Shaun Howard 10/1/15
EECS 484 PS5 smh150

PS5: Feedforward Network with Backpropagation

Compute $dW_{k,j}$ and $dW_{j,i}$:

To begin, I reviewed my notes from class. The most useful part of my notes had to do with the computation of the influence derivatives $dE/dW_{k,j}$ and $dE/dW_{j,i}$. From these I had to abstract the concept of training different layers of neurons and thus be able to successfully compute the derivatives of $dW_{k,j}$ and $dW_{j,i}$. These derivatives represent the derivatives of the weight vectors for the influences of both layer K on layer J and layer J on layer I in the ffwd network. Due to the nature of computation of these derivatives, they can be done in two ways. I chose to implement a nested loop in order to perform these calculations, but there are magic matrix formulas to do the same thing. The formulas for the derivative computation of $dW_{k,j}$ is as follows:

$$dW_{k,j} = \sum_{p=1}^{P} p * error_k * \varphi'_k(p) * \sigma_k(p)$$

That one was the easier of the two. Now for the more complex one, the derivative computation of $dW_{j,i}$:

$$dW_{j,i} = \sum_{p=1}^{P} \sum_{j=1}^{J} p * W_{k,j} * error_k * \varphi'_k(p) * \varphi'_j(p) * \sigma_i(p)$$

Given the perusal of more notes from class, I reflect that the layers of neurons have a logistic sigmoid activation function like so:

$$\sigma = \varphi(u)$$

Due to the fact that the layers in the ffwd network have such an activation function, this reveals that their derivatives are equivalent. Hence, we have the derivative as so:

$$\varphi'(u) = \varphi(u) * (1 - \varphi(u))$$

Solutions in Code:

After figuring out the equations above, I had to program them into the "compute_W_derivs.m" file provided. After getting the above formulas programmed into the Matlab code, I was able to break ground in other incomplete aspects of the provided code. I had to complete the "ps5_fdfwd_net.m" file provided. I added a maximum number of iterations to run the calculations of error metric and added or changed some other aspects of it. I had to experiment with different ways to use this file in terms of node stepping and rms-error convergence, hence, I created two new files with the same prefix as "ps5_fdfwd_net" but discerned appropriately by the stepping technique. The purpose of these is to evaluate the influence of both number of interneurons and choice of epsilon for gradient-descent computations. Both of these files are included with my submission.

Proof of Derivatives:

It is not unordinary to ask for proof that mathematical calculations are correct. Despite the fact that some calculations may provide a good estimate to a problem and thus cause the ffwd network to work sometimes, I will reveal that my calculations do in fact match the numerical approximations of $dW_{k,j}$ and $dW_{j,i}$. In the tables below, one can see that the output from both "numer_est_Wji.m" and "numer_est_Wkj.m" match the derivatives for $dW_{k,j}$ and $dW_{j,i}$ accordingly. Note that I have tested this several times but only included one set of calculations in order to remain brief.

The following tables show the side-by-side comparison of the estimated derivative calculations with my program's derivative calculations upon the first iteration of computation:

dWkj	est_dWkj
-0.15393	-0.15393
-0.11874	-0.11874
-0.10689	-0.10689
-0.10741	-0.10741
-0.07029	-0.07029
-0.07902	-0.07902
-0.08458	-0.08458
-0.09576	-0.09576
-0.06865	-0.06865
-0.08461	-0.08461

est_dWji			dWji		
0	0	0	0	0	0
0.014473	0.009448	0.008202	0.014473	0.009448	0.008202
-0.02525	-0.01301	-0.01303	-0.02525	-0.01301	-0.01303
0.029905	0.01338	0.020389	0.029905	0.01338	0.020389
0.000822	0.00039	0.000419	0.000822	0.00039	0.000419
0.025982	0.013299	0.013685	0.025982	0.013299	0.013685
-0.03727	-0.01898	-0.02084	-0.03727	-0.01898	-0.02084
-0.01372	-0.00934	-0.0056	-0.01372	-0.00934	-0.0056
-3.8E-05	-1.8E-05	-1.9E-05	-3.8E-05	-1.8E-05	-1.9E-05
0.002082	0.00113	0.000976	0.002082	0.00113	0.000976

The data in these tables was generated with the following function calls:

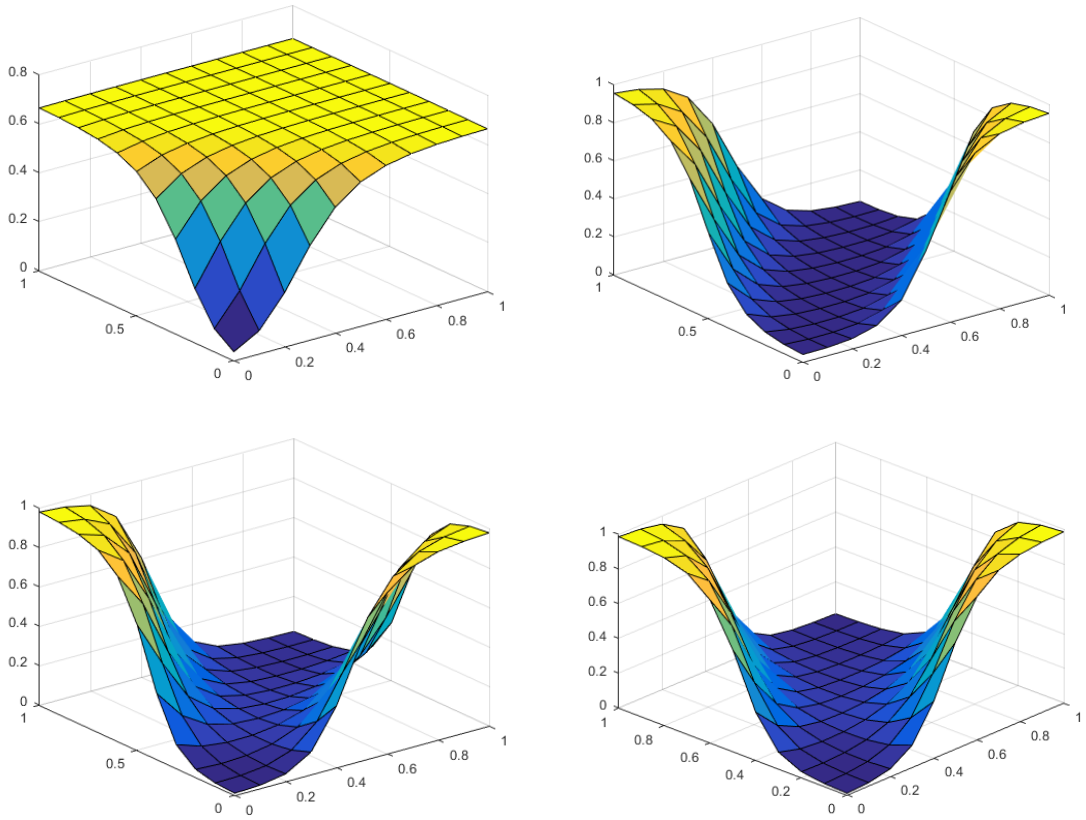
```
xlswrite('dWkj', dWkj)
xlswrite('est_dWkj', est_dWkj)
xlswrite('dWji', dWji)
xlswrite('est_dWji', est_dWji)
```

Shaun Howard
EECS 484 PS5

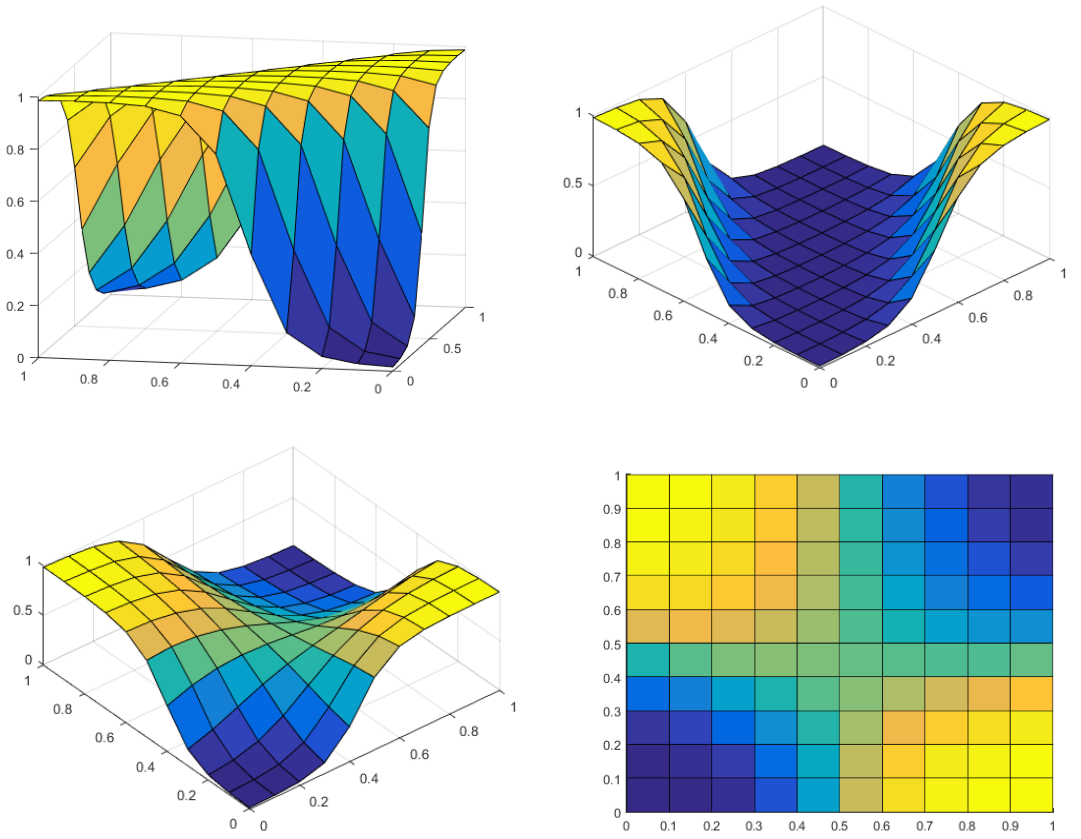
10/1/15
smh150

Influence of Number of Interneurons:

During my experiments, I noticed that as the number of interneurons grew, the results generally seemed to converge smoother to the expected solution for the XOR classification problem. Despite the smoother convergence as the number of neurons grew, the solutions computed were negligibly different after around 4 or 5 interneurons. I noticed that for some multiples of neurons, the graph would change drastically but still reveal the XOR classification solution on its four corners. I ran my program with a node stepping technique that would increase the number of interneurons after each set of net training was complete. Each training set would be computed across a number of iterations (in a while loop) based on the number of interneurons being tested plus the bias multiplied by 2,000. This made the program run very fast and successful at convergence. I ran this from 1 to 9 interneurons with a constant epsilon of 0.4 and graphed the converged results after each training period. This resulted in the production of multiple graphs, of which can be visualized below.



Figures 1-4:
(1) The top-left graph is for 1 interneuron. (2) The top-right graph is for 2 interneurons.
(3) The bottom-left graph is for 4 interneurons. (4) The bottom-right graph is for 6 interneurons.



Figures 5-8:
(5) The top-left graph is for 7 interneurons. (6) The top-right graph is for 8 interneurons.
(7) The bottom-left graph is for 9 interneurons. (8) The bottom-right gradient is for 9 interneurons.

Given the graphed results above, we can see that 1 interneuron could not solve the XOR classification problem, which makes sense because it is a multi-dimensional problem, i.e. linearly inseparable pattern. Following, we can see that 2 or more interneurons solved the classification problem. For some reason, the solution with 7 interneurons has a completely different looking graph than the rest but still solves the problem (maybe some prime number concept is at play behind the scenes). The most interesting graph, in my opinion, is that for 9 interneurons. It has a very smooth gradient pattern, which can be visualized in figure 8 above. Altogether, these graphs display that my program works correctly for different numbers of interneurons.

Influence of Learning Rate:

During my experiments, I noticed that as the learning rate grew, the number of iterations necessary for convergence diminished as well as the amount of time needed to find the proper solution. Growing the learning rate to some large value near 2 is plausible for the simple XOR classification problem, so in my testing, epsilon varied of values in the range [0, 2]. The number of interneurons during this experiment, however, remained constant. As epsilon grew from 0 to 2, the solution graph became somewhat less smooth but still correct within a certain amount of error. The maximum error threshold that I used to end the training process for each epsilon was

RMS error ≤ 0.05. Despite the inversely proportional relationship between the learning rate and the number of iterations taken to reach the maximum threshold of error, the solutions computed were negligibly different after around a learning rate of 2, so I made this my maximum value. I ran my program with an epsilon stepping technique that would increase the rate of learning based on the current step number and a step size of 0.1 for a maximum of 20 steps. Each training set would be computed across a number of iterations (in a while loop) until it converged to have rms_err of less than or equal to 0.05. This made the program run very fast and successful at convergence. I ran this with 5 interneurons and graphed the relationship between learning rate and the number of iterations taken to converge. The graph is below.

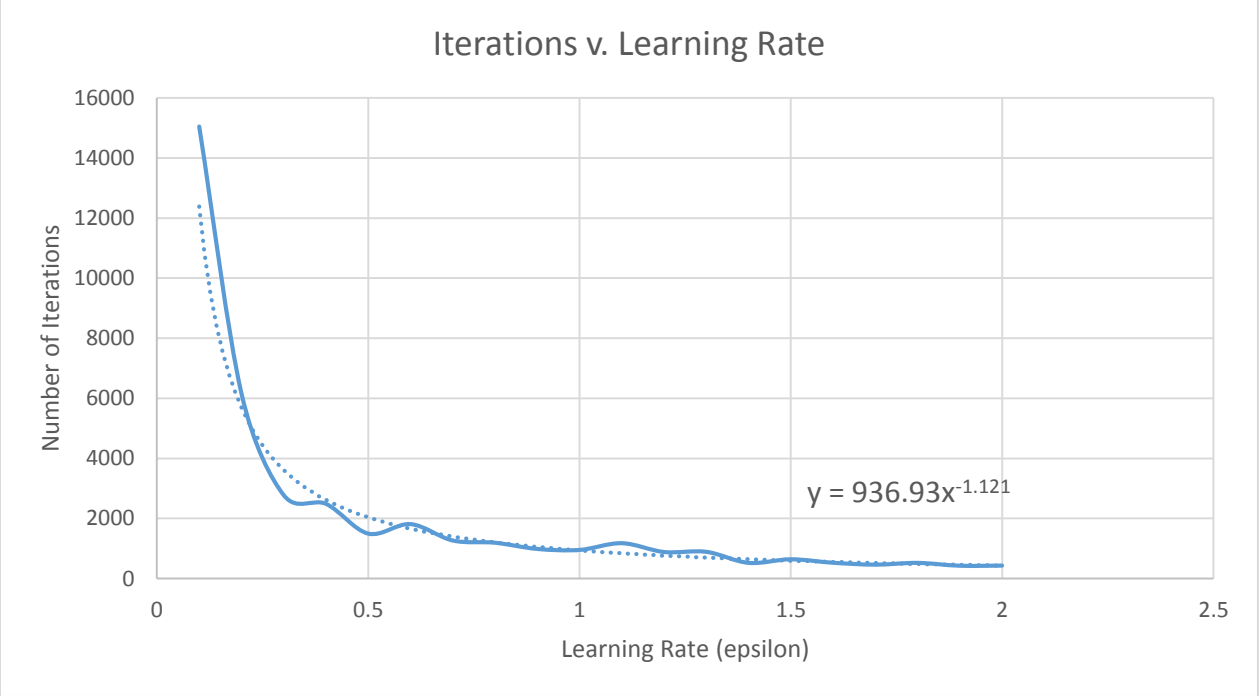


Figure 9: A graph of the relationship between learning rate and the number of iterations necessary for convergence.

One can see that the line of best fit is a power series with the equation: $y = 936.93x^{-1.121}$. This is a complex relationship between the learning rate and the number of iterations. It shows that the value of increasing the learning rate decays rather quickly with respect to the time it takes to compute the XOR classification solution.

Experiments with Matlab NeuralNet Toolbox:

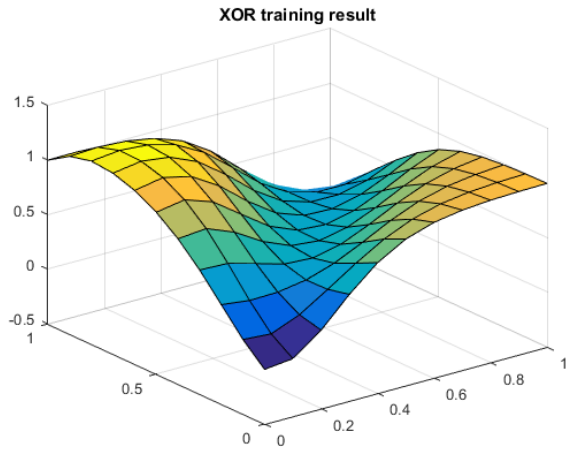
By reading the document on the Matlab NeuralNet Toolbox, I was able to familiarize myself with the concepts of how Matlab organizes its code to create, train, and examine neural nets. As discussed in class, this neural net toolbox uses an interneuron activation function of "tansig" by default. On output, it uses the "purelin" activation function. This produces quick and successful results most of the time. As I believe we also discussed or I found out through research, the default training method is the "Levenberg-Marquardt" method.

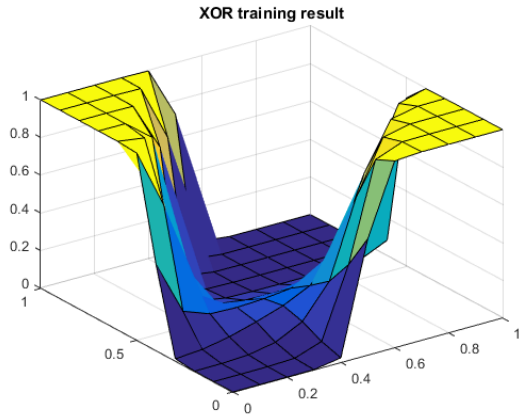
To begin experimenting with this toolbox, I had to modify and complete the provided file "nntbox_example.m." By default, Matlab takes some of the provided training data for testing and validation. We do not want this. In order to circumvent this, I set the following:

```
net.divideFcn='dividetrain';
```

Purelin vs. Tansig:

I experimented with "purelin" vs "tansig" activation functions for the output layer. I used a number of 4 interneurons with a "tansig" interneuron activation function and both "purelin" and "tansig" output activation functions. The results of my experiments in graphical form are as follow:



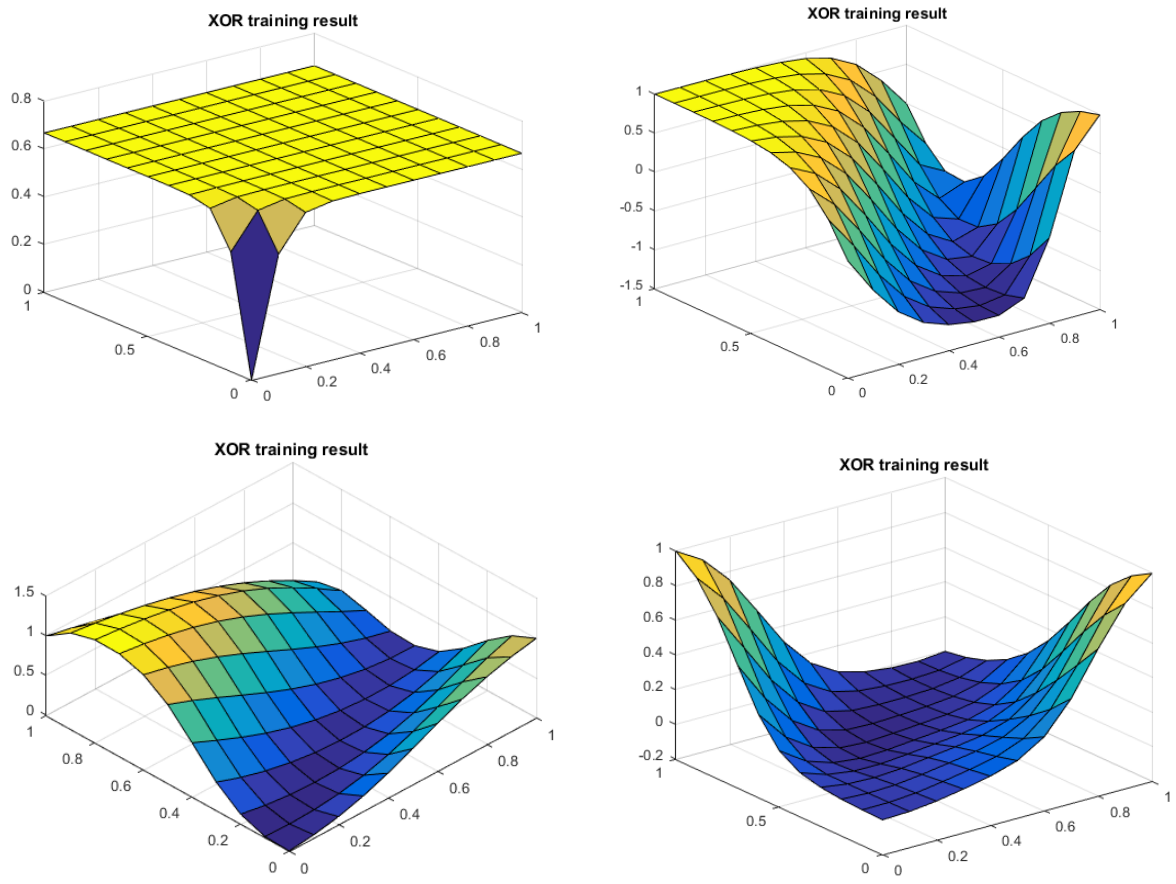


Figures 10-11: (10) On the left is the graph of the 'purelin' output activation function. (11) On the right is the graph of the 'tansig' output activation function.

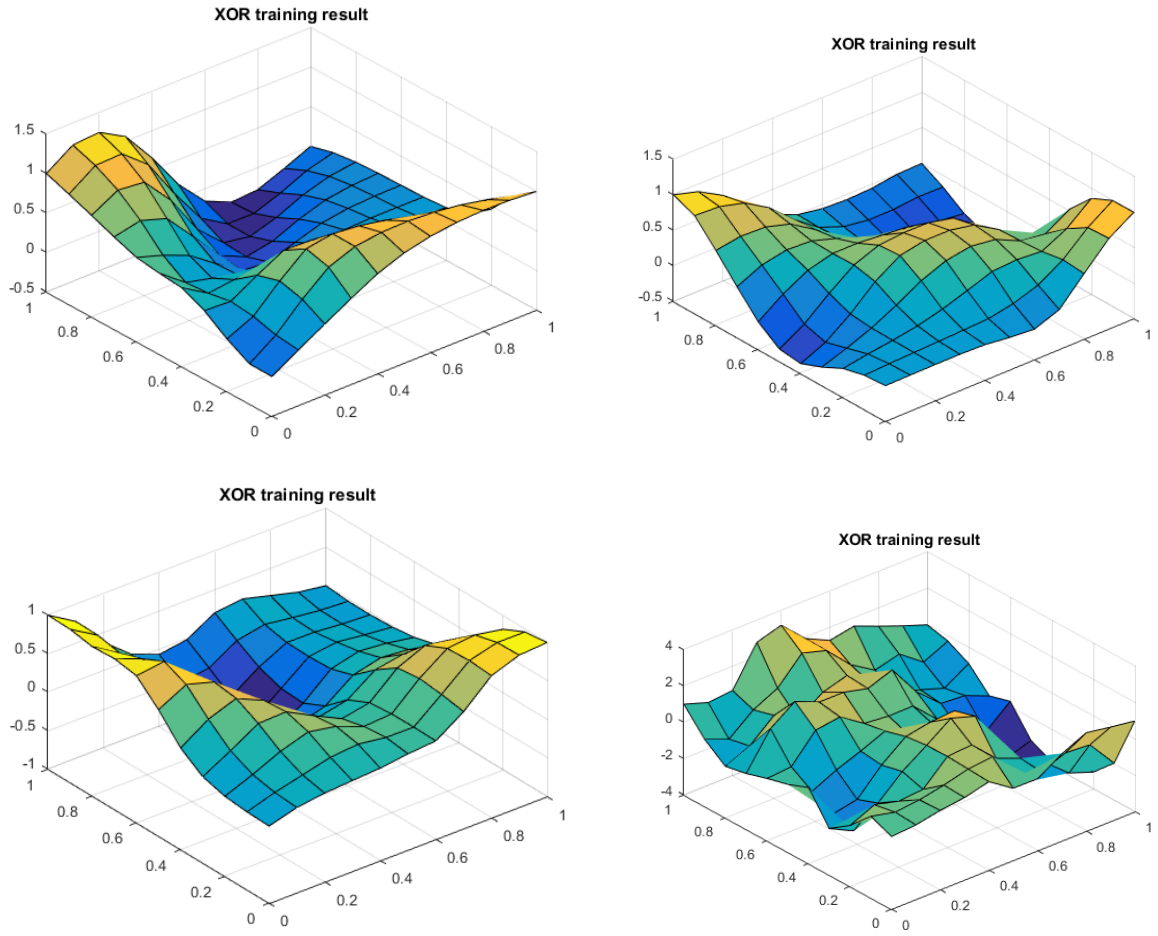
As one can see, the 'purelin' output activation function is much smoother than the 'tansig' output activation function. This reveals the using a linear output activation function would be better for continuous approximation to a solution rather than a discrete approximation. However, the 'tansig' output activation function would be better for discretely classifying data. One can see that figure 10 is less sure about the proper solution to the problem (even though it reaches it) given its continuous nature, whereas figure 11 is very certain about where the solution exists. As we can see, both methods still reach the XOR classification solution among their four corners. Something notable is that training takes longer on the "purelin" output activation function as compared to the "tansig" function. This is probably because the latter method is much more crude than the former method.

Varying Numbers of Interneurons:

For this set of experiments, I switched back to the 'purelin' output activation function. I varied the number of interneurons from 1 to 9, as in my original experiments, for serious graphical comparisons. I also ran the simulation with an interneuron count of 100 for fun. My results in graphical form are as follow:



Figures 12-15:
(12) The top-left graph is for 1 interneuron. (13) The top-right graph is for 2 interneurons.
(14) The bottom-left graph is for 3 interneurons. (15) The bottom-right graph is for 4 interneurons.



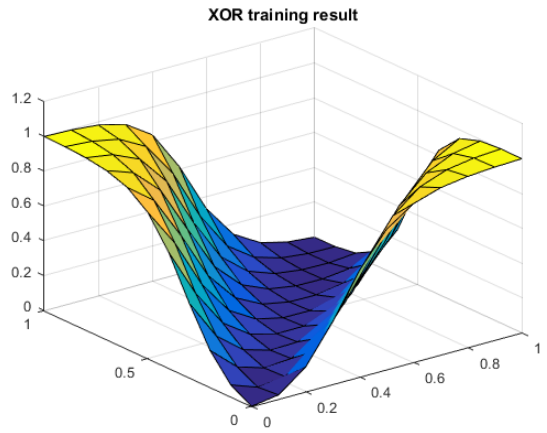
Figures 16-19:
(16) The top-left graph is for 7 interneuron. (17) The top-right graph is for 8 interneurons.
(18) The bottom-left graph is for 9 interneurons. (19) The bottom-right graph is for 100 interneurons.

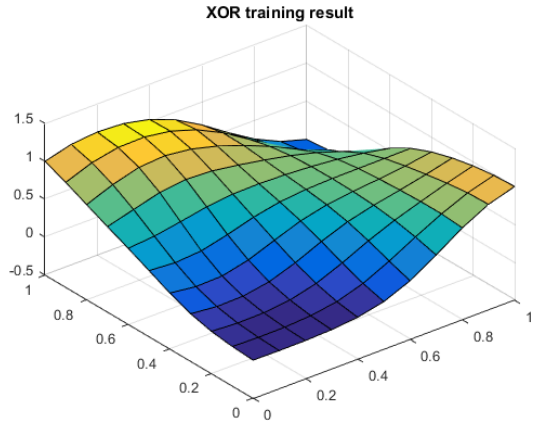
In observation of the graphs above, we can see that they all yield varying results. They results change for each time the neural net training code is run, even without changing the output neuron activation function. This shows that the behavior of the NeuralNet Toolbox is random, although it always finds the 4 corners of the XOR classification problem. We can observe that as the number of interneurons increases, so does the complexity of the geometry of the graph. The graph in figure 19 best exemplifies this observation. For the XOR classification problem, as noted in the original gradient descent method, the best number of interneurons is around 4. This value seems to be the best value for creating a reasonable classification of the data because it is concise and accurate and makes the training process faster than a greater number of neurons.

Training Methods:

I experimented with 3 different training methods, one being the default as above. The other two methods I used were “Resilient Backpropagation” aliased as “trainrp” and “Fletcher-Powell Conjugate Gradient,” aliased as “traincgf.” I held the number of interneurons constant at 4 since

this is the magical value for the other methods discussed earlier. My graphical results were as follow:





Figures 20-21:
(20) The left graph is for 'trainrp.' (21) The right graph is for 'traincgf.'

As we can observe, the results for both of these training methods are not much different than any of the other results. Each run of the code will generate another unique result that solves the classification problem with great performance. The one thing about the NeuralNet toolbox is that it is very fast. Even for the precision of the calculations in the default training method, it only takes a matter of seconds to classify the XOR data.

Conclusion:

In conclusion, we can see that although the hand-written gradient descent training algorithm is very precise and accurate, the Matlab NeuralNet Toolbox provides substantial alternatives to this training method that are much faster but less predictable. From an implementation point of view, it would be extremely hard to test the Matlab NNet toolbox implementation, whereas it would be quite easy to test the gradient descent implementation in comparison. Hence, the gradient descent method is accurate and quite fast if implemented correctly. My implementation happened to take up to 20 minutes to train but nothing far beyond that, and the results were accurate. The Matlab NeuralNet toolbox is a great set of tools to use when a fast and mostly accurate result is needed with lots of data as input. In reflection, many of the principles observed in this lab might not apply to the same methods for harder classification problems. However, given the nature of the XOR classification problem and its simplicity, the report holds.