Cosc 420 – Neural Networks Assignment

Introduction

My report will start with a section on the structure of my code and the choices I made when designing my network. Then I will talk about the things I decided to test my network on and why I wanted to. Then I will present my findings and discussion.

As per the assignment specification I have designed a fully connected feed forward network with a single layer of hidden neurodes. The first thing I considered when designing the network was the overarching structure of my program and how that should be informed by the operation of the network itself.

I started by thinking about how to represent the various elements required for the network to run and I decided on an implicit representation of the weights between layers. By this I mean that as far as the code is concerned there is no explicit connection between units. My reason for doing so is that when doing research prior to starting the assignment I found example code that used the java library class HashMap to explicitly represent connections in a rather complicated way and that seemed to be rather excessive. So with this implicit representation in mind I began by designing a Neurode class to encapsulate all of the state and behavior associated with a neurode.

The Neurode class has a series of global data fields. The weights array is of a size determined by the number of units in the following layer, and at each index of the array is a double that stores the weight from this neurode to the neurode in the next layer indicated by the index. For example, in the weights array of a given hidden neurode, x, weights[0] gives the weight of the connection between x and the first neurode in the output layer. This simple array representation of the

weights forms the basis of the individual algorithms involved in the learning process.

I use two more double array data fields in the Neurode class. The first is the changes array. I realised that I essentially needed two weights arrays as when calculating the weight changes during the backwards pass of the backpropagation algorithm the original weight values from other layers are required. So on the backwards pass I calculate the weight changes and store them in the changes array. The second extra array is used to store the changes from the previous epoch, this is used for applying momentum to weight changes. The bias weight, activation value, and error value are also represented by double data fields.

Most of the behavior of the Neurode class is apparent when looking at the code. The methods of note are the constructor and updateWeights(). The constructor method accepts an int that is the size of the next layer of units, this is used to initialise the weight array, and two Boolean flags that are used to determine which layer the Neurode is in. The neurode is then intialised according to the layer it is in. The weights array and bias are initialised with random weights in the range +/- 0.3. The updateWeights() method is used to apply the weight changes calculated in the backwards pass of the backpropagation algorithm.

All of the file input/output is handled in the file NetworkApp. It is also the file that contains my programs main method. I use a series of scanners to read in all the parameters, input patterns, and expected outputs and store them in arrays. I had to use a separate scanner to read the number of lines in the input file in order to use arrays. I did this because I did not want to use ArrayLists as I find the syntax required to use them unwieldy, and they are slower than traditional arrays. I considered using a separate class for the file input/output in order to keep NetworkApp clean and used only for the interface and set up. But I decided against this as it would have required passing information from some FileIO

class to NetworkApp and then again to NeuralNetwork to actually initialise the network and that seemed to be overly complicated.

Using the information from the input files I create an instance of NeuralNetwork and then program execution goes into a while loop where operation of the network can be specified by user input.

The structure of the network itself and all of the processes involved in the operation of the network are contained within my NeuralNetwork class. The constructor accepts all of the parameters and patterns parsed in the NetworkApp class and it uses these to intialise all of NeuralNetwork's data fields. Each layer of the network is represented by an array of Neurode objects, their position in the array is equivalent to a diagrammatic representation of a neural network. So input[0] is the first neurode in the input layer. All of the code for backpropagation in contained within the learn() method.

I didn't make any particularly interesting choices in the design of my backpropagation algorithm. In calculating the activation function for each neurode I decided upon the standard sigmoidal function as this seemed like the only choice. Similarly when considering the pattern and population error calculations I went with the examples shown in the lecture notes, again because it seemed like the only real choice and researching other functions seemed to be tangential to the purpose of the assignment.

I chose to use the sequential update method when considering how to update the weights in my network; primarily this was due to Haykin section 4.3. In the section he discusses the differences between sequential and batch updating and one of the key differences is: due to the stochastic nature of sequential update the network is less likely to get stuck in a local minimum. I wrote a method called shuffle, which shuffles an array of integers that is the same size and range as the number of input patterns. When choosing from both the pattern array and the

teaching array the order is determined by the integers stored in the shuffled array which ensures that a pattern in the pattern array is compared to the corresponding teacher pattern in the teacher array.

Background

When exploring the network I have built, one of things that interested me the most was how the different parameters affected the learning speed of the network. Particularly the affect of the learning constant and momentum term on the number of epochs required for the population error to fall below the given error criterion.

In Haykin chapter 4, he states that the smaller the learning constant, the smaller the changes to weights will be from one epoch to the next. This results in a smoother trajectory in weight space, but it comes at the cost of a slower learning speed. On the other hand if the learning constant is too large then the changes to weights from one iteration to the next will become too large and the network will become unstable. I am interested to see at what value of learning constant the network becomes unstable and how that point of instability differs between data sets.

I would also like to see if there is any negative aspect of using a momentum term and if so what the ideal momentum term would be for a given data set.

Another aspect of the network that is of particular interest to me is its' ability to generalise. One of a neural networks most useful features is its ability to learn from a training set of data and then apply its "knowledge" to previously unseen data. I want to see how well my neural network can classify patterns in relation to the size of training sets smaller than or equal to the size of the test data set. In Haykin chapter 4.12 he gives a formula for the size of a training set needed for good generalization. The formula is $N = O(W/\varepsilon)$ where w is the total number of

weights in the network and ε is the accepted error and O(.) denotes order of the quantity within. This means that with an accepted error of 10% in the validation set the training set would need to be 10 times the number of weights and biases in the network. Given that I am interested in the generalisation capability of my network from smaller training sets and based on Haykin's formula I would not expect the results to be what you would consider "good" generalisation. But I am interested to see how the network performs in practice.

The Affect of the Learning Constant and Momentum term on Time to Learn

Method

To test the affect of the learning constant and momentum term on time to learn I looked at two different data sets, namely XOR and Iris. I tested each term separately from the other in order to ensure sensible results.

The default learning constant, momentum and error criterion were 0.1, 0.9, and 0.02 respectively.

The process for generating the data for both the learning constant and momentum term were essentially identical. For each interval in the range of the term I wanted to test I ran my network's learning algorithm 50 times and each time returned the number of epochs it took for the population error to fall below the error criterion. I then calculated the mean and the standard deviation over the 50 cycles of my program and output them along with the value of the independent variable I was testing. I had a threshold set in place of 10000 epochs in the case of XOR so if the network became trapped in a local minimum the learning process would cease. When calculating the mean and standard deviation I excluded the times when XOR caused the number of epochs to reach the threshold as I reasoned that this would skew the data.

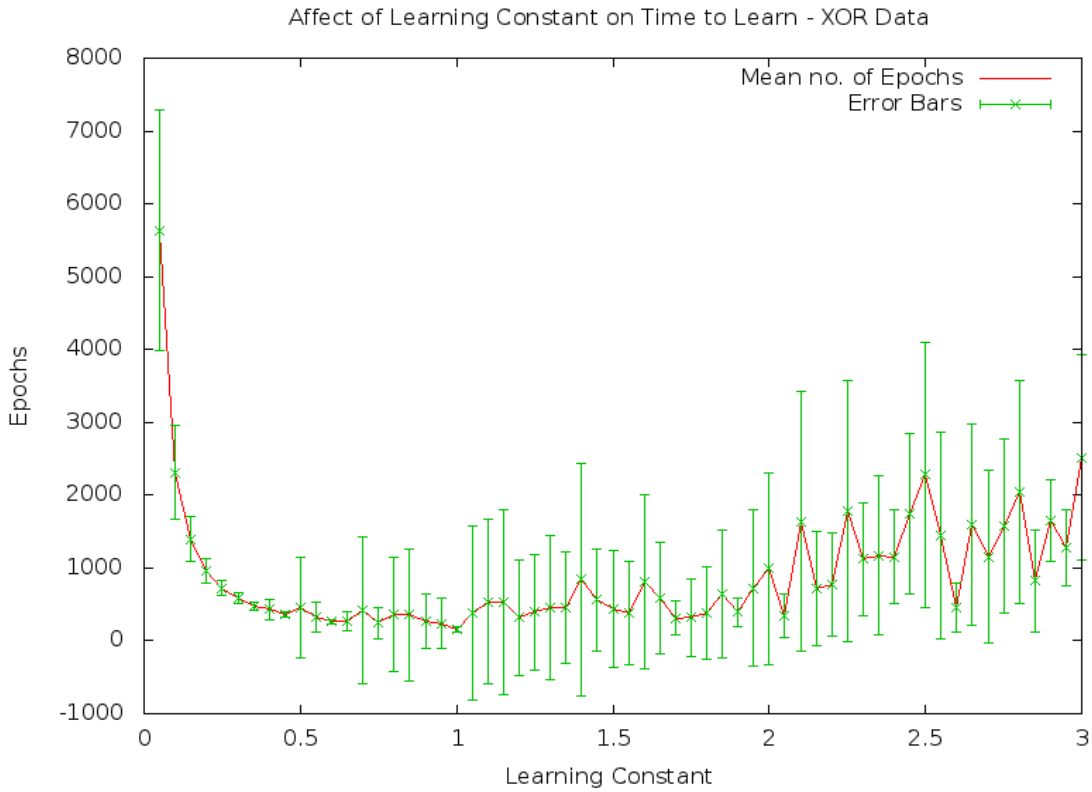
I performed the procedure detailed above for both XOR and Iris. For momentum I used the range $0 \leq \alpha \leq 0.95$ and an interval size of 0.05. Because the momentum term α must be, according to Haykin, $0 \leq \alpha < 1$, the range of values that I tested were the same for both data sets.

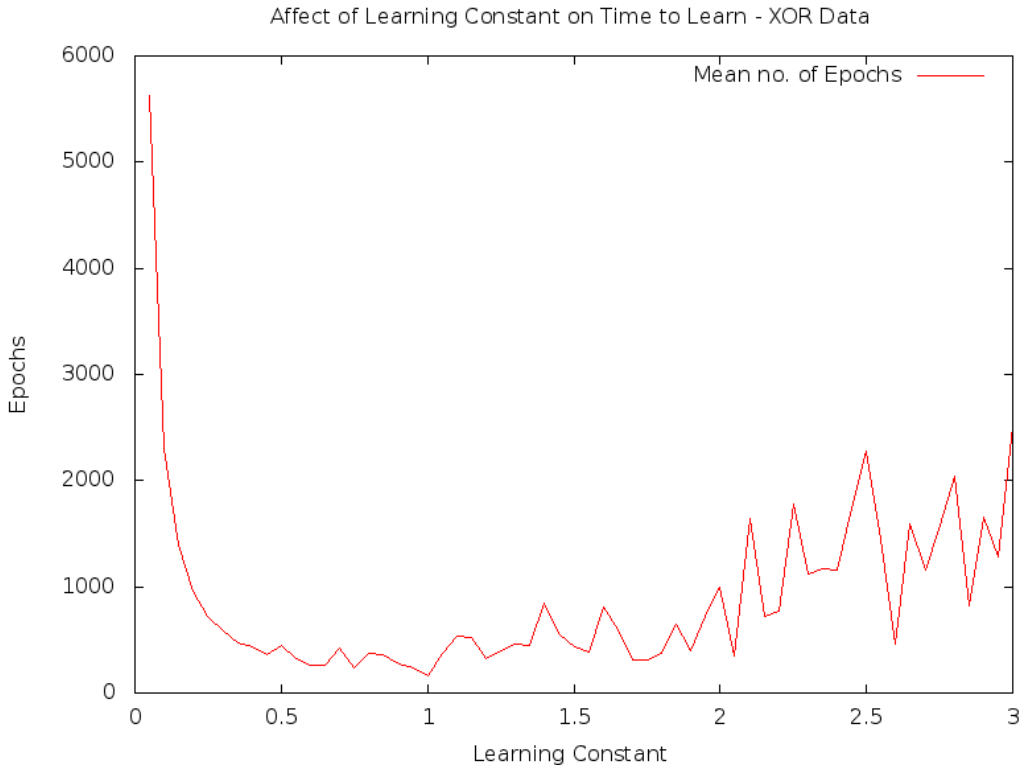
When testing the learning constant, some rough initial testing showed that the different data sets coped with different ranges of learning constant. I decided on

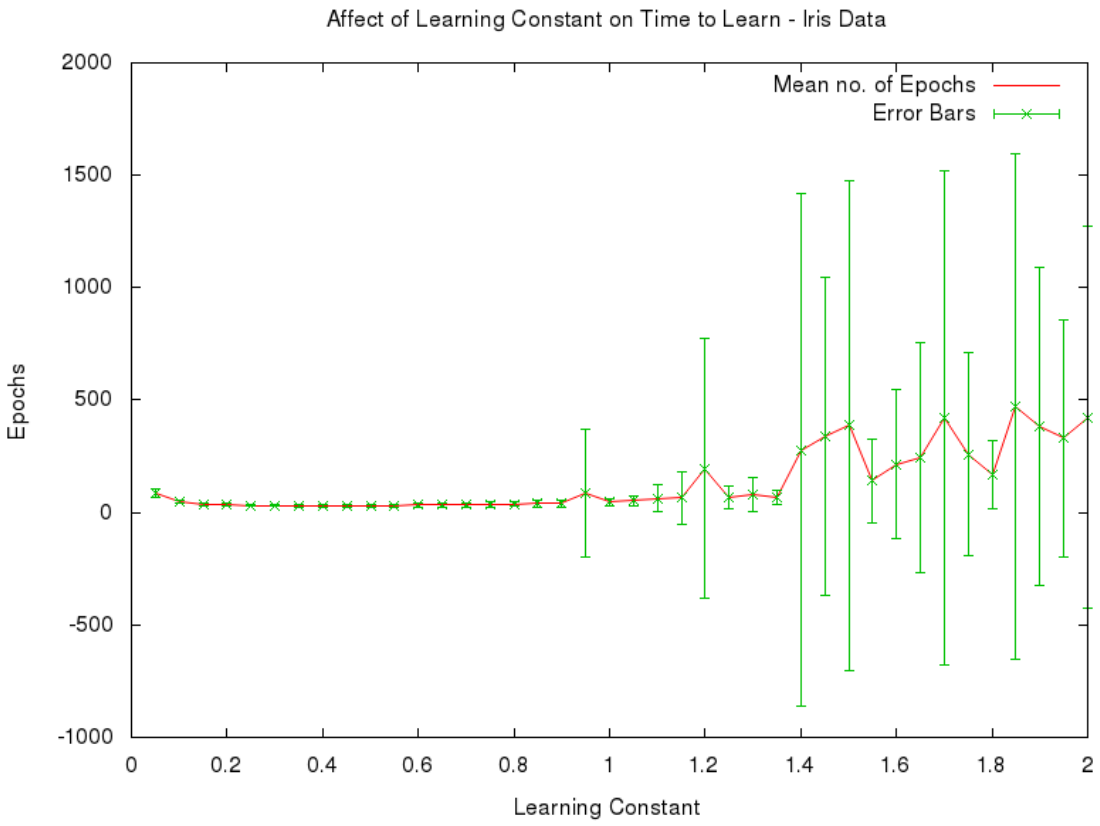
the range $0.05 \leq \alpha \leq 3$ for XOR and $0.05 \leq \alpha \leq 2$ for Iris, with an interval size of 0.05.

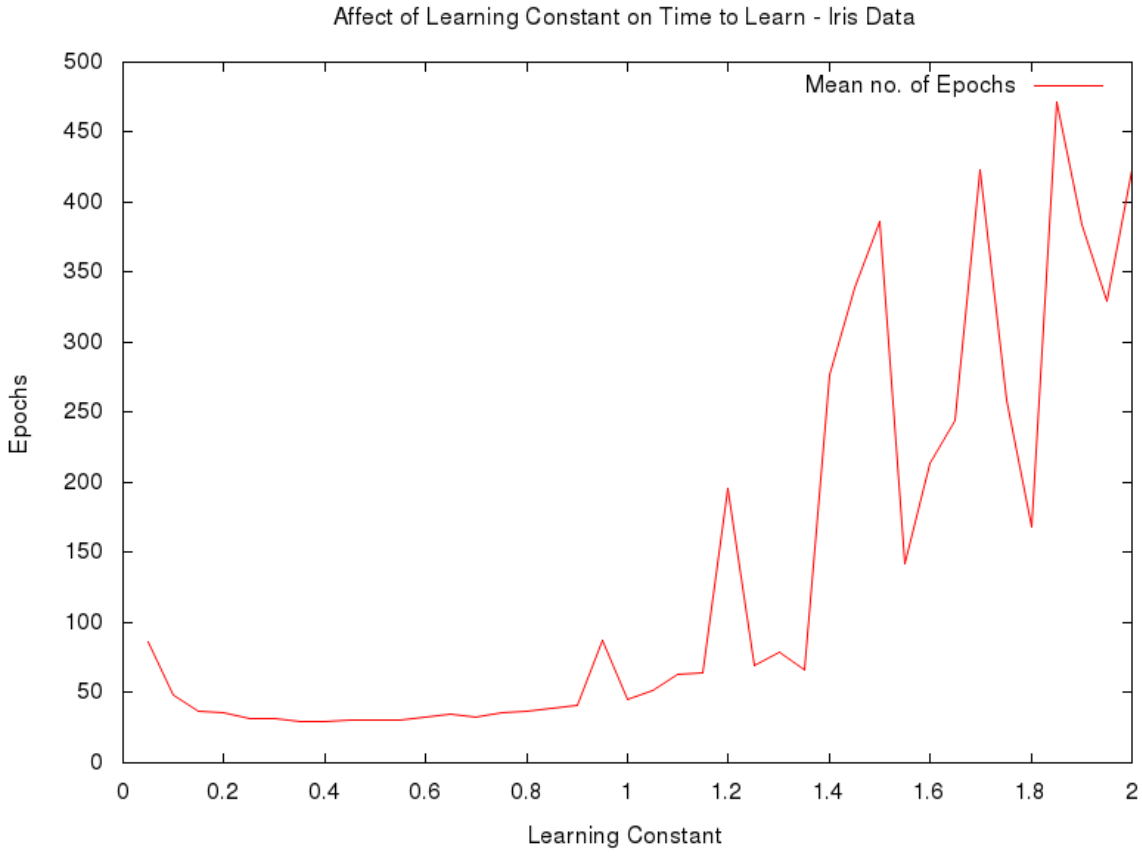
Once I had generated all of this data I graphed it using gnuplot.

Results

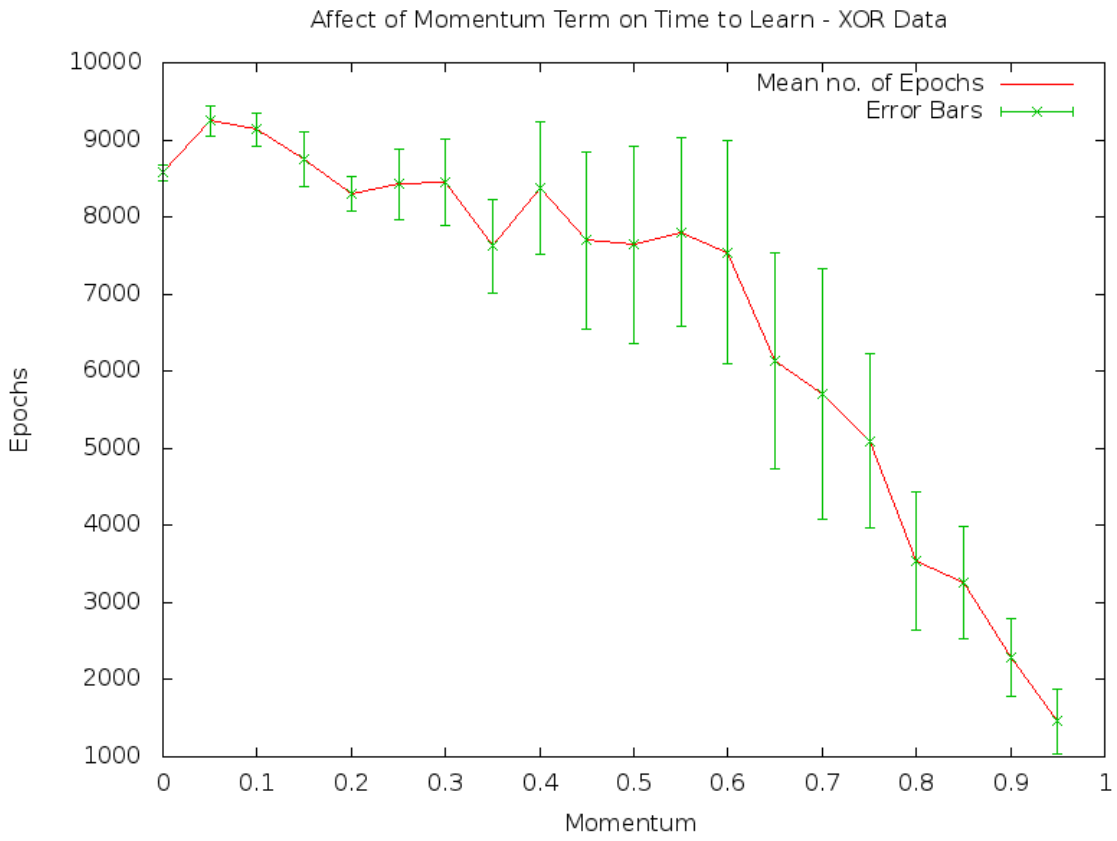

Affect of Learning Constant on Time to Learn - XOR Data
Mean no. of Epochs
Error Bars
Epochs
Learning Constant
8000
7000
6000
5000
4000
3000
2000
1000
0
-1000
0
0.5
1
1.5
2
2.5
3

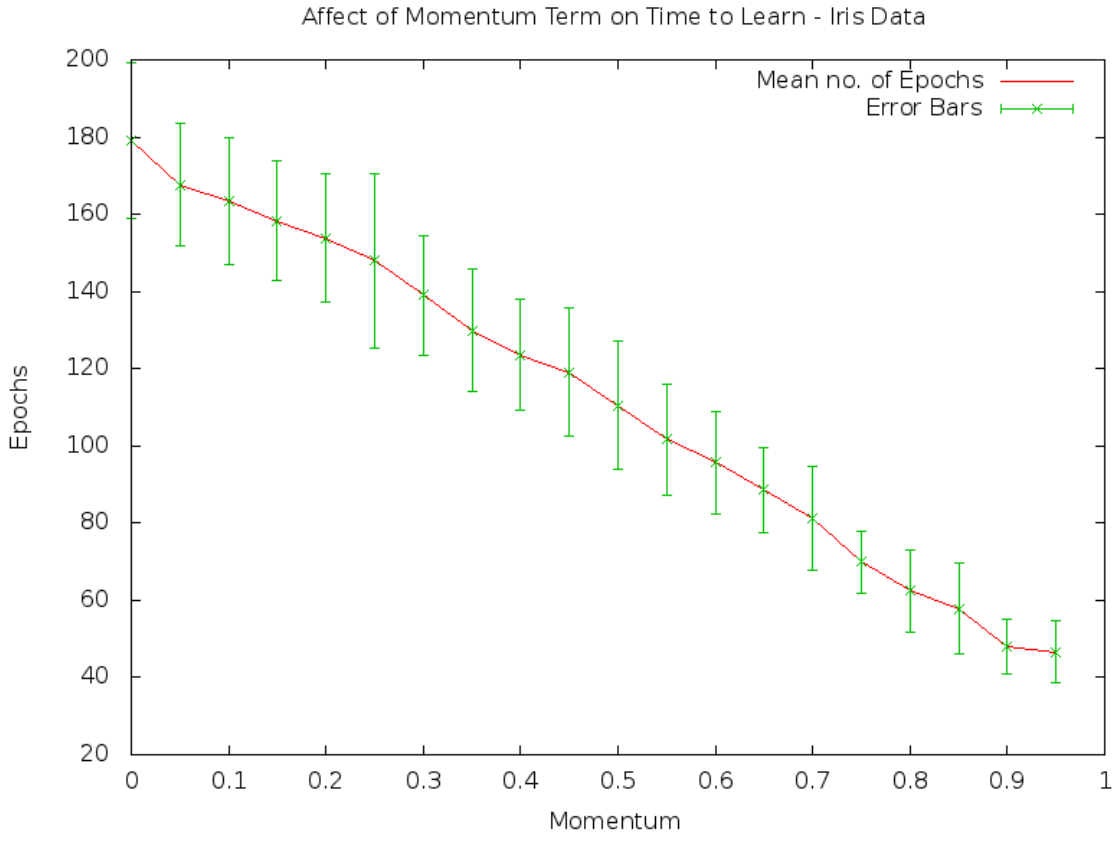

Affect of Learning Constant on Time to Learn - XOR Data
Mean no. of Epochs
Epochs
Learning Constant
6000
5000
4000
3000
2000
1000
0
0
0.5
1
1.5
2
2.5
3


Affect of Learning Constant on Time to Learn - Iris Data
Mean no. of Epochs
Error Bars
Epochs
Learning Constant
2000
1500
1000
500
0
-500
-1000
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2


Affect of Learning Constant on Time to Learn - Iris Data
Mean no. of Epochs
Epochs
Learning Constant
500
450
400
350
300
250
200
150
100
50
0
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2

The graphs displayed above show the results of my experiment to determine the affect of different learning constants on the XOR and Iris data sets. I have made two graphs for each data set, one with error bars and one without. The reason for doing so it that I think the error bars make it more difficult to see the trend and actually skew the graph, particularly in the case of the Iris data set.


Affect of Momentum Term on Time to Learn - XOR Data
Mean no. of Epochs
Error Bars
Epochs
Momentum
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


Affect of Momentum Term on Time to Learn - Iris Data
Mean no. of Epochs
Error Bars
Epochs
Momentum
200
180
160
140
120
100
80
60
40
20
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1

Discussion

I'll first consider the results of the learning constant experiment. At the initial learning constant value of 0.05 XOR took an average of around 5500 epochs for the population error to fall below the error criterion but this number of epochs decreased by half at the next learning constant of 0.1. The time to learn continued to decrease until a learning constant of 0.4 where it didn't change too dramatically until a learning constant of 1 was reached. From a learning constant of 1 the numbers of epochs taken to learn begins to become less stable until around a learning constant of 2 where the graph starts to oscillate sharply. This indicates that the learning constant is too high for efficient learning. Based on these results it seems that an ideal learning constant for the XOR data set is some value in the range 0.4 – 1.

The Iris data set provided an interesting and far more transparent result compared to the XOR data set. The initial learning constant of 0.1 resulted in the network taking somewhere in the region of 80 – 90 epochs to fall below the error criterion threshold. But the learning time appears to stabilise at the very next learning constant of 0.1. The learning time remains consistently under 50 epochs until around 0.9 where it spikes. From there the network begins to destabilise and once at a learning constant of 1.2 the network begins to oscillate very sharply by large numbers of epochs. Based on this result an ideal learning constant for the Iris data set is a value between 0.2 and 0.85.

It is interesting to note the differences between the results of the XOR and Iris data sets. Where the XOR data set started with a huge number of epochs and then decreased rapidly to a far lower number the Iris data set started low and then got slightly lower where it stabilised. I believe this has to do with the number of input patterns present in each data set. The Iris data set has 150 input patterns, where as the XOR data set has only 4, so more weight adjustment would occur in each epoch, in the case of Iris. Another difference worth noting is

that the curve for XOR got slowly more jagged over time, whereas the curve for Iris was very smooth and then suddenly began to spike. Perhaps this is again due to the number of input patterns for each set.

The results of the momentum term experiment are simple and not entirely unexpected. It would seem that regardless of which data set you are using the higher the momentum is the less epochs are required for the population error to fall below the error criterion threshold. So a higher momentum is always better, at least as far as my experiment is concerned.

Testing Generalisation from a Smaller Training set to a Larger Data set

Method

The parameters used when generating the experiment data were 0.1 learning constant, 0.9 momentum, and 0.02 error criterion. Although the error criterion is not used as the following procedure runs for a defined numbers of epochs. The data set used is the Iris data set as it was the only one suitable for testing generalisation capability. I took an equal number of patterns from each flower class for the training sets.

To generate the data that would enable me to determine my network's ability to generalise I had to write a fair bit of extra code. The first thing I did was to write more parsing code so that my program could read from a file containing the test data set and also from another file containing the expected output from each of those elements in the test data set. These patterns are stored in arrays like the input and teacher patterns and are passed to the constructor when a NeuralNetwork object is made.

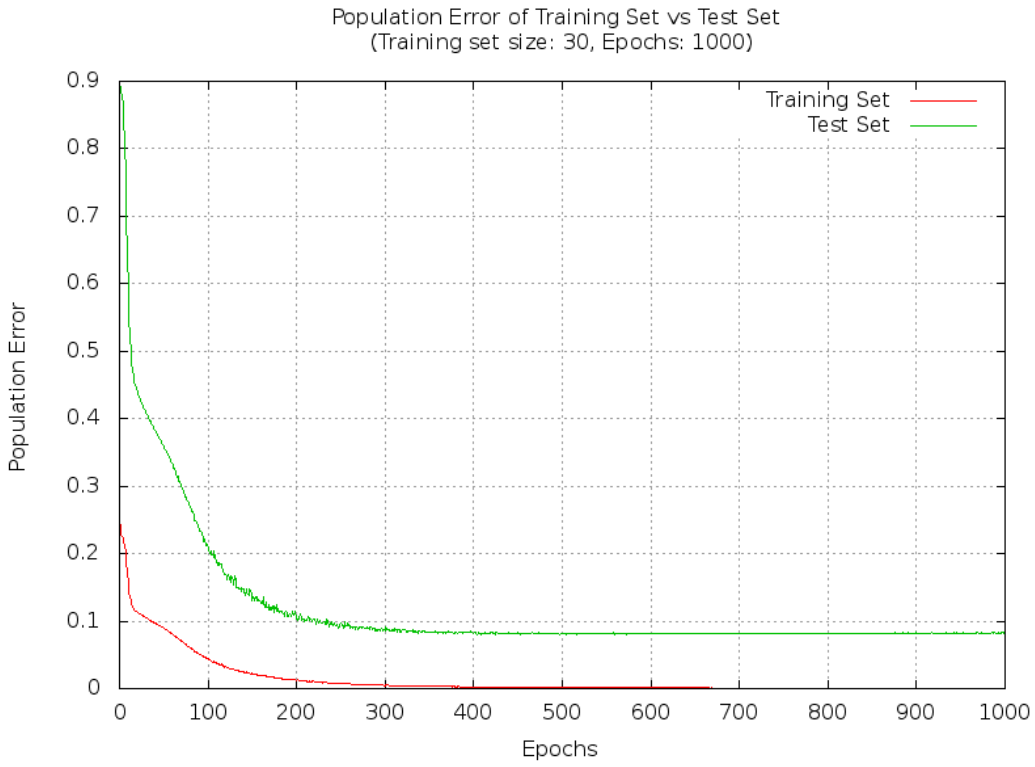
I made a new method called test that calls the methods that calculate the activations of each unit for every pattern in the test data set. It then returns the population error for the test data set.

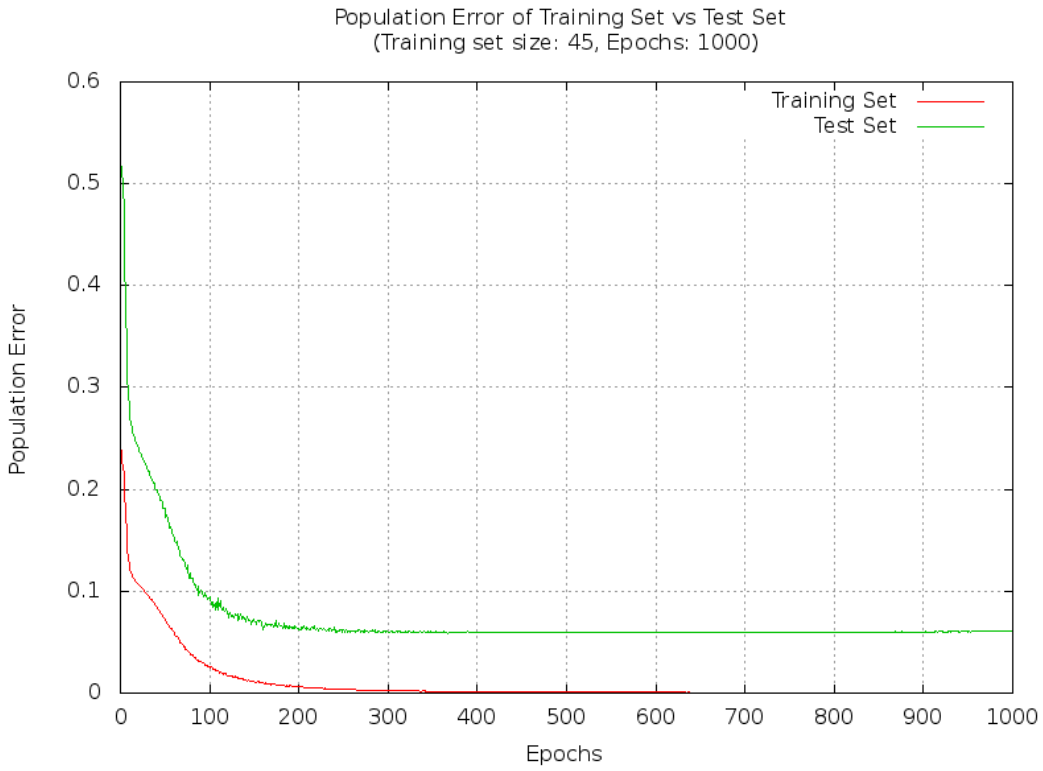
I then modified my learn method, which contains the code that controls backpropagation, so that if passed a true value it would run the test method after every epoch and store the population error of the training set and the test data set in an array. When a given epoch limit had been reached it would return the array containing the population error of the training set and test data set for every epoch.

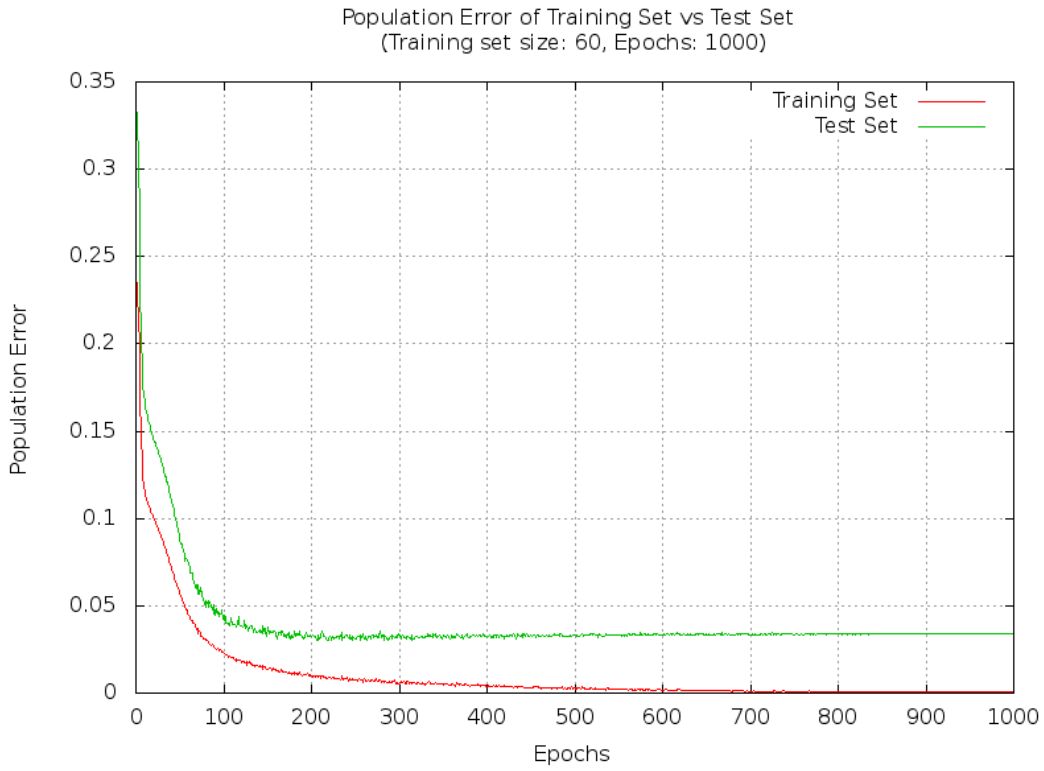
In my NetworkApp class I ran the modified learn method 50 times and summed the population error values that were returned. I then divided each value by 50 to get the average population error at each epoch as this is more representative of the network's performance.

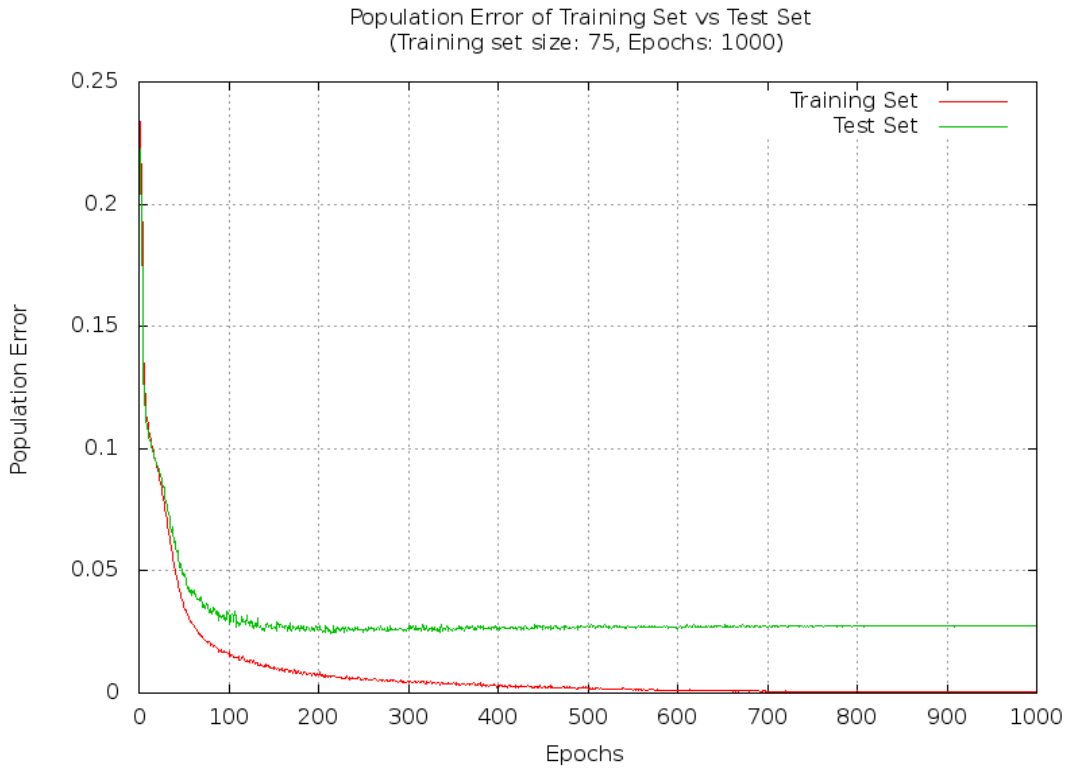
I performed this procedure for different sizes of training set and for different numbers of epochs. The different sizes of training set is to see how generalisation performance changes in relation to the size of the training set. Running the procedure for a different number of epochs is a result of finding the initial graph less informative than I would have liked. The training set, test data set divisions that I tested are 30/120, 45/105, 60/90, and 75/75 and I ran each of these for 250, 500, and 1000 epochs as each time I couldn't be sure that I had enough data to make an accurate statement about the generalisation capability of my network. I will include the data for 1000 epoch runs below.

Results


Population Error of Training Set vs Test Set
(Training set size: 30, Epochs: 1000)
Training Set
Test Set
Population Error
Epochs
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
100
200
300
400
500
600
700
800
900
1000


Population Error of Training Set vs Test Set
(Training set size: 45, Epochs: 1000)
Training Set
Test Set
Population Error
Epochs
0
0.1
0.2
0.3
0.4
0.5
0.6
0
100
200
300
400
500
600
700
800
900
1000


Population Error of Training Set vs Test Set
(Training set size: 60, Epochs: 1000)
Training Set
Test Set
Population Error
Epochs
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0
100
200
300
400
500
600
700
800
900
1000


Population Error of Training Set vs Test Set
(Training set size: 75, Epochs: 1000)
Training Set
Test Set
Population Error
Epochs
0.25
0.2
0.15
0.1
0.05
0
0
100
200
300
400
500
600
700
800
900
1000

Discussion

The formula found in Haykin chapter 4.12 ($N = O(W/\varepsilon)$) suggests for an accepted error of 10% in the test data set you would need the number of training samples to be 10 times the number of weights and biases in your network. If we look at the first graph which represents a training set of size 30 and a test data set of 120 we can see that the population error of the test data set is below 0.1. This contradicts Haykin's formula. This is entirely unexpected and I am unsure why this is the case, especially considering that it is the smallest training set I tested.

Another perplexing aspect of my results is that at no point in any of the graphs does the generalisation peak and begin to get worse. This is not in line with what I expected either. I thought that at some point the network would overfit to the training data and the generalisation would worsen. But you can see from the results that this doesn't appear to happen. In fact, in every graph the generalisation performance appears to plateau and hold steady all the way up to 1000 epochs. Given that in every case the population error of the training set is already so close to zero it seems that it is not possible to overtrain a network with the Iris data set. So it is difficult to make a definitive statement regarding the generalisation capabilities of my network. However, in this case the results could certainly be interpreted as my network providing good generalisation of the Iris data set.

Something that I did expect was a larger training set will result in better generalisation and this can be seen clearly in the results. The smallest training set of 30 elements gave a population error of the test data set of under 0.1. But the largest training set, consisting of 75 elements, making it equal to the test data set, gave a population error of under 0.05.

Conclusion

Designing and implementing my neural network has given me a far deeper understanding of the way backpropagation networks operate. It was very interesting to find that my results differed from what the theory suggested would happen. I would like to know more about why that happened, particularly if it is due to some fault in my implementation or methodology. I would like to implement and explore other types of neural networks to develop a deeper understanding of their operation and I also would like to test my network on other types of data.

Appendix

Compile Neurode.java, NetworkApp.java, and NeuralNetwork.java using javac. To run the program type into the command line: java NetworkApp

The program will then parse the in.txt, teach.txt, and param.txt from the same directory as the program files. Once that is done control is turned over to the user. The operations are as follows:

c: Press c and push enter to create a new network with newly randomised weights.

l: Press l and push enter to run the learning algorithm.

t: Press t and push enter. You will then be prompted to enter an input pattern for testing. Type the input pattern one number at a time and push enter after each number you type. Once a full input pattern has been entered the activations of the units will be calculated and printed out.

w: Press w and push enter to print out the value of the weights in the network.

a: Press a and push enter to print out the activations of the neurodes in the network.

q: Press q to quit the program.

In the event that XOR gets trapped in a local minimum you will have to kill the process by pressing ctrl c.