**Compute** dW\_L **and** dW\_Lminus1:

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 and . 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\_L) and (dW\_Lminus1) . 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 is as follows:

delta\_L\_cum =

That one was the easier of the two. Now for the more complex one, the derivative computation of :

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

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:

**Proof of Derivatives:**

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 and 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 |
| -0.00075338 | -0.00075337 |
| 0.0010742 | 0.0010742 |
| -0.0014961 | -0.001496 |
| 0.00069297 | 0.00069297 |

|  |  |  |  |
| --- | --- | --- | --- |
| dWkj | | est\_dWkj | |
| -0.00057181 | -0.00042702 | -0.00057181 | -0.00042702 |
| 0.0076068 | 0.0047303 | 0.0076068 | 0.0047303 |
| -0.00013177 | -0.00042001 | -0.00013177 | -0.00042001 |
| -0.0054832 | -0.0011212 | -0.0054832 | -0.0011212 |

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

xlswrite('dWkj', dWkj)

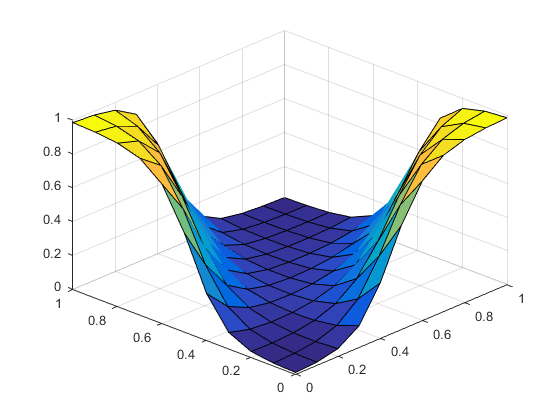
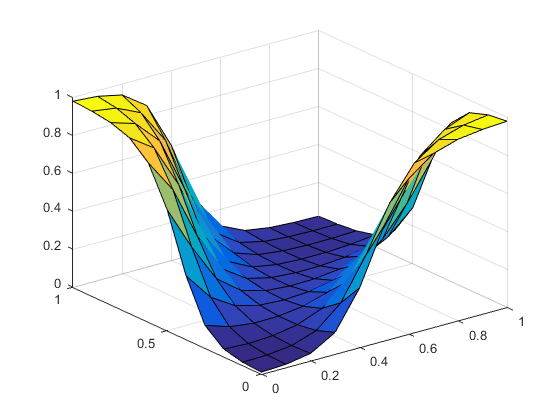
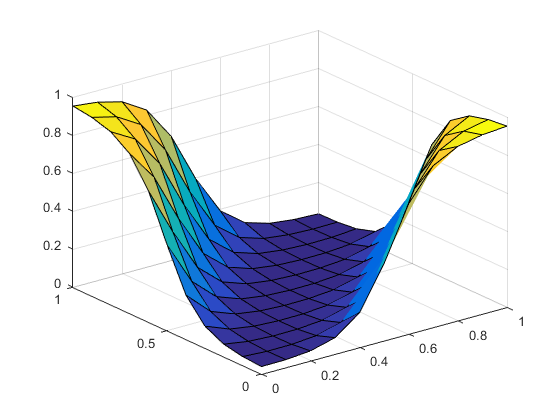
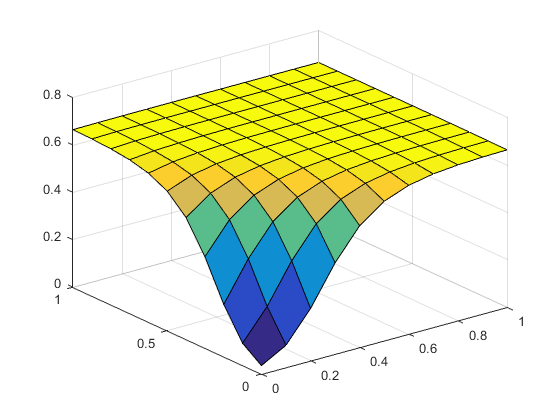
xlswrite('est\_dWkj', est\_dWkj)

xlswrite('dWji', dWji)

xlswrite('est\_dWji', est\_dWji)

**Influence of Number of Interneurons:**

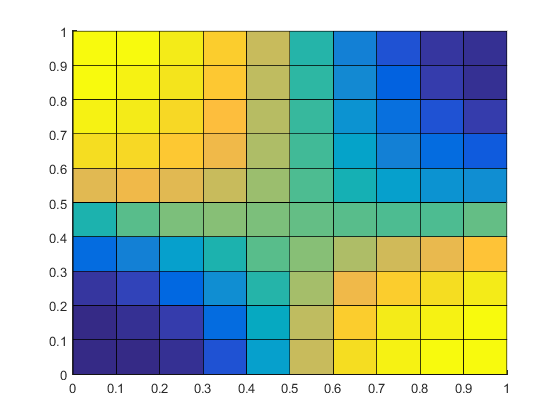
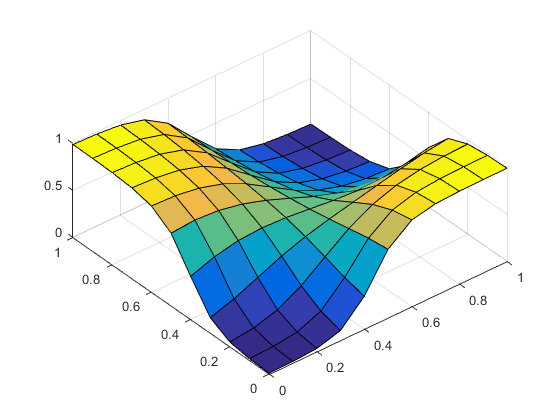
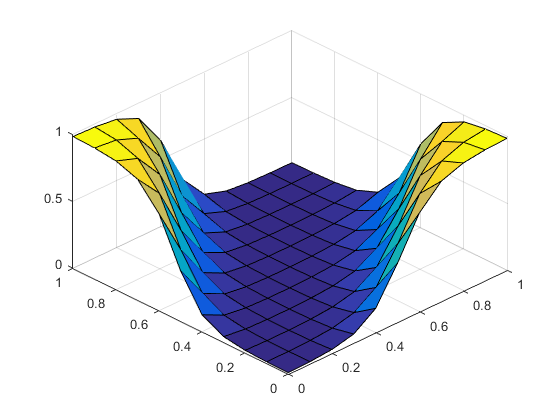
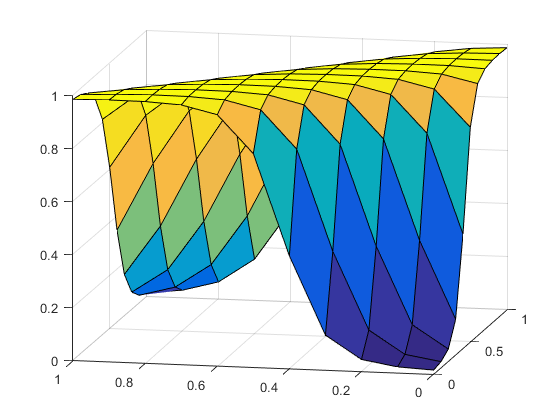
During my experiments, I noticed that as the number of interneurons grew, the results generally seemed to converge smoother to the expected solution for the XOR classification problem. Despite the smoother convergence as the number of neurons grew, the solutions computed were negligibly different after 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.