Nick Peters
COMP 469
04/27/09

# **Assignment #7 - Neural Networks**

The purpose of this assignment is to use a feed-forward neural network to classify points into either type A or type B. Type A points are contained in a unit circle whose center is placed at the origin of a cartesian coordinate system. The circle is enclosed by a square with sides of 4 units. Points that are outside of the circle and inside the square are classified as a type B point.

Points of type A are generated by first choosing a random x-value between -1 and 1. The corresponding y-value is calculated using the formula for a circle ($y=\mathrm{sqrt}(1-x^2)$). This method produces two values: a positive and negative value. Only one of the values are randomly selected and added it to the learning set.

A similar approach is taken to generate points of type B. A random x-value between -2 and 2 and a random y-value between -2 and 2 are generated. The point is then verified to not be inside the circle using the distance formula between the random point and the origin. If the distance is less than or equal to 1, then the point is inside the circle. In this case, another y-value is randomly generated until it is outside of the circle, at which point it is added to the learning set.

The feedforward network consists of 3 input nodes (one of them being the bias node), 8 hidden nodes and 2 output nodes. Two of the input nodes are set to the x and y values of a point in the learning set and the bias node is always set to 1. The eight hidden nodes correspond to lines that make up a polygon that approximates the circle. The output nodes determine if the point is of type A or of type B. All nodes in each layer are connected to nodes in both the next and previous layer. All of the weights between each node is initially assigned a random weight between -.1 and .1.

When the system is given a point, it will take the weighted sum of each input for each of the 8 hidden nodes. It will then use an activation function to determine the activation value of each of the hidden nodes. The activation function used is the sigmoid function since it converges to 0 in the negative direction and 1 in the positive direction. The weighted sums for all of the hidden nodes for each output node and fed into the activation function to determine the final activation value for each output node.

In the case that the point read is used for training, the system would then go through the backpropagation process to update the weights between nodes. This method first calculates delta values for each output node, followed by delta values for each hidden node. Using these values we are able to update the weights. The following formula is used to update the weights of the connections between the nodes: Learning Rate * Delta Value + Momentum Factor * Momentum. The learning rate determines how fast the network learns. A lower learning rate requires more training iterations, while a higher learning rate allows the network to converge more rapidly. However, the higher the learning rate increases the chance of having a non-optimal solution (1). The value used for the learning rate is 0.5. The momentum amplifies the learning rate and causes a faster convergence rate and helps the network escape from local minima (2). The value used for the momentum is 0.1. These values were chosen

simply because they were the default values used in the python script that inspired this Java implementation (3).

For testing, 1,000 points of each type are generated, shuffled together and fed into the neural network 1,000 times. Using this training set and backpropagation, the network is able to learn to distinguish between the two types of points. Backpropagation is a learning method that involves comparing the output of the network to the expected output. Given this information, the network is able to adjust the weights of the connections between the nodes after it reads in each individual point (as opposed to taking a batch approach).

After the neural network learns the data set, 100 more test points are fed in (50 of each type). The error is computed (number of incorrect guesses / total number of points) and displayed for visual inspection. With the setup previously described the neural network got the classification correct 98.5% of the time. For actual results see appendix A and to see the results of different training set sizes, see appendix B. Appendix C will give you a graph of the sum-squared error as the network learns from the sample set.

Determining whether or not a point is classified as type A or B is done by looking at the two output nodes in the network. If the first node is greater than or equal to the second node, then it is classified as type A; otherwise it is classified as type B.

Overall I found this problem very interesting. I originally modified the python script I found, but had overflow issues when using the sigmoid function (the script uses the hyperbolic tangent function instead). In my final program, I could not get correct results. I changed my sigmoid function, changed my backpropagation algorithm, and changed the way I generate sample points. It turns out that the bias input node wasn't returning the correct output. Once I changed that, the neural network worked fine. I think it was all of the troubleshooting that I did that lead me to have a greater understanding of feedforward networks and backpropagation.


1 - http://grb.mnsu.edu/grbts/doc/manual/Backpropagation_Neural_Netw.html
2 - http://homepages.gold.ac.uk/nikolaev/311practbp.htm
3 - http://arctrix.com/nas/python/bpnn.py

**Appendix A: Sample Results**

| X | Y | Output #1 | Output #2 | Guessed Type | Actual Type |
|---|---|---|---|---|---|
| 0.59062147 | -0.8069487 | 0.90640401 | 0.09359602 | A | A |
| 0.29809695 | 0.9545356 | 0.94900579 | 0.0509943 | A | A |
| -0.6335852 | 0.77367293 | 0.99987025 | 1.30E-04 | A | A |
| 0.37062641 | 0.92878203 | 0.97319866 | 0.02680141 | A | A |
| 0.62836099 | 1.4648659 | 1.63E-04 | 0.99983698 | B | B |
| 1.95003782 | -1.2368613 | 1.22E-10 | 1 | B | B |
| 0.51559105 | 0.85683479 | 0.99212769 | 0.00787234 | A | A |
| 1.66415417 | 0.02485885 | 0.05658564 | 0.9434143 | B | B |
| 0.20155597 | -1.7749627 | 0.02861275 | 0.97138718 | B | B |
| 0.65967106 | -1.0207108 | -0.2226113 | 0.77738818 | B | B |

% Error: 0.0%

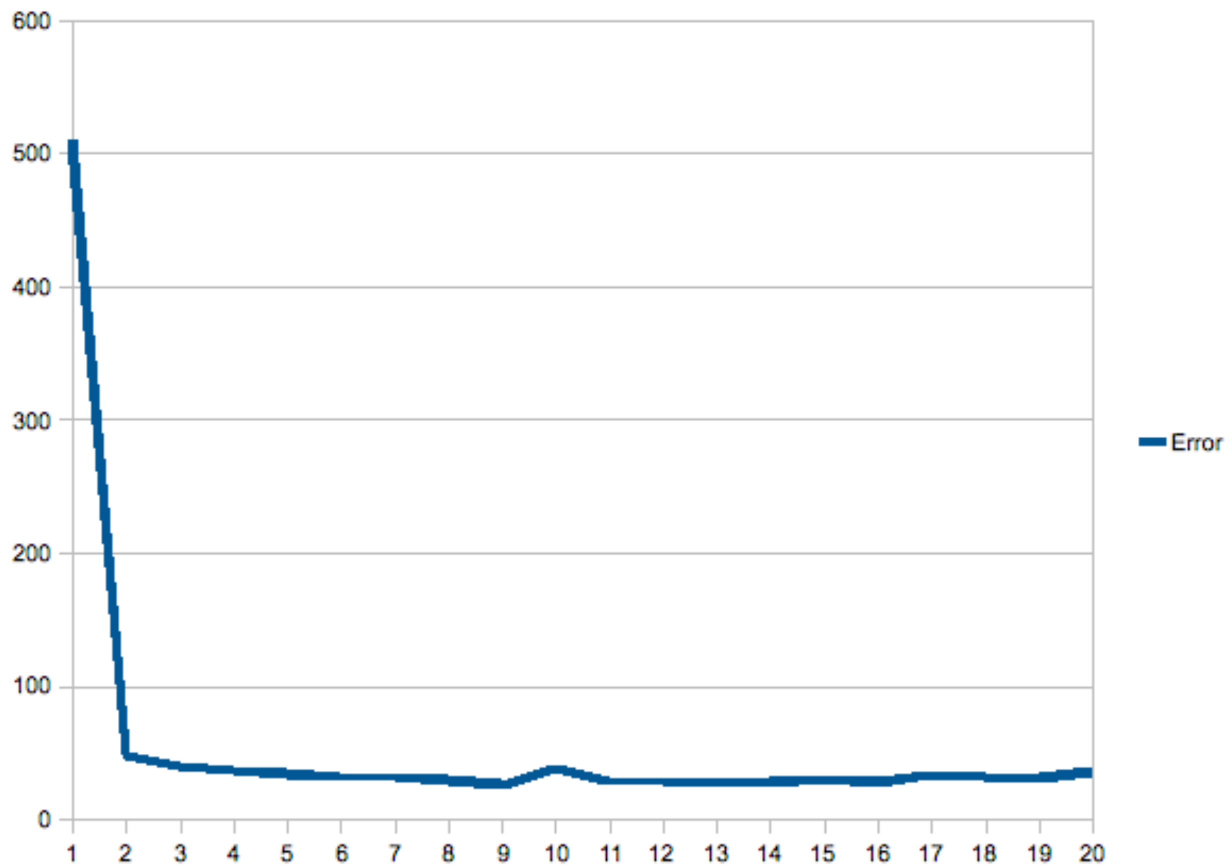
These sample results were a sample set of 1,000 of each type and having the set ran through the network 1,000 times.

**Appendix B: Error Rates**

| Number of Samples of Each Type | % Error |
|---|---|
| 10 | 31.50% |
| 25 | 14% |
| 50 | 7% |
| 100 | 4% |
| 200 | 2% |
| 500 | 2% |
| 1000 | 1.50% |
| 2000 | 0% |
| 5000 | 1% |
| 7500 | 1.50% |
| 10000 | 3% |

This table shows us the % Error we get as we increase the number of samples in our training set. Note that as we introduce more and more samples into our training set, we may see a higher % error which results from overtraining the network.

**Appendix C: Sum Squared Error**




This graph gives us the sum-squared error for each 50 iterations during the learning process. Each iteration is when we pass the learning set through the neural network. These results were from a training set of 1,000 A points and 1,000 B points and ran 1,000 times.