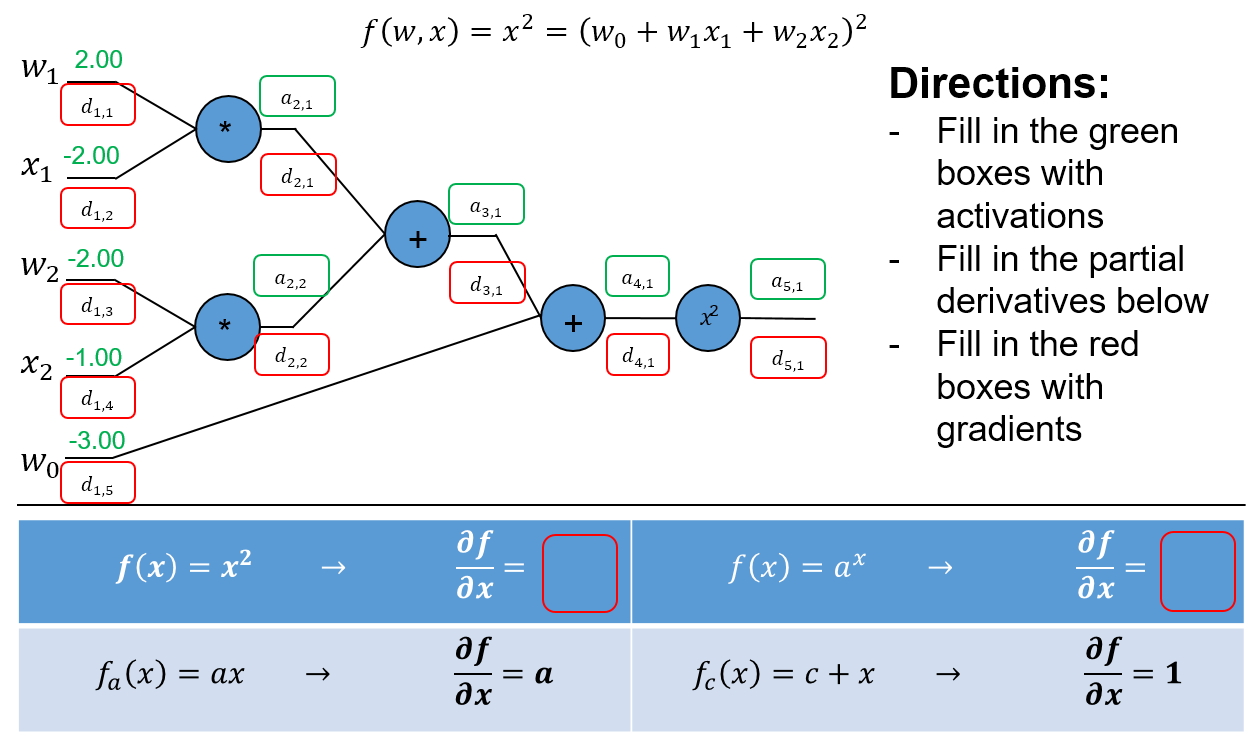
Assignment 1 (Due 4/26/2017)

Understanding Neural Networks

|  |  |
| --- | --- |
| **Name** | Michael Cho |
| **Discussion partner** | Jessica Chen |
| **Comments** | Add comments for the grader here. E.g. How to run the code, or anything to note when grading the code.  For each problem part, I created a new folder which also has a sub folder called train. These folders need to recreated if you are rerunning my code. For the activation, weight, and weight deltas, I sent the images captured at each 10 epochs into this subfolder and I reprinted and sent the final loss and accuracy plots to the main folder for the problem part.  Problem 2:  Part 1 (assignment1\_learningrate.py) –  Ln 15-17 changed variables names of weights so they can be referenced for each learning rate with the same initial weights  Ln 20 changed learning rate to a list of learning rates  Ln 24, 25-27 created a for loop to cycle through each learning rate  with the same starting weights  Ln 30,133 added a start and end timer to time each run  Part 2 (assignment1\_changek.py) –  Ln 20. List of different k  Ln.21. changed the sigmoid function to take in argument k  Ln. 24, 27-29 created for loop to cycle through different k with the same starting weights  Part 3 (assignment1\_init.py) –  Ln. 16-39 set up a list of 5 lists each with initialization for the 3 weights  Ln.45 added a loop to cycle through the different weight initializations  Problem 3:  Part 1 (assignment1\_actvgrad.py) –  Ln.30-32 applied clipping the sigmoid gradient  Part 2(assignment1\_tanh.py) –  Ln.19-21 replaces sigmoid with np.tanh  Ln. 24-26 changed sigmoid gradient with tanh derivative  Part 3 (assignment1\_cross\_entropy.py) –  Ln.16-17 softmax function  Ln.22 replaced losses with crossentrps list  Ln.28 applied softmax to only L3  Ln.33 removed sigmoid gradients for only L3  Ln.42 renames loss to ce and redid formula to cross-entropy cost  Ln.60 redid appending ce to crossentrps list  Part 4 (assignment1\_relu.py) –  Ln.17-18 changed initialization divide by sqrt of fan in  Ln.23 added relu function  Ln.30,31 applied relu function to hidden layers  Ln.38,39 applied relu derivative  Problem 5:  (assignment1\_final.py) –  Ln.8-14 centered and whitened data with PCA  Ln.25-27 used weight initialization of random normal divided by square root of fan in  Ln.30-31 functions for relu and softmax  Ln.36 added crossentrps list to contain individual cross entropies  Ln.40-42 applied relu to L1, tanh to L2, softmax to L3  Ln.45-47 got rid of activation gradient in dW3, applied clipping to dW2, and applied leaky relu to dW1  Ln.54 applied cross entropy cost function |
| **Feedback** | Note any feedback that you’d like to address in a future lecture.   1. We need more direction on the visualizations and what they mean here (show us an example in class). 2. Would be nice to have code suggest a way to test different learning rates, k in one run instead of writing our own code for that portion. 3. Prior to this, I didn’t know that softmax can be only used with an output layer and relu should be only used with hidden layers. Please add these directions to future classes. 4. I would like to know how to prevent overfitting. 5. I would like to know more about the geometry of neural networks in relation to these activation transformations. |

# 1. Neural Networks on paper (5 points)

Fill in the blanks below:



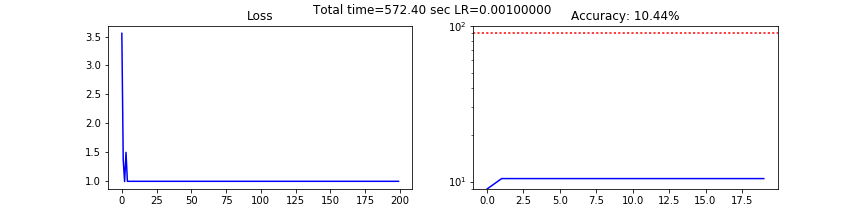
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 20 |  | -4 |  | -2 |  | -5 |  | 25 |
|  | -20 |  | 2 |  | -10 |  | -10 |  | 1 |
|  | 10 |  | -10 |  | |  | |  | |
|  | 20 |  | -10 |  | 2x |
|  | -10 |  | |  | lna |

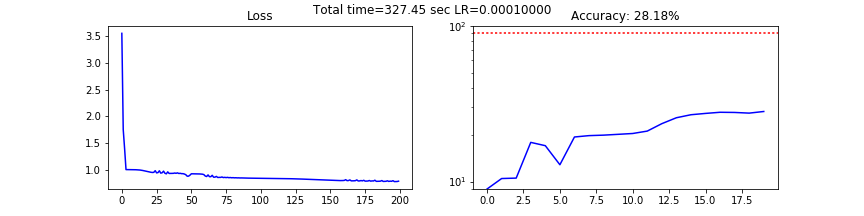
# 2. Neural Networks in code (12 points)

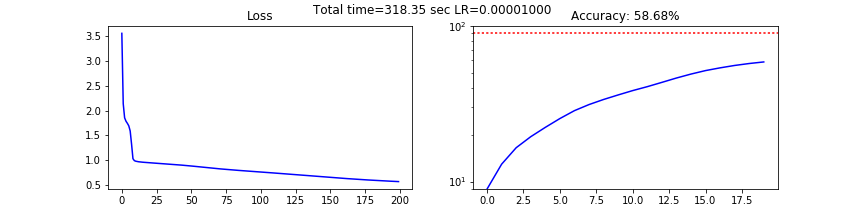
Using the provided example code from Lecture 3, explore the items below and demonstrate how to improve the example code by showing plots of the loss and accuracy curves. For each plot, show the curves for the first 200 iterations (You can stop training after 200 iterations for this part of the assignment). Consider the visualizations for activations, weights and weight updates.

1. **Learning rate:** Adjust the learning rate variable (lr) to try to achieve the “fastest” possible training rate. Show your loss and accuracy curves. What should you generally look for in the visualizations to ensure a “good” learning rate?

Please note that the loss is plotted against all 200 epochs, while accuracy is captured every 10th epoch. I tried three different learning rates of 1e-3, 1e-4, and 1e-5, with 1e-5 having the minimum loss and highest accuracy after 200 epochs. All of these learning rates started with the same initial weights. I calculated the total time for the code to run with each learning rate by using the timer function from the timeit library and this is printed on the final visualizations in secs. along with the corresponding learning rate. The largest learning rate here of 1e-3 actually took the most time (almost 2x as long as the other learning rates) and we can see that early on, the loss actually stalled around 1, while the other learning rates loss value continued to decrease. This is probably because gradient descent became stuck at a saddle point. The accuracy was the best for 1e-5 above with 58.68%. To judge a good learning rate, we want to have the minimum time, minimum loss, and maximum accuracy.

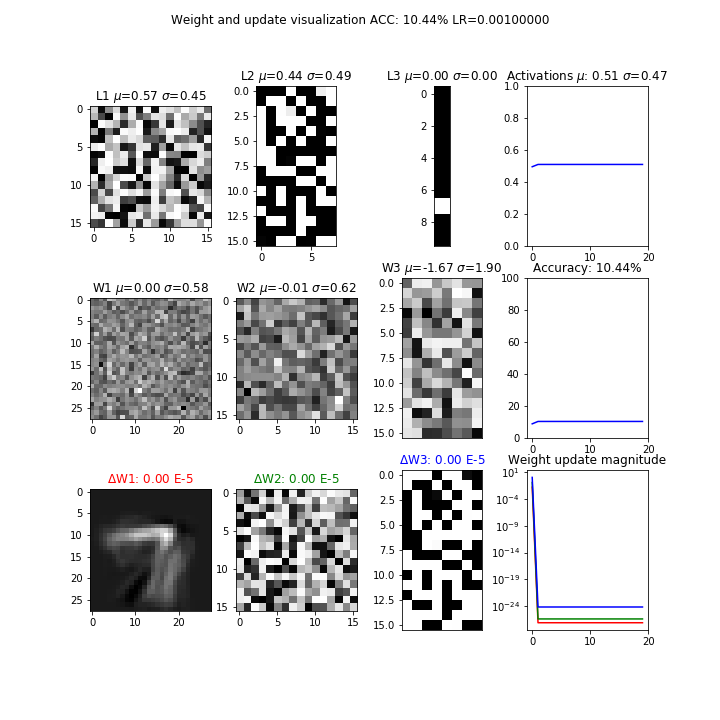
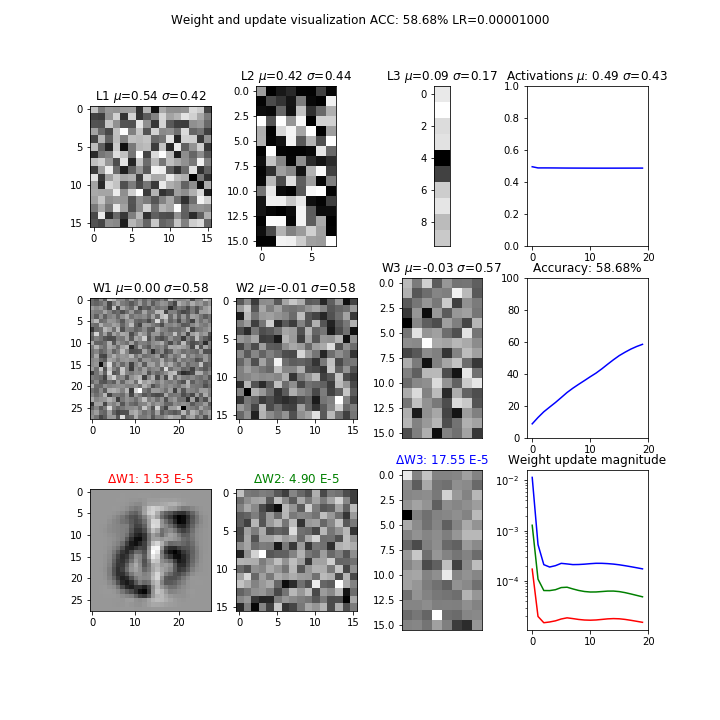






In the visualization shown below, I am showing the activation, weight, weight delta visualizations at the 190th epoch for the optimal learning rate of 1e-5 (best) and 1e-3 (worst). We can see clear differences in the two images. The L3 is shaded darker for 1e-3 while the L3 for

1e-5 is brighter across most digits, indicating that the neural network is considering more digits when making a final determination as to the class. We can also see that there are more white spots for 1e-3 for L2, W3, and delta W2, W3. If we are approaching the end of the epochs, we should expect the deltas to not change much and therefore the delta visualizations should be darker. Therefore, we can tell that 1e-3 is worse because it’s the neurons are activated more in its deltas.

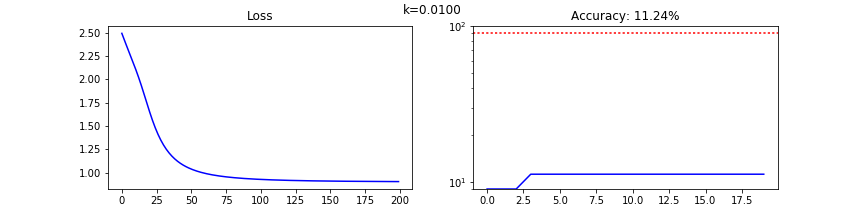


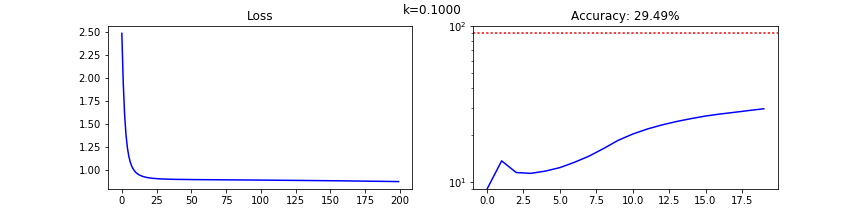
1. **Activation function:** Try changing the sigmoid function to “1.0/(1.0 + np.e\*\*-(k\*x))”, where k is another training parameter. Explain what k does. What is the effect of a small k on training versus a larger value for k? Is there an optimal k for a given learning rate? Use the default learning rate “1e-5” for your experiments. Justify your position in words, and show up to 5 plots.

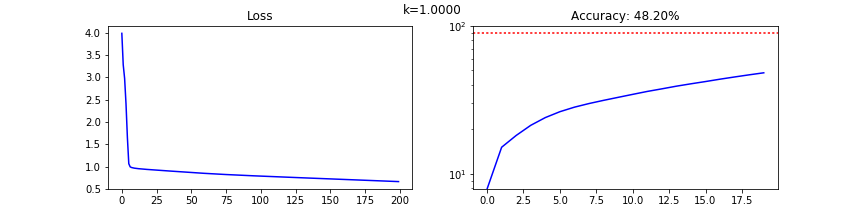
With a smaller k, the denominator of the sigmoid function will increase and thus the sigmoid will decrease in value for each layer, while with a larger k, the denominator of the sigmoid function will decrease and thus the sigmoid will increase in value for each layer.

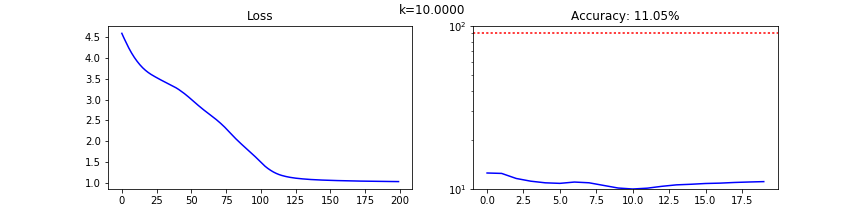
I tried 5 different k values with increasing and decreasing in degrees of 10: 1e-2,1e-1,1e0,1e1, and 1e2.

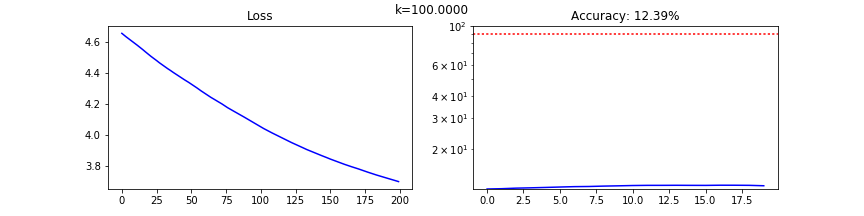
The first series of visualizations is once again the loss and accuracy plots for each k. Based on the loss and accuracy, it seems the best k value is 1 as its loss is near 0.5 and accuracy is 48.20%.



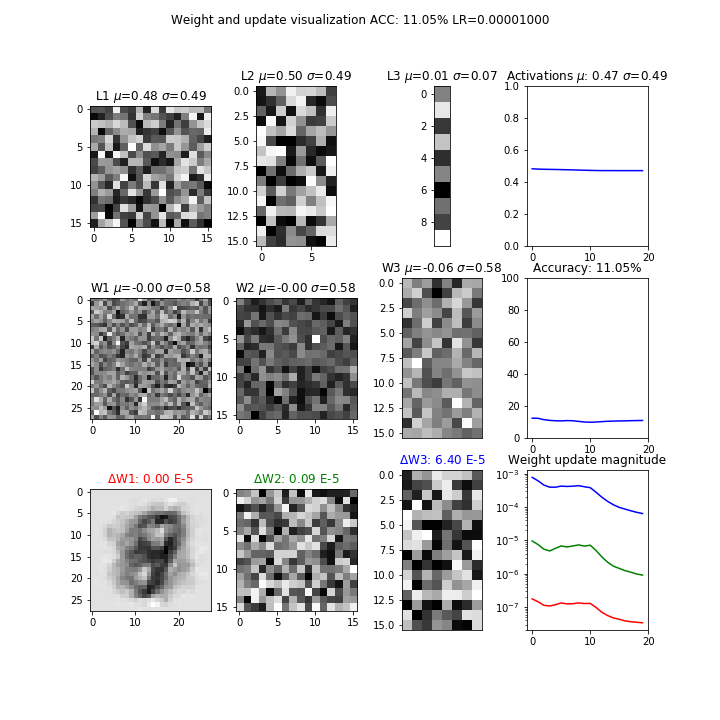
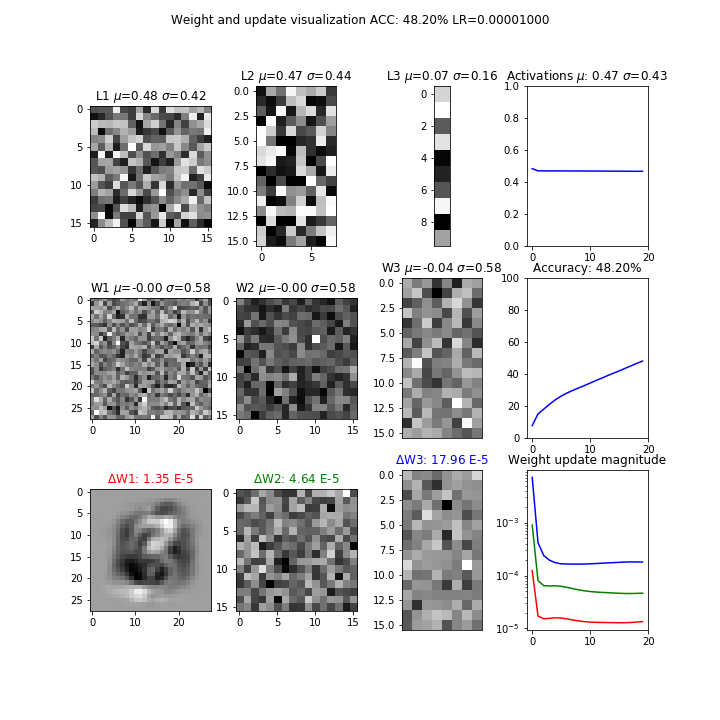








In the visualization shown below, I am showing the activation, weight, weight delta visualizations at the 190th epoch for a k of 1 (best and to the left) and a k of 10(worst and to the right). Once again we see the issue with the deltas as it seems the neurons are more activated for deltas W3 and delta W2 for the worse k. Also we see that there is no steep decline in update magnitude in k=10 as compared to k=1.



1. **Initialization:** In the sample code, the weights W1, W2, W3 are initialized uniformly from -1 to 1. Experiment with various kinds of initialization and report your findings. Justify why your proposed initialization is better than the default initialization. Show up to 5 plots. Hint: how do the visualizations differ for good and bad initializations?

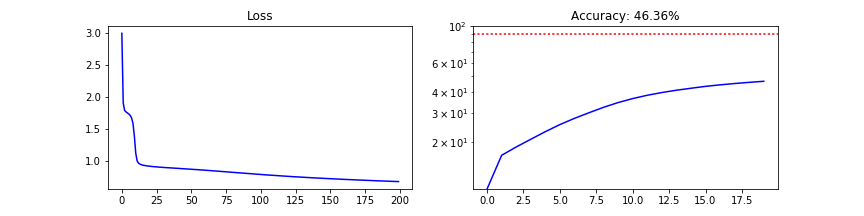
I ran five different kinds of initial weights: the original with range [-1,1], the original modified with range [-4,4], a random normal divided by , a random normal with 0 mean and variance of 2/fan-in, and a glorot uniform with a range of [- ] .

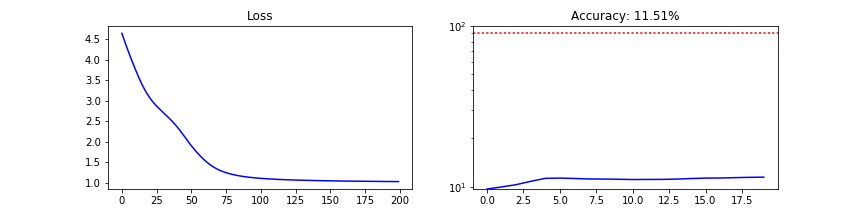
The visualizations are shown below in their respective order. As one can see, the glorot uniform initialization has the lowest loss (under 0.5) and highest accuracy (84%) accuracy after 200 epochs.

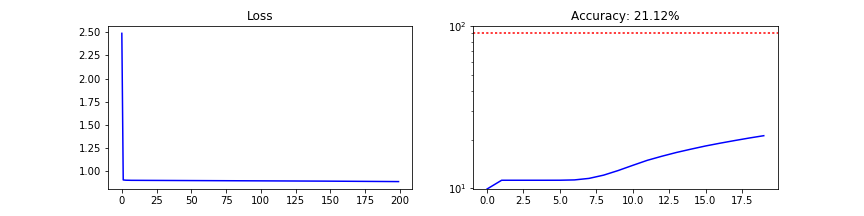
The 2nd initialization came from idea of testing what if we increased the range of the initializations as well as the observation that 95% of the gradient density is within [-4.4]. The 3rd initialization came from the idea that we should divided by square root of the fan in to get unit variance at each layer. The last two initializations came from a stack exchange post quoting the latest academic research around weight initializations. For example, the 4th initialization is said to work with well with the relu activation function and converge much faster.

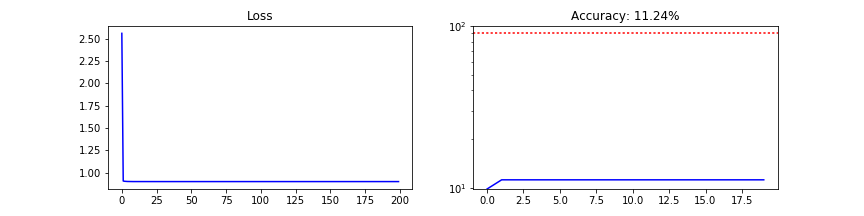
<https://stats.stackexchange.com/questions/47590/what-are-good-initial-weights-in-a-neural-network>

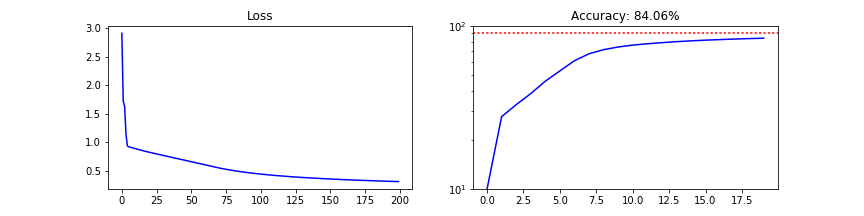
The visualizations are shown below in their respective order. As one can see, the glorot uniform initialization has the lowest loss (under 0.5) and highest accuracy (84%) accuracy after 200 epochs.



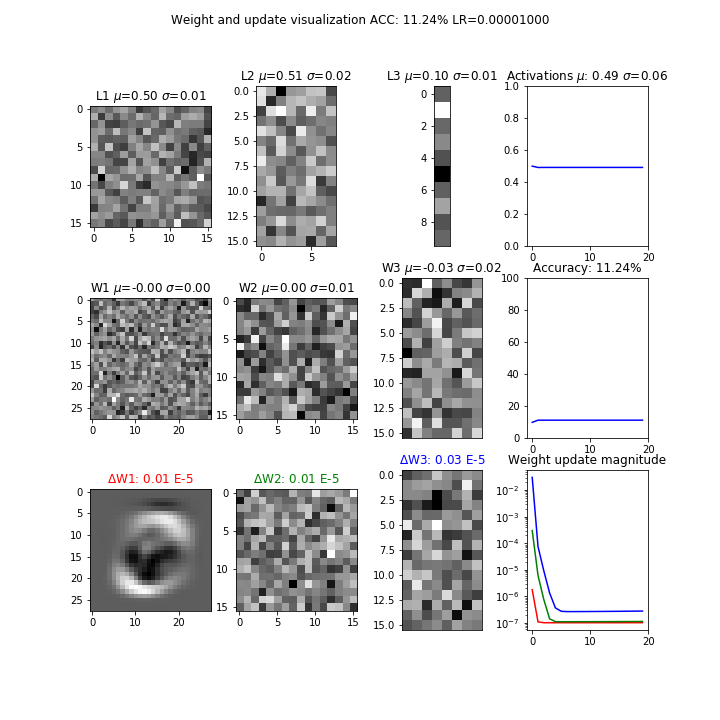
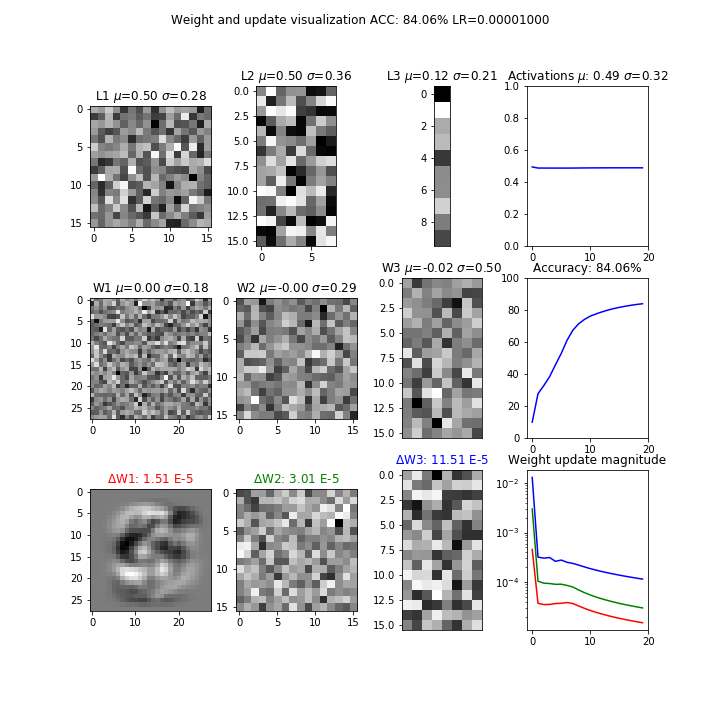








In the visualization shown below, I am showing the activation, weight, weight delta visualizations at the 190th epoch for the glorot uniform (best and to the left) and the 4th initialization (worst and to the right). Strangely enough this time, the delta W2 and delta W3 for the better initialization shows more highlighted regions. This is probably because the epochs were cut short to 200 and the accuracy could improve by a good margin with the glorot uniform. Also notice that the weight update magnitude for the glorot uniform hasn’t hit a plateau at the bottom and is still trending downwards.



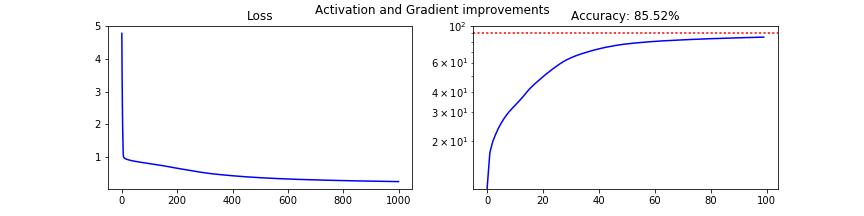
# 3. Optimization in code (16 points)

Using the example code from lecture 3, demonstrate your understanding of the principles in lecture 4 by doing the following (submit your final code for these in part 4 below, but show your changes here):

1. **Activations and Gradients:** Examine the activations and gradients visualized during training. Justify why the mean and standard deviation of the activation and gradient matrices are “optimal” or not. Propose some ways to “fix” the activations/gradients to improve training. Show some plots to illustrate how your proposed “fix” improves training.

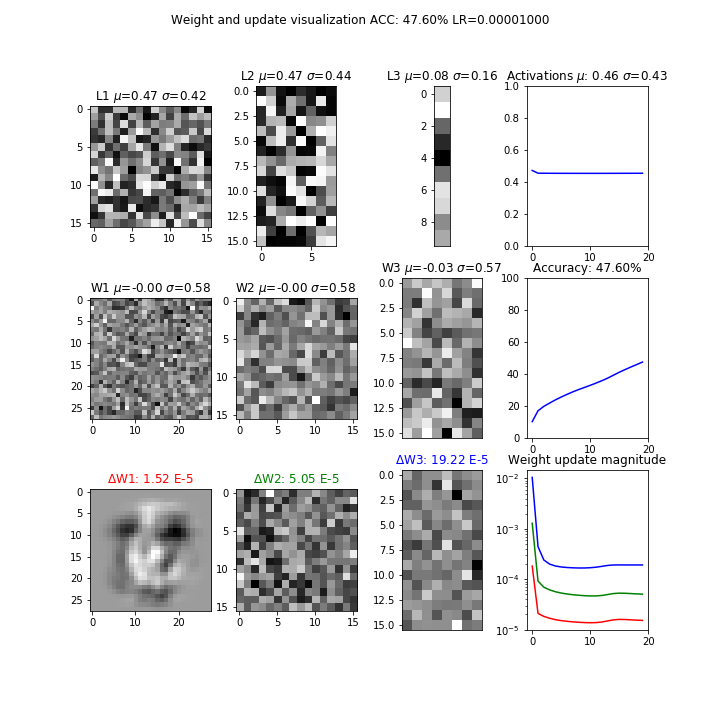
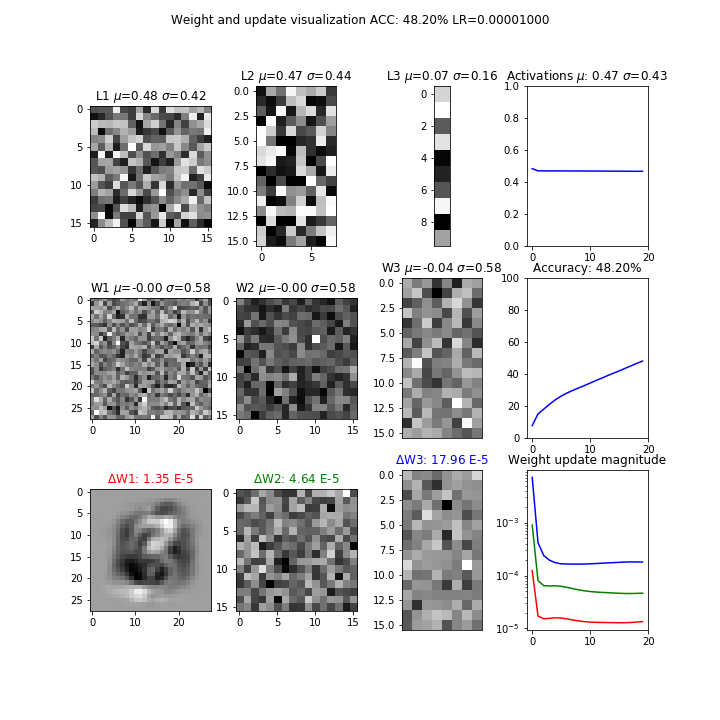
There are three improvements I would suggest. One is to clip the gradient to some small value such as 0.001 so that gradients won’t be saturated using np where as an if else statement for each matrix element. Next is to normalize the data by zero centering and dividing by the category standard deviation. Finally, we can divide sigmoid inputs by sqrt(# of inputs to each layer) to have unit variance.

I had problems with normalizing the data by dividing by the row standard deviation as it turned out nans when dividing by 0. Also, dividing sigmoid inputs by the sqrt(fan-in) didn’t provide any improvements. I was however successful with clipping the gradient to 0.01. I ended up with 85.52% accuracy after 1000 epochs.



I compared the original code without any changes and the clipping code at the 190th epoch shown below. L2, W2, and W3 is much brighter indicating that less neurons are dead when clipping.

Original code Clipping code



1. **Tanh:** Implement tanh(x) instead of the sigmoid. Explain why tanh(x) may be better, and show plots. Hint: what is the derivative of tanh(x)?

I first replaced sigmoid with np.tanh in the code.

The only other code change required was to account for derivative of tanh(x), which is

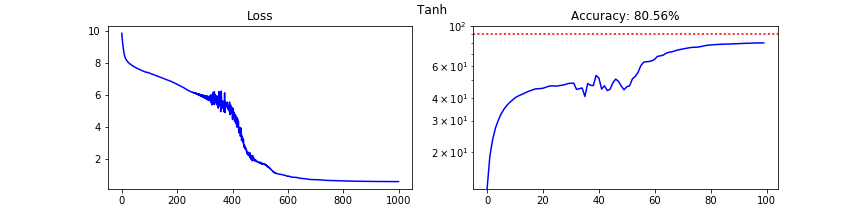
in the overall gradient which is shown below. The original sigmoidal gradient was of the form .

dW3 = (L3 - T) \* (1 - L3\*\*2)

dW2 = W3.T.dot(dW3)\*(1-L2\*\*2)

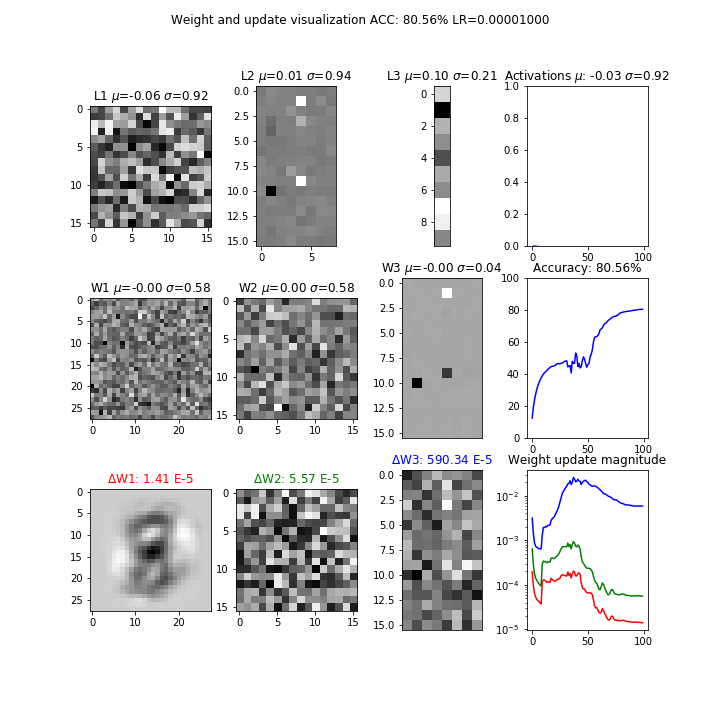
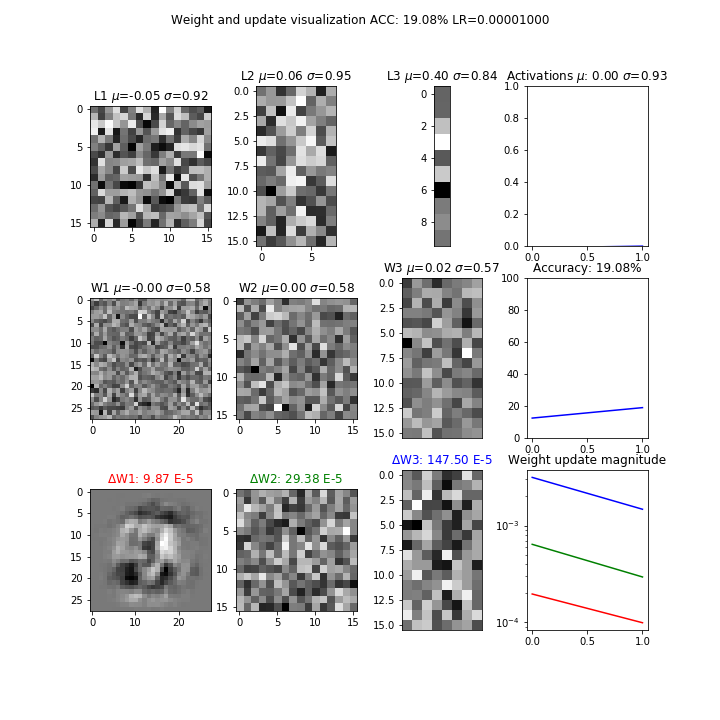
dW1 = W2.T.dot(dW2)\*(1-L1\*\*2)

The tanh function turned out an accuracy of 80.56% according to the graph below. The tanh transformation is better than the sigmoid because it centers the mean and it takes care of the zig zagging issue seen when updating the gradient direction.



There are some strange things with these visualizations. First, there seem to be spikes in the loss and accuracy. With 990th epoch visualizations L2, W3, and delta W3 are shaded very heavily in comparison to the 10th epoch visualization indicating that a majority of the activation regions are dead/near dead. We also see the spikes with the weight update magnitude in the 990th epoch visualization.

10th epoch 990th epoch



1. **Cross Entropy:** Implement cross entropy. Show plots of how “Cross-entropy” improves training.

First, I added a softmax function to be used with cross entropy.

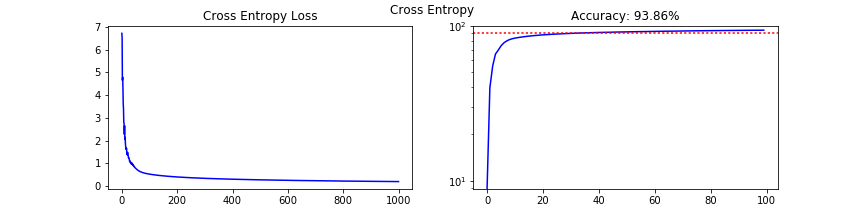
def softmax(z):

return np.exp(z) / np.sum(np.exp(z), axis = 0)

Next, I replaced the losses list with a list called crossentrps where I can append individual cross entropies called ce.

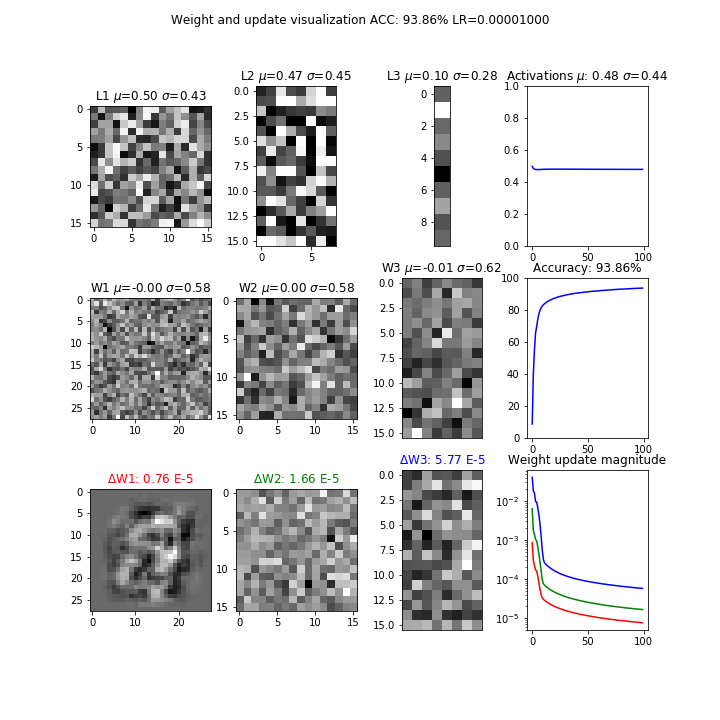
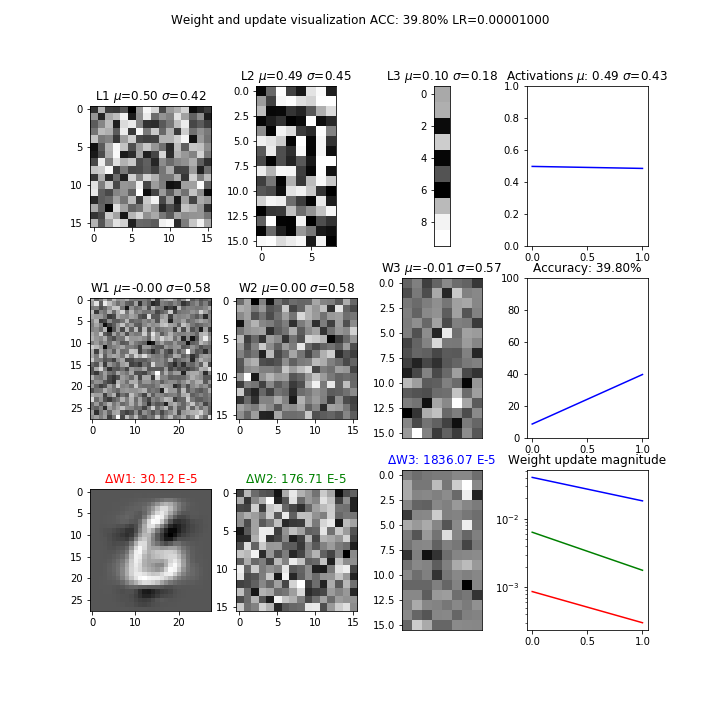
I only applied softmax to the output layer L3 and removed the sigmoid gradients for dW3, as cross entropy works best with soft max and softmax should only be used with the last output layer. The activation gradients for the hidden layers are retained.

I also changed the loss function as well as the display message to show Cross Entropy Loss instead of MSE Loss.



As seen from visualizations above and below, cross entropy seems to rapidly train the model within the first 100 epochs and then plateaus. The 10th epoch shown below already returned a 39.8% accuracy. Delta W2 and W3 show the most change from the 10th to the 990th epoch but in no discernable pattern.

10th epoch 990th epoch



1. **ReLU:** Implement rectified linear units and justify why they may be better. Show plots. Hint: what is the derivative of relu(x)? Did any of your neurons “die”? What do dead neurons look like in the visualizations? How can we “fix” dead neurons?

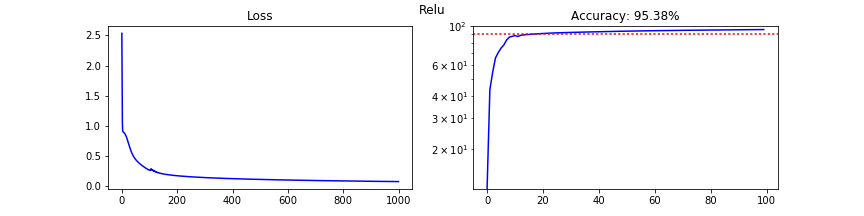
I applied relu to only the hidden layers (kept the sigmoid output) and accordingly switched the weight updates according to the relu derivative, shown below. Basically, np.where acts as an if else statement for each matrix element. If the input to the relu corresponding to the relu is less than or equal to 0, than the matrix element is 0 otherwise it is the MSE gradient. I also experimented with a number of initializations and found the best one was standard normal divided by from the square root of with fan-in being the number of inputs to each layer.

dW3 = (L3 - T) \* L3\*(1 - L3)

dW2 = np.where(W2.dot(L1)<=0,0,W3.T.dot(dW3))

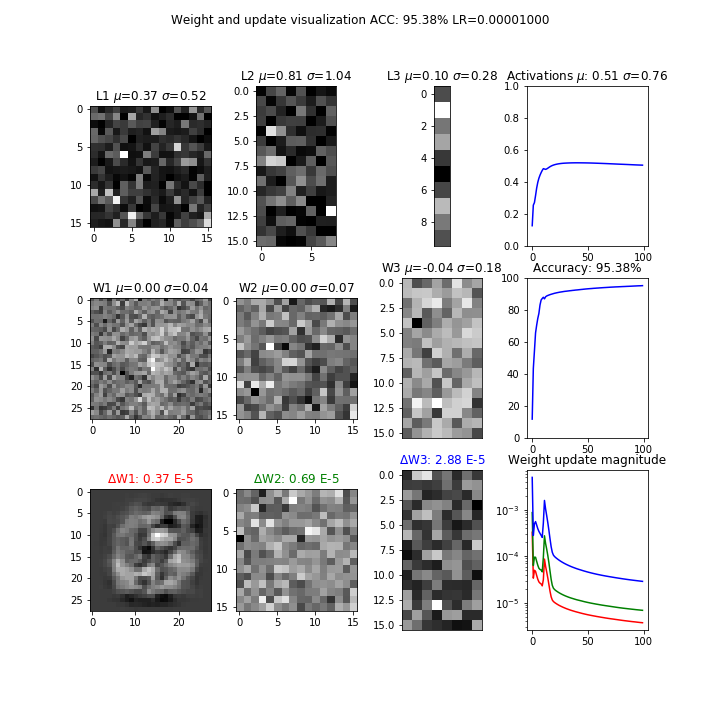
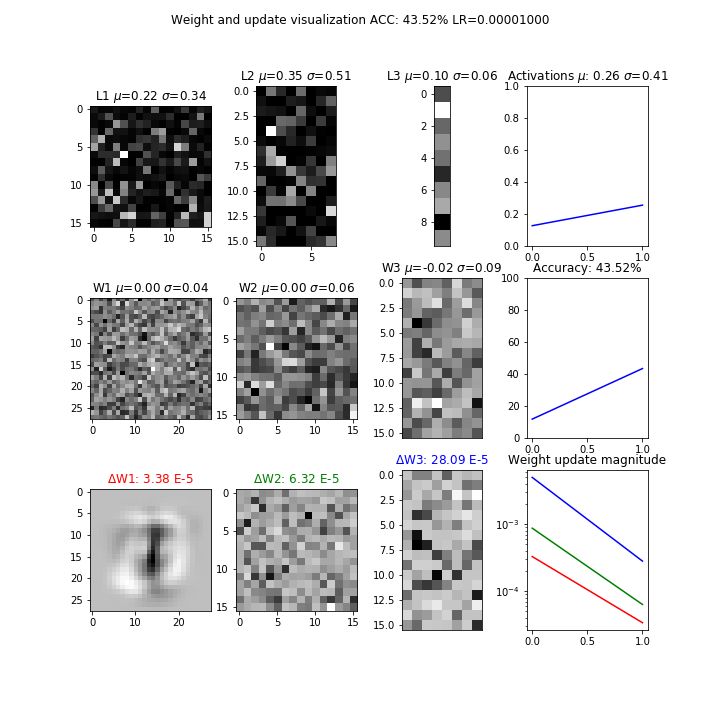
dW1 = np.where(W1.dot(X)<=0,0,W2.T.dot(dW2))

Relu has two main advantages over the sigmoid. Number one it addresses the sparsity of the matrix when the activation region is less than or equal to 0. Second, while Relu also suffers from vanishing gradients, which we can resolve with clipping, it is less prone to the problem as the gradient is constant as opposed to the sigmoid where the gradient gets smaller as it approaches the bounds of the activation region. This constant gradient therefore results in faster training.



There are some interesting things in the visualizations below. First, delta W3 is shaded much darker by the 990th epoch corresponding to the plateauing of the loss and accuracy curves. Second, a large number of activation regions in L1 and L2 seem to be dead.

10th epoch 990th epoch



We can fix dead neurons by clipping them to a small value like 0.001x in the relu derivative when x<=0 instead of 0 as seen in a leaky relu (x is a single matrix element from the corresponding layer). The code change would look like the following.

dW3 = (L3 - T) \* L3\*(1 - L3)

dW2 = np.where(W2.dot(L1)<=0,0.01\* W2.dot(L1),W3.T.dot(dW3))

dW1 = np.where(W1.dot(X)<=0,0.01\* W1.dot(X),W2.T.dot(dW2))

# 4. Understanding the weights (7 points)

Looking at the visualizations of the activations, weights and weight updates, explain what each plot means. Refer to the images in the “train” subfolder. Don’t forget to delete or rename old runs.

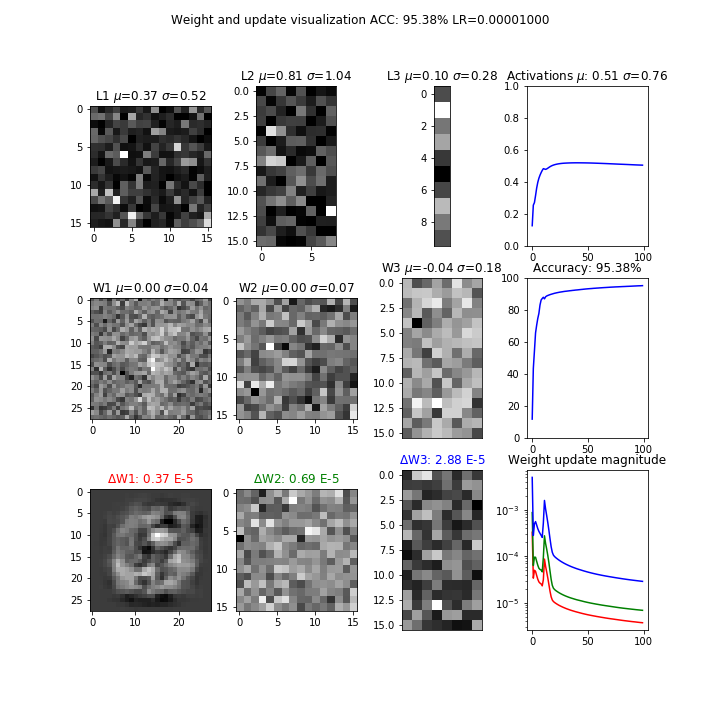
- How do the visualizations/plots differ for Tanh, ReLU and cross entropy?

- How does the weight/update magnitude change as training progresses? How are the magnitudes similar or different depending on the depth of the layer?

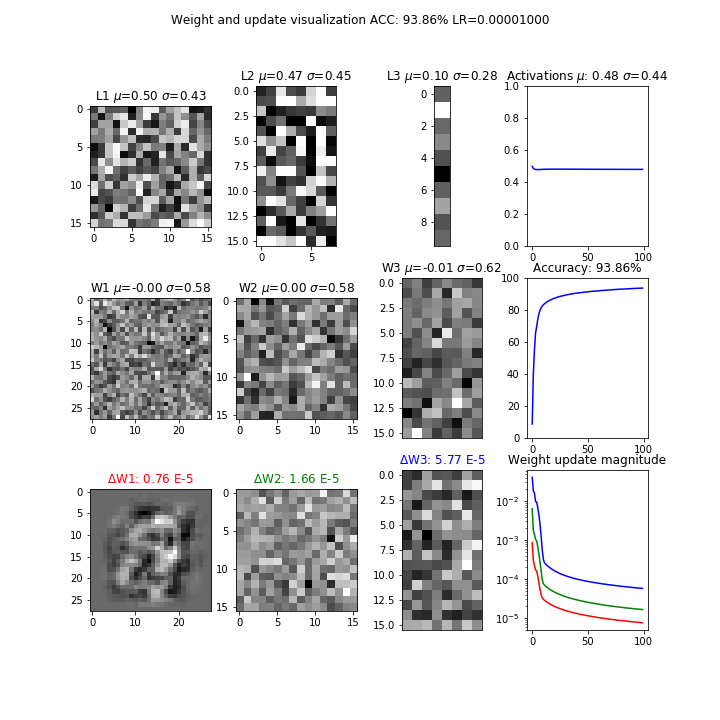
- What are signs that the network is “stuck”, and how should the plots look as the network reaches the final trained state?

- Does the network “prefer” certain activation/weight settings? Or do the activations/weights change with more training? Does this depend on initialization? Why?

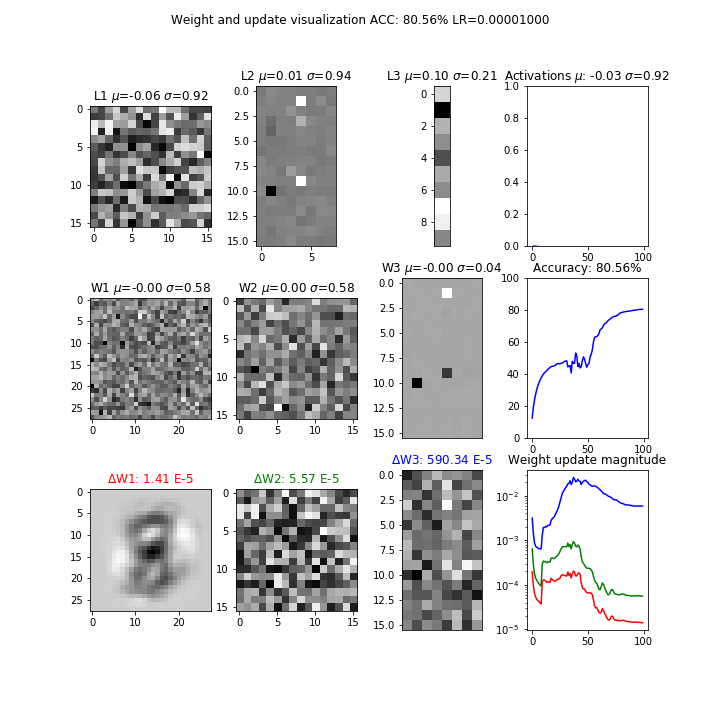
The activation, weights, and weight updates show which neurons are activated with black regions showing that they are “dead” and lighter regions showing up as “more alive” as the layers feed into one anoter. Let’s take for example the relu images at 990th epoch shown above. Relu is great at recognizing sparsity in the inputs and therefore we see these pixels in L1 and L2 as really dark. Without the aid of leaky relu, these regions will not be activated with each epoch as the relu takes the max of 0 and the activation value of the input matrix element.



Cross entropy is similar to relu in training speed as it quickly plateaus within a few hundred epochs, but notice how much more of the layers are activated in comparison at the 990th epoch, especially in L1 and L2.



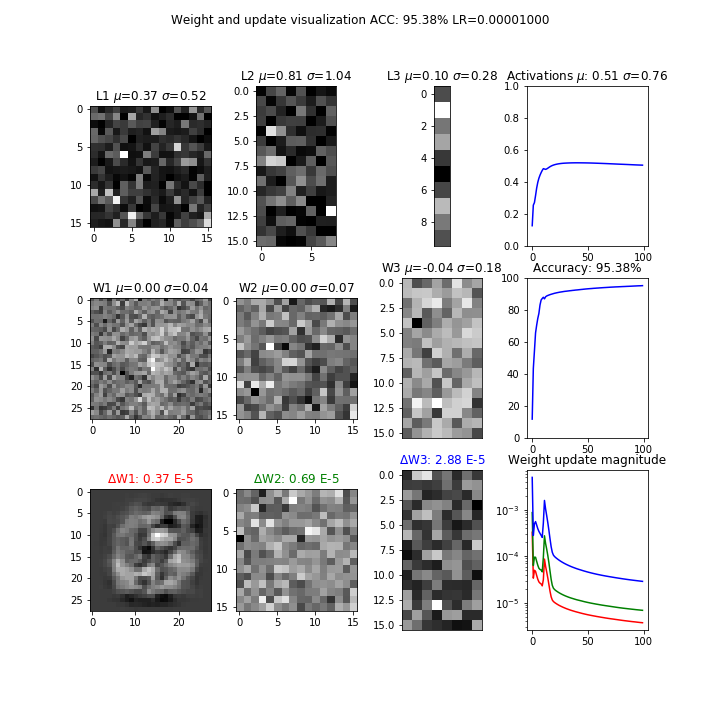
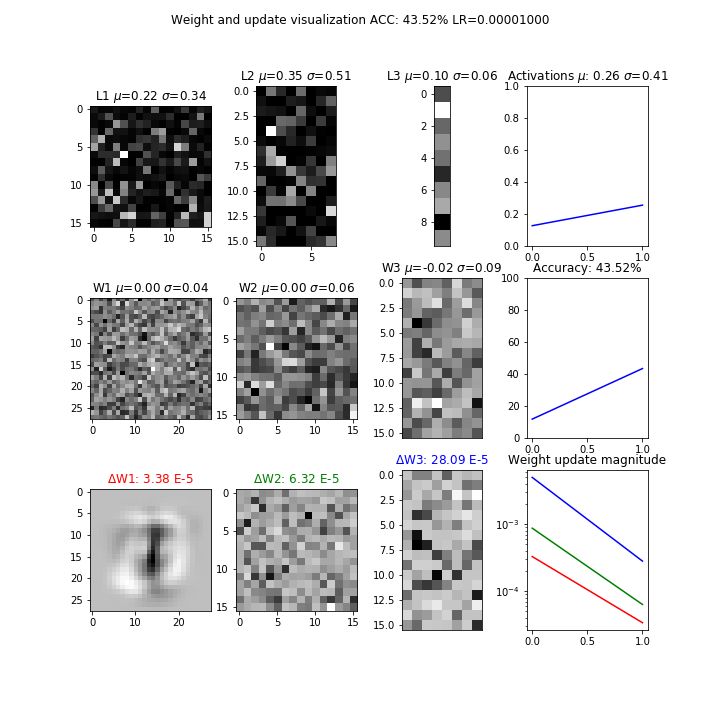
Finally tanh major improvement over sigmoid is that it mean centers the data and gets rid of the zig zagging nature when gradients are all positive or negative. Therefore we see a lot more gray at the 990th epoch.



The weight update magnitude rapidly changes in the first hundred epochs and then seem to generally plateau as there is less gradient to update weights with diminishing returns. The last layer (delta W3) seems to always have a higher weight update in comparison than the previous layers and this makes sense as backpropagation feeds the output weight updates into the earlier layers and changes their weight updates accordingly but each time on a smaller scale.

If we see the tanh visualization again up above, we see spikes in its accuracy and weight updates. This indicates that the network is stuck as it went off track on the gradient and is finding again how best approach the minima. As networks reach the final state, we will see three things shown below as described with the relu example. We should should plateauing with the accuracy and weight update magnitude and the deltas for each layers should get darker since there is less “bang for the buck” with diminishing returns.

10th epoch 990th epoch



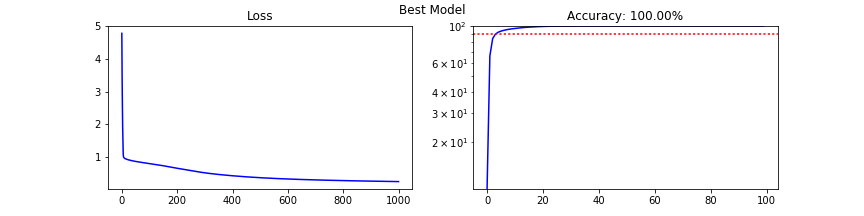
The networks don’t prefer certain activation/weight settings. They will change with more training and given enough epochs, most neural networks will approach similar accuracies and losses. However, the speed with which they arrive at certain accuracies and losses is dictated by how well We saw with relu, tanh, and cross entropy that choice of activation, weight initialization, and cost function greatly affected the speed with which we arrived near 90% accuracy in the first few epochs, compared to the original model with only sigmoids and mse loss with which we only achieved 50% accuracy in the first 200 epochs.

# 5. Putting it all together (10 points)

Starting with the example code from lecture 3, integrate all your improvements from part 3 (Tanh, Cross entropy, ReLU and others that you can think of) together to attain the best possible training conditions. Comment your code thoroughly, and show plots of how your code improves upon the example. Explain thoroughly what you did and why it works. Submit your final code, but comment out the lines that you aren’t using, e.g. tanh.

I made a number of changes for the final code listed below:

1. zero centered and whitened the data using PCA dividing by singular values
2. initialized weights dividing random normal by square root of fan in
3. L1 – relu, L2 – tanh, L3 – softmax with cross entropy cost
4. Applied clipping to the tanh derivative in dW2
5. Applied leaky relu of 0.01s to relu derivative in dW1



My model had a 100% accuracy which indicates possibly overfitting. I whitened the data instead of dividing the pixel values by 255 and then taking a random uniform from range [-1,1]. I chose to use the initialization of dividing random normal by square root of fan in because it works pretty well with relu and gives unit variance as well. I had originally tried to use relu on all hidden layers with a softmax in the output but kept getting NAN values in the softmax, probably because relu sparsity creates a lot of zeros which feed into softmax in L3. That is why I used a tanh in L2. I applied the clipping for both the relu and tanh to take care of vanishing gradients. Finally, I decided to use cross entropy to speed up training as it doesn’t need to consider the local gradient in the weight update in dW3.

The only discernible difference in the plots below is the 990th epoch dW2 is darker than the 10th epoch’s while the 990th epoch’s DW3 is brighter than the 10th epoch’s dW3. Also the 990th epoch’s dW1 is more grayed out from the bottom than the 10th epoch’s image. By the 10th epoch, the model had already 65.98% and it quickly plateaued above 90% accuracy within the first 100 epochs.

10th epoch 990th epoch

