples of a font, and the processing power wasn’t available to train on much more data than was used.

3.3: Final Dataset Information

The more data presented to the neural networks, the better they will be at recognizing unseen characters. Tests with back-propagation showed a steady increase in recognition as the amount and variety of training data increased. However, a greater variety of alphabets requires that each letter be bigger, or at least be contained within a bigger square. Due to the nature of the networks, a small increase in the dimensions of each letter results in a dramatic increase of the number of weights in the network.

Consider a network with 15 hidden neurons, which should be sufficient to learn to recognize 26 separate characters. Assuming the input is square, the size of input is plotted against the size of the entire network (the number of weights) in figure 1. It can be seen that the network size is not linearly related to the input size.
The number of weights in the network greatly affects the speed of forward propagations, the most computationally expensive part of all training algorithms. Back-propagation training, which relies on a single network propagation per generation, still performs quickly with large input sizes. However, both genetic algorithms and particle swarm optimization need to perform one propagation per training item per population member – this is easily as high as 50000 propagations per generation. Several attempts were made to optimize the fitness algorithms to reduce the number of propagations (see section 5.4), but they were not completely successful.

As such, the size of each neural network had to be minimized. Minimizing the size of the input will do this, as the number of hidden and output neurons is fixed. A reduced input size means that the letters read by the network must be smaller. As mentioned above, this reduces the variety of the training data, which reduces the recognition of unseen characters, but it was unavoidable. The computing power to train
large networks was not available for this project – even on a 2 ghz computer, it took 10 hours to train a network with an input size of 10x10 and 50 hidden layer neurons. The largest character set created had letters of size 21x28, which would result in a prohibitively long training time.

The final character set consists of characters of size 9x10, which means that networks trained on these characters will have an input layer size of 90 neurons. This was the minimum size obtainable – with smaller input sizes, the letters from most fonts are distorted and difficult to differentiate. Sufficient variety could not be obtained to generalize to test data. 20 separate fonts were used to create this character set, providing a total of 520 characters. From this, 4 separate datasets were created, each with a different proportion of training/testing data items. The datasets are listed in table 2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Training Examples</th>
<th>Testing Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset 1</td>
<td>78</td>
<td>442</td>
</tr>
<tr>
<td>Dataset 2</td>
<td>208</td>
<td>312</td>
</tr>
<tr>
<td>Dataset 3</td>
<td>312</td>
<td>208</td>
</tr>
<tr>
<td>Dataset 4</td>
<td>390</td>
<td>130</td>
</tr>
</tbody>
</table>

*Table 2: Training/Testing Datasets*

Tests for generalization and comparison of algorithms will be performed using these datasets. They can be found on the CD as detailed in appendix A. The different proportions of the data sets are designed to simulate different conditions under which a network may operate – a particular training algorithm may perform better when there is limited training data, or when there is a large amount.
Chapter 4: Back-propagation Training

4.1: Overview

Back-propagation is a gradient descent-training algorithm, designed to modify the connection weights of a neural network to reduce the error in the observed output of a network given a specific training example. It is a standard training algorithm for neural networks, and has been the subject of much research. Faster versions, such as quickprop, have been developed. For purposes of comparing with evolutionary methods, standard back-propagation was used.

In back-propagation, the error found at the outputs is propagated back through the network to adjust the weights for the output layer and hidden layer. The error for a particular set of inputs is determined first, using the mean squared error formula shown in equation 3. The error doesn’t need to be divided by the number of outputs since the shape of the error curve is the important factor, not the actual values.

\[ E = \sum_{n=1}^{\text{outputs}} (D_n - O_n)^2 \]  

**Equation 3**

To adjust the weights of the network based on the MSE, it is necessary to calculate the gradient of the error function with respect to each network weight. The weight can then be adjusted slightly in the opposite direction to the gradient. The opposite direction is used so that we arrive at a lower point on the gradient surface. To calculate the error gradient, the derivative of the forward propagation equation, the sigmoid function, is needed. It is straightforward and shown in equation 4.
\[
\frac{d}{dx} \text{sgm}(x) = (1 - \text{sgm}(x)) \text{sgm}(x)
\]

Equation 4

The actual error gradient is the partial derivative of the error function with respect to each of the variables. The basic MSE error, when derived, will not give the error gradient shown. To derive the proper gradient function, it is necessary to replace \( O_n \) by the variables used to obtain it. \( O_n \) is merely a result of multiplications of the inputs of the network by the weights of the network. However, this results in complex formulas, and is well detailed in literature (reference to website), so it was not included here. The final error gradient expression is shown in equation 5. (Note that

\[
S^O = \sum_{m=1}^{\text{#hidden}} W_{mn}^O \text{ and } H_m = \text{outputOfHiddenLayerNeuron}. \]

\[
\nabla E_m^O = -2(D_n - O_n)(1 - \text{sgm}(S^O))\text{sgm}(S^O)H_m
\]

Equation 5

The gradient is calculated for each of the incoming weights for the output layer, according to the error obtained for each output neuron and the output of each hidden neuron. This gives a vector of error gradients corresponding to the vector of output layer weights. Before updating the output layer weights, the vector of error gradients with respect to the hidden layer weights must be calculated. This is based on the output layer weights, so it must be calculated before they are updated.
The update of the hidden layer weights is slightly more complicated than the update of the output layer weights. The equation becomes very long, so a new error variable was introduced as in equation 6.

\[
\delta_n^O = -2(D_n - O_n)(1 - \text{sgm}(S^O))\text{sgm}(S^O) \tag{Equation 6}
\]

The error gradient function for the hidden layer weights is shown in equation 7.

\[
\nabla E_h^m = ((1 - \text{sgm}(S^h))\text{sgm}(S^h)) \sum_{n=1}^{\text{#hidden}} \delta_n^O W_m^O I_1 \tag{Equation 7}
\]

Once the vector of error gradients have been calculated for both the hidden layer and the output layer, the weights themselves are updated. This is accomplished by multiplying each vector by a step size parameter, called the learning rate, and then adding the vector to the corresponding vector of network weights (hidden or output). This learning rate controls how much the network will change according to the current training example that it has been shown. This parameter must not be set too high, or the network will experience significant change given each training example. Rather, the network should experience a multitude of small changes that result in a network capable of remembering all examples and generalizing well to test data.

Due to the nature of the back-propagation training algorithm, it can get stuck at local minima in the solution space – solutions that are optimal compared to nearby solutions, but that are not the global optimal. This is a known problem with gradient
descent methods. If the network is re-randomized and training is begun again, then the search will hopefully avoid the sub optimal solution that it found in the first run. This is one of the main benefits of evolutionary methods over the back-propagation – they are not gradient descent methods, and are not subject to becoming trapped at local minima. There is also some danger of the network being over-fitted to the training data. That is, if the error of the network is made too small, the network will not be able to generalize to previously unseen data, as seen in section 4.3.

4.2: Implementation

The actual function for performing the back-propagation is contained in the file “backprop.h”. There is no separate class, as the algorithm deals directly with a single neural network. The function merely sets the network inputs, runs the network, and then performs back-propagation. A generation consists of processing a single character, and an epoch consists of processing all training characters. Every 10 epochs, information about the current performance of the network is output to the screen and to a log file.

The NeuralNet and Neuron classes perform much of the work of back-propagation. The NeuralNet will call a back-propagation in each neuron that calculates the error from the outputs, or output layer. These classes are contained in “NeuralNet.h” and “Neuron.h”, and can be found on the CD as detailed in appendix A.

4.3: Testing

The back-propagation method is extremely good at learning to recognize the given training cases. It learned to recognize more than 98% of training examples on
every single test run, and was quicker and more efficient than the other two methods. There is only one parameter that can be adjusted – the learning rate. This was tested for three separate values – 0.05, 0.10, and 0.15. Changing the learning rate does not significantly change the speed with which the network learns to recognize the training data. The performance on the test (unseen) data was considered to be the most important factor for the learning rate. A data set with 260 training examples and 1040 testing examples was used. Ideally, a much larger set would be used, but performance issues limited the ability to perform multiple tests on a large amount of data. Since the recognition of unseen data decreases after it reaches a peak point (because the network begins to be over-fitted on the training data), the tests were only run for 3000 generations each. The number of correct classifications of testing data are plotted in figure 2.

Figure 2: Back-propagation Learning Rate

4.4: Conclusion
The learning rate of 0.05 proved superior. Although a higher learning rate causes the network to recognize a greater number of unseen data faster, the lower learning rate results in a higher number of correct classifications a bit later in the training run. This is understandable – the higher learning rate causes the network to begin over-fitting to the training data sooner than the lower learning rate. A lower learning rate, of 0.02, does not achieve as high a success rate. For future tests, a learning rate of 0.05 will be used for back-propagation. This still allows the network to learn to recognize all training data very quickly, and ensures that the maximum possible amount of unseen data will be recognized.
Chapter 5: Fitness Functions

5.1: Overview

Both PSO and GA algorithms rely on fitness functions to determine the quality of an individual population member. A fitness function for a neural network relies on propagating training data through the network; the more training data the network classifies correctly, the higher the fitness.

Forward propagations are computationally expensive, so it is important to minimize the number required. Back-propagation uses only a single forward propagation per generation; after seeing a single training example, the weights of the network are updated. However, this isn’t practical in the case of genetic algorithms – an individual may have a high fitness when shown an “A”, but a low fitness when shown a “B”. Each generation, evolution will progress in a different direction, rendering it useless. In the case of particle swarm optimization, the global best position will be optimized for a certain character each generation, and the particles will likely not converge. Despite these misgivings, early attempts were made to implement this online learning paradigm, but it was found that populations did not converge.

As such, it was determined that the fitness function for these evolutionary algorithms could not depend on a single letter. It was surmised that the fitness must be based on the classification of all training examples. This becomes very slow when there are a large number of training examples, which is required for the networks to generalize to test data. Some attempts were made to use only samples of the data for training, as detailed in the description of fitness functions below. A sampled fitness function that
examines only part of the training data was successfully used for genetic algorithms, but not for particle swarm optimization.

5.2: Class Fitness

This is the first fitness function developed. It propagates all training data through the network, and compares the result of the network with the desired result for each piece of training data. It returns the number of correct classifications obtained by the network.

This fitness function did not provide enough resolution to drive the evolution towards an answer. There will be many different network configurations that will result in the same number of correct answers; either algorithm will have a difficult time evolving when there are many population members with the same fitness. This fitness function may be more appropriate when there are a greater number of test examples, but the slow speed of the function with many training examples limited the amount of testing.

5.3: Mean Squared Error Fitness

The mean squared error is as follows:

\[
MSE = \frac{1}{N} \sum (actualOutput - desiredOutput)^2
\]  
\textbf{Equation 8}

The mean squared error is calculated separately for each data item (letter). Each output node of the neural network is compared against the desired output for that node, and this is used to calculate the MSE. The fitness function returns the average MSE of all data items.

This method is slightly slower than the class fitness, since finding the MSE is computationally more expensive than finding the number of correct classifications. It
resulted in successful evolution in both particle swarm optimization and genetic algorithms, but they did not converge quickly. Through extensive observation of training runs, it was observed that a decrease in the MSE fitness sometimes resulted in the network obtaining fewer correct answers. This is understandable, as a better MSE does not always mean that the network will classify more items correctly. This was originally believed to be a problem, as it could take longer to learn a training set. However, the decreased MSE may result in better generalization ability.

5.4: Sampled Fitness

Several sampled fitness routines were developed and tested. These routines aimed to decrease the number of network propagations required by using a different subset of the training data for each generation. At least one letter of each classification is used to determine the fitness for each member, in order to avoid the problems associated with online learning (see section 5.1). Each generation, a different subset of training data is tested to guarantee that the evolved networks will recognize all training characters.

One sampled fitness function tested each letter one time, but chose each letter from a random alphabet in the training data. One attempted to test at least one letter from each alphabet. A third tested each letter n times from n random alphabets.

They did result in a considerable speedup in fitness evaluation times, particularly when there were a large number of training examples. However, neither evolutionary algorithm converged when using these fitness functions, even after as much as 20,000 generations. Much the same problem exists as with online learning. A single population
member might receive very different fitness evaluations from one generation to the next, so it is difficult for the algorithm to consistently evolve towards a single goal.

A successful sampled fitness function was developed for use with genetic algorithms. It simply tests a different character set from the training data each generation. If there are 5 alphabets, it will take 5 generations to test all training data. To find the fitness of each character set, it uses the combination fitness function described in section FF.5. This worked extremely well in the case of genetic algorithms – much better than all other fitness functions. It didn’t work for particle swarm optimization – please see chapter 7 for details.

5.5: Combination Fitness

The combination fitness combines both class fitness and MSE fitness. It calculates the number of incorrect answers the network provides for the training data, and adds the MSE to this number. Given enough training examples, after a few generations the number of incorrect answers is far greater than the MSE. This means that this fitness function values the number of incorrect answers more than the MSE. In the case where two population members report different numbers of incorrect answers, this fitness function usually prefers the member with a lower number of incorrect answers. This ensures that a network with a higher number of correct answers will usually be preferred. In the case where two population members have the same number of incorrect answers (and the same number of correct answers), this function prefers the member with a lower MSE. Thus, evolutionary drive exists even when two members of the population report the same number of incorrect answers.
This worked very well, and resulted in good convergence. It does not require more forward propagations than class fitness or MSE fitness, as the MSE and validity of the answer can both be calculated after one propagation. As propagations are the most time-consuming part of any fitness function, its’ running time is comparable to the other non-sampled fitness functions. It avoids the problems of both class fitness and MSE fitness, and allows the two training methods to easily learn small amounts of training data. In the case of genetic algorithms, it is inferior to the sampled fitness described above when a large amount of training data is used.

5.6: Conclusion

The best fitness function to use depends on the exact application. For learning a small amount of data (a single alphabet), the combination fitness function was superior. It was used in the parameter optimization tests for both particle swarm optimization and genetic algorithms. It provides consistent evolutionary drive, and usually ensures that networks with a higher number of correct answers are preferred. However, it is very slow – it must examine all training data to return a fitness value.

The sampled fitness function that examines a different complete character set from the training data each generation proved superior for genetic algorithms. It was not used when there was only one training alphabet, as it is the same as the combination fitness function. When a large amount of training data was used, it was vastly superior to the combination fitness function. It also performs much quicker than the combination fitness function.
The attempts at reducing the number of forward propagations for PSO via the sampled fitness functions were unsuccessful. It appears that all training data must be propagated each generation, which means that the running time of either evolutionary training method corresponds directly to the number of training examples used. This is certainly not ideal, as a large amount of training data must be used to ensure that the algorithm generalizes to previously unseen testing data. Optimization of training speed must be done elsewhere – perhaps in design of the training/testing data.
Chapter 6: Genetic Algorithms

6.1 Overview

A genetic algorithm is an evolutionary search method. The first evolutionary strategies came about in the 1950s and 1960s, and much research and work in the field has been done since then. John Holland created the first genetic algorithms in the 1960s, as attempt to study and apply natural adaptation to computers [Mitchell, 2002]. It attempts to apply the principles of biological evolution to function optimization or search problems. A population of potential solutions to a problem is maintained, with each solution being represented by a “chromosome”. The fittest members of the population are allowed to reproduce and create the next generation through the use of genetic operators such as mutation and crossover. Typically, chromosomes consisted of binary bit strings, with each bit representing a single gene. However, for more complicated applications, each gene can be represented by a real number, which would be complicated to encode into a bit string. Any chromosome representation can be used as long as appropriate genetic operators are defined. Each population member is randomized at the start of the genetic algorithm run. A selection operator chooses which members of the population are able to reproduce – the fitter members of the population will reproduce more often than less fit members. The fitness of each population member is specific to the problem – it is a measure of how “good” a potential solution to that problem is. It is important to note that the best members of the population are not the only members to reproduce – less fit members are allowed to reproduce because they may contain valuable genetic code that, when combined with other individuals, may produce fitter individuals. Each individual in the population may reproduce more than
once, or not at all. Selection is a stochastic process that doesn’t guarantee that the best members will reproduce. When two population members reproduce, they are combined via genetic operators to produce two offspring. One primary operator is crossover – it exchanges sub parts of the chromosomes, so that part of each child’s chromosome comes from each parent. It is designed to mimic biological recombination of DNA. The other primary operator is mutation – this randomly changes the value of the genes within each chromosome. It simulates biological mutation, and can result in the population trying very different solutions. Crossover will result in the population trying similar solutions that are recombinations of previous solutions. Many other genetic operators have been proposed and used, but these two are primary. Holland’s original genetic algorithm implementation also used an operator called “inversion”, but this is rarely used today.

Many problems require searching many possibilities to find an appropriate solution. Genetic algorithms are appealing because many problems require complex solutions that are difficult to develop and code. A genetic algorithm provides a bottom up method for solving solutions – very simple rules are created and then used to develop more complex solutions. This allows genetic algorithms to find solutions to problems that are very difficult to solve through conventional methods. They avoid certain problems of traditional search methods – most notably, the problem of local minima. Typical gradient descent search methods find themselves trapped at solutions that are not the global best, but are superior to all nearby solutions. Genetic algorithms are not subject to these problems – mutation may move the search to a vastly different solution space.
6.2: Implementation

A Chromosome class, contained in the file “chromosome.h”, represents the chromosomes used by the genetic algorithms. Each chromosome consists of an array of floating point numbers, dynamically sized on creation. The array represents the weights of the chromosome. Before use, a chromosome must be initialized with a neural network – this sizes the array and sets up appropriate variables. The chromosome class contains functions to implement crossover, mutation, and all implemented fitness functions. The selection methods used are all contained in the file “selection.h”. When the fitness functions are evaluated, each chromosome is read into a neural network, using a function defined in the NeuralNet class.

The functions that actually run the genetic algorithm are all contained in “GA.h”. An initial population is randomly initialized to provide a starting point. The standard genetic algorithm loop is then implemented. The fitness of each population member is determined, and the parents are chosen stochastically. The genetic operators of crossover and mutation, both contained in “GA.h”, are applied to create the next generation of individuals, which is copied over the current generation. Every 10 generations, information about the current state of the network is output to the screen and to a log file.

6.3: Notes on Parameter Optimization

Genetic algorithms have a large number of parameters available for change, and there are many different genetic operators, selection methods, etc. There is no well-defined way to pick a certain parameter set, so tests had to be conducted to find the best
parameters and operators for this particular implementation. Ideally, full tests would be conducted, measuring the training success rate and the generalization ability of each network. This would require many training examples and a lot of time – the processing power to do this was not available. As such, limited tests were done instead. A test alphabet was created by hand, with each character being 7x8 pixels in size. All optimization tests were run with this training alphabet, which is stored on the CD as detailed in appendix A. Only the ability of the network to learn the training set was measured, not the generalization ability (which would be minimal, considering there were only 26 training examples). Parameter sets that resulted in faster learning were considered superior.

6.4: Network to Chromosome Encoding

6.4.1: Overview

A major factor impacting the success of the genetic algorithm is the method of encoding a neural network into a chromosome. Many typical genetic algorithms use strings of bits to represent chromosomes. This implementation uses real numbers instead – each weight is a real number, and encoding it into a binary string would make for excessively long chromosomes. As it is, a neural network can have up to 20,000 weights, so it was important to keep the chromosome size as small as possible. Using real numbers for the chromosome is an accepted method of encoding for genetic algorithms. A chromosome consists of an array of floating point numbers. Two different encoding methods were tested, as explained below.
6.4.2: Feature Detector Encoding

For this encoding method, each hidden layer neuron was considered to be a feature detector, recognizing certain parts of each letter. Both incoming and outgoing weights for each hidden layer neuron are contained in a single gene in the chromosome. The incoming weights are copied (in order) into the gene, followed by the outgoing weights (which are actually incoming weights for the output layer). The number of genes in the chromosome will be equal to the number of hidden layer neurons. As the number of hidden layer neurons was kept as low as possible, chromosomes encoded via this method have a fairly low number of genes.

6.4.2: Neuronal Encoding

Each neuron in the hidden and output layers was encoded separately under this encoding method. The incoming weights for each neuron are copied, in order, into the gene. The hidden layer neurons are copied first, followed by the output layer neurons. This will result in a greater number of genes per chromosome than the feature-detector encoding method has. Since there are 26 classifications for each network, there will be 26 more genes – though the number of floating point numbers remains the same. This results in different behaviour under crossover and mutation.

6.4.3: Permutation Problem

Each hidden layer neuron is connected to every input layer neuron, and every output layer neuron. As such, the order of the hidden layer neurons does not matter – each will still contribute to the output in the exact same method, regardless of the
position. This means that neural nets that are similar in behaviour may have vastly
different chromosomes, preventing the successful recombination of building blocks.
There are actually $n!$ possible ways to encode a neural network with $n$ hidden units
[Hancock, 1992]. Units that are identical in the network may be encoded at different
places in the chromosome, so crossover may result in multiple copies of some units, and
no copies of other important units. This can result in two highly fit parents producing
offspring that have poor fitness. There hasn’t been much work done in resolving this
problem – theoretically, it would be possible to examine each parent chromosome for
identical genes, and pass those genes along with a high probability. However, Hancock
discusses ways of overcoming this problem in [Hancock, 1992] and concludes that the
permutation problem is not serious in practice. Due to his conclusion, no attempts were
made to address this issue.

6.4.5: Testing

Since the encoding method will affect the operation of the crossover and mutating
operators, testing must be done to determine which encoding method is superior. Testing
of the encoding method was not done until after all parameter testing was done, to ensure
that non-optimal parameters did not affect the results of the tests. The encoding method
was tested along with the genetic algorithm paradigm. The tests are contained in section
6.8.

6.5: Selection Methods

6.5.1: Overview
The process of selection is the process of choosing members of the population to create offspring for the next generation, and determining how many offspring each will have. Selection methods should emphasize fitter members of the population, with the intent that their offspring should be fitter still. However, it is important that less fit population members are selected to produce offspring as well – they may contain valuable genetic code. Allowing less fit individuals to reproduce ensures that there is genetic diversity in the population, and will allow a genetic algorithm to explore new solution spaces. Care must be taken with the selection method – if it emphasizes fitter individuals too much, the population will prematurely converge on a sub-optimal solution. If fitter members are not emphasized enough, the population remain diverse and never converge at all.

Elitism and steady-state genetic algorithms are closely related to selection methods. However, both these paradigms themselves use selection methods, so they have been details in section 6.7. Many early selection methods were fitness proportional, in which an individual’s expected number of offspring was proportional to the individual’s fitness divided by the total fitness. This was avoided due to major flaws with this method. Early in the search, a small number of highly fit individuals dominate and the population converges too quickly. Near the end of the search, the fitness of each individual is nearly the same, leaving no real evolutionary drive, as no individual is better. Two different selection methods were tested, as detailed below.

The fitness methods used are discussed separately in chapter 5.

6.5.2: Rank Selection
6.5.2.1: Overview

Rank selection was designed to prevent premature convergence of the population. Individuals are ranked according to their fitness, and their expected number of offspring are determined by their ranking rather than by absolute differences in fitness. This ensures that highly fit individuals don’t dominate early, and that selection pressure is maintained when the population has almost converged. The method used was originated by Baker and described in <GA Textbook Reference>.

Each individual is ranked in increasing order of fitness. The expected value of the highest ranked population member can be changed, and is called max. The expected value of the lowest ranked member is called min, and is set to (2-max). The expected value of each member i of a population of size N is given by equation 9.

\[ ExpVal(i) = \min + (\max - \min) \frac{\text{rank}(i) - 1}{N - 1} \]  

Equation 9

Once the expected value for all population members is known, stochastic universal sampling is used to sample the population.

A simple quicksort algorithm was used to sort the population, as this could be a potentially time-consuming operation.

6.5.2.2: Parameter Testing

The value of the max parameter significantly influences the operation of the algorithm. The higher the value of max, the more emphasis is placed on the fittest individuals in the population. Thus it will influence the speed of convergence of the
population. Several tests were performed to determine the best value of $max$. The original value suggested by Baker was 1.1. Higher values of 1.2, 1.4, and 1.7 were tested. Three tests were performed for each value, and the average was plotted, to ensure that the stochastic nature of the algorithm did not significantly influence the results. The results have been plotted in figure 3.

![Figure 3: RankSelection MAX Parameter](image)

The value of $max$ that performed best was 1.4 – it converged slower than the others at first, but ended up converging to a higher value. 1.7 was almost always inferior – this emphasizes fitness too much, and resulted in quick early gains and an eventual plateau. 1.2 converged to a high value quickly – this $max$ value may be better in
situations where the training time is limited. However, 1.4 was used for other parameter tests, as there is a full 2000 generations available for training in each.

6.5.3: Tournament Selection

6.5.3.1: Overview

Tournament selection is a common selection algorithm. It provides similar selection pressure to rank selection, but isn’t as computationally expensive as rank selection. A number of individuals are randomly chosen from the population to compete in a “tournament”. With a certain probability, the best individual is selected as a parent. If the best individual isn’t chosen, then a random member of the tournament participants is chosen to be a parent. A tournament is run for each potential parent. The probability that the best individual is chosen is a major factor for selection pressure, so tests were conducted to find the best value.

6.5.3.2: Pick Best Member Probability

A constant tournament size of 10 individuals was used for testing the probability that the best individual is chosen – this is 10% of the population size, and seemed a reasonable value. Values of 0.75, 0.85, and 0.95 were tested, as 0.75 seemed the minimum value that maintained good selection pressure. Three tests were conducted for each value to ensure that the stochastic nature of the algorithm didn’t affect the results too badly. The results have been plotted in figure 4.
A very high value (0.95) is obviously inferior – it converges on a sub-optimal value. Both other probabilities are comparable, but a probability of 0.75 seems to allow the algorithm to converge quicker. The increased population diversity benefited the algorithm, and a value of 0.75 was chosen for the probability that the fittest member of a tournament wins the tournament.

**SM.6.5.3.3: Tournament Size**

The size of each tournament is also a major player in the evolutionary drive. A smaller tournament size means that it is less likely that the tournament will include the best members of the population. Since there is always a fairly good chance that the population member with best fitness will win the tournament, a greater chance of the
fittest members being included means that the selection method emphasizes the fittest members of the population. Several different tournament sizes were tested; the average of three tests each is plotted on figure 5.

![Figure 5: Tournament Size](image)

The tests indicate that a tournament size of 20 individuals is superior – it converges significantly quicker than the other tournament sizes. A tournament size of 30 tends to lead to premature convergence – the graph plateaus several times and only classifies all examples after a large number of generations. As such, 20 individuals will be used for further testing.

6.6: Genetic Operators

6.6.1: Overview
In order for the genetic algorithm to work correctly, the offspring of each generation must be a recombination of the genetic code of the parent generation. If the fittest members are simply passed along to the next generation, innovation will never occur. As in a typical GA, two genetic operators are used here – crossover and mutation. When two parents “mate”, the crossover operator is applied to make one or two new children. Crossover doesn’t occur every time two parents mate – a probabilistic check is made, and if it does not pass, both parents survive unchanged to the next generation. Every offspring, whether the result of crossover or not, has some chance of randomly mutating. The mutation operation also probabilistically, but the chance of a mutation is much lower than the chance of crossover (less than 20% chance of mutation). These probabilities are subject to change, and tests will be performed to find the ideal values. Each paradigm (elitist, steady-state, non-elitist) will react best to different operators, and tests must be done separately.

6.6.2: Crossover

6.6.2.1: Overview

A simple uniform crossover is used for developing these neural networks. Crossover is handled slightly differently for the two different encoding methods discussed in section 6.4, but the principle is the same. For each gene in the chromosome, there is a 50% chance that it will come from the first parent, and a 50% chance that it will come from the second parent. It is important to note that a gene consists of all connection weights for a particular neuron in the network. This is discussed previously in section
6.4. It means that crossover does not result in innovation within particular neurons – it only rearranges and swaps them.

Several other crossover operators were tried. A simple crossover method that picks a single crossover point anywhere on the array of real numbers making up the chromosome was tried, but it did not seem to effectively lead the population to create more successful generations. An injection crossover method was also tried – a section of real numbers was taken from one parent and injected into the second parent to create the child. This seemed to perform roughly equally to the simple crossover method.

6.6.2.2: Crossover Probability

As noted above, crossover occurs with a certain probability. This probability is very important in controlling the rate of convergence and the amount of diversity in the population, so tests were conducted to find the optimal value. Crossover probabilities of 0.65, 0.75, 0.85, and 0.95 were tested, and the results are plotted in figure 6. Three tests for each value were performed due to the stochastic nature of the algorithm. Separate tests were run for the elitist and non-elitist paradigms, as they may well have different optimal crossover rates. The steady state paradigm does not use a crossover probability.
Figure 6: Elitist Crossover Rate

Figure 7: Non-Elistist Crossover Rate
For the elitist paradigm, a crossover rate of 0.85 allowed early gains, but resulted in a plateau as the algorithm progressed. The crossover rate of 0.75 showed slow steady progress that resulted in an eventual perfect convergence sooner than the others, and so will be used as the crossover rate for elitist genetic algorithms. For the non-elitist paradigm, 0.95 was superior in the middle, but again reached a sub-optimal plateau. 0.85 resulted in perfect convergence, and was chosen as the crossover rate for non-elitist.

6.6.3: Mutation

6.6.3.1: Overview

Mutation occurs with a relatively low probability, but it can cause major changes to the weights in a particular chromosome. Once a chromosome has been marked for mutation, each gene has a 40% chance of mutating. A gene is mutated by adding a random number (ranging from -1 to 1) to each weight within the gene. This is very similar to the mutation algorithm suggested in [Mitchell, 2002], but it gives each gene a chance of mutating instead of merely selecting a number of genes to mutate. As crossover keeps all connection weights of each neuron together, with changes occurring between neurons, mutation is the primary drive towards change in individual genes/neurons.

6.6.3.2: Mutation Rate Testing

The mutation rate is another important factor in the rate of convergence of the population. Separate mutation rate tests were run for each paradigm. Mutation rates of
0.04, 0.07, 0.10, and 0.15 were tested for each, and are graphed below on figures 8, 9, and 10.

**Figure 8: Elitist Mutation Rate**
Figure 9: Steady State Mutation Rate

Figure 10: Non-Elitist Mutation Rate
For the elitist paradigm, a high mutation rate of 0.20 was superior. This is understandable as the best members of each generation always survive to the next, allowing the high mutation rate to increase diversity without lowering the best fitness. The same applies to the non-elitist paradigm. The best mutation rate for the steady-state GA was found to be 0.15 – although the rate of 0.25 made good initial gains, it reached a plateau and was surpassed by 0.15.

6.7 Genetic Algorithm Paradigms

6.7.1: Overview

The paradigm is very closely related to the selection methods used – however, they are slightly different. The paradigm specifies how many members of each generation are replaced, and how they are replaced. Many texts discuss treat this as one and the same with selection methods, but I have chosen to treat it differently. All of these paradigms themselves use the different selection methods, just in different ways.

6.7.2: Non-Elitist

Under the non-elitist paradigm, all population members of each generation are replaced. If the probabilistic crossover check fails, a current member may survive to the next generation, but otherwise they will all be different.

6.7.3: Elitist
Under the elitist paradigm, the best n members of each generation automatically survive to the next generation. In this case, the best 6 members always survive. The other population members are all replaced unless surviving the crossover test.

### 6.7.4: Steady State

Under the steady state paradigm, only a small portion of the population is replaced. The n members with the highest fitness are used to create n offspring, which replace the n least-fit members of the current population. Standard genetic operators and selection methods are used, but only the top n individuals are available to be parents. In this case, n was set to 20 – much lower and each generation would see no significant change.

### 6.8: Final Testing

#### 6.8.1: Overview

The three most important factors in the performance of the genetic algorithm were determined to be the encoding method, the paradigm, and the selection method. The encoding/selection methods were tested for each paradigm, which determined the best encoding/selection method for each paradigm. The best of each paradigm were then tested against each other to determine the final genetic algorithm design.
6.8.2: Testing

The first round of tests tested the encoding/selection method for each paradigm, and can be found on figures 11, 12, and 13. Each line is labelled only with a test number, due to spacing constraints. Table 3 explains what each test is.

<table>
<thead>
<tr>
<th>Label</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1</td>
<td>Neuronal Encoding, Rank Selection</td>
</tr>
<tr>
<td>Test2</td>
<td>Neuronal Encoding, Tournament Selection</td>
</tr>
<tr>
<td>Test3</td>
<td>Feature-Detector Encoding, Rank Selection</td>
</tr>
<tr>
<td>Test4</td>
<td>Feature-Detector Encoding, Tournament Selection</td>
</tr>
</tbody>
</table>

Table 3: Paradigm Tests

Figure 11: Elitist Encoding/Selection
Figure 12: Steady State Encoding/Selection

Figure 13: Non-Elitist Encoding/Selection
For the elitist paradigm, it is obvious that tournament selection is superior, performing significantly better than rank selection. The encoding methods were similar in performance, but neuronal encoding gave slightly better performance.

For the steady state paradigm, it is equally obvious that rank selection is much superior to tournament selection. Feature-based encoding makes great progress early in the run, but is surpassed by neuronal-based encoding in the end.

For the non-elitist paradigm, things are not so clear. Neuronal-based tournament selection and feature-detector-based rank selection are extremely close in performance. Both will need to be tested in the final tests.

The four candidates just discussed for the final genetic algorithm implementation are listed in table 4, and plotted in figure 14.

<table>
<thead>
<tr>
<th>Label</th>
<th>Test</th>
<th>Test</th>
<th>Test3</th>
<th>Test3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1</td>
<td>Elitist, Neuronal Encoding, Tournament Selection</td>
<td>Steady State, Neuronal Encoding, Rank Selection</td>
<td>Non-Elitist, Neuronal Encoding, Tournament Selection</td>
<td>Non-Elitist, Feature-Detector Encoding, Rank Selection</td>
</tr>
</tbody>
</table>

Table FT2: Final Genetic Algorithm Implementation Candidates
The elitist tournament selection and non-elitist tournament selection tests performed much better than either other test. Their performance is very comparable. The non-elitist paradigm reaches perfection recognition sooner than the elitist paradigm. Tournament selection does not guarantee that the best member of each generation will be passed along. I initially surmised that this would mean the elitist paradigm would be superior, but this is apparently not the case. The increased genetic diversity of creating an entirely new generation must be beneficial with the limited population size in use. Final tests will be conducted using the non-elitist paradigm with tournament selection.

6.9: Conclusion
The non-elitist paradigm was found to be the paradigm that obtains perfect recognition of the training data the fastest. The elitist paradigm was comparable, but slightly slower, while the steady state paradigm lagged far behind. The network-to-chromosome encoding method was not a major factor in the performance of the genetic algorithm – while the neuronal encoding was slightly faster in all tests, the feature-detector encoding method never lagged far behind. Tournament selection was found to be the best selection method for the non-elitist paradigm – the rank selection method was only effective for the steady state paradigm. The tournament selection that was the most effective picked the tournament participant with the best fitness 75% of the time. The best tournament size was found to be 20 individuals – it is important to note that this is 20% of the population size. If a greater population size were used, it would likely be important to maintain the percentage of the population involved in each tournament, rather than maintain the absolute number. The best crossover probability was found to be 0.85, and the best mutation rate was 0.20. The mutation rate is fairly high – this is explained by the fact that crossover only chooses crossover points in between genes, so mutation is the primary drive towards genetic advancement within a single gene.

All these settings will be used for the final genetic algorithm design. The fitness functions will either be the combination fitness detailed in section 5.5, or the sampled fitness function detailed in section 5.4.
Chapter 7: Particle Swarm Optimization

7.1: Overview

Particle swarm optimization is based on artificial life techniques designed to simulate the behaviour of swarms of animals, and was first presented in a 1995 paper by Dr. Eberhart and Dr. Kennedy [Eberhart and Kennedy, 1995]. They originally intended to simulate the behaviour of flocks of birds. This was accomplished by creating a population of agents (“birds”), which moved in two-dimensional space and was attracted to a particular position (the global best position). Each agent stochastically matched its’ nearest neighbour’s velocity, and was also attracted to its’ personal best position (nearest to the global best position in the space). After finding that they could optimize simple functions with this method, they eliminated unneeded variables, and came up with what they call particle swarm optimization.

A particle swarm consists of a population of agents called “particles”. Each particle moves through n-dimensional space, and keeps track of its’ current position, best position found so far, and current velocity. Every generation, each particle’s velocity is updated according to its’ personal best position, and the best position found by all particles so far (the “global best”). The velocity update function is shown in equation 10.

\[
velocity = velocity + 2 \times rand() \times (personalBest - currentPos) + 2 \times rand() \times (globalBest - currentPos)
\]

**Equation 10: Particle Swarm Velocity Update**

The current position is then updated by simply adding the velocity to the current position. The global best position and personal best positions of each particle are then
updated. The original position and velocity of the particles were randomized.

Conceptually, these particles “fly” through hyperspace, being attracted to the global best and their personal best positions. When a superior solution is found, the particles will converge, and in doing so they will explore the space surrounding the solution. The velocity of a particle will allow it to “fly” past the best solution and explore solutions in unknown space, even while it’s velocity is being adjusted to return back to the current optimal solution.

In their original paper, Dr. Eberhard and Dr. Kennedy applied this technique to optimizing artificial neural networks. They found that particle swarm optimization was comparable to back-propagation for training neural networks, and was even superior in some respects. This is why particle swarm optimization was chosen as a training method for this project. However, it is important to note that their neural networks were very limited – the network they describe in their paper used only 13 connection weights. The neural networks used in here have anywhere from 1600 to 22000 connection weights, so it is yet to be seen if particle swarm optimization is comparable to back-propagation in a non-trivial context. My early tests of particle swarm optimization showed that they did indeed converge very quickly networks with approximately 400 weights.

Particle swarm optimization is beneficial in several ways. It is an extremely simple concept, and can be implemented without complex data structures. No complex or costly mathematical functions are used, and it doesn’t require a great amount of memory. Tests have shown that particle swarm optimization can be superior to back-propagation in generalizing to previously unseen test data. The tendency of particles to
“overfly” the global and personal best solutions leads to the algorithm exploring new space, and avoid the problem of being stuck on local minima.

**7.2: Implementation**

The particle swarm optimization implementation consists of the “Particle” class and the “ParticleSwarm” class. Each particle maintains its’ current position, current velocity, lowest personal fitness, and personal best position (i.e. historical position with the lowest fitness). The particle swarm maintains the global lowest fitness and best position found. Each generation, the position of each particle is updated with the particle’s velocity, and the global and personal best positions/fitnesses are updated. Each particle’s velocity is then updated according to formula 10, found in the particle swarm optimization overview. Due to the nature of the combination fitness function, when all training examples are classified correctly the global best fitness will be less than 1. The training loop stops when the number of generations specified has passed.

All particles start at a random position in the range [-1.0,1.0] for each dimension. The velocity of each particle is randomized to a small value to provide initial random impetus to the swarm. The population size was limited to 100 particles. Unlike the genetic algorithm, the scheme used to encode neural network weights into a particle is not important. Each dimension of the particle is changed independently from the others, so the order of the weights is not important.

**7.3: Maximum Velocity Decay**
Original tests were performed without any maximum velocity, and it was found that these swarms did not converge in a reasonable amount of time (2000 generations). A maximum velocity of 0.5 was introduced, and the swarms would sometimes converge. However, they often got stuck at a certain fitness level without being able to progress to a greater number of correct classifications. I hypothesized that the particles may be circling around a correct optimum location, but are not able to converge because they have attained relatively high velocity.

In order to resolve this issue, I introduced a maximum velocity decay function. The idea is that the particles converge fairly quickly on a good solution space. Once they have converged on this space, a lower maximum velocity will force the particles to explore the space with smaller steps. Solutions that are closer together will be considered by the particles. Thus they will converge to a smaller solution space, at which point the maximum velocity is reduced again. Eventually this will result in correct convergence. This was initially believed to be an original idea, but after all decay functions were created and tested, references to maximum velocity were found in some literature [Eberhart and Shi, 1998]. However, this literature used a constant maximum velocity, along with inertia weights (which were not implemented). The decay of maximum velocity may still remain an original idea.

There were many possible velocity decay functions. Tests were conducted without any decay function and with four different decay functions (see section 7.5). The first decay function monitors the number of correct classifications in the global best solution. If 100 generations pass without an increase in this number, the velocity is halved. The second decay function halves the speed every 500 generations. The third
halves the speed every 250 generations. The fourth decay function monitors the global best fitness, and will halve the speed if this does not decrease within 100 generations. This is similar to the first, but is sensitive to changes in the MSE as well as changes in the number of correct answers.

The first and fourth decay functions may reduce the speed before the population has found a good solution space, if the population is stuck for some time at a certain fitness level. This will prevent the swarm from exploring other, potentially better, solution spaces. To counter this, if the maximum velocity drops below a preset value, it is reset to the original value. This ensures that the swarm will explore new spaces if the current space has turned out to be unproductive.

7.4: Fitness Functions

The combination fitness function discussed in section 5.5 was used to evaluate the fitness of each population member. This requires one forward propagation through the network for each training item per population member, which is extremely computationally expensive. When a sampled fitness function was used (section 5.4), particle swarm optimization did not converge on a good solution. After the first few generations, the global best fitness did not change whatsoever. The particles would converge on the best position for a single alphabet, and ignore the others. The algorithm was changed such that it maintained a separate best fitness and best position for each character set within the training data. Each generation, the particles would be attracted to their own personal best fitness, as well as the average of all the global best fitnesses. Again, after a few generations, the best fitness for each character set didn’t change – the
particles were unable to find better solutions than they had already found. The necessity of using an unsampled fitness function means that particle swarm optimization runs much slower than back-propagation or the sampled fitness genetic algorithm.

7.5: Testing

There are many changeable parameters in the particle swarm optimization algorithm. A constant population size of 100 particles was use – while a bigger population may be beneficial, the computational time required to run the algorithm with a higher population size limits testing ability. The velocity update function was also constant, and in accordance with the function described in [Eberhart and Kennedy, 1995]. The primary testing was done on the velocity decay function, as it seemed to drastically affect the performance of the algorithm. The maximum and initial velocities were also tested.

All tests were done with the minimal alphabet created by hand. Each letter in this alphabet is only 7x8 pixels, so training is sufficiently fast to do a large number of tests. Only 20 hidden units were used -- this proved sufficient to learn to recognize the entire alphabet. The different test runs are compared based on their ability to learn the training set. As there is only one alphabet to train on, the networks created are unlikely to recognize a significant amount of unseen data. However, this is not considered a problem, as these tests were run specifically to optimize parameters. Correct convergence was desired above all else.

It is important to note that the particle swarm optimization algorithm is randomized -- the initial starting positions are random, as are the velocity updates.
Sometimes a certain parameter set will converge, sometimes not. All graphs are the result of the average of three test runs for each parameter set.

### 7.5.1: Test Set 1

The first tests measured all decay functions against each other, with different starting and maximum velocities. Early tests showed that both the starting and maximum velocities greatly affect the ability to converge. Figure 15 shows tests with a maximum velocity of 1.0, and a starting velocity ranging from –1.0 to 1.0. Figure 16 shows tests with a maximum velocity of 0.5, and a starting velocity ranging from –0.5 to 0.5. Figure 19 shows tests with a maximum velocity of 0.5, and a starting velocity ranging from -0.125 to 0.125. The decay functions are numbered on the figures, according to table 5.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Decrease max velocity after 100 similar # of correct answers</td>
</tr>
<tr>
<td>2</td>
<td>Decrease max velocity after 500 generations</td>
</tr>
<tr>
<td>3</td>
<td>Decrease max velocity after 250 generations</td>
</tr>
<tr>
<td>4</td>
<td>Decrease max velocity after 100 similar best fitness</td>
</tr>
<tr>
<td>5</td>
<td>No max velocity decrease</td>
</tr>
</tbody>
</table>

*Table 5: Particle Swarm Velocity Decay Functions*
Figure 15: 1.0 Maximum Velocity, [-1.0,1.0] Starting Velocity

Figure 16: 0.5 Maximum Velocity, [-0.5,0.5] Starting Velocity
These tests show several important things. When the starting/max velocities were 1.0, the algorithm performed very poorly – none of the decay methods caused the network to converge. This velocity is too high to allow convergence in a reasonable amount of time. It is obvious that the particle swarm optimization tests without any maximum velocity decay performed much worse than all others. The hypothesis in section 7.3 has proved to be correct. The first and third decay functions perform best.

7.5.1: Test Set 2

Three parameter settings are superior to the rest, yet comparable to each other. With a starting/maximum velocity of 0.5, the first decay function is superior. With a starting velocity in the range [-0.125,0.125], the first and third decay functions are superior. Plotted against each other, they are very close in performance. As such, a second round of testing was done specifically to compare these three parameter settings. Five tests each are run, to try and ensure that the randomness of the algorithm does not influence the results. The results have been plotted in 18. The first number in the graph legend represents the maximum starting velocity –
“2” represents a maximum starting speed of 0.5, while “3” represents a maximum starting speed of 0.125. The second number in the graph legend represents the velocity decay function as detailed in Table Particle Swarm Optimization.1.

![Figure 18: Final Particle Swarm Design Testing](image)

7.6: Conclusion

The best velocity decay function was found to be the function that automatically reduces the velocity after 250 generations. It is important to note that this is so when the algorithm is run for 250 generations. When training with the intent of generalizing to unknown data, the particle swarm optimization may be run for many more generations. If this were the case, after 2000 generations the maximum velocity would get so small that little progress could be made towards a correct solution. Instead of using 250 generations, a percentage of the total running time should be used – in this case, 12.5%. 
Generalization tests performed in later sections of the report showed that this was not always the best method. The velocity decay function that reduces the speed after 100 similar fitnesses was also found to be extremely effective, as well as the velocity decay function that reduces the speed after 12.5% of the total running time. When testing generalization ability, there were many more training examples, which explains why it did not perform quite so well here. With few training examples, the number correct is more important than the MSE, but when training with many examples, a decrease in the MSE may be important as it may lead to an increased number of correct answers in the next few generations.
Chapter 8: Generalization

8.1: Overview

The true power of a neural networks lies in its’ ability to generalize to previously unseen data. Even when a network classifies training examples almost perfectly, there is still a fair bit of error in the network outputs. The output with the highest value is considered to be the classification of the network, but other outputs may also have high values. This error allows the network to generalize – instead of learning the specific training examples; it is learning the structures of the training examples. That is, it may learn to detect the features of each character presented, rather than learning the peculiarities of the characters it has already seen. If the network has been shown sufficient different examples of each character, the differences between them will be ignored, and the structure will be recognized. The network will recognize a letter “O” because it is a circle, while the letter “Q” will be recognized as a circle with a small tail on the bottom right. However, the network will not be able to recognize characters that are vastly different. For example, all fonts used for training were not cursive – the characters were oriented vertically, and didn’t have much noise distorting them. If a trained network is shown cursive fonts, its’ recognition will be much lower than for non-cursive fonts. If the network had been trained using cursive fonts as well, it would be able to recognize both types. Please note that cursive fonts were not used – the processing power was not available to train the networks on the number of characters that would have been required for both non-cursive and cursive fonts.

Care must be taken to avoid over-fitting. This occurs when the error on the training data becomes too low – the network has memorized the training examples too
well to generalize to unseen data. This will occur if the network has been trained for too long. In the back-propagation tests of section 8.3, the accuracy on unseen data clearly declines after a certain optimal point. A primary method for avoiding over-fitting is to minimize the number of weights in the network. There should be sufficient weights to learn the training examples, but not many more. The additional weights provide too much opportunity for the network to simply memorize the data. Since the number of inputs and outputs are fixed, this means that the number of hidden nodes must be minimized. The structure of the network can also be optimized such that not all nodes connect to every node in the next layer. Several methods have been developed to avoid over-fitting. One primary method is early stopping [Caruana et al., 2000]. When the network reaches an optimal performance for unseen data, training is stopped. The optimal point can be determined via an algorithm that monitors the progress of the training. Early stopping can overcome the disadvantages of excess hidden units. This was not implemented for this project – evolutionary methods took so long to train that they did not encounter this problem, and back-propagation was run for an equal number of generations.

8.2: Implementation

The method for testing the generalization of a neural network was simple. A training function was written to implement the training method chosen. The networks were trained using the datasets described in section 3.3. These datasets consists of both training and testing character sets. The network was trained as normal, but every 10 generations/epochs, the performance on unseen data was tested. Every character in the
testing dataset was propagated through the network, and the number of correct
classifications was output to the screen and to a file. The network then continued
learning using the training data. This allowed graphs to be produced showing the
generalization performance as compared to the accuracy on the training data, which was
also logged every 10 generations/epochs.

8.3: Testing

The four datasets introduced in section 3.3 were used. Each dataset consists of
the same character sets, but they are separated into training/testing data in different
proportions. Early tests showed that back-propagation was superior to evolutionary
methods. However, when there was limited training data available, their generalization
performance was more comparable. Thus, the datasets range from using 15% of
available data to using 75% of available data. The results of the four datasets have been
plotted on separate graphs, shown on figures 19-23. Five different training methods were
tested against each other. For purposes of the graphs, they are numbered as shown in
table 6. The two particle swarm tests used different maximum velocity decay functions,
as listed in table 5, chapter 7. Each graph shows a training method’s classification
success for training and testing data in the same color – the success rate for training is
always the higher line. All tests were run for 7500 generations.

<table>
<thead>
<tr>
<th>Number</th>
<th>Training Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>Genetic Algorithm, Combination Fitness</td>
</tr>
<tr>
<td>(2)</td>
<td>Genetic Algorithm, Sampled Fitness</td>
</tr>
<tr>
<td>(3)</td>
<td>Particle Swarm Optimization, Decay</td>
</tr>
<tr>
<td>Function 3</td>
<td>Function 2</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>(4) Particle Swarm optimization, Decay</td>
<td>(BP) Back-propagation</td>
</tr>
</tbody>
</table>

Table 6: Generalization Tests

Figure 19: Dataset 1 Generalization
Figure 20: Generalization Dataset 2

Figure 21: Generalization Dataset 3
8.4: Conclusion

For all datasets, it is apparent that back-propagation far outperforms the two evolutionary methods for learning the training data. Back-propagation learns the data almost perfectly in a very small number of generations. This faster performance is to be expected when using a large number of network weights – back-propagation learns based directly on the observed error, while the two evolutionary methods are essentially random searches.

Back-propagation is also the most effective at generalizing to unseen data for all datasets. It clearly suffers from over-fitting after a very short time – a few hundred epochs after the network has reached near-perfect recognition on the training data, the accuracy on unseen data decreases. This could be detected and prevented by the methods mentioned in section 8.1. If back-propagation is run for an equal amount of time as the
evolutionary methods, the generalization ability becomes comparable. This is particularly true in the case of the genetic algorithm using a sampled fitness function. In all tests, this training method’s generalization ability approached that of back-propagation at the end of the run. While the generalization success of back-propagation was decreasing at the end of the run, the success of the sampled genetic algorithm was almost always still increasing. This suggests that a network trained using this genetic algorithm may very well be comparable to the best performance of a network trained using back-propagation, if sufficient training time is available.

Of the evolutionary methods, the genetic algorithm using a sampled fitness function is clearly the best performer. The non-sampled genetic algorithm did not perform well at all; in the second two tests, it plateaued early and at a low level. In all tests, it was inferior to the particle swarm optimization techniques. The generalization ability generally stayed below 20%. The two particle swarm training methods were close behind the sampled genetic algorithms, but they also tended to reach a maximum level and not increase. The velocity decay function itself may explain this. While it is extremely beneficial for early convergence, it may limit the ability of the network to explore new space near the end of the run. The decay functions reset the speed if it dropped below a certain level, in an attempt to prevent this from occurring. Perhaps better would have been to start with a higher maximum velocity, to allow the particles to explore a bigger solution space before their velocities are slowed. The technique may also have benefited from a higher minimum velocity.
References


Chapter 10: Conclusions and Future Directions

Back-propagation performed better than both particle swarm optimization and genetic algorithms. It trained a network to recognize the training data more accurately and faster than either evolutionary method. The generalization ability was also superior – given a very large amount of training data, it approached 90% accuracy on previously unseen data. However, both evolutionary methods showed great promise. The genetic algorithm using a sampled fitness function was very nearly equal to back-propagation when training networks of a fairly small size. It did not suffer from the problem of over-fitting, though this may be due to the fact that it didn’t often approach 100% accuracy on the training examples, as back-propagation did. Particle swarm optimization training didn’t perform quite as well as either other training methods. However, its’ performance was significantly increased by using improved fitness functions, and by introducing a maximum velocity and a maximum velocity decay function.

The computation cost of evolutionary computing proved to be a major obstacle in the attempts to improve the performance to the level of back-propagation. Since the fitness of a population of individuals has to be evaluated each generation, many more forward propagations through the network are required for evolutionary training methods. This means that the population size had to be kept low, the size of the characters had to be kept small, and the number of training examples also had to be relatively small. Back-propagation did not suffer from this as it trains on only a single network. The two evolutionary training methods would reap many benefits from having more computing power available, particularly if parallel processing was used. Each individual, or a group of individuals, could reside on a separate processor. This would make the training time
comparable to back-propagation. With greater computing power, a greater population size could also be used. Tests showed that the evolutionary methods performed much better with smaller networks, likely due to the limited population size. A greater population size could allow these training methods to be much more successful when there are a large number of parameters to optimize. This larger size, combined with a speedup in processing, may very well make the two evolutionary methods comparable to back-propagation in all respects.

There are many possible future directions from this work. Ideally, the particle swarm training would be further optimized. It should be possible to develop a sampled fitness function that does not depend on propagating all training data each generation. This would greatly increase the speed, and if the genetic algorithm fitness functions were any indication, it would also greatly improve the accuracy. The concept of inertia could also be introduced, as discussed in [Eberhart and Shi, 1998]. The performance of the genetic algorithm is very good when using a sampled fitness function. Further genetic operators could be investigated – perhaps some different crossover or mutation operators would have a beneficial effect.

There has been some discussion in literature of using evolutionary methods to train a network and provide a starting point for back-propagation. In all tests, the accuracy of the evolutionary methods increased very slowly after many generations. It would beneficial to train a network to the point where the evolutionary methods begin to slow, and then use this as a starting point to run a back-propagation training algorithm. This helps to avoid the problem of local minima that any gradient descent algorithm
suffers from. Evolutionary methods can also be used to optimize the structure of a network.
Appendix A: CD Contents

Updates.txt – Any last minute updates to CD contents

/data
  /utilities
    /RawConverter
      Rawconverter program
    /WhiteSpace
      WhiteSpace program
    /TTMaker
      Train/Test Maker program
  /font image files
    Images used to make character sets
  /character sets
    Character sets used to make data
  /generalization data sets
    Datasets used for generalization tests
  /parameter character sets
    Datasets used for GA/PSO optimization
  /scaleability
    Datasets used for scaleability tests

/nn
  /backprop.h
    Backpropagation training methods
  /chromosome.h
    Chromosome class for GA
  /fileReader.h
    .dat File Reading Routines
  /GA.h
    Genetic algorithm training methods
  /NeuralNet.h
    Neural network class
  /Neuron.h
    Neuron class
  /particleSwarm.h
    Particle/ParticleSwarm classes
  /quicksort.h
    Quicksort implementation for GA
  /selection.h
    Selection methods for GA
  /nn.cpp
    Main file to launch all training methods