EEE511
Artificial Neural Networks
Homework Assignment Two

Jason Harris

527334236

Problem One

The discriminating line produced by a single hidden node is perpendicular to the weight vector for that node's inputs. The output of a node with a threshold activation function is 1 on the side of the discriminating line pointed to by the weight vector and zero on the side of the discriminating line away from the weight vector (see Figure 1).

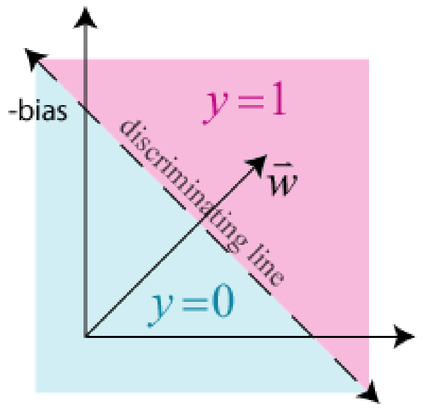

Figure 1: The discriminating line produced by a single node.

Thus, to distinguish the two classes, an output node must produce a shape that will isolate one class from another. This can be accomplished by placing several discriminating lines together to form the shape shown in Figure 2.

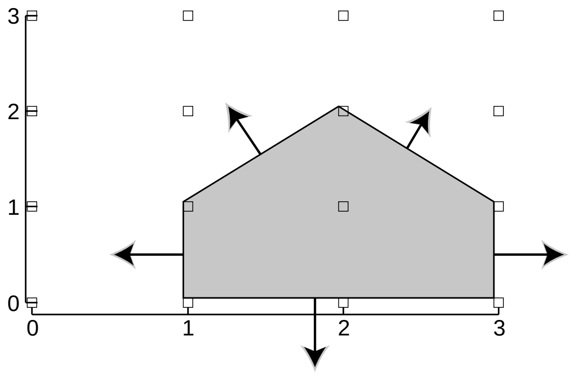

Figure 2: A summation of discriminating lines that will separate the two classes. Weight vectors for each discriminating line are shown.

The weight vectors and biases that form the shape in Figure 2 are shown in Table 1. Notice that the discriminating line follows the formula $v = w_1 x_1 + w_2 x_2 + bias$. This can be used to find the bias when $x_2 = 0$.

Weight Vector	Bias
<-1, 0>	1
<-1, 1>	0
<1, 1>	-4
<1, 0>	-3
<0, -1>	0

Table 1: Weights and biases needed to form the shape in Figure 2.

These discriminating lines can be summed together to form a classifier that produces zero when the input is inside of the shape and one when it's outside. Thus, the first output node is simply the sum of these discriminating lines, that is, all of its weights are 1 and it has no bias.

The second output is the opposite of the first. With a threshold activation function, the opposite output can be obtained by inverting the induced local field *v* and adding a bias so that zero inputs become positive, without causing activated hidden nodes to become positive. Thus, all of the weights for the second output are -1 and the bias is any number $0 < bias \leq 1$.

The following page shows the weights and biases for each node. Figure 3 shows the value of each output for all possible input combinations.

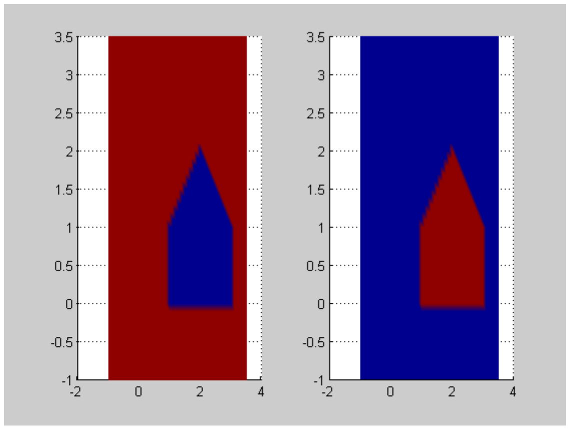

Figure 3: Output one (left) and output two (right) for all inputs.

Hidden Node One	
Weight	Value
w_{11}	-1
w_{12}	0
bias	1

Hidden Node Two	
Weight	Value
w_{21}	-1
w_{22}	0
bias	1

Hidden Node Three	
Weight	Value
w_{31}	1
w_{32}	1
bias	-4

Hidden Node Four	
Weight	Value
w_{41}	1
w_{42}	0
bias	-3

Hidden Node Five	
Weight	Value
w_{51}	0
w_{52}	-1
bias	0

Output Node One	
Weight	Value
w_{11}	1
w_{12}	1
w_{13}	1
w_{14}	1
w_{15}	1
bias	0

Output Node Two	
Weight	Value
w_{11}	-1
w_{12}	-1
w_{13}	-1
w_{14}	-1
w_{15}	-1
bias	{0,1]

Problem 2A

Figure 4 shows how the length of the error vector varies as the number of iterations changes for a given value of the learning rate. This is interesting as it shows that, for a single test case, there are several different domains. The computation initially converges fairly quickly, and then does nothing more for many iterations, at which point it again begins to converge quickly. Eventually, of course, a limiting point is reached.

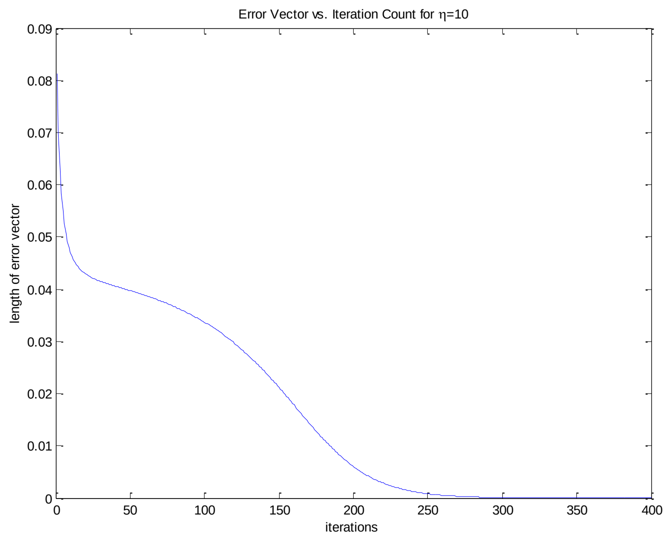

Figure 4: The dependence of the error vector on the number of iterations for a given learning rate.

Figure 4 suggests the one may be able to perform several computations using a low number of iterations in order to obtain the general shape of the curve and then estimate the number of iterations needed to obtain a desired accuracy. Of course, this single test case is not sufficient to draw this general conclusion.

Figure 5 shows how the length of the error vector changes as both the iteration count and the learning rate are varied. Isocontours are also shown to aid visualization. This plot suggests that one could increase the learning rate arbitrarily to speed computation, which is of course not true. This plot does, however, show that quick convergence can be obtained by using a large learning rate.

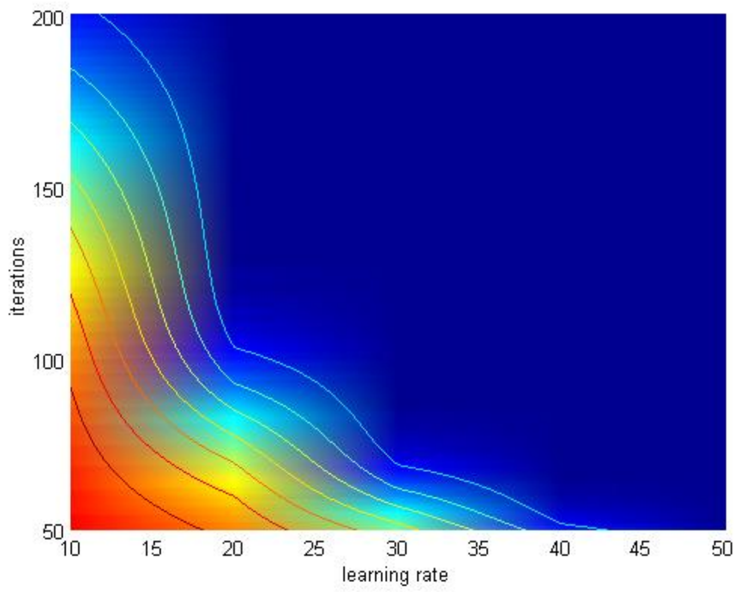

Figure 5: The error magnitude's dependence upon both the learning rate and the number of iterations.

Finally, Figure 6 shows that increasing the learning rate too much will keep the computation from converging. Thus, an optimal combination is probably somewhere around $\eta = 350$, iterations=50. This combination gives accuracy within MATLAB's numerical limits, virtually instantaneously.

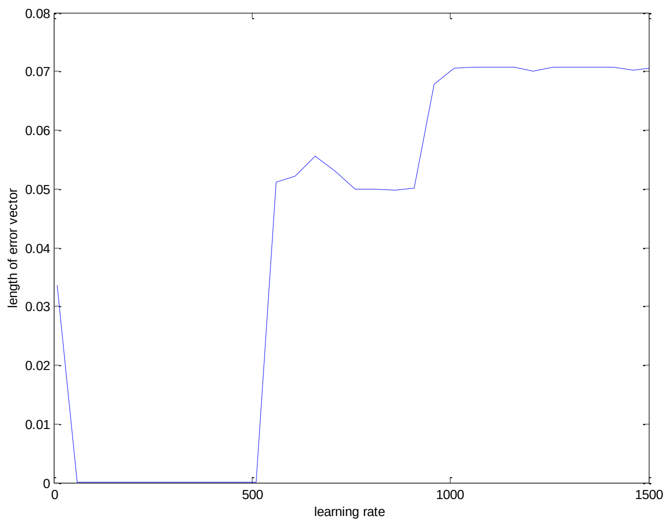

Figure 6: If the learning rate is too large, convergence cannot be achieved.

The code that produced these simulations can be found on my web page at http://www.geekspiff.com/academics/eee511/hw2.

Problem 2B

Figure 7 shows the value of output one for all possible input values in the problem space. Output one is one when the binary inputs are identical and zero when they are not. Since the inputs can be real-valued instead of integer-valued, the output gives intermediate results for non-integer inputs. Output two is the opposite of output one. The code that produced this plot is available on the web.

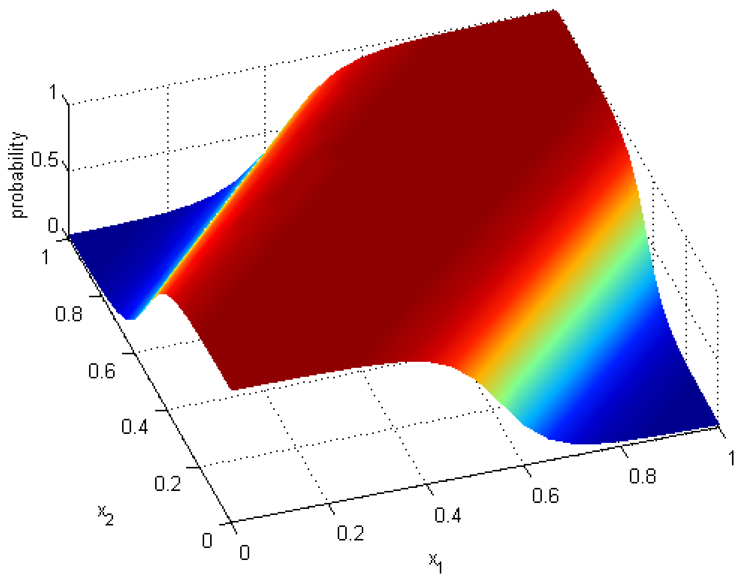

Figure 7: The probability of output 1 correctly classifying a set of two inputs x_1 and x_2 using the XOR relationship.

I used 500 iterations and $\eta = 10$ to produce Figure 7. I found that this computation was more sensitive to the learning rate. Using a learning rate less than 2 or greater than 50 did not result in convergence within a reasonable number of iterations (up to 5000).

Problem 2C

Figure 8 and Figure 9 show the response of an overly flexible network to all possible inputs after training. The network shown in Figure 8 was trained for 3783 epochs (until the mean error magnitude was 0.1) with a learning rate of 0.1. There were 200 nodes (plus a bias) in a single hidden layer, 2 inputs (plus a bias) and two outputs. All nodes used a logistic sigmoid (with unity weight in the exponential) as the activation function. The network shown in Figure 9 had the same geometry, but used a different random seed. The final results are quite similar, but this network required a substantially longer computation time (4661 epochs).

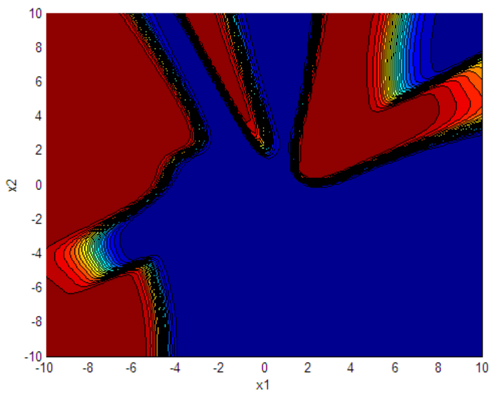

Figure 8: Contour plot of the response of the first output node to all possible inputs in an overly flexible network.

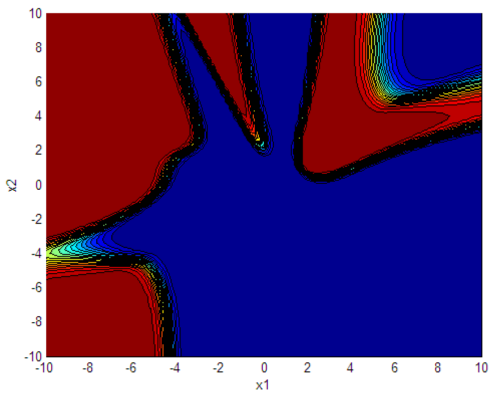

Figure 9: The same system trained using a different random seed. This computation took 4661 epochs.

Figure 10 shows how the error magnitude changed over successive epochs. It appears that the network could have been trained longer, since the error magnitude had not flattened out.

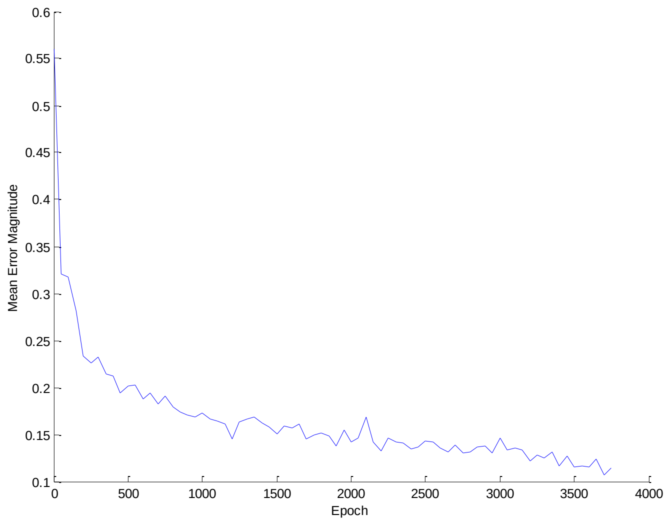

Figure 10: Evolution of the error magnitude over successive epochs. It appears that the network would benefit from additional training, because the error has not leveled off.

Problem 2d

Adding a weight-decay regularizer has the effect of reducing the magnitude of the weights that do not contribute to the classification. However, as implemented, the regularizer attempts to reduce *all* weights, making it more difficult to train the network.

Table 2 shows how the error magnitude and the number of large synaptic weights in the network vary as the decay constant is changed. A “large weight” is defined as a synaptic weight with a magnitude greater than one. It is clear that the regularizer is good for reducing the number of weights needed in the network, but also makes it more difficult to train the network.

Decay Constant	Final Error Magnitude	Large Weight Vectors
0	0.11	177
1×10^{9}	0.11	177
1×10^{7}	0.11	169
1×10^{5}	0.17	65
1×10^{3}	0.45	0

Table 2: As the decay constant grows, the weight vectors decrease, but it becomes progressively more difficult to train the network.