Abstract

This project looks at combining Neural Networks and Genetic Algorithms to classify 5 letters namely A, B, C, D and E from the letter recognition dataset of the UCI machine learning repository with 20000 patterns. Genetic Algorithm is used to generate an optimal set of weights to train the Neural Network using simple backpropogation algorithm. An error threshold of 0.00693221 has been obtained when ran the backpropagation algorithm for 50 epochs with 50 generations.

Acknowledgements

I would like to thank the course lecturer, Dr. Napoleon Reyes, who gave us a challenging project which helped us to understand the importance of Machine Learning Algorithms like Neural Networks and Genetic Algorithms to solve a real-world problem of Letter Recognition. I would also like to acknowledge Bobby Anguelov for the C++ Implementation of Neural Network and Mat Buckland for the implementation of a Genetic Algorithm.

List of Figures

List of Tables

1. Introduction

There are many different types of neural networks and techniques for training but the classic one is back propagation neural network (BPNN). The back propagation refers to the fact that any mistakes made by the network during training get sent backwards through it in an attempt to correct it and so teach the network what's right and wrong.

1.1 Backpropogation algorithm

The BPNN learns during a training epoch. It will probably go through several epochs before the network has sufficiently learnt to handle all the data and the end result is satisfactory. A training epoch is described below:

For each input entry in the training data set:

- check output against desired value and feed back error (back-propagate)

Where back-propagation consists of:

- calculate error gradients
- update weights

Step 1 Put one of the patterns to be learned on the input units feed input data in (feed forward)
Step 2 Find the values for the hidden and the output unit check output against desired value and feed back error (back-propagate)
Step 3 Find out how large the error is on the output unit calculate error gradients
Step 4 Use one of the backpropogation formulas to adjust the weights leading into the output unit update weights
Step 5 Use another formula to find out errors for the hidden layer unit calculate error gradients
Step 6 Adjust the weights into the hidden layer unit via another formula update weights
Step 7 Repeat steps 1to 6 for all the patterns

Figure 1 Iterative minimization of error over training set

The procedure described in Fig. 1 is also referred to as **Iterative minimization of error over training set.**

1.1.1 Activation Functions

An activation function is a function used at each neural processing unit to generate the output signal from the weighted average of inputs. Most common is the sigmoid function.

Table 1 above gives a list of activation functions that can be used to generate the output at hidden and output layers.

Activation Function	$g(a)$	$g'(a)$
Linear	$g(a) = a$	$g'(a) = 1$
Logistic	$g(a) = \frac{1}{1+e^{-a}}$	$g'(a) = g(a)(1-g(a))$
Hyperbolic-tangent	$g(a) = \tanh(a)$	$g'(a) = \operatorname{sech}^2(a) = 1-\tanh^2(a)$
Squash	$g(a) = \frac{a}{1+\lvert a\rvert}$	$g'(a) = \frac{1}{(1+\lvert a\rvert)^2} = (1-\lvert g(a)\rvert)^2$

Table 1 Activation Functions and Their Derivatives

Note1: Any function that is differentiable can be used as an activation function. Sometimes referred to as the transfer/squashing function.

Note2: An epoch happens when all patterns have been trained. Usually, several epochs will be required to train the network to achieve a desired accuracy.

1.1.2 Learning rate and Momentum

Learning Rate and Momentum affects how fast and/or well a BPNN learns.

Momentum is a technique used to speed up the training of a BPN. Remember that the weight update is a move along the gradient of the activation function in the direction specified by the output error. Momentum is just that, it basically keeps you moving in the direction of the previous step. This also has the added bonus that when you change direction you don't immediately jump in the opposite direction but your initial step is a small one. Basically if you overshoot you've missed your ideal point and you don't wish to overshoot it again and so momentum helps to prevent that.

The only change to the implementation is in regards to the weight updates, remember from part one that the weights updates were as follows:

$$w_{ij} = w_{ij} + \Delta w_{ij} \text{ and } w_{jk} = w_{jk} + \Delta w_{jk}$$

where $\Delta w_{ij}(t) = \alpha . inputNeuron_i . \delta_j$ and $\Delta w_{jk}(t) = \alpha . hiddenNeuron_j . \delta_k$

$\alpha - learning\ rate$

$\delta - error\ gradient$

Those weight updates now become:

$$\Delta w_{ij}(t) = \alpha . inputNeuron_i . \delta_j + \beta . \Delta w_{ij}(t-1)$$

$$\Delta w_{jk}(t) = \alpha . hiddenNeuron_j . \delta_k + \beta . \Delta w_{jk}(t-1)$$

$where\ \beta - momentum\ constant$

Figure 2 The generalized delta rule

Momentum is usually set to a high value between 0 and 1. You'll notice that with momentum set to 0 the weight updates are identical to original ones. The effect of this momentum term is shown below for our training problem with the learning rate set to 0.01 and the momentum set to 0.9.

The momentum formula's as shown in Fig. 2 above are also known as **the generalized delta rule.**

1.1.3 Batch / Online learning

Stochastic (Online) learning occurs when the neuron weights are updated after each individual piece of data is passed through the system. The FFNN therefore changes with every piece of data and is in a constant state of change during training.

Batch Learning on the other hand stores each neuron weight change when it occurs, and only at the end of each epoch does it update the weights with the net change over the training set. This means the neural network will only update once at the end of each epoch.

1.1.4 Training the Neural Network

Fig. 3 below shows a step-by-step procedure for training a multi-layer feed-forward backpropogation neural network:

1. **Given:** Data set, desired outputs and a Neural Net with m weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

2. Split data set (randomly) into three subsets:
   - Training set – used for picking weights
   - Validation set – used to stop training
   - Generalization (Test) set – used to evaluate performance

3. Pick random, small weights as initial values.

4. Perform iterative minimization of error over training set.

5. Stop when error on validation set reaches a minimum (to avoid overfitting).

6. Repeat training (from Step 2) several times (to avoid local minima).

7. Use best weights to compute error on test set, which is estimate of performance on new data. Do not repeat training to improve this.

Figure 3 Training a Neural Network

Now we come to an important point that people often overlook, every time you train the network the data sets are different (if you shuffled the initial dataset as recommended earlier) and the weights are random, so obviously your results will differ each time you train the network. Usually the only difference is the number of epochs required to reach the NN's accuracy ceiling.

1.2 Genetic Algorithm (GA)

A genetic algorithm is an adaptation procedure based on the mechanics of natural selection and genetics.

Genetic Algorithms (GAs) have 2 main components:

- A genetic representation of the solution domain
- A fitness function to evaluate the solution domain

1. Initialize the algorithm. Randomly initialize each individual chromosome in the population of size N (N must be even), and compute each individual's fitness.

2. Select N/2 pairs of individuals for crossover. The probability that an individual will be selected for crossover is proportional to its fitness.

3. Perform crossover operation on N/2 pairs selected in Step1. Randomly mutate bits with a small probability during this operation.

4. Compute fitness of all individuals in new population.

5. (Optional Optimization) Select N fittest individuals from combined population of size 2N consisting of old and new populations pooled together.

6. (Optional Optimization) Rescale fitness of population.

7. Determine maximum fitness of individuals in population.

   If |max fitness – optimum fitness| < tolerance

   Then Stop

   Else

   Go to Step1.

Figure 4 Genetic Algorithm

Fig. 4 can be explained more formally as follows:

At the beginning of a run of a genetic algorithm a large population of chromosomes is created. Each one, when decoded will represent a different solution to the problem at hand. Let's say there are *N* chromosomes in the initial population. Then, the following steps are repeated until a solution is found:

- Test each chromosome to see how good it is at solving the problem at hand and assign a ***fitness score*** accordingly. The fitness score is a measure of how good that chromosome is at solving the problem to hand.

- Select two members from the current population. The chance of being selected is proportional to the chromosomes fitness. ***Roulette wheel*** selection is a commonly used method.

- Dependent on the ***crossover rate*** crossover the bits from each chosen chromosome at a randomly chosen point.

- Step through the chosen chromosomes bits and flip dependent on the ***mutation rate***.

- Repeat steps 2, 3, 4 until a new population of N members has been created.

Genetic Algorithms have been found to be robust and practical optimization methods. In a genetic algorithm a possible solution of the problem under consideration is represented by a chromosome. In the initialization step of the algorithm a set of chromosomes is created randomly. The actual set of chromosomes is called the population. A fitness function is defined to represent the quality of the solution given by a chromosome. Only the chromosomes with the highest values of this fitness function are allowed to reproduce. In the reproduction phase new chromosomes are created by fusing information of two existing chromosome (crossover) and by randomly changing them (mutation). Finally the chromosomes with the lowest values of the fitness function are

removed. This reproduction and elimination step is repeated until a predefined termination condition is become true.

1.2.1 Roulette Wheel Selection

This is a way of choosing members from the population of chromosomes in a way that is proportional to their fitness. It does not guarantee that the fittest member goes through to the next generation, merely that it has a very good chance of doing so. It works like this:

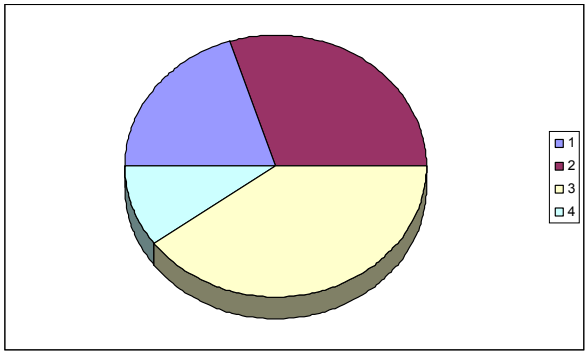

Figure 5 Roulette Wheel Selection

Imagine that the population's total fitness score is represented by a pie chart as shown in Fig. 5, or roulette wheel. Now you assign a slice of the wheel to each member of the population. The size of the slice is proportional to that chromosomes fitness score. i.e. the fitter a member is the bigger the slice of pie it gets. Now, to choose a chromosome all you have to do is spin the ball and grab the chromosome at the point it stops.

1.2.2 Crossover

Crossover is performed by selecting a random gene along the length of the chromosomes and swapping all the genes after that point. This is dependent on the Crossover Rate which is simply the chance that two chromosomes will swap their bits. A good value for this is around 0.7.

e.g. Given two chromosomes (green and red) as shown in Fig. 6

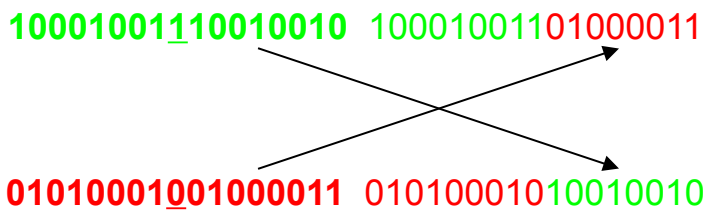

Figure 6 Single-point crossover

Choose a random bit along the length, say at position 9, and swap all the bits after that point. This point is referred to as the Single-point crossover.

This function crosses over two members with a certain probability at a single random point.

1.2.3 Mutation

This is the chance that a bit within a chromosome will be flipped (0 becomes 1, 1 becomes 0) as can be seen from Fig.7. This is usually a very low value for binary encoded genes, say 0.001.

So whenever chromosomes are chosen from the population the algorithm first checks to see if crossover should be applied and then the algorithm iterates down the length of each chromosome mutating the bits if applicable.

Mutation based on probability is used:

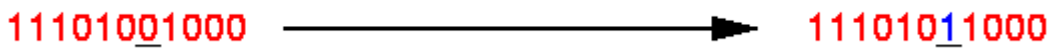

Figure 7 Mutation

Each bit in the bit-string chromosomes has an equal and very low probability of mutating.

1.3 Combining Neural Networks and Genetic Algorithms (GA-NN)

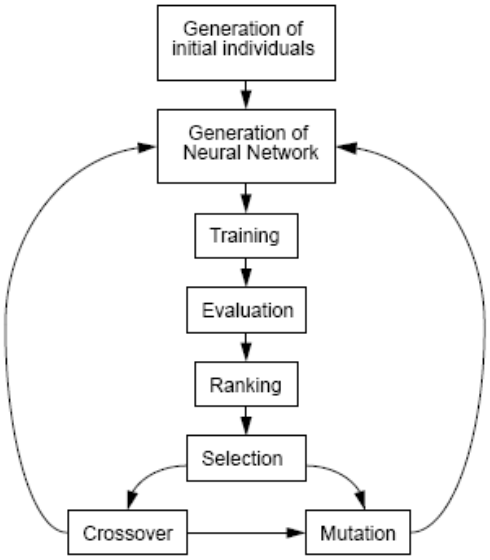

Figure 8 GA-NN

Fig. 8 shows the process of a combined genetic algorithm GA and backpropagation approach for neural network training. The GA selects the initial set of weights by applying genetic operators. The theory goes that a global GA search is combined with a local BP search for a more effective training procedure.

The full algorithm is as follows:

- Generate an initial population with random weights of values within a defined range between -1 and 1

- Repeat for G generations

    1. Evaluate the individuals of the population - train them using a Neural Network and assign a fitness value based on the mean squared error.
    2. Select the N fittest individuals and apply a mating or crossover function. Doing this guarantees diversity.
    3. Replace the unfit individuals with new ones

- Use the best individuals to obtain the testing error

2. Problem Description

We want to find a set of optimal weights for use in a back-propagation neural network by using a genetic algorithm to reach an error tolerance below 0.007 when recognizing letters A, B, C, D, E on unseen data.

2.1 Letter Recognition Dataset

The Letter Recognition Dataset [4,5] consists of 20,000 patterns of 26 capital letters from A-Z in the English alphabet. This dataset is used as a benchmark for testing new and improving old machine learning algorithms.

2.1.1 Formatting Dataset

The letter recognition dataset [4] from the UCI machine learning repository is used for training. The first thing is to format the dataset, in our case we had a list of 16 attributes and the letter those attributes represent. To make it easy to understand, all letters were converted to a binary representation as shown in Table 2.

Letter	Binary
A	10000
B	01000
C	00100
D	00010
E	00001
F-Z	00000

Table 2 Encoding Scheme For Target

Therefore, the neural network's architecture is now 16 input neurons and 5 output neurons and the value for the hidden neuron will have to be determined by trial and error but is used as 10. It must be noted that letters other than A, B, C, D and E are not recognized. So an input pattern representing these letters could be any letter between F-Z.

Please refer to Section 4.1 for a C++ implementation.

2.1.2 Training Dataset

So in our case we have this massive data set of 20000 patterns (entries), we can't just stick all of the data into our network since the network will learn that data and we have no way of checking how well the network will do with unseen data. This problem is referred to as over-fitting, Basically the network starts remembering the input data and will fail to correctly handle unseen data.

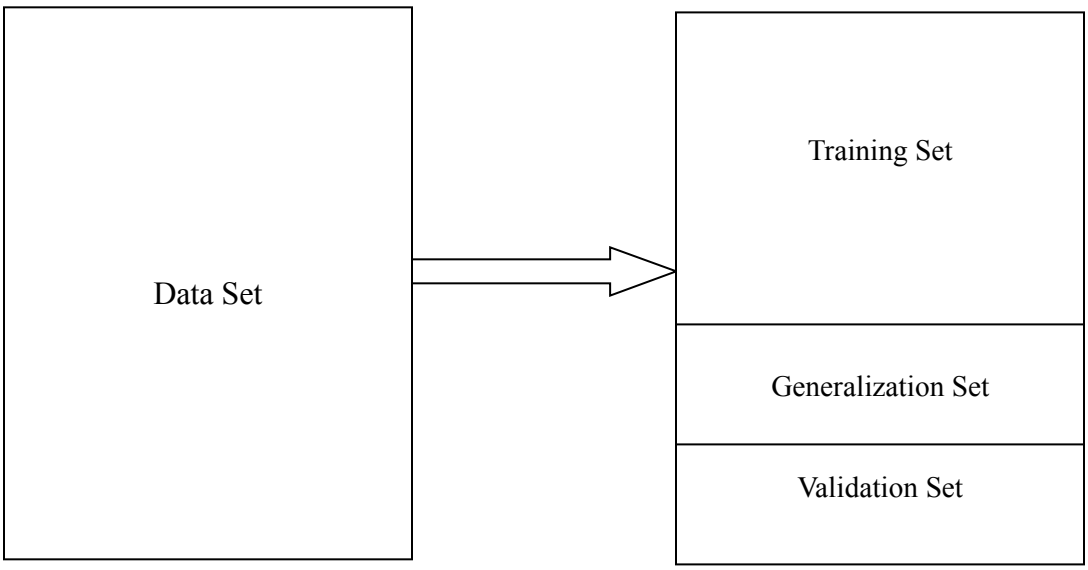

Figure 9 Splitting of Data Set

So we don't want the network to memorize the input data, so obviously we'll have to separate our training set. Classically we split the data set into three parts: the training data, the generalization data and the validation set as shown in Fig. 9. It is also often recommended to shuffle the initial dataset before splitting it to ensure that your data sets are different each time.

- Training set is what we used to train the network and update the weights with so it must be the largest chunk.
- Validation set will be run though the neural network once training has completed (i.e. The stopping conditions have been met), this gives us our final validation error.
- Generalization set is data that we'll run through the NN at the end of training to see how well the network manages to handle unseen data.

The classic split of the dataset is 60%, 20%, 20% for the training, validation and generalization data sets respectively. The training set is used to train the network, so for every piece of data (pattern) in the training set the following occurs:

- Feed pattern through NN
- Check errors and calculate error gradients
- Update weights according to error gradients (back propagation)

Once we have processed all the patterns in the training data set, the process begins again from the start.

3. Design

3.1 NN Design

Multi-layer feedforward network is the architecture chosen for our Neural Network as shown in Fig.10 with 16 Input Nodes, 10 Hidden Nodes and 5 Output Nodes. It also has 2 bias nodes, one for the input layer and the other for the hidden layer.

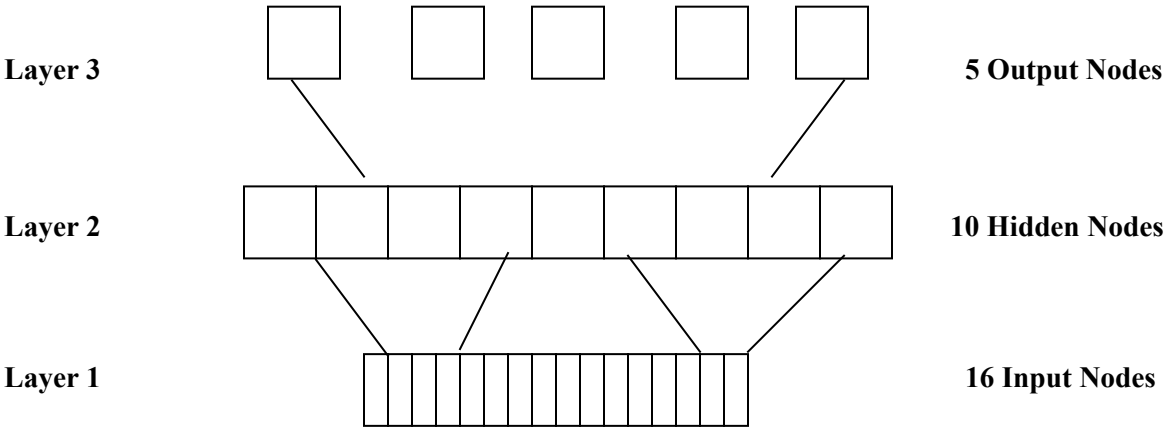

Figure 10 Neural Network Architecture

3.1.1 Initialization of Neural Network

The weights between the layers are randomly initialized using a Genetic Algorithm. These are set to small values between in the range of [-1, 1]. The whole point of setting the initialization range and using a genetic algorithm is to reduce the number of epochs required for training and converge to a solution a lot faster.

3.1.2 Stopping Conditions

The following are several measures that could be used to decide when to stop training:

- **Maximum Epochs Reached** – The first measure is really easy, all it means is that the NN will stop once a set number of epochs have elapsed.
- **Training Set Accuracy** – This is the number of correctly classified patterns over the total number of patterns in the training set for an epoch. Obviously the higher the accuracy the better. But remember that this is the accuracy on previously seen patterns so you can use this alone to stop your training.
- **Generalization Set Accuracy** – This is the number of correctly classified patterns over the total number of patterns in the generalization set for an epoch. This gets calculated at the end of an epoch once all the weight changes have been completed. This represents the accuracy of the network in dealing with unseen patterns. Again you can't use this measure alone since this could have a much higher accuracy that the training set error.
- **Training Set Sum Of Squared Error (SSE)** – This is the sum of the squared errors (desired – actual) for each pattern in the training set as shown in Fig. 11.
- **Validation Set Sum Of Squared Error (SSE)** – This is the sum of the squared errors (desired – actual) for each pattern in the validation set.
- **Training Set Mean Squared Error (MSE)** – This is the average of the sum of the squared errors (desired – actual) for each pattern in the training set as shown in Fig. 12.
- **Validation Set Mean Squared Error (MSE)** – This is the average of the sum of the squared errors (desired – actual) for each pattern in the validation set.

$$E = \frac{1}{2}\sum_{p}\left(\sum_{k}(t_{pk} - o_{pk})^2\right)$$

Figure 11 Sum Of Squared Error

$$MSE = \frac{\sum_{i=0}^{n}(desired\ value\ i - actual\ value\ i)^2}{n}$$
$$where\ n = number\ of\ patterns\ in\ set$$

Figure 12 Mean Squared Error

Both MSE and SSE give a more detailed measure of the current networks accuracy, the smaller the MSE or SSE the better the network performs.

Initial experimets, suggested that MSE takes a very long time to converge in terms of the computation involved, therefore SSE was used.

So now we have these measures so how do we stop the network, well what I use is both the training and validation SSE's in addition to a maximum epoch. So I'll stop once both my training and validation set SSE's are below some value. This way all bases will be covered.

3.1.3 NN Setup

The following parameters and their values were set for training the Neural Network:

- LEARNING_RATE = 0.01
- MOMENTUM = 0.9
- MAX_EPOCHS = 30
- DESIRED_SSE = 0.01

The Sum of Squared error (SSE) is a measure with a much higher resolution so is used for stopping conditions and as a measure of fitness for a chromosome

3.2 GA Design

Our genetic algorithm population is basic and standard. Its number of chromosomes and size of each chromosome can vary, but should be kept large (50-100) for better results. In addition, the probability of crossover and probability of mutation can be specified.

3.2.1 Encoding and Decoding

First we need to encode a possible solution as a string of bits - a chromosome. The representation of a set of weights for the individual classifiers by a chromosome is straightforward. Each chromosome is represented by an array of real numbers between -1 and 1. 9 bits are used to encode a chromosome where the 8 bits are used as weight encoding bits and the 9^{th} bit is used as an index it which determines the sign of the value. This 8-bit binary representation is converted to a decimal value and is multipled by a scaling factor which scales the value bweteen -1 and 1.

3.2.2 Choosing a Fitness Function

This can be the most difficult part of the algorithm to figure out. It really depends on what problem you are trying to solve but the general idea is to give a higher fitness score the closer a chromosome comes to solving the problem. A fitness value of 999.0f is assigned to the chromosome when fitness which is the sum of squared errors for that chromosome obtained from the Neural Network is below a certain error threshold of 0.006 otherwise a fitness value that's inversely proportional to the difference between the desired value and the target value a decoded chromosome represents.

3.3 Overall GA-NN Design

The following values constitute a combined GA-NN setup:

- CROSSOVER_RATE = 0.7
- MUTATION_RATE = 0.0333
- POP_SIZE = 50
- CHROMO_LENGTH = 2025
- GENE_LENGTH = 9
- MAX_ALLOWABLE_GENERATIONS = 100
- LEARNING_RATE = 0.01
- MOMENTUM = 0.9
- MAX_EPOCHS = 30
- DESIRED_SSE = 0.01

4. C++ Implementation

The neural network code was adapted from work by Bobby Anguelov [3]. It uses a standard sigmoid activation function, and allows to specify the number of input, hidden, and output units, as well as the learning rate and momentum for each data set it trains on.

The genetic algorithm code was adapted from work by Mat Buckland [6]. It allows to specify the population size, length of the chromosome, probability of crossover and mutation and selection of individuals using Roulette Wheel.

4.1 Formatting Data

The following program employs unary encoding for recognizing letters A, B, C, D and E:

```
#include <iostream>
#include <fstream>
#include <string>
#include <math.h>
#include <algorithm>
#include <cctype>
#include <iomanip>
#include <vector>
#include <iterator>
#include <stdlib.h>
#include <string.h>

using namespace std;

void processLine( string &line );

ofstream outputFile1;

int countForLetterA = 0;
int countForLetterB = 0;
int countForLetterC = 0;
int countForLetterD = 0;
int countForLetterE = 0;

int main() {
	int i=0;

	ifstream inputFile;
	ofstream outputFile;

	inputFile.open("dataset.txt", ios::in);
	outputFile1.open("ABCDE.csv", ios::out);

	string line;
	if ( inputFile.is_open() )
	{
		while ( getline(inputFile, line)) {

			//process line
			processLine(line);

			char *cstr = new char[line.size()+1];
			strcpy(cstr, line.c_str()); // creates a c string equivalent
			switch(cstr[0]) {

				case 'A': outputFile1 << "1" << "," << "0" << "," << "0" << ","
<< "0" << "," << "0" << endl;
					countForLetterA++;
					break;
```

```
                                        case 'B': outputFile1 << "0" << "," << "1" << "," << "0" << ","
<< "0" << "," << "0" << endl;
                                                                countForLetterB++;
                                                                break;
                                        case 'C': outputFile1 << "0" << "," << "0" << "," << "1" << ","
<< "0" << "," << "0" << endl;
                                                                countForLetterC++;
                                                                break;
                                        case 'D': outputFile1 << "0" << "," << "0" << "," << "0" << ","
<< "1" << "," << "0" << endl;
                                                                countForLetterD++;
                                                                break;
                                        case 'E': outputFile1 << "0" << "," << "0" << "," << "0" << ","
<< "0" << "," << "1" << endl;
                                                                countForLetterE++;

                                                                break;
                                        default : outputFile1 << "0" << "," << "0" << "," << "0" << "," <<
"0" << "," << "0" << endl;
                                                                break;

                                }
                        }
                        inputFile.close();
                }
                else
                {
                        cout << "Error Opening Input File: " << "dataset.csv" << endl;
                        return false;
                }

 cout << "Finished writing file" << endl;
 cout << "======================" << endl;
 cout << "Count for Letter A = " << countForLetterA << endl;
cout << "Count for Letter B = " << countForLetterB << endl;
cout << "Count for Letter C = " << countForLetterC << endl;
cout << "Count for Letter D = " << countForLetterD << endl;
cout << "Count for Letter E = " << countForLetterE << endl;


outputFile1.close();
}

/*************************************************************************
* Processes a single line from the data file
*************************************************************************/
void processLine( string &line )
{
        int* pattern = new int[17];
        char *cstr = new char[line.size()+1];
        char* t;

        strcpy(cstr, line.c_str()); // creates a c string equivalent

        //tokenise
        int i = 1;
```

```
        t=strtok (cstr,",");

        while ( t!=NULL && i < 17)
        {

                if(i < 17) {

                        t = strtok(NULL,","); //move token onwards
                                pattern[i] = atoi(t);
                                outputFile1 << pattern[i] << "," ;
                }

                i++;
        }
}
```

The following is the output of the program:

// Output

Finished writing file

Count for Letter A = 789

Count for Letter B = 766

Count for Letter C = 736

Count for Letter D = 805

Count for Letter E = 768

4.1.1 Splitting

The dataset is split into 3 separate datasets as follows:

```
// split data set
trainingDataEndIndex = (int) ( 0.6 * data.size() );
int gSize = (int) ( ceil(0.2 * data.size()) );
int vSize = (int) ( data.size() - trainingDataEndIndex - gSize );
```

4.1.2 Shuffling

Shuffling is done by using the random_shuffle function

```
// shuffle data
srand(time(0));
random_shuffle(data.begin(), data.end());
```

4.2 NN

A NN is implemented by 2 main classes:

1. neuralNetwork
2. neuralNetworkTrainer

The *neuralNetwork* class defines the number of nodes for the input, hidden and output layers, weights associated with the layers and a standard sigmoid activation function. The class has the ability of loading weights from a file, saving weights to a file, executing the feedforward operation and clamping the output when for a given input pattern the output is less than 0.1 and greater than 0.9 is treated as a 0 and 1 respectively. This is then used to calculate the acuuracy and the sum of squared error for recognizing letters A, B, C, D and E.

The *neuralNetworkTrainer* class as the name suggests is used for training the NN using backpropagation. It allows one to specify the learning rate (LR), momentum, maximum number of epochs and the desiredAccuracy. Also, error gradients at the output and the hidden layers are calculated and then backpropagated using the backpropagate function. It also has the ability of making use of batch learning.

4.2.1 Feed Forward Operation

```
/*****************************************************************
* Feed Forward Operation
*****************************************************************/
void neuralNetwork::feedForward(double* pattern)
{
	//set input neurons to input values
	for(int i = 0; i < nInput; i++) inputNeurons[i] = pattern[i];

	//Calculate Hidden Layer values - include bias neuron
	//---------------------------------------------------------------------------------------------
	for(int j=0; j < nHidden; j++)
	{
		//clear value
		hiddenNeurons[j] = 0;

		//get weighted sum of pattern and bias neuron
		for( int i=0; i <= nInput; i++ ) hiddenNeurons[j] += inputNeurons[i] *
wInputHidden[i][j];
		//set to result of sigmoid
		hiddenNeurons[j] = activationFunction( hiddenNeurons[j] );
	}
	//Calculating Output Layer values - include bias neuron
	//---------------------------------------------------------------------------------------------
	for(int k=0; k < nOutput; k++)
	{
		//clear value
		outputNeurons[k] = 0;

		//get weighted sum of pattern and bias neuron
		for( int j=0; j <= nHidden; j++ ) outputNeurons[k] += hiddenNeurons[j] *
wHiddenOutput[j][k];

		//set to result of sigmoid
		outputNeurons[k] = activationFunction( outputNeurons[k] );
```

```
        }
}
```

4.2.2 Backpropogation

```
/*****************************************************************
* Propagate errors back through NN and calculate delta values
*****************************************************************/
void neuralNetworkTrainer::backpropagate( double* desiredOutputs )
{
        //modify deltas between hidden and output layers
        //------------------------------------------------------------------------------------------------
        for (int k = 0; k < NN->nOutput; k++)
        {
                //get error gradient for every output node
                outputErrorGradients[k] = getOutputErrorGradient( desiredOutputs[k], NN->outputNeurons[k] );

                //for all nodes in hidden layer and bias neuron
                for (int j = 0; j <= NN->nHidden; j++)
                {
                        //calculate change in weight
                        if ( !useBatch ) deltaHiddenOutput[j][k] = learningRate * NN->hiddenNeurons[j] * outputErrorGradients[k] + momentum * deltaHiddenOutput[j][k];
                        else deltaHiddenOutput[j][k] += learningRate * NN->hiddenNeurons[j] * outputErrorGradients[k];
                }
        }

        //modify deltas between input and hidden layers
        //------------------------------------------------------------------------------------------------
        for (int j = 0; j < NN->nHidden; j++)
        {
                //get error gradient for every hidden node
                hiddenErrorGradients[j] = getHiddenErrorGradient( j );

                //for all nodes in input layer and bias neuron
                for (int i = 0; i <= NN->nInput; i++)
```

```
                {
                        //calculate change in weight
                        if ( !useBatch ) deltaInputHidden[i][j] = learningRate * NN->inputNeurons[i]
* hiddenErrorGradients[j] + momentum * deltaInputHidden[i][j];
                        else deltaInputHidden[i][j] += learningRate * NN->inputNeurons[i] *
hiddenErrorGradients[j];

                }
        }

        //if using stochastic learning update the weights immediately
        if ( !useBatch ) updateWeights();
}
```

4.2.3 Train the NN

```
/*****************************************************************
* Train the NN using gradient descent
*****************************************************************/
double neuralNetworkTrainer::trainNetwork( trainingDataSet* tSet )
{

        clock_t startTime, endTime;
        cout      << endl << "Neural Network Training Starting: " << endl
                        <<
"==========================================================================
===========" << endl
                        << "LR: " << learningRate << ", Momentum: " << momentum << ", Max
Epochs: " << maxEpochs << endl
                        << NN->nInput << " Input Neurons: " << NN->nHidden << " Hidden
Neurons: " << NN->nOutput << " Output Neurons: " << endl
                        <<
"==========================================================================
===========" << endl << endl;

        //reset epoch and log counters
        epoch = 0;
```

```
	lastEpochLogged = -logResolution;
	startTime = clock();
	//train network using training dataset for training and stop training trainging using both the training and the validation dataset

	while ( epoch < maxEpochs )
	{
		//use training set to train network
		trainingSetSSE = runTrainingEpoch( tSet->trainingSet );

		//get validation set accuracy and SSE
	  validationSetAccuracy = NN->getSetAccuracy(tSet->validationSet);
	  validationSetSSE = NN->getSetSSE(tSet->validationSet);

		cout << "Epoch :" << epoch;
		cout << " TSet Acc:" << trainingSetAccuracy << "%, SSE: " << trainingSetSSE ;
		cout << " VSet Acc:" << validationSetAccuracy << "%, SSE: " << validationSetSSE << endl;


		//Log Training results
		if ( loggingEnabled && logFile.is_open() && ( epoch - lastEpochLogged == logResolution ) )
		{
			logFile << epoch << "," << trainingSetSSE << "," << validationSetSSE << endl;
			lastEpochLogged = epoch;
		}

		//print out change in training / validation mse
		if ( trainingSetSSE <= desiredAccuracy && validationSetSSE <= desiredAccuracy)
		{
			desiredAccuracy /= 2; // successively lower the error threshold
			cout << "=====================================" << endl;
			cout << "Lowering the error threshold to " << desiredAccuracy << endl;
```

```
                cout << "=====================================" << endl;
                if(trainingSetSSE < 0.007 && validationSetSSE < 0.007) { // stop when error threshold below 0.007 is reached
                        break;
                } break;
                }
        }
        //~ else {
                //~ learningRate /= 2; // successively lowering the learning rate if desired training and validation error level not reached
        //~ }

        //once training set is complete increment epoch
        epoch++;
    }//end while

  cout << endl;
    endTime = clock();
    cout << "Time taken for backpropagation training = " << (endTime - startTime)/1000/60 << " minutes" << endl;

    //get training set accuracy and SSE
    trainingSetAccuracy = NN->getSetAccuracy( tSet->trainingSet );
    trainingSetSSE = NN->getSetSSE( tSet->trainingSet );

    //get validation set accuracy and SSE
    validationSetAccuracy = NN->getSetAccuracy(tSet->validationSet);
    validationSetSSE = NN->getSetSSE(tSet->validationSet);

    //out validation accuracy and MSE
    cout << endl << "Training Complete!!! - > Elapsed Epochs: " << epoch << endl;

    return trainingSetSSE;
}
```

```
/*****************************************************************
* Run a single training epoch
*****************************************************************/
double neuralNetworkTrainer::runTrainingEpoch( vector<dataEntry*> trainingSet )
{
	//incorrect patterns
	double incorrectPatterns = 0;
	double sse = 0;



	//for every training pattern
	for ( int tp = 0; tp < (int) trainingSet.size(); tp++)
	{
		//feed inputs through network and backpropagate errors
		NN->feedForward( trainingSet[tp]->pattern );
		backpropagate( trainingSet[tp]->target );

		//pattern correct flag
		bool patternCorrect = true;

		//check all outputs from neural network against desired values
		for ( int k = 0; k < NN->nOutput; k++ )
		{
			//int temp = NN->clampOutput( NN->outputNeurons[k] ) ;
			//
		  //out << temp << "," << trainingSet[tp]->target[k] << endl;
			//pattern incorrect if desired and output differ
			if ( NN->clampOutput( NN->outputNeurons[k] ) != trainingSet[tp]->target[k] )
{
				patternCorrect = false;

			}
			//calculate SSE
			sse +=  pow(( NN->outputNeurons[k] - trainingSet[tp]->target[k] ), 2);
```

```
                }


                //if pattern is incorrect add to incorrect count
                if ( !patternCorrect ) incorrectPatterns++;

        }//end for

        //if using batch learning - update the weights
        if ( useBatch ) updateWeights();

        //update training accuracy and SSE
        trainingSetAccuracy = 100 - (incorrectPatterns/trainingSet.size() * 100);
        trainingSetSSE = (0.5 * sse) / ( NN->nOutput * trainingSet.size() );

        return trainingSetSSE;

}
```

4.3 GA

The Genetic Algorithm is implemented by a structure *chromo_typ* which holds the chromosome (a possible solution) and its fitness.

```
struct chromo_typ
{
  //the binary bit string is held in a std::string
  string  bits;
  float     fitness;

chromo_typ(): bits(""), fitness(0.0f){};
chromo_typ(string bts, float ftns): bits(bts), fitness(ftns){};
};
```

4.3.1 Generation of Chromosomes.

```
//storage for our population of chromosomes.
        chromo_typ Population[POP_SIZE];
        //first create a random population, all with zero fitness.
        for (int i=0; i<POP_SIZE; i++)
        {
                Population[i].bits = GetRandomBits(CHROMO_LENGTH);
                Population[i].fitness = 0.0f;
        }
//----------------------------------GetRandomBits-----------------------------------------
//      This function returns a string of random 1s and 0s of the desired length./
//----------------------------------------------------------------------------------------
string GetRandomBits(int length)
{
        string bits;

        for (int i=0; i<length; i++)
        {
                if (RANDOM_NUM1 > 0.5f)

                        bits += "1";

                else

                        bits += "0";
        }
```

```
        return bits;
}
```

4.3.2 Roulette Wheel Selection

```
//---------------------------------Roulette-------------------------------------------
//      selects a chromosome from the population via roulette wheel selection
//------------------------------------------------------------------------------------
string Roulette(double total_fitness, chromo_typ* Population)
{
        //generate a random number between 0 & total fitness count
        double Slice = (RANDOM_NUM1 * total_fitness);

        //go through the chromosones adding up the fitness so far
        double FitnessSoFar = 0.0f;

        for (int i=0; i<POP_SIZE; i++)
        {
                FitnessSoFar += Population[i].fitness;

                //if the fitness so far > random number return the chromo at this point
                if (FitnessSoFar >= Slice)

                        return Population[i].bits;

                else return Population[i].bits;
         }
}
```

4.3.3 Crossover

```
//---------------------------------- Crossover ----------------------------------------
//
//  Dependent on the CROSSOVER_RATE this function selects a random point along the
//  length of the chromosomes and swaps all the  bits after that point.
//------------------------------------------------------------------------------------
string Crossover(string &offspring1, string &offspring2)
{
  //create a random crossover point not at the boundary
  int crossover = (int) (RANDOM_NUM1 * (CHROMO_LENGTH-1));

  //dependent on the crossover rate
  if (RANDOM_NUM1 <= CROSSOVER_RATE)
  {

    string t1 = offspring1.substr(0, crossover) + offspring2.substr(crossover, CHROMO_LENGTH-1);
    string t2 = offspring2.substr(0, crossover) + offspring1.substr(crossover, CHROMO_LENGTH-1);

    offspring1 = t1;
    offspring2 = t2;

  }
}
```

4.3.4 Mutation

```
//-------------------------------------Mutate----------------------------------------
//
//          Mutates a chromosome's bits dependent on the MUTATION_RATE
//-------------------------------------------------------------------------------------
void Mutate(string &bits)
{
          //cout << "Mutate" << endl;
          for (int i=0; i<bits.length(); i++)
          {
                    if (RANDOM_NUM1 < MUTATION_RATE)
                    {
                              if (bits.at(i) == '1')

                                        bits.at(i) = '0';

                              else

                                        bits.at(i) = '1';
                    }
          }

          return;
}
```

4.3.5 Assigning Fitness

Based on the sse obtained from the trainng set on a chromosome, an error threshold is chosen say 0.006 and a fitness value of 999.0f is assigned if they are equal otherwise a fitness which is inversely proportional to their difference is assigned.

```
//---------------------------------AssignFitness--------------------------------------
// given a string of bits and a target value this function will calculate its
// representation and return a fitness score accordingly
//------------------------------------------------------------------------------------
double AssignFitness(string bits, double desiredValue, double targetValue)
{
        //holds float values of gene sequence
        float buffer[(int)(CHROMO_LENGTH / GENE_LENGTH)];

        int num_weights = ParseBits(bits, buffer);
        int i;
        // ok, we have a buffer filled with valid values of weights
        // now we calculate what this represents.

        // Now we calculate the fitness. First check to see if a solution has been found
        // and assign an arbitarily high fitness score if this is so.

        if ( desiredValue == ceil(targetValue)) {
                return 999.0f;
        }
        else {
                //cout << "ok" << endl;
                return (1/fabs(desiredValue-ceil(targetValue)));
        }
  }
```

5. Testing and Evaluation

5.1 Preliminary Results

There are many parameters that affect the performance of the neural network. Therefore, some initial testing was testing to get good learning rate and momentum values.

Effect of Scaling Factor

Initally, scaling factor was set to 0.002 which resulted no change in SSE. Scaling factor was then change to 0.0002 which resulted in much faster changes to SSE.

Effect of LR:

No. of epochs = 20

As can be seen from Fig. 13, when the learning rate is set below 0.02, SSE takes a dip and goes back up when set below 0.009. Therefore, LR was set to 0.009 for experimentation purposes.

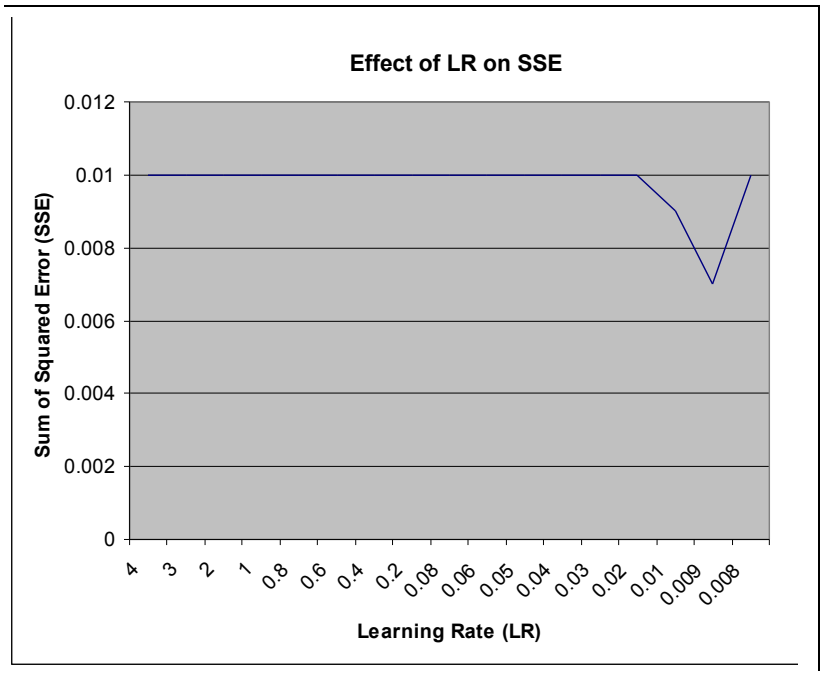

Figure 13 Effect of LR on SSE

Effect of Momentum

As you can see from the Fig. 14 and 15, the BPNN converges a lot faster with momentum added than without it and it also has the added benefit of allowing the back-propagation to avoid local minima's in the search space and to traverse areas where the error space doesn't change.

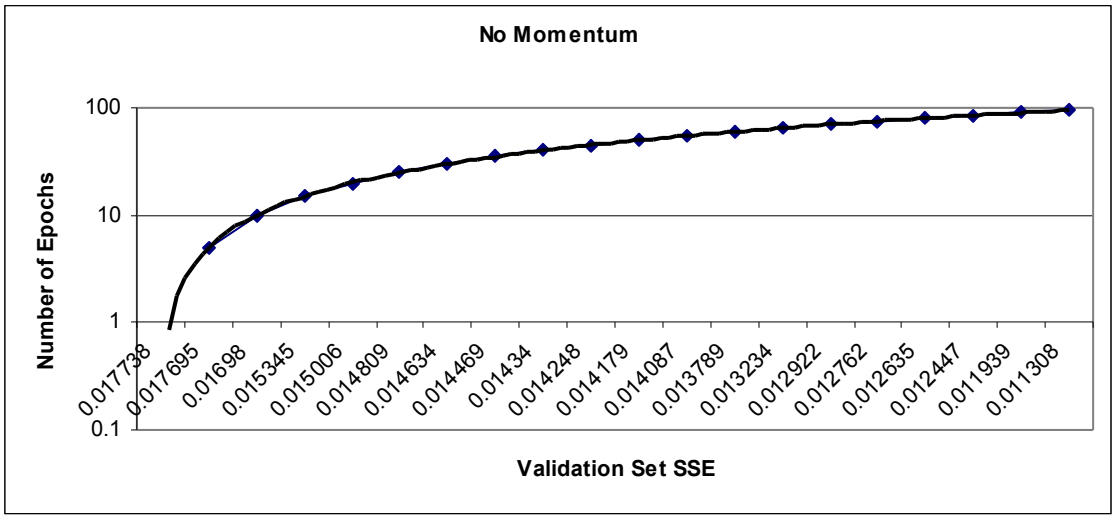

Figure 14 Effect of No Momentum on Validation Set SSE

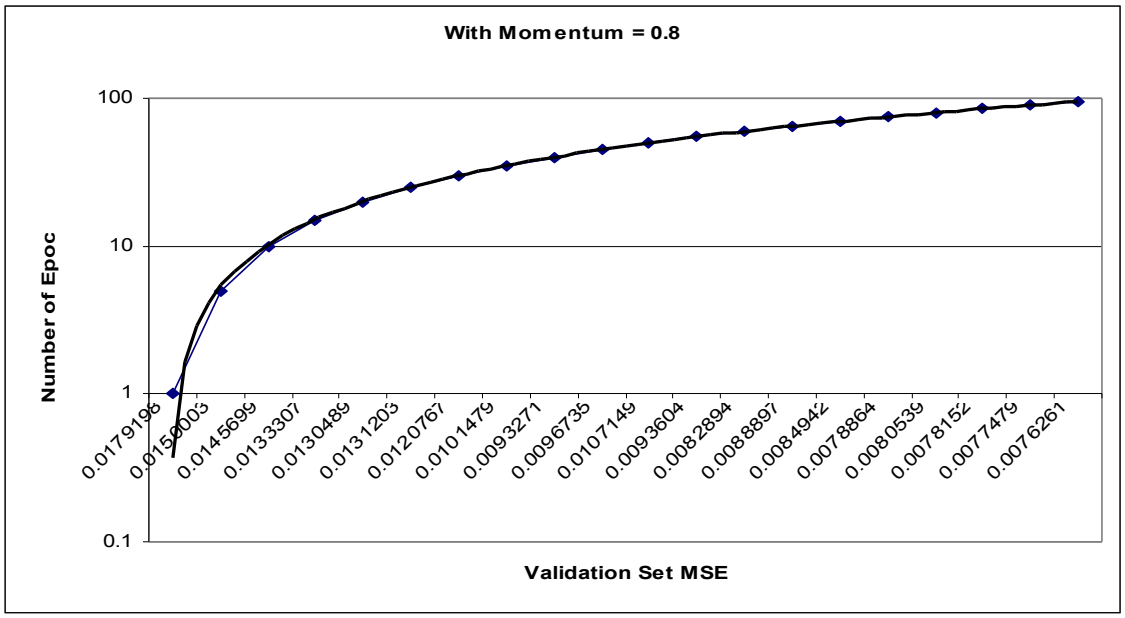

Figure 15 Effect of Momentum = 0.8 on Validation Set SSE

5.2 Experiments

Run 1: Standard Run

From the preliminary results the following values were set:

- CROSSOVER_RATE = 0.7
- MUTATION_RATE = 0.0333
- POP_SIZE = 100
- CHROMO_LENGTH = 2025
- GENE_LENGTH = 9
- MAX_ALLOWABLE_GENERATIONS = 0
- LEARNING_RATE = 0.009
- MOMENTUM = 0.8
- MAX_EPOCHS = 500
- DESIRED_SSE = 0.01
- Scaling Factor = 0.0002

This was the standard run in which the GA ran for just 1 generation to look at at how well does simple backpropagation perform on training, validation and the generalization set.

Run 2: Varying the number of GA Generations

For this run, the number of BP epochs was fixed to 10 and number of GA generations were varied from 0 to 50.

Run 3: Varying amount of BP Training Epochs/Generation

For this the number of GA generations was fixed to 50 and the number of BP epochs was varied from 10 to 25.

Final Run: Testing using the Generalization Set

When a solution is found (when SSE goes below 0.006 for both the training and validation set), we test it on the Generalization Set as we want to be able to measure the performance of the NN on unseen data.

5.3 Results and Discussion

There will always be a tradeoff between the time taken to train a network and the accuracy. Therefore, an optimum balance must be found depending on the problem at hand.

Results from Standard Run

It took 24 minutes for running 500 epochs.

No solution was found (as the validation set SSE did not go below 0.007) as shown in Fig. 17

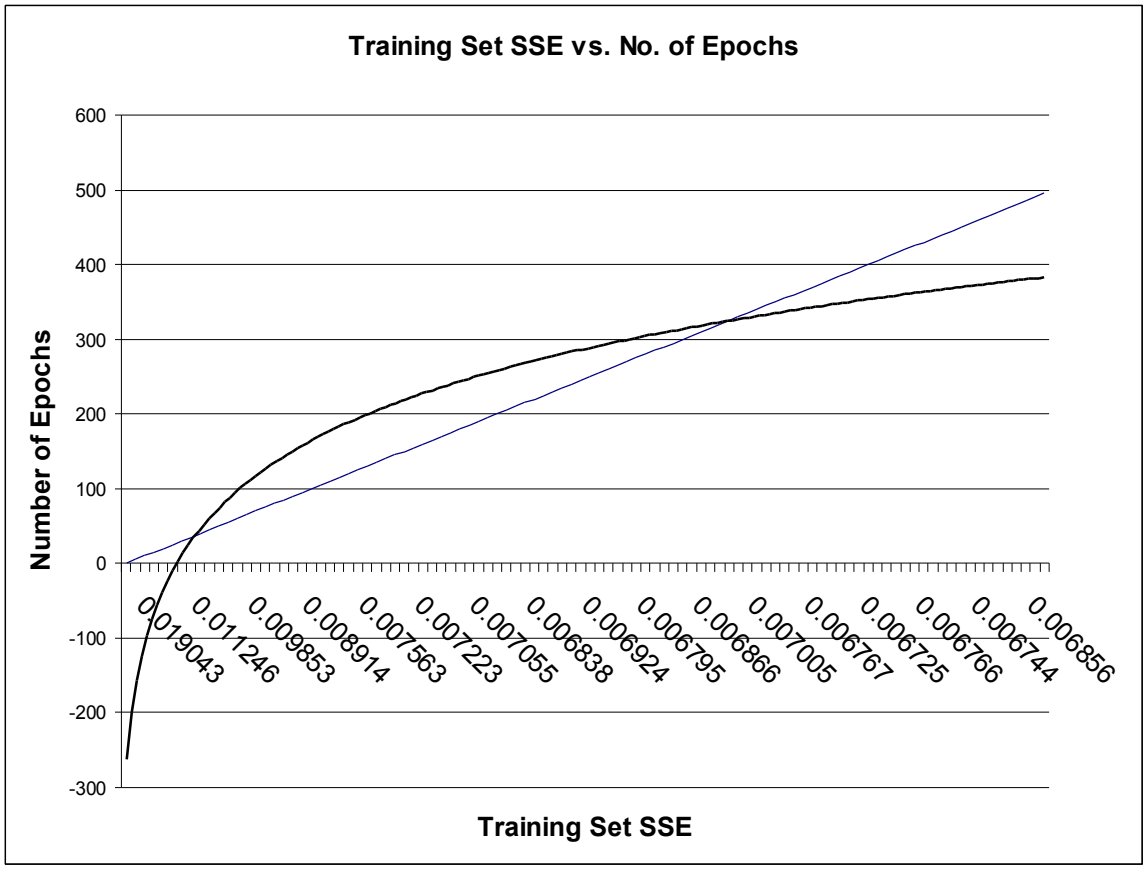

Figure 16 Training Set SSE vs. No. of Epochs

Only SSE for the Training Set came below the threshold as can be seen from Fig. 16.

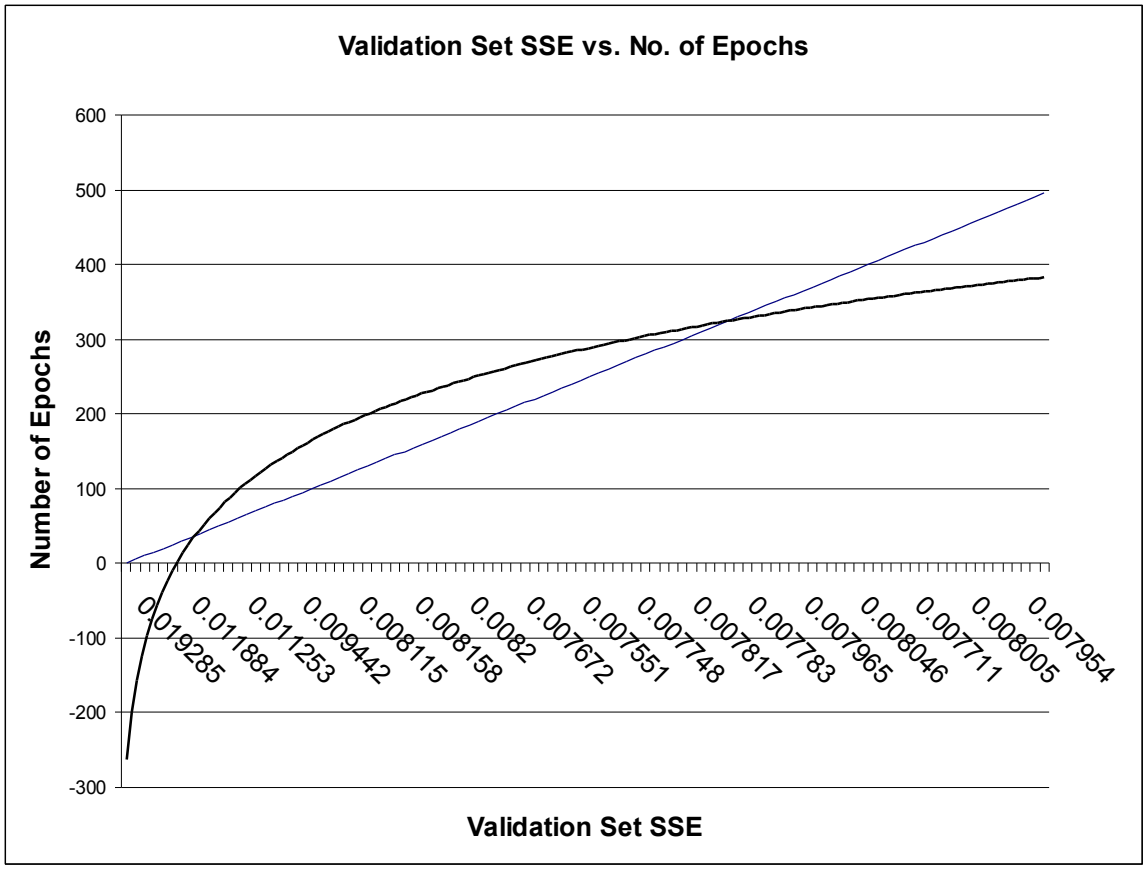

Figure 17Training Set SSE vs. No. of Epochs

Effect of Varying the number of GA Generations

Started at 8:16 a.m. Finished at 9:16a.m.

For this run, the number of BP epochs was fixed to 10 and number of GA generations was set to 50.

After 20 generations, both training set and the validation set SSE came below 0.007. It took an hour to find the solution.

Effect of Varying amount of BP Training Epochs/Generation

For this the number of GA generations was fixed to 50 and the number of BP epochs was varied from 10 to 25.

Started at 9:17 a.m. Finished at 10:00 a.m. (40 minutes)

Solution found after 10 generations compared to Run 2 which took 20 generations.

Generalization Set Accuracy = 81.325%
Generalization Set SSE = 0.00693232

This shows that when the number of epochs per generation is increased the neural network converges to a solution a lot faster.

Started at 10:05 a.m. Finished at 10.10 a.m. (5 minutes)

Now increased the number of epochs per generation to 50.

Solution found in 1 generation

Generalization Set Accuracy = 71.975%
Generalization Set SSE = 0.00693221

This shows that by setting the number of epochs to 50/generation, error threshold was reached very quickly.

Also, it must be noted that the Accuracy is not good measure of the performance therefore SSE should be looked at.

Obviously, one would want an error threshold of as low as possible. But this would require more testing.

5.3 Final Optimal Setup

From the results the following setup produces an error threshold of 0.00693221 on the testing set:

- CROSSOVER_RATE = 0.7
- MUTATION_RATE = 0.0333
- POP_SIZE = 100
- CHROMO_LENGTH = 2025
- GENE_LENGTH = 9
- MAX_ALLOWABLE_GENERATIONS = 50
- LEARNING_RATE = 0.009
- MOMENTUM = 0.8
- MAX_EPOCHS = 50
- DESIRED_SSE <= 0.007

6. C++ Source Code

6.1 Formatting Data

- ABCDE.cpp
  This program reads in the Letter Recognition Dataset in raw form and encodes the dataset to recognize letters A, B, C, D and E

- dataReader.h [3]
  CSV Data File Reader and Training Set Creator

- dataReader.cpp [3]
  Function definitions for formatting data
- dataset.txt [4]
  Letter Recognition Dataset

6.2 BPNN Code/Libraries [3]

- neuralNetwork.h
  Basic Feed Forward Neural Network Class

- neuralNetwork.cpp
  Function definitions for initializing a NN

- neuralNetworkTrainer.h
  Neural Network Training Class

- neuralNetworkTrainer.cpp
  Function definitions for training a NN

6.3 GA Code/ Libraries [6]

- ga.h
  Header file for defining a data structure for a chromosome and function prototypes
- ga.cpp
  Function definitions for selecting indivuals using Roulette Wheel, evolving population using crossover and mutation, decoding and assigning a fitness.

6.4 Main Program

- main.cpp
  Main program which executes the hybrid GA-NN to recognize letters A, B, C, D and E
- makefile
  Makefile for the project to link all files and compile an executable.

6.5 Running the program

All of the above files need to be present in the same directory to compile the program.

7. Conclusions & Future Work

- In this project a C++ implementation of a Neural Network with a Genetic Algorithm is used to classify letters A, B, C, D and E with an accuracy of 84.325% and a sum of squared error of 0.0066.

- Future Work will look at using the Genetic Algorithm to get optimal learning parameters (learning rate and momentum) and the number of hidden layers or the number of hidden nodes for the multi-feed forward NN architecture used.

- Try advanced data portioning methods which will speed the process of training. Also try different variants of simple backpropagation like quickprop, rprop, supersab etc to improve training time and the accuracy of the overall system.

- From the Genetic Algorithm's point of view, there are different selection techniques to use, different crossover and mutation operators to try and more complicated things like fitness sharing and speciation.. All or some of these techniques will improve the performance of genetic algorithms considerably.

References

[1] **Reyes, N.H. (2008).** *Lecture Notes: 159.734, Studies in Machine Learning.* Massey University.

[2] **Bishop, C.M. (1995).** *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford - New York.

[3]**Anguelov, B. (2008).** *Neural Network Implementation in C++*. Retrieved from http://takinginitiative.wordpress.com

[4] **Slate, D.J. (1991).** *Letter Image Recognition Data*. UCI Machine learning Repository. Retrieved from http://archive.ics.uci.edu/ml/datasets/Letter+Recognition

[5] **Frey, P. W. and Slate, D. J. (1991).** *Letter Recognition Using Holland-style Adaptive Classifiers, Machine Learning Volume 6 #2*

[6] **Buckland, Mat (2008).** *Basic Genetic Algorithm Tutorial*. Retrieved from http://www.ai-junkie.com/ga/intro/gat1.html

Appendix Complete C++ Source Code

ABCDE.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <math.h>
#include <algorithm>
#include <cctype>
#include <iomanip>
#include <vector>
#include <iterator>
#include <stdlib.h>
#include <string.h>

using namespace std;

void processLine( string &line );

ofstream outputFile1;

int countForLetterA = 0;
int countForLetterB = 0;
int countForLetterC = 0;
int countForLetterD = 0;
int countForLetterE = 0;

int main() {
	int i=0;

	ifstream inputFile;
	ofstream outputFile;

	inputFile.open("dataset.txt", ios::in);
	outputFile1.open("ABCDE.csv", ios::out);

	string line;
	if ( inputFile.is_open() )
	{
		while ( getline(inputFile, line)) {

			//process line
			processLine(line);
```

```
                        char *cstr = new char[line.size()+1];
                        strcpy(cstr, line.c_str()); // creates a c string equivalent
                        switch(cstr[0]) {

                                case 'A': outputFile1 << "1" << "," << "0" << "," << "0" << "," << "0" <<
"," << "0" << endl;
                                                countForLetterA++;
                                                break;
                                case 'B': outputFile1 << "0" << "," << "1" << "," << "0" << "," << "0" <<
"," << "0" << endl;
                                                countForLetterB++;
                                                break;
                                case 'C': outputFile1 << "0" << "," << "0" << "," << "1" << "," << "0" <<
"," << "0" << endl;
                                                countForLetterC++;
                                                break;
                                case 'D': outputFile1 << "0" << "," << "0" << "," << "0" << "," << "1" <<
"," << "0" << endl;
                                                countForLetterD++;
                                                break;
                                case 'E': outputFile1 << "0" << "," << "0" << "," << "0" << "," << "0" <<
"," << "1" << endl;
                                                countForLetterE++;

                                                break;
                                default : outputFile1 << "0" << "," << "0" << "," << "0" << "," << "0" <<
"," << "0" << endl;
                                                break;

                        }
                }
                inputFile.close();
        }
        else
        {
                cout << "Error Opening Input File: " << "dataset.csv" << endl;
                return false;
        }

  cout << "Finished writing file" << endl;
  cout << "======================" << endl;
        cout << "Count for Letter A = " << countForLetterA << endl;
        cout << "Count for Letter B = " << countForLetterB << endl;
        cout << "Count for Letter C = " << countForLetterC << endl;
        cout << "Count for Letter D = " << countForLetterD << endl;
        cout << "Count for Letter E = " << countForLetterE << endl;


        outputFile1.close();
}

/*****************************************************************
* Processes a single line from the data file
*****************************************************************/
void processLine( string &line )
{
        int* pattern = new int[17];
        char *cstr = new char[line.size()+1];
        char* t;

        strcpy(cstr, line.c_str()); // creates a c string equivalent
```

```
		//tokenise
		int i = 1;
		t=strtok (cstr,",");

		while ( t!=NULL && i < 17)
		{

			if(i < 17) {

				t = strtok(NULL,","); //move token onwards
					pattern[i] = atoi(t);
					outputFile1 << pattern[i] << "," ;
			}

			i++;
		}
}
```

datasetReader.h

```
#ifndef NNetworkTrainer
#define NNetworkTrainer

//standard includes
#include <fstream>
#include <vector>

//neural network header
#include "neuralNetwork.h"

//Constant Defaults!
#define LEARNING_RATE 0.001
#define MOMENTUM 0.9
#define MAX_EPOCHS 20000
#define DESIRED_ACCURACY 90
#define DESIRED_SSE 0.01

/*************************************************************************
* Basic Gradient Descent Trainer with Momentum and Batch Learning
*************************************************************************/
class neuralNetworkTrainer
{
		//class members
		//---------------------------------------------------------------------------------------

private:

		//network to be trained
		neuralNetwork* NN;

		//learning parameters
		double learningRate;			// adjusts the step size of the weight update
		double momentum;		// improves performance of stochastic learning (don't use for batch)

		//epoch counter
		long epoch;
		long maxEpochs;

		//accuracy/SSE required
```

```
        double desiredAccuracy;

        //change to weights
        double** deltaInputHidden;
        double** deltaHiddenOutput;

        //error gradients
        double* hiddenErrorGradients;
        double* outputErrorGradients;

        //accuracy stats per epoch
        double trainingSetAccuracy;
        double validationSetAccuracy;
        double generalizationSetAccuracy;
        double trainingSetSSE;
        double validationSetSSE;
        double generalizationSetSSE;

        //batch learning flag
        bool useBatch;

        //log file handle
        bool loggingEnabled;
        std::fstream logFile;
        int logResolution;
        int lastEpochLogged;

        //public methods
        //-------------------------------------------------------------------------------------------
public:

        neuralNetworkTrainer( neuralNetwork* untrainedNetwork );
        void setTrainingParameters( double lR, double m, bool batch );
        void setStoppingConditions( int mEpochs, double dAccuracy);
        void useBatchLearning( bool flag ){ useBatch = flag; }
        void enableLogging( const char* filename, int resolution );

        double trainNetwork( trainingDataSet* tSet );

        //private methods
        //-------------------------------------------------------------------------------------------
private:
        inline double getOutputErrorGradient( double desiredValue, double outputValue );
        double getHiddenErrorGradient( int j );
        double runTrainingEpoch( std::vector<dataEntry*> trainingSet );
        void backpropagate(double* desiredOutputs);
        void updateWeights();
};


#endif
```

datasetReader.cpp

```
//include definition file
#include "dataReader.h"

#include <iostream>
#include <fstream>
#include <string>
#include <math.h>
#include <algorithm>

using namespace std;

/*************************************************************************
* Destructor
*************************************************************************/
dataReader::~dataReader()
{
	//clear data
	for (int i=0; i < (int) data.size(); i++ ) delete data[i];
	data.clear();
}
/*************************************************************************
* Loads a csv file of input data
*************************************************************************/
bool dataReader::loadDataFile( const char* filename, int nI, int nT )
{
	//clear any previous data
	for (int i=0; i < (int) data.size(); i++ ) delete data[i];
	data.clear();
	tSet.clear();

	//set number of inputs and outputs
	nInputs = nI;
	nTargets = nT;

	//open file for reading
	fstream inputFile;
	inputFile.open(filename, ios::in);

	if ( inputFile.is_open() )
	{
```

```
		string line = "";

		//read data
		while ( !inputFile.eof() )
		{
			getline(inputFile, line);

			//process line
			if (line.length() > 2 ) processLine(line);
		}


		//shuffle data
		srand(time(0));
		random_shuffle(data.begin(), data.end());

		//split data set
		trainingDataEndIndex = (int) ( 0.6 * data.size() );
		int gSize = (int) ( ceil(0.2 * data.size()) );
		int vSize = (int) ( data.size() - trainingDataEndIndex - gSize );

		//generalization set
		for ( int i = trainingDataEndIndex; i < trainingDataEndIndex + gSize; i++ )
tSet.generalizationSet.push_back( data[i] );

		//validation set
		for ( int i = trainingDataEndIndex + gSize; i < (int) data.size(); i++ )
tSet.validationSet.push_back( data[i] );

		//print success
		cout << "Input File: " << filename << "\nRead Complete: " << data.size() << " Patterns
Loaded"  << endl;

		//close file
		inputFile.close();

		return true;
	}
	else
	{
		cout << "Error Opening Input File: " << filename << endl;
		return false;
	}
}

/*****************************************************************
* Processes a single line from the data file
*****************************************************************/
void dataReader::processLine( string &line )
{
	//create new pattern and target
	double* pattern = new double[nInputs];
	double* target = new double[nTargets];

	//store inputs
	char* cstr = new char[line.size()+1];
	char* t;
	strcpy(cstr, line.c_str());

	//tokenise
	int i = 0;
	t=strtok (cstr,",");
```

```
		while ( t!=NULL && i < (nInputs + nTargets) )
		{
			if ( i < nInputs ) pattern[i] = atof(t);
			else target[i - nInputs] = atof(t);

			//move token onwards
			t = strtok(NULL,",");
			i++;
		}

		//~ cout << "pattern: ";
		//~ for (int i=0; i < nInputs; i++)
		//~ {
			//~ cout << pattern[i] << ",";
		//~ }

		//~ cout << " target: ";
		//~ for (int i=0; i < nTargets; i++)
		//~ {
			//~ cout << target[i] << " ";
		//~ }
		//~ cout << endl;


		//add to records
		data.push_back( new dataEntry( pattern, target ) );
}
/*****************************************************************
* Selects the data set creation approach
*****************************************************************/
void dataReader::setCreationApproach( int approach, double param1, double param2 )
{
		//static
		if ( approach == STATIC )
		{
			creationApproach = STATIC;

			//only 1 data set
			numTrainingSets = 1;
		}

		//growing
		else if ( approach == GROWING )
		{
			if ( param1 <= 100.0 && param1 > 0)
			{
				creationApproach = GROWING;

				//step size
				growingStepSize = param1 / 100;
				growingLastDataIndex = 0;

				//number of sets
				numTrainingSets = (int) ceil( 1 / growingStepSize );
			}
		}

		//windowing
		else if ( approach == WINDOWING )
		{
			//if initial size smaller than total entries and step size smaller than set size
```

```
                if ( param1 < data.size() && param2 <= param1)
                {
                        creationApproach = WINDOWING;

                        //params
                        windowingSetSize = (int) param1;
                        windowingStepSize = (int) param2;
                        windowingStartIndex = 0;

                        //number of sets
                        numTrainingSets = (int) ceil( (double) ( trainingDataEndIndex - windowingSetSize ) /
windowingStepSize ) + 1;
                }
        }

}

/*****************************************************************
* Returns number of data sets created by creation approach
*****************************************************************/
int dataReader::getNumTrainingSets()
{
        return numTrainingSets;
}
/*****************************************************************
* Get data set created by creation approach
*****************************************************************/
trainingDataSet* dataReader::getTrainingDataSet()
{
        switch ( creationApproach )
        {
                case STATIC : createStaticDataSet(); break;
                case GROWING : createGrowingDataSet(); break;
                case WINDOWING : createWindowingDataSet(); break;
        }

        return &tSet;
}
/*****************************************************************
* Get all data entries loaded
*****************************************************************/
vector<dataEntry*>& dataReader::getAllDataEntries()
{
        return data;
}

/*****************************************************************
* Create a static data set (all the entries)
*****************************************************************/
void dataReader::createStaticDataSet()
{
        //training set
        for ( int i = 0; i < trainingDataEndIndex; i++ ) tSet.trainingSet.push_back( data[i] );
}
/*****************************************************************
* Create a growing data set (contains only a percentage of entries
* and slowly grows till it contains all entries)
*****************************************************************/
void dataReader::createGrowingDataSet()
{
        //increase data set by step percentage
        growingLastDataIndex += (int) ceil( growingStepSize * trainingDataEndIndex );
```

```
		if ( growingLastDataIndex > (int) trainingDataEndIndex ) growingLastDataIndex = trainingDataEndIndex;

		//clear sets
		tSet.trainingSet.clear();

		//training set
		for ( int i = 0; i < growingLastDataIndex; i++ ) tSet.trainingSet.push_back( data[i] );
}
/*****************************************************************
* Create a windowed data set ( creates a window over a part of the data
* set and moves it along until it reaches the end of the date set )
*****************************************************************/
void dataReader::createWindowingDataSet()
{
		//create end point
		int endIndex = windowingStartIndex + windowingSetSize;
		if ( endIndex > trainingDataEndIndex ) endIndex = trainingDataEndIndex;

		//clear sets
		tSet.trainingSet.clear();

		//training set
		for ( int i = windowingStartIndex; i < endIndex; i++ ) tSet.trainingSet.push_back( data[i] );

		//increase start index
		windowingStartIndex += windowingStepSize;
}
```

neuralNetwork.h

```
#ifndef NNetwork
#define NNetwork

#include "dataReader.h"

class neuralNetworkTrainer;

class neuralNetwork
{
	//class members
private:

	//number of neurons
	int nInput, nHidden, nOutput;

	//neurons
	double* inputNeurons;
	double* hiddenNeurons;
	double* outputNeurons;

	//weights
	double** wInputHidden;
	double** wHiddenOutput;

	//Friends
	friend class neuralNetworkTrainer;

	//public methods

public:

	//constructor & destructor
	neuralNetwork(int numInput, int numHidden, int numOutput);
	~neuralNetwork();

	//weight operations
	bool loadWeights(char* inputFilename);
	bool saveWeights(char* outputFilename);
	int* feedForwardPattern( double* pattern );
	double getSetAccuracy( std::vector<dataEntry*>& set );
```

```
        double getSetSSE( std::vector<dataEntry*>& set );

        //private methods

private:

        void initializeWeights();
        inline double activationFunction( double x );
        int clampOutput( double x );
        void feedForward( double* pattern );

};

#endif
```

neuralNetwork.cpp

```
//standard includes
#include <iostream>
#include <vector>
#include <fstream>
#include <math.h>

//include definition file
#include "neuralNetwork.h"

using namespace std;

/*********************************************************************
* Constructor
*********************************************************************/
neuralNetwork::neuralNetwork(int nI, int nH, int nO) : nInput(nI), nHidden(nH), nOutput(nO)
{
        //create neuron lists
        //-------------------------------------------------------------------------------------------------
        inputNeurons = new( double[nInput + 1] );
        for ( int i=0; i < nInput; i++ ) inputNeurons[i] = 0;

        //create input bias neuron
        inputNeurons[nInput] = -1;

        hiddenNeurons = new( double[nHidden + 1] );
        for ( int i=0; i < nHidden; i++ ) hiddenNeurons[i] = 0;

        //create hidden bias neuron
        hiddenNeurons[nHidden] = -1;
        //outputNeurons[nOutput] = -1;

        outputNeurons = new( double[nOutput] );
        for ( int i=0; i < nOutput; i++ ) outputNeurons[i] = 0;

        //create weight lists (include bias neuron weights)
        //-------------------------------------------------------------------------------------------------
        wInputHidden = new( double*[nInput + 1] );
        for ( int i=0; i <= nInput; i++ )
        {
                wInputHidden[i] = new (double[nHidden]);
```

```
                for ( int j=0; j < nHidden; j++ ) wInputHidden[i][j] = 0;
        }

        wHiddenOutput = new( double*[nHidden + 1] );
        for ( int i=0; i <= nHidden; i++ )
        {
                wHiddenOutput[i] = new (double[nOutput]);
                for ( int j=0; j < nOutput; j++ ) wHiddenOutput[i][j] = 0;
        }

}







/*****************************************************************
* Destructor
*****************************************************************/
neuralNetwork::~neuralNetwork()
{
        //delete neurons
        delete[] inputNeurons;
        delete[] hiddenNeurons;
        delete[] outputNeurons;

        //delete weight storage
        for (int i=0; i <= nInput; i++) delete[] wInputHidden[i];
        delete[] wInputHidden;

        for (int j=0; j <= nHidden; j++) delete[] wHiddenOutput[j];
        delete[] wHiddenOutput;
}

/*****************************************************************
* Load Neuron Weights
*****************************************************************/
bool neuralNetwork::loadWeights(char* filename)
{
        //open file for reading
        fstream inputFile;
        inputFile.open(filename, ios::in);

        if ( inputFile.is_open() )
        {
                vector<double> weights;
                string line = "";

                //read data
                while ( !inputFile.eof() )
                {
                        getline(inputFile, line);

                        //process line
                        if (line.length() > 2 )
                        {
                                //store inputs
                                char* cstr = new char[line.size()+1];
```

```
                                        char* t;
                                        strcpy(cstr, line.c_str());

                                        //tokenise
                                        int i = 0;
                                        t=strtok (cstr,",");

                                        while ( t!=NULL )
                                        {
                                                weights.push_back( atof(t) );

                                                //move token onwards
                                                t = strtok(NULL,",");
                                                i++;
                                        }
                                }
                        }

                        //check if sufficient weights were loaded
                        if ( weights.size() != ( (nInput + 1) * nHidden + (nHidden +  1) * nOutput ) )
                        {
                                cout << endl << "Error - Incorrect number of weights in input file: " << filename <<
endl;

                                //close file
                                inputFile.close();

                                return false;
                        }
                        else
                        {
                                //set weights
                                int pos = 0;

                                for ( int i=0; i <= nInput; i++ )
                                {
                                        for ( int j=0; j < nHidden; j++ )
                                        {
                                                wInputHidden[i][j] = weights[pos++];

                                        }
                                }

                                for ( int i=0; i <= nHidden; i++ )
                                {
                                        for ( int j=0; j < nOutput; j++ )
                                        {
                                                wHiddenOutput[i][j] = weights[pos++];

                                        }
                                }

                                //print success
                                cout << endl << "Neuron weights loaded successfuly from " << filename << endl;

                                //close file
                                inputFile.close();

                                return true;
                        }
                }
                else
```

```
		{
			cout << endl << "Error - Weight input file '" << filename << "' could not be opened: " << endl;
			return false;
		}
}
```

```
/*************************************************************************
* Save Neuron Weights
*************************************************************************/
bool neuralNetwork::saveWeights(char* filename)
{
	//open file for reading
	fstream outputFile;
	outputFile.open(filename, ios::out);

	if ( outputFile.is_open() )
	{
		outputFile.precision(50);

		//output weights
		for ( int i=0; i <= nInput; i++ )
		{
			for ( int j=0; j < nHidden; j++ )
			{
				outputFile << wInputHidden[i][j] << ",";
			}
		}

		for ( int i=0; i <= nHidden; i++ )
		{
			for ( int j=0; j < nOutput; j++ )
			{
				outputFile << wHiddenOutput[i][j];
				if ( i * nOutput + j + 1 != (nHidden + 1) * nOutput ) outputFile << ",";
			}
		}

		//print success
		cout << endl << "Neuron weights saved to '" << filename << "'" << endl;

		//close file
		outputFile.close();

		return true;
	}
	else
	{
		cout << endl << "Error - Weight output file '" << filename << "' could not be created: " << endl;
```

```
                return false;
        }
}
```

```
/*************************************************************
* Initialize Neuron Weights
*************************************************************/
void neuralNetwork::initializeWeights()
{
        //set range
        double rH = -1; // 1/sqrt( (double) nInput);
        double rO = 1; // 1/sqrt( (double) nHidden);

        //set weights between input and hidden
        //--------------------------------------------------------------------------------------------------
        for(int i = 0; i <= nInput; i++)
        {
                for(int j = 0; j < nHidden; j++)
                {
                        //set weights to random values
                        wInputHidden[i][j] = ( ( (double)(rand()%100)+1)/100  * 2 * rH ) - rH;

                }
        }

        //set weights between input and hidden
        //--------------------------------------------------------------------------------------------------
        for(int i = 0; i <= nHidden; i++)
        {
                for(int j = 0; j < nOutput; j++)
                {
                        //set weights to random values
                        wHiddenOutput[i][j] = ( ( (double)(rand()%100)+1)/100 * 2 * rO ) - rO;
                }
        }
}


/*************************************************************
* Feed pattern through network and return results
*************************************************************/
int* neuralNetwork::feedForwardPattern(double *pattern)
{
        feedForward(pattern);

        //create copy of output results
```

```
	int* results = new int[nOutput];
	for (int i=0; i < nOutput; i++ ) results[i] = clampOutput(outputNeurons[i]);

	return results;
}









/*****************************************************************
* Return the NN accuracy on the set
*****************************************************************/
double neuralNetwork::getSetAccuracy( std::vector<dataEntry*>& set )
{
	double incorrectResults = 0;

	//for every training input array
	for ( int tp = 0; tp < (int) set.size(); tp++)
	{
		//feed inputs through network and backpropagate errors
		feedForward( set[tp]->pattern );

		//correct pattern flag
		bool correctResult = true;

		//check all outputs against desired output values
		for ( int k = 0; k < nOutput; k++ )
		{
			//set flag to false if desired and output differ
			if ( clampOutput(outputNeurons[k]) != set[tp]->target[k] ) correctResult = false;
		}

		//inc training error for a incorrect result
		if ( !correctResult ) incorrectResults++;

	}//end for

	//calculate error and return as percentage
	return 100 - (incorrectResults/set.size() * 100);
}


/*****************************************************************
* Return the NN sum of  squared error on the set
*****************************************************************/
double neuralNetwork::getSetSSE( std::vector<dataEntry*>& set )
{
	double sse = 0;


	//for every training input array
```

```
	for ( int tp = 0; tp < (int) set.size(); tp++)
	{
		//feed inputs through network and backpropagate errors
		feedForward( set[tp]->pattern );
		//check all outputs against desired output values
		for ( int k = 0; k < nOutput; k++ )
		{
			//sum all the MSEs together
			sse += (0.5 * pow((outputNeurons[k] - set[tp]->target[k]), 2));
		}

	}//end for

	//calculate error and return as percentage
	return sse/(nOutput * set.size());
}



/*****************************************************************
* Activation Function
*****************************************************************/
inline double neuralNetwork::activationFunction( double x )
{
	//sigmoid function
	return 1/(1+exp(-x));
}


/*****************************************************************
* Output Clamping
*****************************************************************/
inline int neuralNetwork::clampOutput( double x )
{
	if ( x < 0.1 ) return 0;
	else if ( x > 0.9 ) return 1;
	else return -1;
}


/*****************************************************************
* Feed Forward Operation
*****************************************************************/
void neuralNetwork::feedForward(double* pattern)
{
	//set input neurons to input values
	for(int i = 0; i < nInput; i++) inputNeurons[i] = pattern[i];

	//Calculate Hidden Layer values - include bias neuron
	//------------------------------------------------------------------------------------------------
	for(int j=0; j < nHidden; j++)
	{
		//clear value
		hiddenNeurons[j] = 0;

		//get weighted sum of pattern and bias neuron
		for( int i=0; i <= nInput; i++ ) hiddenNeurons[j] += inputNeurons[i] * wInputHidden[i][j];

		//set to result of sigmoid
		hiddenNeurons[j] = activationFunction( hiddenNeurons[j] );
	}
```

```
		//Calculating Output Layer values - include bias neuron
		//-------------------------------------------------------------------------------------------------
		for(int k=0; k < nOutput; k++)
		{
			//clear value
			outputNeurons[k] = 0;

			//get weighted sum of pattern and bias neuron
			for( int j=0; j <= nHidden; j++ ) outputNeurons[k] += hiddenNeurons[j] * wHiddenOutput[j][k];

			//set to result of sigmoid
			outputNeurons[k] = activationFunction( outputNeurons[k] );
		}
}
```

neuralNetworkTrainer.h

```
#ifndef NNetworkTrainer
#define NNetworkTrainer

//standard includes
#include <fstream>
#include <vector>

//neural network header
#include "neuralNetwork.h"

//Constant Defaults!
#define LEARNING_RATE 0.001
#define MOMENTUM 0.9
#define MAX_EPOCHS 20000
#define DESIRED_ACCURACY 90
#define DESIRED_SSE 0.001

/////////////////////////////////////////////////////////////////
* Basic Gradient Descent Trainer with Momentum and Batch Learning
/////////////////////////////////////////////////////////////////*/
class neuralNetworkTrainer
{
		//class members
		//-------------------------------------------------------------------------------------------

private:

		//network to be trained
		neuralNetwork* NN;

		//learning parameters
		double learningRate;		// adjusts the step size of the weight update
		double momentum;		// improves performance of stochastic learning (don't use for batch)

		//epoch counter
		long epoch;
		long maxEpochs;

		//accuracy/MSE required
```

```
        double desiredAccuracy;

        //change to weights
        double** deltaInputHidden;
        double** deltaHiddenOutput;

        //error gradients
        double* hiddenErrorGradients;
        double* outputErrorGradients;

        //accuracy stats per epoch
        double trainingSetAccuracy;
        double validationSetAccuracy;
        double generalizationSetAccuracy;
        double trainingSetSSE;
        double validationSetSSE;
        double generalizationSetSSE;

        //batch learning flag
        bool useBatch;

        //log file handle
        bool loggingEnabled;
        std::fstream logFile;
        int logResolution;
        int lastEpochLogged;

        //public methods
        //-------------------------------------------------------------------------------------------
public:

        neuralNetworkTrainer( neuralNetwork* untrainedNetwork );
        void setTrainingParameters( double lR, double m, bool batch );
        void setStoppingConditions( int mEpochs, double dAccuracy);
        void useBatchLearning( bool flag ){ useBatch = flag; }
        void enableLogging( const char* filename, int resolution );

        double trainNetwork( trainingDataSet* tSet );

        //private methods
        //-------------------------------------------------------------------------------------------
private:
        inline double getOutputErrorGradient( double desiredValue, double outputValue );
        double getHiddenErrorGradient( int j );
        double runTrainingEpoch( std::vector<dataEntry*> trainingSet );
        void backpropagate(double* desiredOutputs);
        void updateWeights();
};


#endif
```

neuralNetworkTrainer.cpp

```
//standard includes
#include <iostream>
#include <fstream>
#include <math.h>

//include definition file
#include "neuralNetworkTrainer.h"

using namespace std;

/*****************************************************************
* constructor
*****************************************************************/
neuralNetworkTrainer::neuralNetworkTrainer( neuralNetwork *nn ) :       NN(nn),
                                                                        epoch(0),
                                                                        learningRate(LEARNING_RATE),
                                                                        momentum(MOMENTUM),
                                                                        maxEpochs(MAX_EPOCHS),
                                                                        desiredAccuracy(DESIRED_SSE),
                                                                                useBatch(false),
                                                                        trainingSetAccuracy(0),
                                                                        validationSetAccuracy(0),
                                                                        generalizationSetAccuracy(0),
                                                                        trainingSetSSE(0),
                                                                        validationSetSSE(0),
                                                                        generalizationSetSSE(0)
```

```
{
	//create delta lists
	//-----------------------------------------------------------------------------------------------
	deltaInputHidden = new( double*[NN->nInput + 1] );
	for ( int i=0; i <= NN->nInput; i++ )
	{
		deltaInputHidden[i] = new (double[NN->nHidden]);
		for ( int j=0; j < NN->nHidden; j++ ) deltaInputHidden[i][j] = 0;
	}

	deltaHiddenOutput = new( double*[NN->nHidden + 1] );
	for ( int i=0; i <= NN->nHidden; i++ )
	{
		deltaHiddenOutput[i] = new (double[NN->nOutput]);
		for ( int j=0; j < NN->nOutput; j++ ) deltaHiddenOutput[i][j] = 0;
	}


	//create error gradient storage
	//-----------------------------------------------------------------------------------------------
	hiddenErrorGradients = new( double[NN->nHidden + 1] );
	for ( int i=0; i <= NN->nHidden; i++ ) hiddenErrorGradients[i] = 0;

	outputErrorGradients = new( double[NN->nOutput + 1] );
	for ( int i=0; i <= NN->nOutput; i++ ) outputErrorGradients[i] = 0;
}


/*****************************************************************
* Set training parameters
*****************************************************************/
void neuralNetworkTrainer::setTrainingParameters( double lR, double m, bool batch )
{
	learningRate = lR;
	momentum = m;
	useBatch = batch;
}


/*****************************************************************
* Set stopping parameters
*****************************************************************/
void neuralNetworkTrainer::setStoppingConditions( int mEpochs, double dAccuracy )
{
	maxEpochs = mEpochs;
	desiredAccuracy = dAccuracy;
}

/*****************************************************************
* Enable training logging
*****************************************************************/
void neuralNetworkTrainer::enableLogging(const char* filename, int resolution = 1)
{
	//create log file
	if ( ! logFile.is_open() )
	{
		logFile.open(filename, ios::out);

		if ( logFile.is_open() )
		{
```

```
                        //enable logging
                        loggingEnabled = true;

                        //resolution setting;
                        logResolution = resolution;
                        lastEpochLogged = -resolution;
                }
        }
}

/*****************************************************************
* calculate output error gradient
*****************************************************************/
inline double neuralNetworkTrainer::getOutputErrorGradient( double desiredValue, double outputValue)
{
        //return error gradient
        return  (outputValue * ( 1 - outputValue ) *( desiredValue - outputValue ));
}



/*****************************************************************
* calculate input error gradient
*****************************************************************/
double neuralNetworkTrainer::getHiddenErrorGradient( int j )
{
        //get sum of hidden->output weights * output error gradients
        double weightedSum = 0;
        for( int k = 0; k < NN->nOutput; k++ ) weightedSum += NN->wHiddenOutput[j][k] *
outputErrorGradients[k];

        //return error gradient
        return NN->hiddenNeurons[j] * ( 1 - NN->hiddenNeurons[j] ) * weightedSum;
}

/*****************************************************************
* Train the NN using gradient descent
*****************************************************************/
double neuralNetworkTrainer::trainNetwork( trainingDataSet* tSet )
{

        clock_t startTime, endTime;
        cout        << endl << "Neural Network Training Starting: " << endl
                        <<
"================================================================================
====" << endl
                        << "LR: " << learningRate << ", Momentum: " << momentum << ", Max Epochs: "
<< maxEpochs << endl
                        << NN->nInput << " Input Neurons: " << NN->nHidden << " Hidden Neurons: " <<
NN->nOutput << " Output Neurons: " << endl
                        <<
"================================================================================
====" << endl << endl;

        //reset epoch and log counters
        epoch = 0;
        lastEpochLogged = -logResolution;
        startTime = clock();
        //train network using training dataset for training and stop training trainging using bothe training abd
validation dataset
        //-------------------------------------------------------------------------------------------------------------
```

```
	//write log file header
	//~ logFile << "Epoch, Training Set SSE, Validation Set SSE" << endl;

	while ( epoch < maxEpochs )
	{
		//use training set to train network
		trainingSetSSE = runTrainingEpoch( tSet->trainingSet );

		//get validation set accuracy and SSE
	  validationSetAccuracy = NN->getSetAccuracy(tSet->validationSet);
	  validationSetSSE = NN->getSetSSE(tSet->validationSet);

		cout << "Epoch :" << epoch;
		cout << " TSet Acc:" << trainingSetAccuracy << "%, SSE: " << trainingSetSSE ;
		cout << " VSet Acc:" << validationSetAccuracy << "%, SSE: " << validationSetSSE << endl;


		//Log Training results
		if ( loggingEnabled && logFile.is_open() && ( epoch - lastEpochLogged == logResolution ) )
		{
			logFile << epoch << "," << trainingSetSSE << "," << validationSetSSE << endl;
			lastEpochLogged = epoch;
		}


		//print out change in training / validation mse
		if ( trainingSetSSE <= desiredAccuracy && validationSetSSE <= desiredAccuracy)
		{
			desiredAccuracy /= 2; // successively lower the error threshold
			cout << "=====================================" << endl;
			cout << "Lowering the error threshold to " << desiredAccuracy << endl;
			cout << "=====================================" << endl;
			if(trainingSetSSE < 0.007 && validationSetSSE < 0.007) { // stop when error threshold
below 0.007 is reached
				break;
			}
		}
		//~ else {
			//~ learningRate /= 2; // successively lowering the learning rate if desired training and
validation error level not reached
		//~ }

		//once training set is complete increment epoch
		epoch++;


	}//end while

  cout << endl;
	endTime = clock();
	cout << "Time taken for backpropagation training = " << (endTime - startTime)/1000/60 << " minutes" <<
endl;

	//get training set accuracy and SSE
	trainingSetAccuracy = NN->getSetAccuracy( tSet->trainingSet );
	trainingSetSSE = NN->getSetSSE( tSet->trainingSet );

	//get validation set accuracy and SSE
	validationSetAccuracy = NN->getSetAccuracy(tSet->validationSet);
	validationSetSSE = NN->getSetSSE(tSet->validationSet);
```

```
	//out validation accuracy and MSE
	cout << endl << "Training Complete!!! - > Elapsed Epochs: " << epoch << endl;

	return trainingSetSSE;
}









/*****************************************************************
* Run a single training epoch
*****************************************************************/
double neuralNetworkTrainer::runTrainingEpoch( vector<dataEntry*> trainingSet )
{
	//incorrect patterns
	double incorrectPatterns = 0;
	double sse = 0;


	//for every training pattern
	for ( int tp = 0; tp < (int) trainingSet.size(); tp++)
	{
		//feed inputs through network and backpropagate errors
		NN->feedForward( trainingSet[tp]->pattern );
		backpropagate( trainingSet[tp]->target );

		//pattern correct flag
		bool patternCorrect = true;

		//check all outputs from neural network against desired values
		for ( int k = 0; k < NN->nOutput; k++ )
		{
			//int temp = NN->clampOutput( NN->outputNeurons[k] ) ;
			//
		  //out << temp << "," << trainingSet[tp]->target[k] << endl;
			//pattern incorrect if desired and output differ
			if ( NN->clampOutput( NN->outputNeurons[k] ) != trainingSet[tp]->target[k] ) {
				patternCorrect = false;

			}
			//calculate SSE
			sse +=  pow(( NN->outputNeurons[k] - trainingSet[tp]->target[k] ), 2);
		}

		//if pattern is incorrect add to incorrect count
		if ( !patternCorrect ) incorrectPatterns++;

	}//end for
```

```
		//if using batch learning - update the weights
		if ( useBatch ) updateWeights();

		//update training accuracy and SSE
		trainingSetAccuracy = 100 - (incorrectPatterns/trainingSet.size() * 100);
		trainingSetSSE = (0.5 * sse) / ( NN->nOutput * trainingSet.size() );

		return trainingSetSSE;

}









/////////////////////////////////////////////////////////////////////
// Propagate errors back through NN and calculate delta values
/////////////////////////////////////////////////////////////////////
void neuralNetworkTrainer::backpropagate( double* desiredOutputs )
{
		//modify deltas between hidden and output layers
		//--------------------------------------------------------------------------------------------------
		for (int k = 0; k < NN->nOutput; k++)
		{
			//get error gradient for every output node
			outputErrorGradients[k] = getOutputErrorGradient( desiredOutputs[k], NN->outputNeurons[k] );

			//for all nodes in hidden layer and bias neuron
			for (int j = 0; j <= NN->nHidden; j++)
			{
				//calculate change in weight
				if ( !useBatch ) deltaHiddenOutput[j][k] = learningRate * NN->hiddenNeurons[j] *
outputErrorGradients[k] + momentum * deltaHiddenOutput[j][k];
				else deltaHiddenOutput[j][k] += learningRate * NN->hiddenNeurons[j] *
outputErrorGradients[k];
			}
		}

		//modify deltas between input and hidden layers
		//--------------------------------------------------------------------------------------------------
		for (int j = 0; j < NN->nHidden; j++)
		{
			//get error gradient for every hidden node
			hiddenErrorGradients[j] = getHiddenErrorGradient( j );

			//for all nodes in input layer and bias neuron
			for (int i = 0; i <= NN->nInput; i++)
			{
				//calculate change in weight
				if ( !useBatch ) deltaInputHidden[i][j] = learningRate * NN->inputNeurons[i] *
hiddenErrorGradients[j] + momentum * deltaInputHidden[i][j];
				else deltaInputHidden[i][j] += learningRate * NN->inputNeurons[i] *
hiddenErrorGradients[j];

			}
```

```
		}

		//if using stochastic learning update the weights immediately
		if ( !useBatch ) updateWeights();
}









/*****************************************************************
* Update weights using delta values
*****************************************************************/
void neuralNetworkTrainer::updateWeights()
{
		//input -> hidden weights
		//------------------------------------------------------------------------------------------------
		for (int i = 0; i <= NN->nInput; i++)
		{
			for (int j = 0; j < NN->nHidden; j++)
			{
				//update weight
				NN->wInputHidden[i][j] += deltaInputHidden[i][j];

				//clear delta only if using batch (previous delta is needed for momentum
				if (useBatch) deltaInputHidden[i][j] = 0;
			}
		}

		//hidden -> output weights
		//------------------------------------------------------------------------------------------------
		for (int j = 0; j <= NN->nHidden; j++)
		{
			for (int k = 0; k < NN->nOutput; k++)
			{
				//update weight
				NN->wHiddenOutput[j][k] += deltaHiddenOutput[j][k];

				//clear delta only if using batch (previous delta is needed for momentum)
				if (useBatch)deltaHiddenOutput[j][k] = 0;
			}
		}
}
```

ga.h

```
#ifndef _GA
#define _GA

#include <string>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <time.h>
#include <math.h>

using std::string;
using namespace std;

#define CROSSOVER_RATE          0.6
#define MUTATION_RATE           0.0333
#define POP_SIZE                50
#define CHROMO_LENGTH           2025
#define GENE_LENGTH             9
#define MAX_ALLOWABLE_GENERATIONS 50

//returns a float between 0 & 1
#define RANDOM_NUM1             ((float)rand()/(RAND_MAX+1))

//returns a float between -1 & 1
#define RANDOM_NUM2             ((float)rand()/(RAND_MAX * (-1) ))

//-------------------------------------------------------------------------------
//          define a data structure which will define a chromosome/
//-------------------------------------------------------------------------------
struct chromo_typ
{
  //the binary bit string is held in a std::string
  string   bits;
  float     fitness;

          chromo_typ(): bits(""), fitness(0.0f){};
          chromo_typ(string bts, float ftns): bits(bts), fitness(ftns){};
};
```

```
////////////////////////////////prototypes/////////////////////////////////////////////////////

void    PrintGeneSymbol(int val);
string  GetRandomBits(int length);
int     BinToDec(string bits);
double  AssignFitness(string bits, double desiredValue, double targetValue);
void    PrintChromo(string bits);
void    PrintGeneSymbol(float val);
int     ParseBits(string bits, float* buffer);
string  Roulette(double total_fitness, chromo_typ* Population);
void    Mutate(string &bits);
string  Crossover(string &offspring1, string &offspring2);

//////////////////////////////////////////////////////////////////////////////////////////////
#endif
```

ga.cpp

```
#include <string>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <time.h>
#include <math.h>

//include definition file
#include "ga.h"


using namespace std;
using std::string;

//-----------------------------------GetRandomBits-------------------------------------------
//
//          This function returns a string of random 1s and 0s of the desired length.
//
//-------------------------------------------------------------------------------------------
string GetRandomBits(int length)
{
        string bits;

        for (int i=0; i<length; i++)
        {
                if (RANDOM_NUM1 > 0.5f)

                        bits += "1";

                else

                        bits += "0";
        }

        return bits;
}

//-----------------------------------BinToDec-----------------------------------------
```

```
//
//          converts a binary string into a decimal
//
//-----------------------------------------------------------------------------
int BinToDec(string bits)
{

          int val = 0;
          int value_to_add = 1 ;
          int indexBit = 0;

          for (int i = bits.length()-1; i > 0; i--)
          {
                    if (bits.at(i-1) == '1') {
      val += value_to_add;//value_to_add;
    }
                    value_to_add *= 2 ;

          }//next bit

          //return val;
          if (bits.at(indexBit) == '1') { return -1 * val; }  // look at the 9th bit which is the index bit and make the value
negative
          else return val;

}


//-----------------------------------ParseBits--------------------------------------------
//
// Given a chromosome this function will step through the genes one at a time and insert
// the float values of each gene into a buffer. Returns the number of weights in the buffer.
//-----------------------------------------------------------------------------------------
int ParseBits(string bits, float* buffer)
{
          //set range
          double rH = -1;
          double rO = 1;

          //counter for buffer position
          int cBuff = 0;

          // step through bits a gene at a time until end and store float values
          // of valid numbers.

          //storage for float value of currently tested gene
          float this_gene = 0.0f;

          for (int i=0; i<CHROMO_LENGTH; i+=GENE_LENGTH)
          {
                    //convert the current gene to float ; set weights to random values between -1 and 1
                    this_gene =   0.002 * BinToDec(bits.substr(i, GENE_LENGTH)) ; // ( ( (double)(rand()
%100)+1)/100  * 2 * rH ) - rH; //
                    buffer[cBuff++] = this_gene;

          } // next gene


          return cBuff;
}

//-----------------------------------AssignFitness---------------------------------------
```

```
//
//          given a string of bits and a target value this function will calculate its
// representation and return a fitness score accordingly
//-------------------------------------------------------------------------------
double AssignFitness(string bits, double desiredValue, double targetValue)
{
          //cout << "Assign Fitness" << endl;

          //holds float values of gene sequence
          float buffer[(int)(CHROMO_LENGTH / GENE_LENGTH)];

          int num_weights = ParseBits(bits, buffer);
          int i;

          // Now we calculate the fitness. First check to see if a solution has been found
          // and assign an arbitarily high fitness score if this is so.

          if ( desiredValue == ceil(targetValue)) {
                    return 999.0f;

          }
          else {
                    return (1/fabs(desiredValue-ceil(targetValue)));
          }

}

//----------------------------------PrintChromo-----------------------------------------
//
// decodes and prints a chromo to screen
//-------------------------------------------------------------------------------
void PrintChromo(string bits)
{
          //cout << "PrintChromo" << endl;
          //holds floating point values of gene sequence
          float buffer[(int)(CHROMO_LENGTH / GENE_LENGTH)];

          //open file for writing
          fstream outputFile;
          outputFile.open("weights.csv", ios::out);
          outputFile.precision(5);

          //parse the bit string
          int num_weights = ParseBits(bits, buffer);

          for (int i=0; i<num_weights; i++)
  {
                    outputFile << buffer[i] << "," ;
                    //PrintGeneSymbol(buffer[i]); uncomment to print weights
  }
          outputFile << endl;

          //close file
          outputFile.close();

          return;
}

//---------------------------------------PrintGeneSymbol-----------------------------
//
//          given a float this function outputs its symbol to the screen
//-------------------------------------------------------------------------------
```

```
void PrintGeneSymbol(float val)
{

		cout << val << " "; // prints the weight value
}

//-------------------------------------Mutate----------------------------------------
//
//		Mutates a chromosome's bits dependent on the MUTATION_RATE
//-----------------------------------------------------------------------------------
void Mutate(string &bits)
{
		//cout << "Mutate" << endl;
		for (int i=0; i<bits.length(); i++)
		{
			if (RANDOM_NUM1 < MUTATION_RATE)
			{
				if (bits.at(i) == '1')

					bits.at(i) = '0';

				else

					bits.at(i) = '1';
			}
		}

		return;
}

//---------------------------------- Crossover ---------------------------------------
//
//  Dependent on the CROSSOVER_RATE this function selects a random point along the
//  length of the chromosomes and swaps all the  bits after that point.
//-----------------------------------------------------------------------------------
void Crossover(string &offspring1, string &offspring2)
{

  //dependent on the crossover rate
  if (RANDOM_NUM1 < CROSSOVER_RATE)
  {
     //create a random crossover point not at the boundary
     int crossover = (int) (RANDOM_NUM1 * (CHROMO_LENGTH-1));

    std::string t1 = offspring1.substr(0, crossover) + offspring2.substr(crossover, CHROMO_LENGTH-1);
    std::string t2 = offspring2.substr(0, crossover) + offspring1.substr(crossover, CHROMO_LENGTH-1);

    offspring1 = t1;
    offspring2 = t2;


  }
}


//-----------------------------------Roulette-----------------------------------------
//
//		selects a chromosome from the population via roulette wheel selection
//-----------------------------------------------------------------------------------
string Roulette(double total_fitness, chromo_typ* Population)
{
		//cout << "Roulette" << endl;
```

```
		//generate a random number between 0 & total fitness count
		double Slice = (RANDOM_NUM1 * total_fitness);

		//go through the chromosones adding up the fitness so far
		double FitnessSoFar = 0.0f;

		for (int i=0; i<POP_SIZE; i++)
		{
			FitnessSoFar += Population[i].fitness;

			//if the fitness so far > random number return the chromo at this point
			if (FitnessSoFar >= Slice)

				return Population[i].bits;

			else return Population[i].bits;
		}
}
```

main.cpp

```
/*************************************************************************
* This program uses a Hybrid GA-NN approach to solve the problem of
* Letter Recognition from the UCI Machine Learning Repository dataset
* ----------------------------------------------------------------------
* Vineet Kashyap
* 02302675
*************************************************************************/

//standard libraries
#include <iostream>
#include <ctime>
#include <string>
#include <math.h>

//custom includes
#include "neuralNetwork.h"
#include "neuralNetworkTrainer.h"
#include "ga.h"

using namespace std;

int main()
{
		float mseForTestingFitness;
		float errorThreshold = 0.006;   // can be changed
		float result;

		//seed random number generator
		srand( (unsigned int) time(0) );

		//create data set reader and load data file
		dataReader d;
		d.loadDataFile("ABCDE.csv",16,5);
		d.setCreationApproach( STATIC, 10 );

		//create neural network
		neuralNetwork nn(16,10,5);

		//create neural network trainer
```

```
	neuralNetworkTrainer nT( &nn );
	nT.setTrainingParameters(0.009, 0.8, false);
	nT.setStoppingConditions(100, 0.01);
	nT.enableLogging("log.csv", 5);

	//storage for our population of chromosomes.
	chromo_typ Population[POP_SIZE];

	//first create a random population, all with zero fitness.
	for (int i=0; i<POP_SIZE; i++)
	{
		Population[i].bits = GetRandomBits(CHROMO_LENGTH);
		Population[i].fitness = 0.0f;
	}

	cout	<< endl << "Genetic Algorithm Starting: " << endl
			<<
"=============================================================================
====" << endl
			<< "POP SIZE: " << POP_SIZE << ", MAX ALLOWABLE GENERATIONS: " <<
MAX_ALLOWABLE_GENERATIONS << ", CHROMO LENGTH: " << CHROMO_LENGTH << endl
			<< "GENE LENGTH: " << GENE_LENGTH << " CROSSOVER RATE: " <<
CROSSOVER_RATE << " MUTATION RATE: " << MUTATION_RATE << endl
			<<
"=============================================================================
====" << endl << endl;

	int GenerationsRequiredToFindASolution = 0;

	//we will set this flag if a solution has been found
	bool bFound = false;

	//enter the main GA loop
  while(!bFound)
	{
		//this is used during roulette wheel sampling
		double TotalFitness = 0.0f;
    //train neural network on data sets


		// test and update the fitness of every chromosome in the population

		for (int i=0; i<POP_SIZE; i++)
		{
			PrintChromo(Population[i].bits); // initialize weights for backpropagation
			//print success
			cout << endl << "Neuron weights saved to weights.csv"  << endl;
			nn.loadWeights("weights.csv");
			cout <<
"=========================================================" << endl;
			cout << "Generations: " << GenerationsRequiredToFindASolution << endl;
			cout <<
"=========================================================" << endl;

			for (int j=0; j < d.getNumTrainingSets(); j++ )
		{
			mseForTestingFitness = nT.trainNetwork( d.getTrainingDataSet() );
		}

			//~ if(mseForTestingFitness < 0.0001) {
				//~ cout << "\nSolution found in " << GenerationsRequiredToFindASolution
<< " generations!" << endl << endl;
```

```
                                        //~ cout << "Testing using the generalization data set" << endl;
                                        //~ //get generalization set accuracy and MSE
                                        //~ double generalizationSetAccuracy =
nn.getSetAccuracy( d.tSet.generalizationSet );
                                        //~ double generalizationSetSSE = nn.getSetSSE( d.tSet.generalizationSet );

                                        //~ cout << "Generalization Set Accuracy: " << generalizationSetAccuracy
<< "%, SSE: " << generalizationSetSSE << endl;

        //~ bFound = true;
        //~ break;
                                //~ }

                                Population[i].fitness = AssignFitness(Population[i].bits, errorThreshold,
ceil(mseForTestingFitness));
        TotalFitness += Population[i].fitness;

                                if (Population[i].fitness == 999.0f)
                                {
                                        cout << "\nSolution found after " << GenerationsRequiredToFindASolution
<< " generations!" << endl << endl;
                                        cout << "Testing using the generalization data set" << endl;

                                        //get generalization set accuracy and SSE
                                        double generalizationSetAccuracy =
nn.getSetAccuracy( d.tSet.generalizationSet );
                                        double generalizationSetSSE = nn.getSetSSE( d.tSet.generalizationSet );

                                        cout << "Generalization (Testing) Set Accuracy: " <<
generalizationSetAccuracy << "%, SSE: " << generalizationSetSSE << endl;

        bFound = true;
        break;
                                }

                                // create a new population by selecting two parents at a time and creating offspring
                                // by applying crossover and mutation. Do this until the desired number of offspring
                                // have been created.

                                //define some temporary storage for the new population we are about to create
                                chromo_typ temp[POP_SIZE];

                                int cPop = 0;

                                //loop until we have created POP_SIZE new chromosomes
                                while (cPop < POP_SIZE)
                                {

                                        // we are going to create the new population by grabbing members of the old
population
                                        // two at a time via roulette wheel selection.
                                        string offspring1 = Roulette((int)TotalFitness, Population);
                                        string offspring2 = Roulette((int)TotalFitness, Population);


                                        //add crossover dependent on the crossover rate
                                        Crossover(offspring1, offspring2);

                                        //now mutate dependent on the mutation rate
                                        Mutate(offspring1);
                                        Mutate(offspring2);
```

```
                                        //add these offspring to the new population. (assigning zero as their fitness
scores)
                                        temp[cPop++] = chromo_typ(offspring1, 0.0f);
                                        temp[cPop++] = chromo_typ(offspring2, 0.0f);

                                }//end loop

                                //copy temp population into main population array
                                for (int i=0; i<POP_SIZE; i++)
                                {
                                        Population[i] = temp[i];
                                }

                                ++GenerationsRequiredToFindASolution;

                                // exit app if no solution found within the maximum allowable number
                                // of generations
                                if (GenerationsRequiredToFindASolution > MAX_ALLOWABLE_GENERATIONS)
                                {
                                        cout << "No solutions found after " << GenerationsRequiredToFindASolution
<< " Generations" << endl ;
                                        bFound = true;
                                        break;
                                }
                        }
                }

                cout << endl << endl << "-- END OF PROGRAM --" << endl;
}
```

makefile

```
main.exe        :       main.o  dataReader.o    neuralNetwork.o  neuralNetworkTrainer.o
        ga.o
        g++ -Wl,-s -o main.exe main.o dataReader.o    neuralNetwork.o  neuralNetworkTrainer.o    ga.o

main.o          :       main.cpp  dataEntry.h     dataReader.h     neuralNetwork.h
        neuralNetworkTrainer.h
        g++ -c -fpermissive -fconserve-space main.cpp

dataReader.o            :       dataReader.cpp   dataReader.h
        g++ -c -fpermissive -fconserve-space dataReader.cpp

neuralNetwork.o         :       neuralNetwork.cpp neuralNetwork.h
        g++ -c -fpermissive -fconserve-space neuralNetwork.cpp

neuralNetworkTrainer.o          :       neuralNetworkTrainer.cpp  neuralNetworkTrainer.h
        g++ -c -fpermissive -fconserve-space neuralNetworkTrainer.cpp

ga.o            :       ga.cpp  ga.h
        g++ -c -fpermissive -fconserve-space ga.cpp
```