Edward Venator

EECS 484 Assignment 4a: Feedforward Neural Networks with Error Back Propagation

Basic Methodology

I began by defining the derivative equations for the weight matrices W_{ji} and W_{kj}.my derivative computation function is implemented as a nested for loop. It would run faster if I had implemented it using matrix operations, but since this code is not meant to be used in a production environment, I decided that the for-loop implementation was adequate.

My equations for the derivatives of the weight matrices are as follows:

$$dW_{kj} = \sum_{p=p1}^{P} e_k * (p) * \dot{\varphi}_k(p) * \sigma_k(p)$$

$$dW_{ji} = \sum_{p=p1}^{P} \sum_{j=1}^{J} e_k * (p) * \dot{\varphi}_k(p) * \dot{\varphi}_j(p) * W_{kj} * \sigma_i(p)$$

Since both layers have a logistic sigmoid activation function, both have the same derivative:

$$\dot{\varphi} = \varphi * (1 - \varphi)$$

Once this function was written, I made some minor tweaks to the code and collected data. My findings are described below.

I then moved on to learning the MATLAB neural network toolbox. First, I updated the code to use the new feedforwardnet() constructor. I noticed that by default, Matlab reserves some of the training data for testing and validation. This causes problems with our training data because it is the minimal definition of our function. By setting net.divideFcn to 'dividetrain,' I forced Matlab to use all of the data for training. After making this change, I found that Matlab's results were very similar to the results of the hand-written code, although the Matlab toolbox trained much faster. When the training function was set to 'traingd' (gradient descent), the results became even more similar, but the training took longer.

Findings and Conclusions

Verifying my Derivative Calculations

Table 1 shows that my derivative calculation function matches the estimated derivative perfectly. I tested this over several iterations, but in the interest of brevity, only one is shown.

dWkj	est_dWkj	dWji			est_dWji		
-0.0813	-0.0813	0	0	0	0	0	0
-0.0271	-0.0271	-0.0079	-0.0043	-0.0046	-0.0079	-0.0043	-0.0046
-0.0251	-0.0251	0.0118	0.0075	-0.0014	0.0118	0.0075	-0.0014
-0.0517	-0.0517	0.0121	-0.0000	0.0032	0.0121	-0.0000	0.0032
-0.0216	-0.0216	0.0190	0.0122	0.0113	0.0190	0.0122	0.0113
-0.0635	-0.0635	-0.0149	-0.0105	-0.0083	-0.0149	-0.0105	-0.0083
-0.0329	-0.0329	-0.0196	-0.0041	-0.0037	-0.0196	-0.0041	-0.0037
-0.0418	-0.0418	0.0102	0.0010	0.0008	0.0102	0.0010	0.0008
-0.0524	-0.0524	0.0010	0.0004	-0.0003	0.0010	0.0004	-0.0003
-0.0214	-0.0214	-0.0048	-0.0007	-0.0036	-0.0048	-0.0007	-0.0036

Table 1: Verification of Derivative Calculations

Varying Interneuron Numbers

I found that for such a simple problem, the number of interneurons was not very important. A greater number of interneurons generally yielded a smoother curve, but the benefits of more than about five interneurons were marginal at best.

Figure 1 shows the results of varying the number of interneurons. The learning rate ε was held constant at .5 and each ran for 10,000 iterations of training.

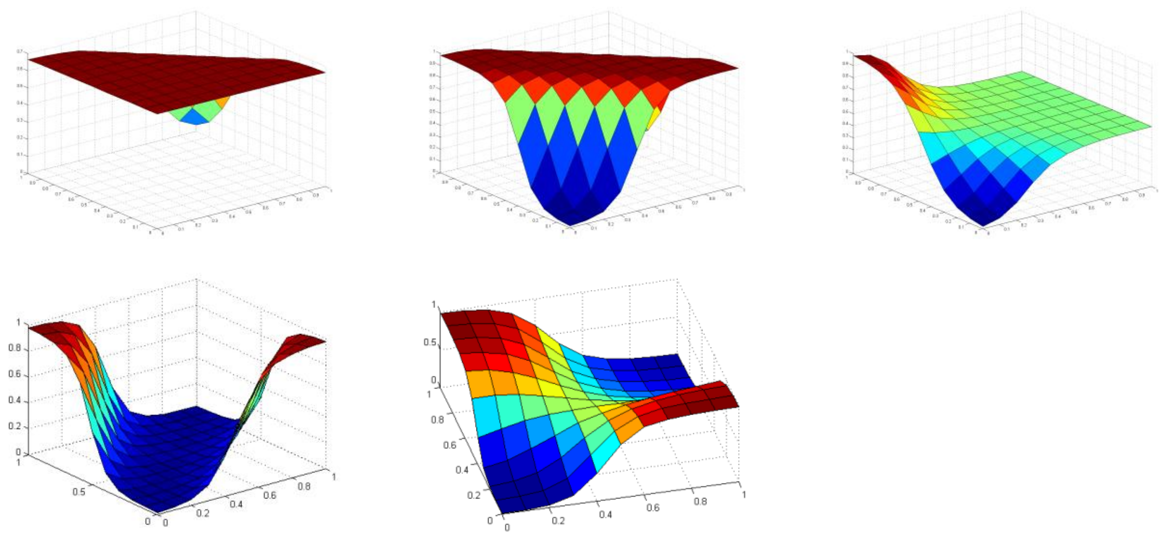


Figure 1: Results of the neural network after training for 10,000 iterations with ε=.5 and 1 interneuron (top left), 2 interneurons (top middle), 3 interneurons (top right), 4 interneurons (bottom left) and 9 interneurons (bottom middle). Note that the last is viewed from a different angle to show the surface shape.

As you can see, the neural network was unsuccessful with only one interneuron. With two interneurons, it was successful. However, for three interneurons, it failed. It seems that for four neurons or greater, there is almost no difference, although the surface develops smoother, more interesting geometry, as shown in the last image in figure 1. Full RMS error data for these trials are available in Appendix A.

Varying Learning Rate

From the results above, I chose to use four interneurons. I then began varying the learning rate of the network ε. Using the completion metric of $E_{rms}<.05$, I trained the network to completion with various learning rates. I found that all produced a surface that was similar to the point of being indistinguishable. As you can see in figure 2, the completion time seems to be inversely proportional to step size. For this problem, increasing step size above about 2 seems to have negligible benefit.

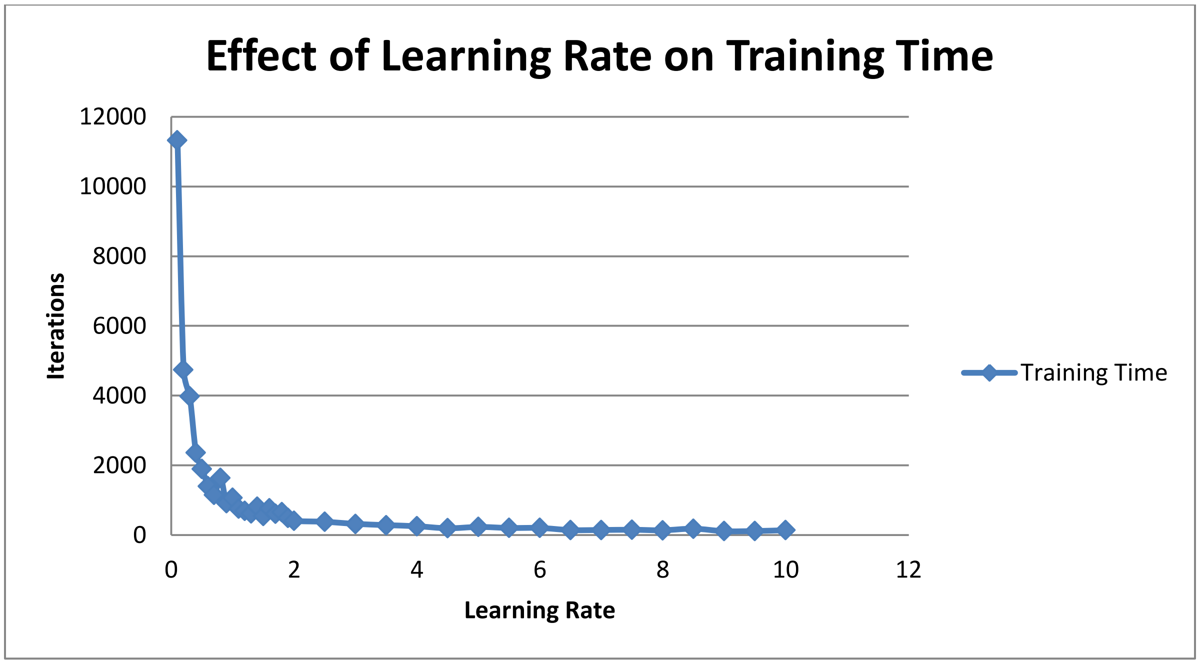


Figure 2: Effect of learning rate on training time with four interneurons. Training is considered complete when RMS Error is less than .05.

Tansig vs. Purelin vs. Logsig (Matlab Neural Net Toolbox)

Matlab's neural network toolbox defaults to using tansig for the interneuron activation function and purelin for the output neuron. Using the default training method (Levenberg-Marquardt), I experimented with changing the output activation function. I kept the number of interneurons constant at four. The table below shows my results. With the exception of tangsig->tansig, all of the activation functions yielded good results. However, the tansig->purelin combination that Matlab uses by default yielded the lowest mean squared error (MSE) and trained in only 4 iterations. I did not thoroughly experiment with changing the training method to gradient descent to match the handwritten code, but a few trial runs showed that it trains much more slowly (hundreds of iterations).

Training Method	Input Activation Function	Output Activation Function	Training Time (Epoch)	MSE
Trainlm	Tansig	Tansig	1	0.25
Trainlm	Tansig	Purelin	4	4.54E-21
Trainlm	Logsig	Logsig	7	1.93E-06
Trainlm	Logsig	Purelin	4	1.83E-15

Table 2: Comparison of activation functions with the Matlab Neural Network Toolbox

Number of Interneurons and Speed of Convergence (Matlab Neural Net Toolbox)

Similar to my results above, the number of interneurons has a strong influence on the training time and quality of the results for very small numbers of interneurons (<4), but for larger numbers of interneurons, there is very little if any difference. As you can see from my results below, any more than four interneurons is overkill—the network will converge just as quickly.

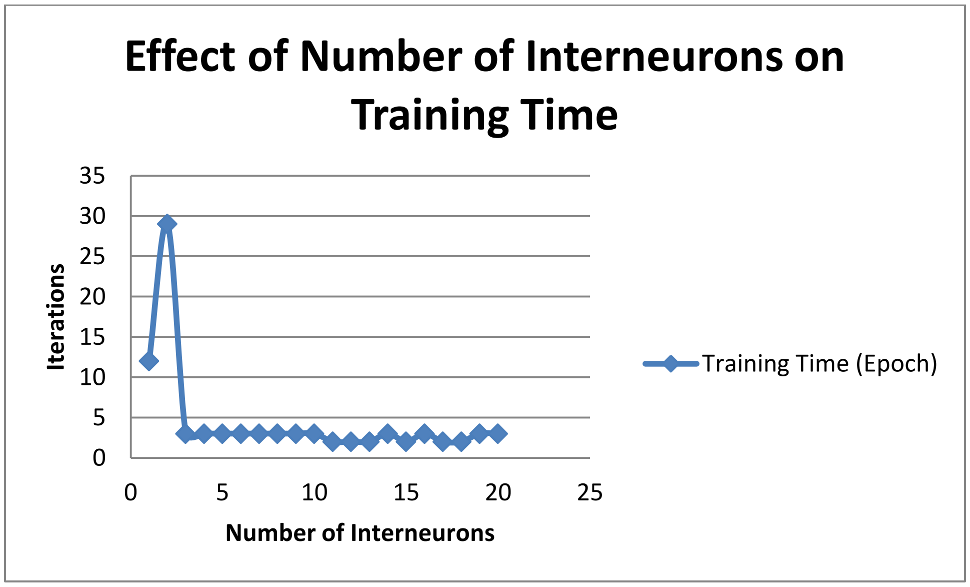
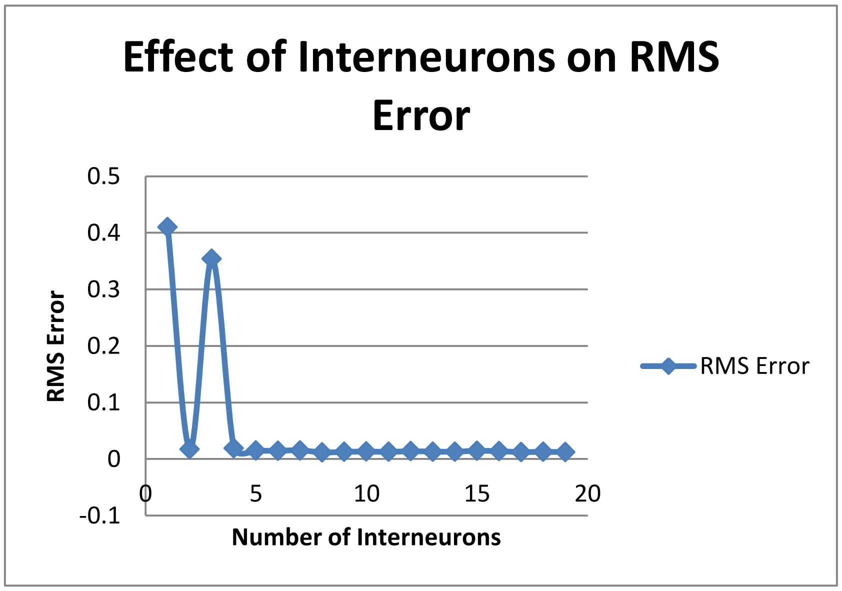


Figure 3: The effect of the number of interneurons on the training time

Appendix A: RMS Error Data for Varying Number of Interneurons

Number of Interneurons	RMS Error
1	0.4102
2	0.0173
3	0.3543
4	0.0192
5	0.015
6	0.0148
7	0.0154
8	0.0118
9	0.0127
10	0.0133
11	0.0127
12	0.0139
13	0.0132
14	0.013
15	0.0147
16	0.0139
17	0.0123
18	0.0127
19	0.0123


Effect of Interneurons on RMS Error
RMS Error
Number of Interneurons
RMS Error