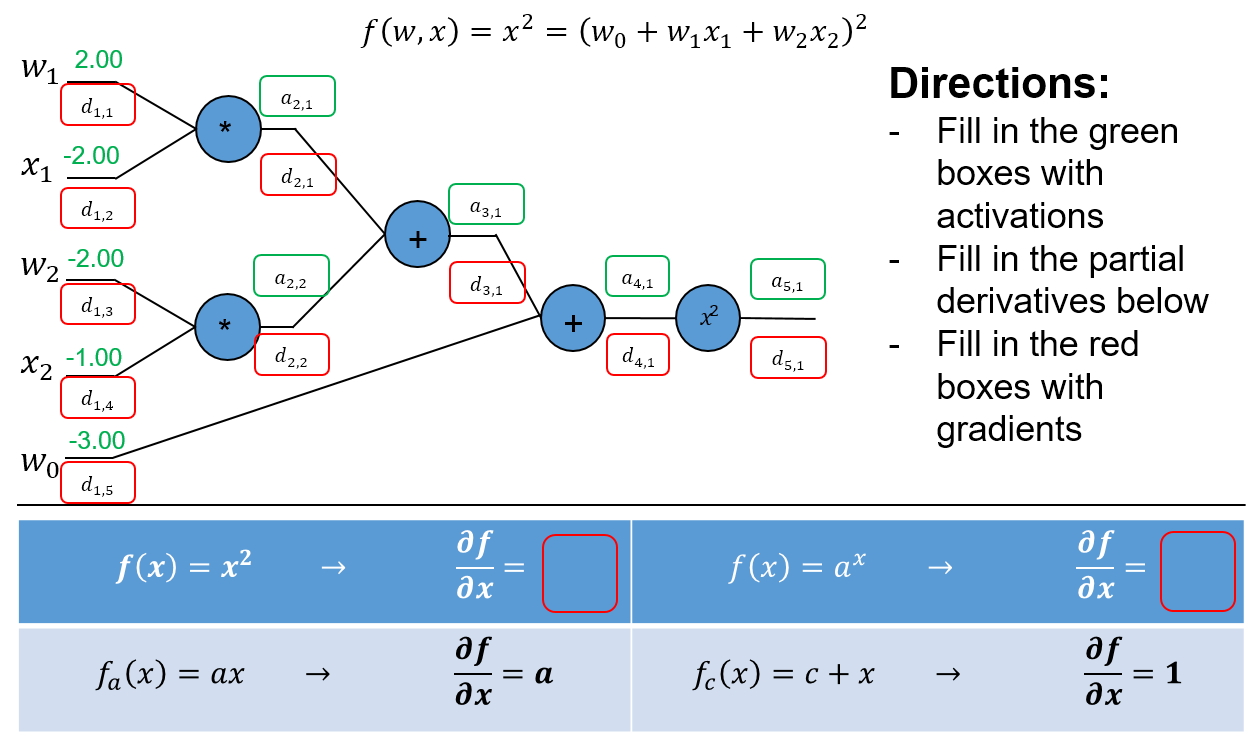
Assignment 1 (Due 4/26/2017)

Understanding Neural Networks

|  |  |
| --- | --- |
| **Name** | Eric Chang |
| **Discussion partner** |  |
| **Comments** | I included every file I ran to run the models. However, all the activations, gradients and costs are implemented in the final model, titled “4-final-model.py”, so that should be the only one needed for grading |
| **Feedback** | I and a couple of my classmates were confused about “dead” neurons when using ReLU activations. I think it would be great to show how to induce dead neurons, for understanding. |

# 1. Neural Networks on paper (5 points)

Fill in the blanks below:



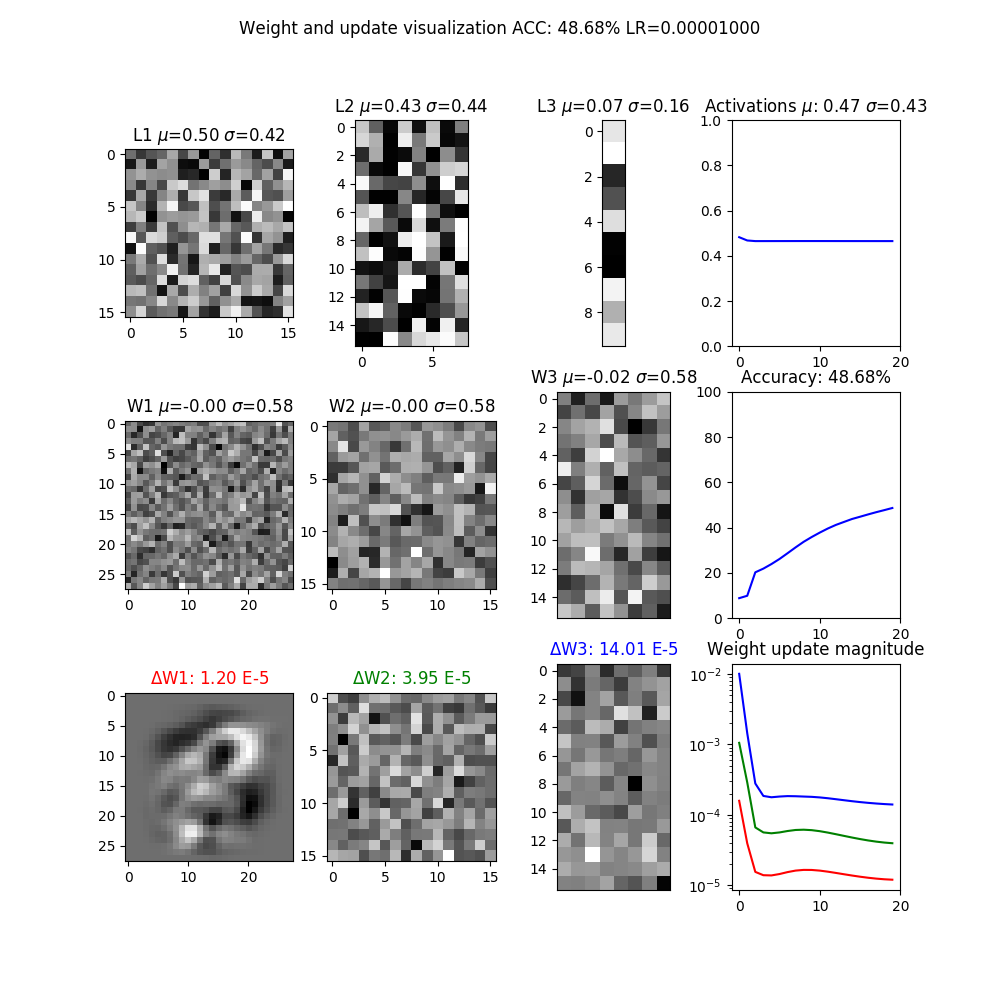
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  | |  | |  | |
|  |  |  |  |  |  |
|  |  |  | |  |  |

# 2. Neural Networks in code (12 points)

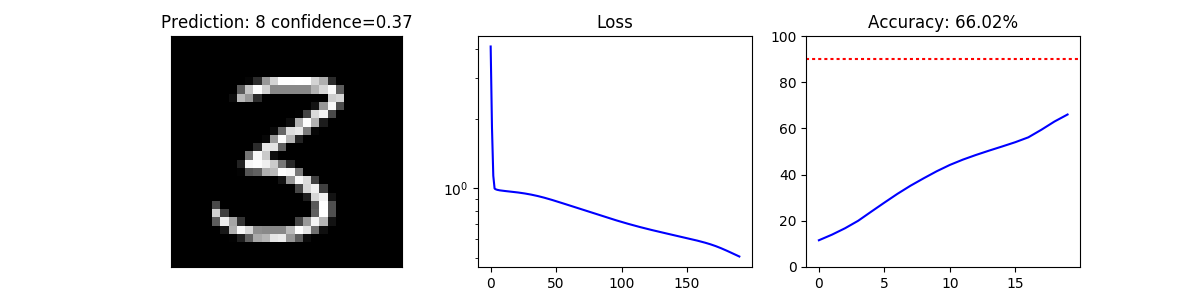
Using the provided example code from Lecture 3, explore the items below and demonstrate how to improve the example code by showing plots of the loss and accuracy curves. For each plot, show the curves for the first 200 iterations (You can stop training after 200 iterations for this part of the assignment).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

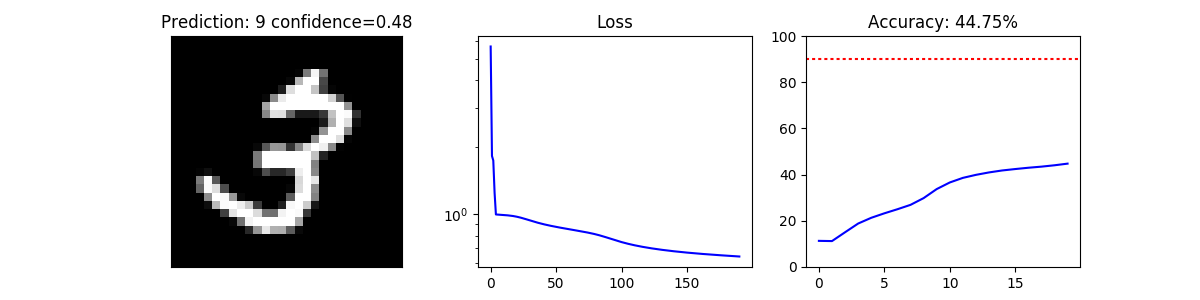
Through around 10 trials of learning rates ranging from 1e-5 to 1e04, I found that the ideal learning rate for the initial model at 200 trials was around 1.8e-5. A good learning rate may look different depending on the activation function and cost function, but the learning rate should generally result in a quickly growing accuracy in the beginning. Too slow and the ascent will not be steep or fast enough, too fast and it might overshot.

****

**eta = 1e-5 (initial)**

****

**eta = 1.8e-5 (just right)**



**eta = 4e -5 (too fast)**

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.

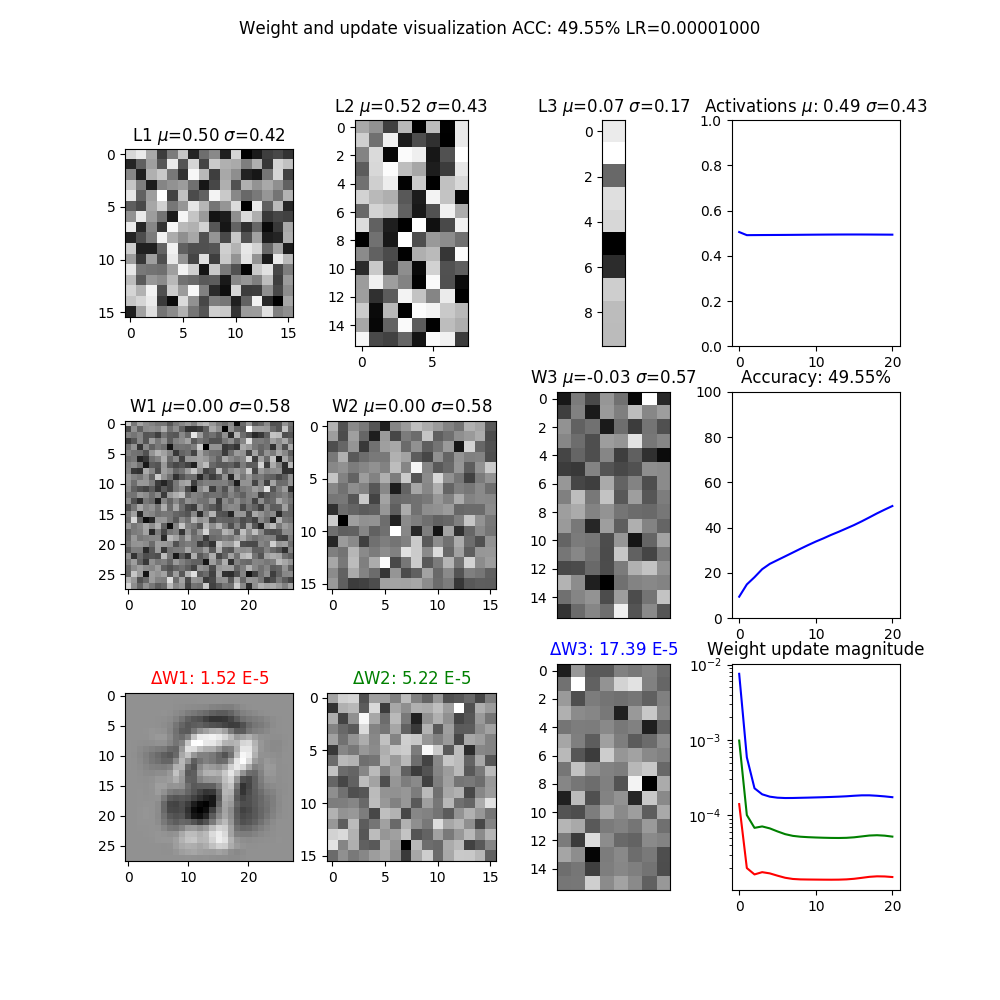
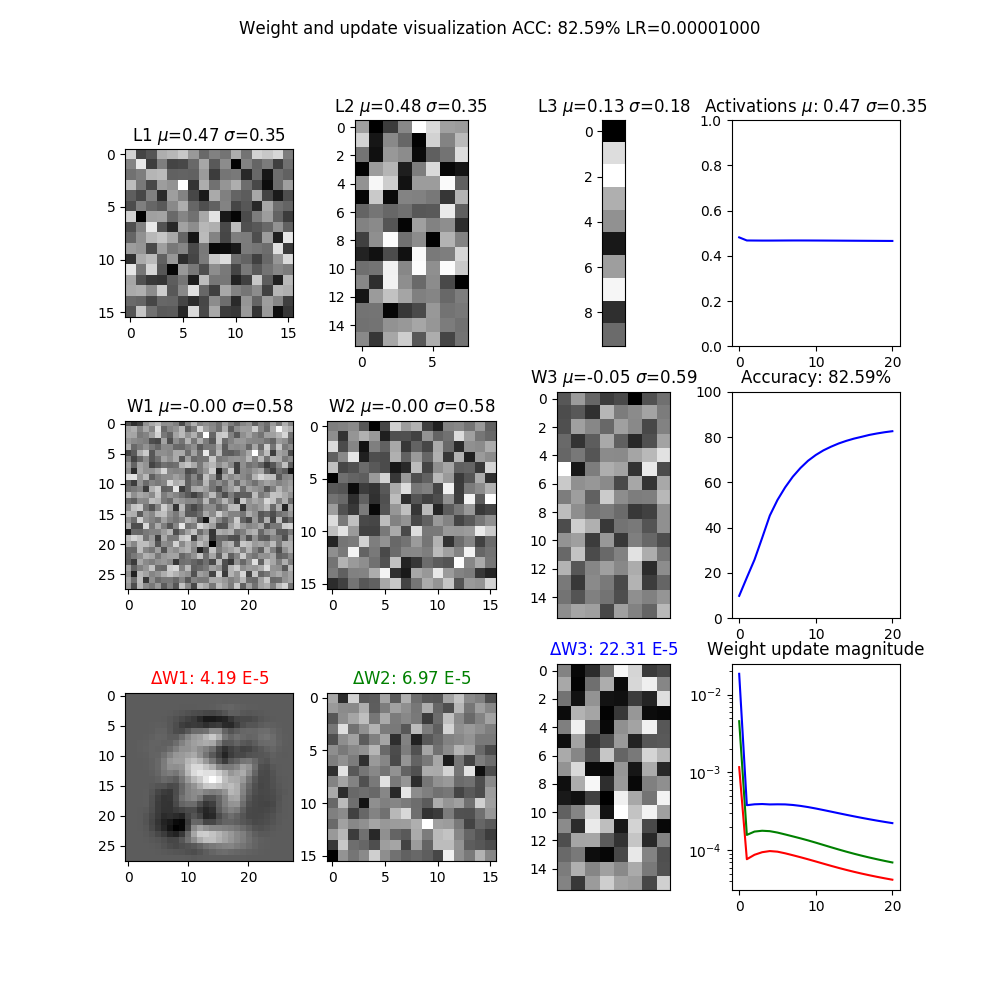
k is a constant by which the input value is multiplied. A large value of k (greater than 1) will translate to artificially large inputs for x, forcing y values closer to the extremes of the sigmoidal function and “narrowing” the function. A small value of k (between 0 and 1) will do the opposite, “widening” the function by bringing inputs closer to zero, the center of the sigmoid function.

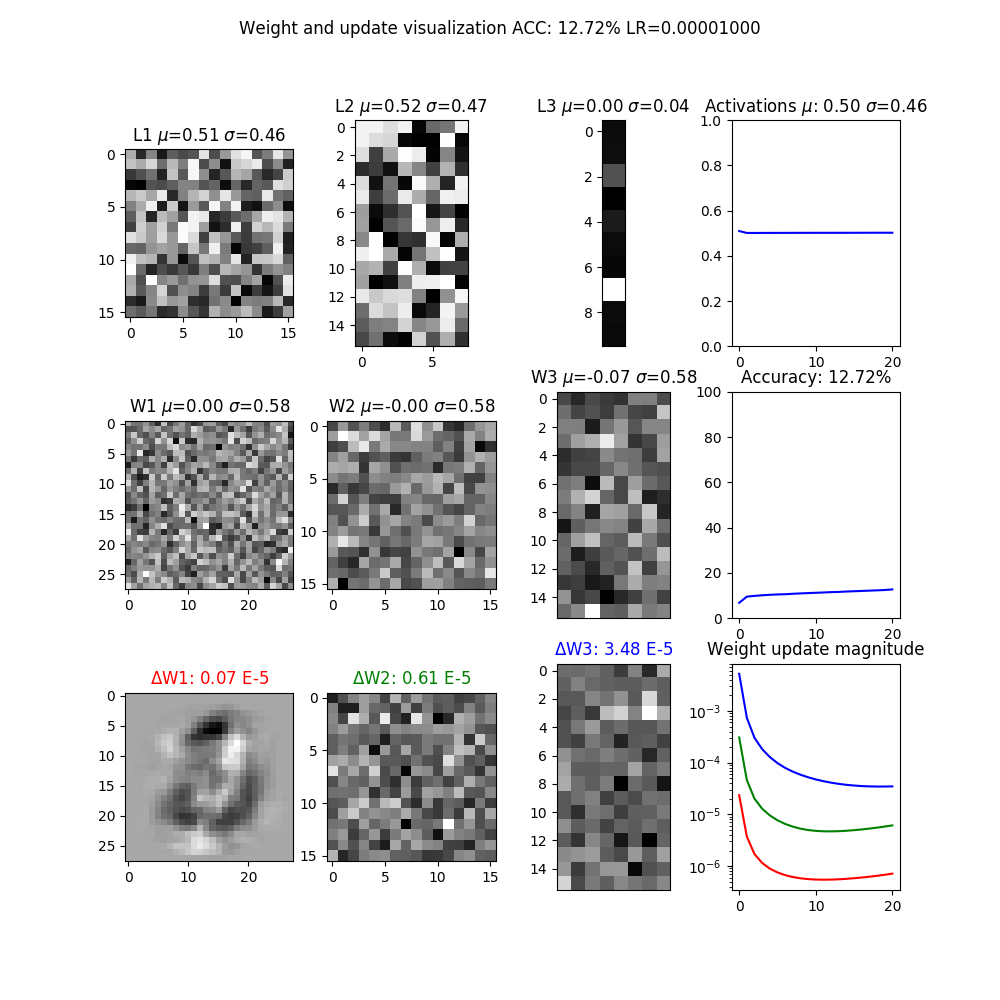
****

As we discussed in Lecture 4: “Active regions of Sigmoids”, training slows down drastically at the edges of the sigmoid function. For this reason, I hypothesize that a smaller value of k will widen active region of the sigmoid function and speed up the training speed. One potential drawback of using this transformation is artificially large gradients – we may be more susceptible to overshooting.

Below are outputs at the 200th epoch for three values of k – 0.5, 1, and 2. These results prove our hypothesis correct with the following observations:

1. Accuracy: As k increases, the accuracy of the network at 200 epochs falls drastically. This is consistent with a slower training rate.
2. Weight updates: As k increases, the weight update / gradient matrices also fall drastically. When we compare the values of delta W3 when k = 0.5 (22.31e-5) and k = 2 (3.48e-5), we already see a fairly large difference in magnitude, where k = 0.5 results in an update around 8 times larger than k = 2. As we move backwards to the input layer, the effect of the sigmoid function multiplier grows stronger. When we compare the values of delta W1 when k = 0.5 (4.19e-5) and k = 2 (7.00e-7), we see a difference of *two orders of magnitude*.

****



**k = 2**

**k = 1**

**k = 0.5**

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?

In Lecture 4, we discussed an issue with randomly initialized weights. Poor initializations can result in overly large gradients – if a weight is too large, the gradient calculated during backpropagation will in turn be very large.

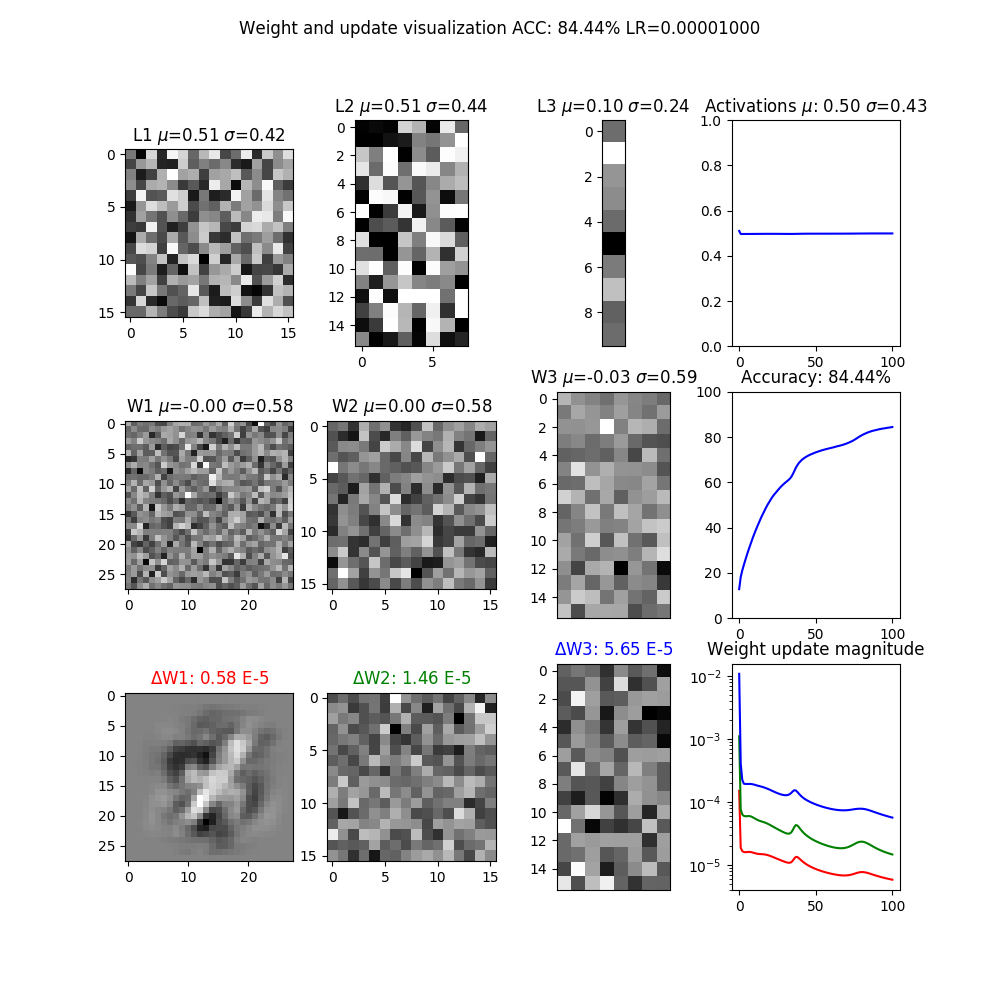
One possible fix is to sample from a standard normal distribution instead of a uniform distribution. This can be justified because all inputs were standardized to the range [0, 1] – therefore, we expect the weights to be centered around 0. Sampling with the standard normal distribution allows us to obtain a vector of random weights with most weights clustered around zero and few extreme values.

However, since adding normal distributions results in a sum of their variances as well, initializing using standard normal distributions will result in an inflation of variance proportional to the number of inputs (or the number of pixels, in our case). The solution is to standardize the variance by dividing the weight vector by 1 / sqrt(n), where n is the length of the input vector.

# 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):

**To evaluate the effects of our optimizations, here is the learning curve for the original set of parameters (sigmoid activation, MSE cost, learning rate 1e-5).**



Activation: sigmoid(x)

Cost: MSE

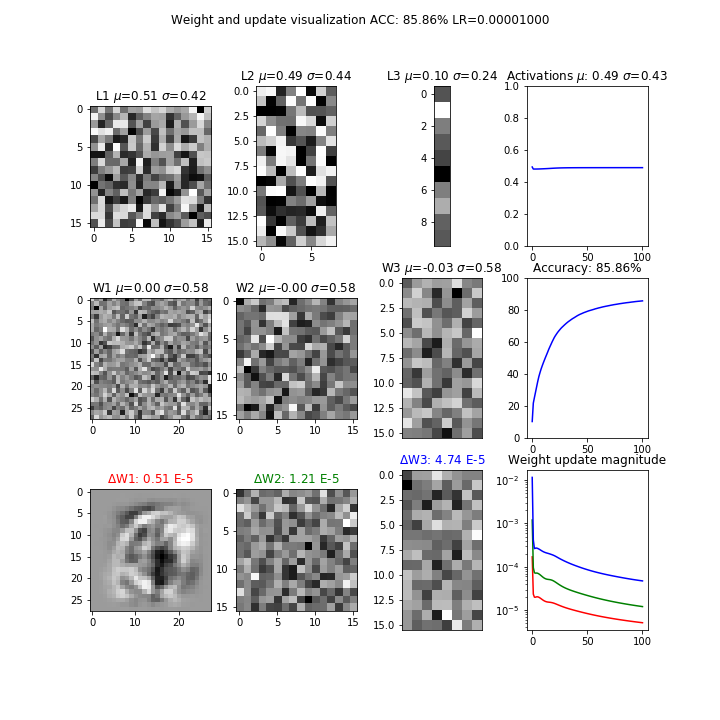
Epoch: 1000

Learning rate: 1e-5

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.

The most pressing issue we note in our original run of the network is the vanishing gradient issue. As gradients are backpropagated through the network, they tend to get smaller and smaller since we are, by the chain rule, multiplying gradients by smaller gradients. This tends to be an issue with the sigmoid activation function. A possible fix to this is to truncate the sigmoid gradient at 0.01 – if the gradient is smaller than 0.01, it is increased to 0.01.

Another issue with the activations is that the mean is 0.50. An activation function that is not zero-centered will result in biased inputs to the hidden layers and output layer. This can be remedied by implementing an activation function with mean 0 like tanh(x), which we will do in part (2).

****

**Activation: truncated sigmoid**

Cost: MSE

Epoch: 1000

Learning rate: 1e-5

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)?

Resources used:

CS231, “Commonly used activation functions”: <http://cs231n.github.io/neural-networks-1/#actfun>

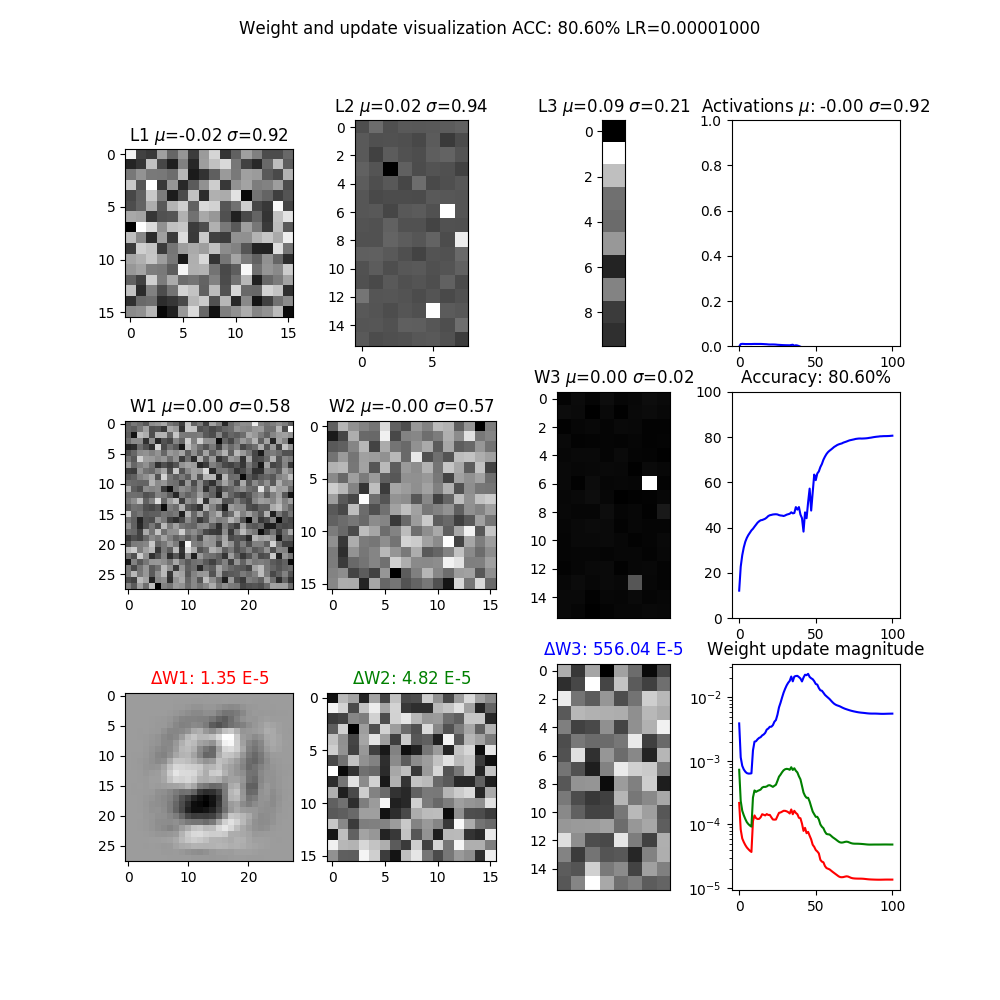
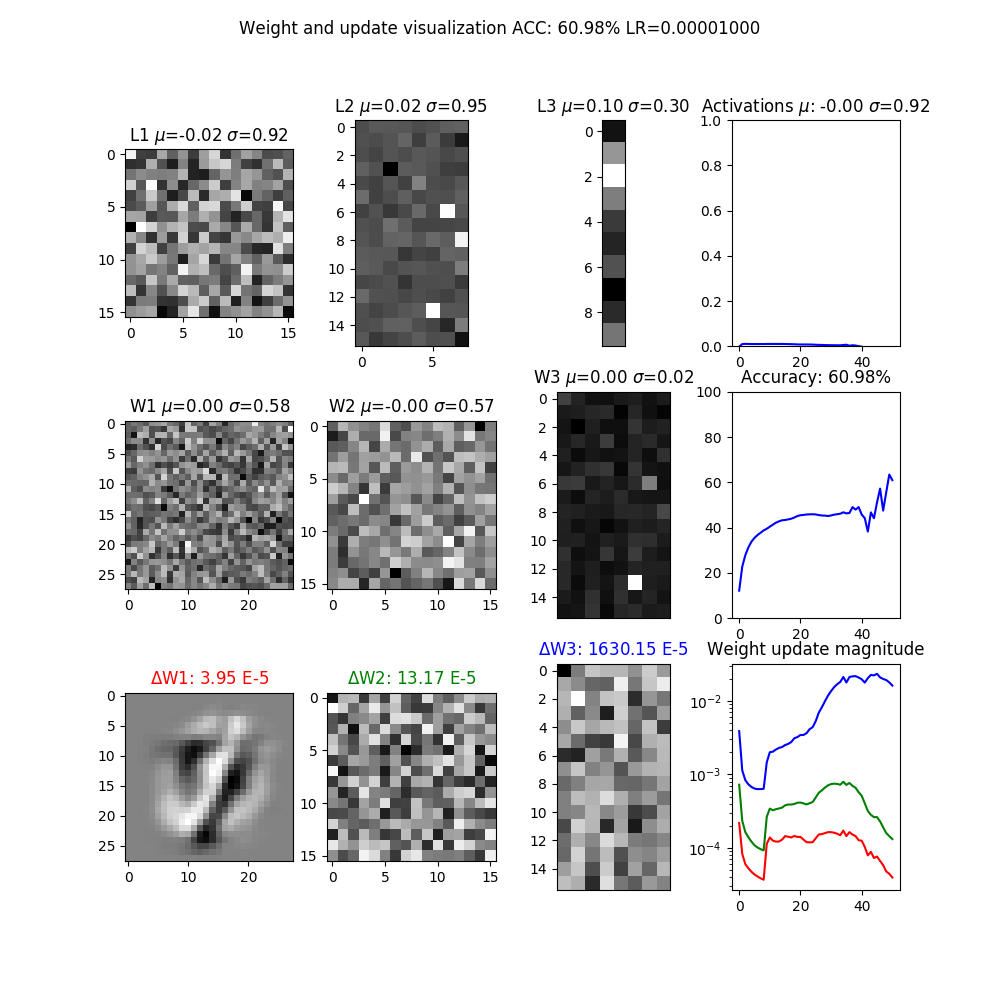
Tanh(x) solves the “zigzag” problem and the activation bias issue. Because the sigmoid function maps all inputs to the range [0, 1].

1. Activation bias: Sigmoid activations have a mean of 0.5 and a range of [0, 1]. This bias will result in all activations and therefore all inputs to the hidden layers and output layer to be positive.
2. “Zigzag” problem: A consequence of the activation bias is the zigzag issue. Because all activations are positive, the gradients can only be positive or negative. This can slow the model convergence, as the path to the optimum can only be taken by a manhattan distance path.

Using tanh(x), does not, however, solve the disappearing gradient issue, since the gradient near the extrema of tanh(x) is still very small.

We observe this improvement in when looking at the activations and accuracy of our tanh(x) plot.

1. The mean of activations for our original run was 0.50, and now it is 0.00.
2. The effect of the faster training is seen in the weight update matrices. For L3, the matrix is 2 orders of magnitude greater than that of the original run. The result of this is that the accuracy converges to its plateau of 80% at around the 600th epoch, whereas our original run needed nearly the full 1000 epochs to converge.

****

**Activation: tanh(x)**

Cost: MSE

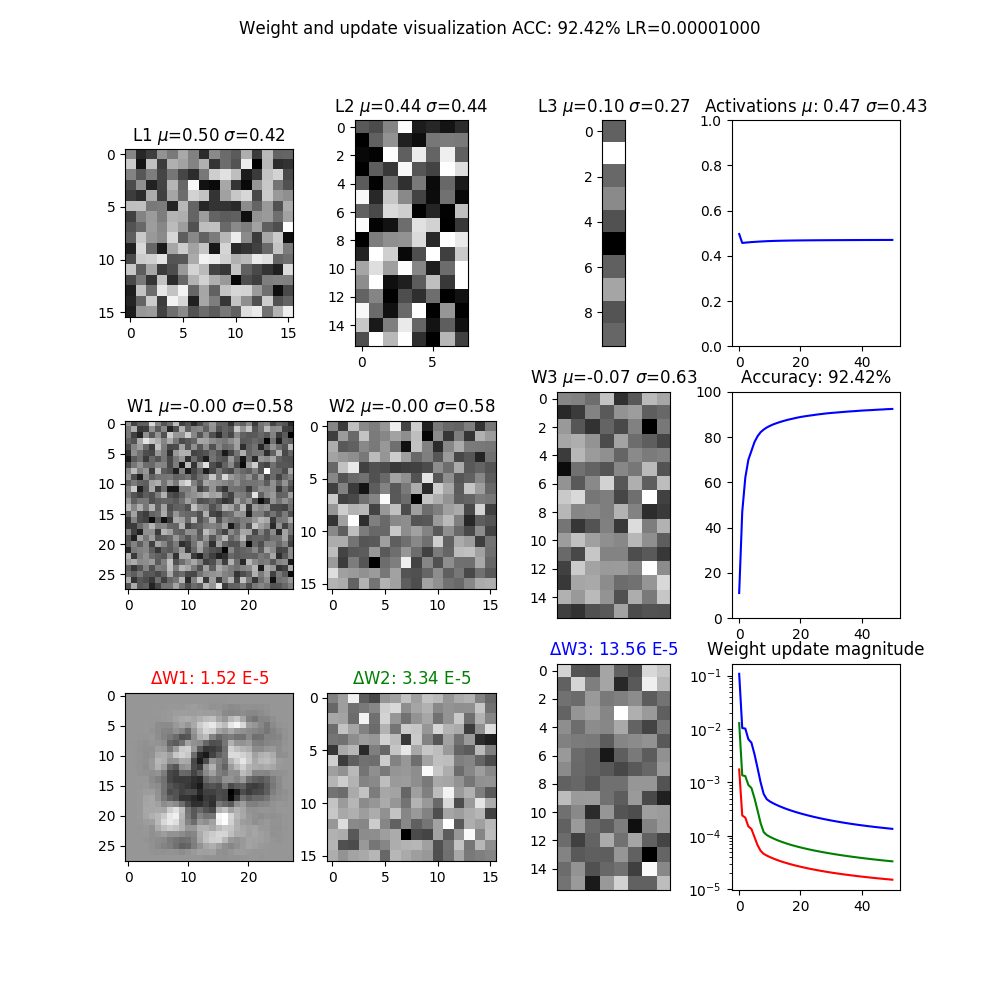
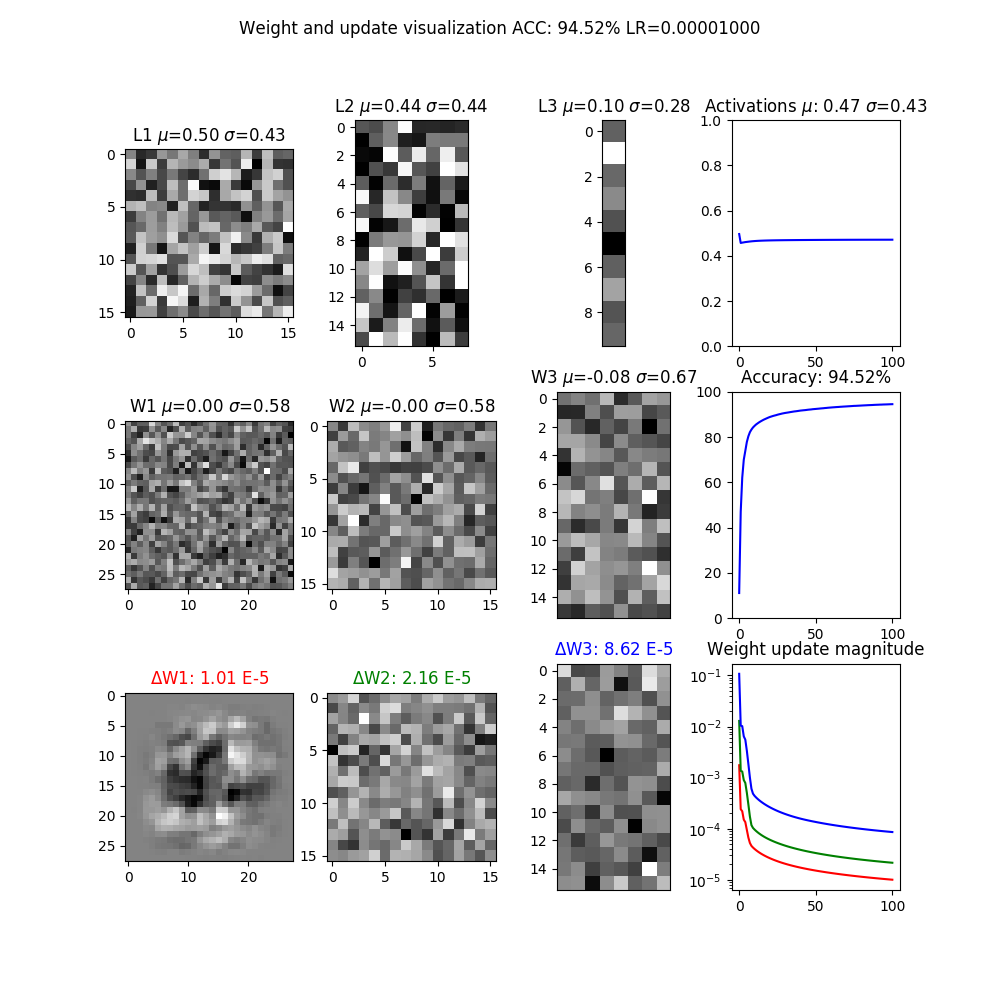
Learning rate: 1e-5

1000 epochs

500 epochs

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

Using cross entropy as our cost function gets rid of the chain rule term (multiplying by the gradient of the sigmoid or tanh(x)) at the first step of backpropagation. This is supposed to reduce the effect of concatenating small gradients and expediting training. In our result, we see that this results in a huge speed up in training – an accuracy of around 90% is reached before the first 100 epochs. The effect of removing the sigmoid gradient term can be seen in the mean of dW3 – it is around twice as big compared to the

****

500 epochs

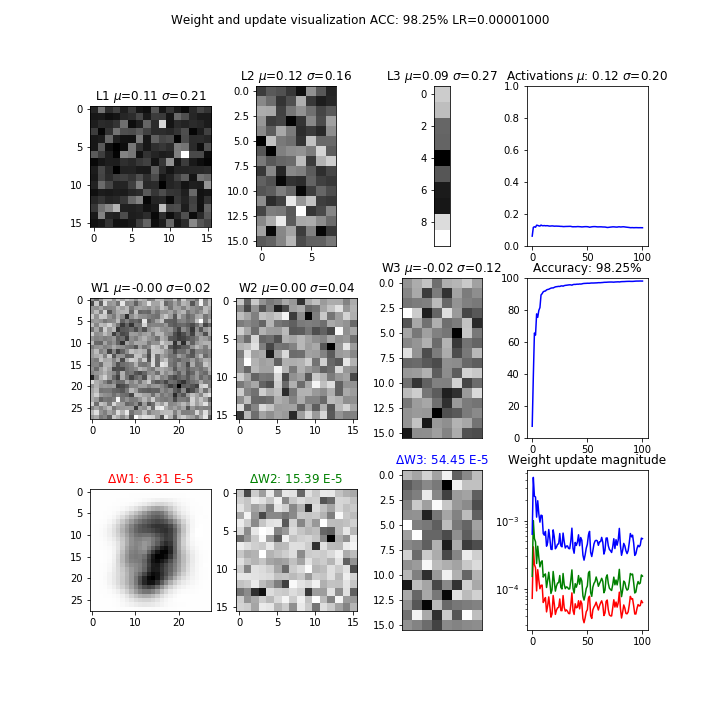
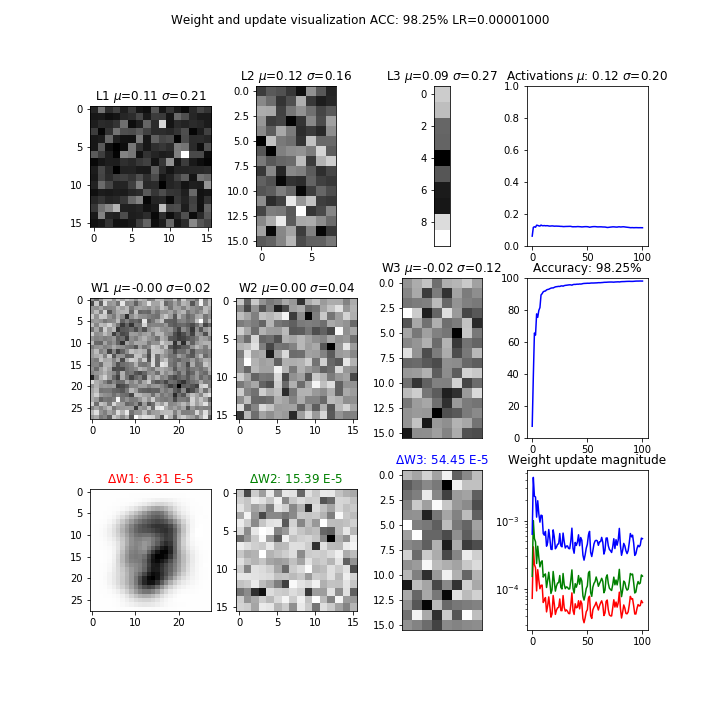
Activation: sigmoid(x)

**Cost: cross entropy**

Learning rate: 1e-5

1000 epochs

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?



**Activation: relu(x)**

Cost: MSE

Epoch: 1000

Learning rate: 1e-5

Resources used:

<https://stats.stackexchange.com/questions/273927/when-using-relu-is-it-normal-for-the-activations-to-go-up-at-each-layer/273933>

The ReLU activation function’s major advantage over tanh(x) and sigmoid(x) is in its simplicity. Sigmoid and tanh both require function evaluations with exponentials for both the activation and the gradient, whereas ReLU’s activation is just a simple if/else statement and its gradient is an indicator function. This results in noticeably faster iterations compared to the other two activations. In addition, the higher gradient values result in faster convergence as well – we see in the weight update matrices that the means of the weight updates are around 5 times bigger than the sigmoid activation across the board.

“Dead” neurons form when a neuron’s weights update in a way that results in a 0 activation for all inputs. If this happens, since the gradient of the ReLU is an indicator function for whether the activation value is nonzero, every gradient flowing through that neuron from that point forward will be 0. On the visualizations, this will appear as a pixel that is always black on the weight updates.

Unfortunately, for some reason, I could not get my neurons to die. I even tried to force dead neurons by manually inserting an entire row of 0 activations in the middle of training, I still didn’t observe any dark pixels in the dW matrices.

We can fix “dead” neurons by using Leaky ReLU, where the floor for the gradient is set to a small value like 0.01 instead of 0. This allows neurons that have “died” a chance to recover with a small positive gradient.

# 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?

Tanh:

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

The weight update magnitudes are usually large at the beginning of training. As the network converges to its fully trained state, the magnitudes will get smaller and smaller – when using ReLU activations, the updates may also oscillate. When the network is stuck in a plateau where it has seen no improvement in accuracy for a while, the weight updates will spike.

The weight updates should always be higher the closer they are to the output layer. This is due to the backpropagation algorithm. Since we are propagating small gradients backwards, by the chain rule, we will be multiplying small gradients by small gradients as we move through the network. Therefore, the magnitude of the weight updates will always follow the patter dW3 > dW2 > dW1.

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

A “stuck” network can be seen in the accuracy curve – when the network is stuck, it will manifest itself in a plateau on the accuracy curve. At the same time, the network tries to escape its stuck state by increasing the magnitude of its weight updates, resulting in a spike in the weight update magnitude plot. This situation is most prominent with our tanh(x) example.

With a good learning rate, as the network converges, the weight updates should decrease since it is nearing its final trained state. The exception is with ReLU activations. With the ReLU activations, a proper learning rate should manifest oscillations in weight update magnitude near the end of training.

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

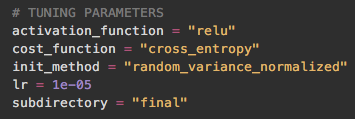
Yes. Looking at the speed and final accuracy of training, the ReLU activation is objectively the best out of the 3 we implemented.

When implementing the ReLU activation, I noticed that if I did not adjust the variance of the inputs to each layer by dividing by sqrt(n), where n is the dimensionality of the fan-in, my activations exploded across the hidden layers. I believe this is because the sigmoid and tanh activation functions mapped all activations to between [0, 1] or [-1, 1], but ReLU allows large activations. Therefore, failing to normalize results in a variance of , where n is the size of the fan-in and is the variance of the distribution used to initialize the weights. Large values then compound and cause exploding activations as high as 10e5 in the final layer.

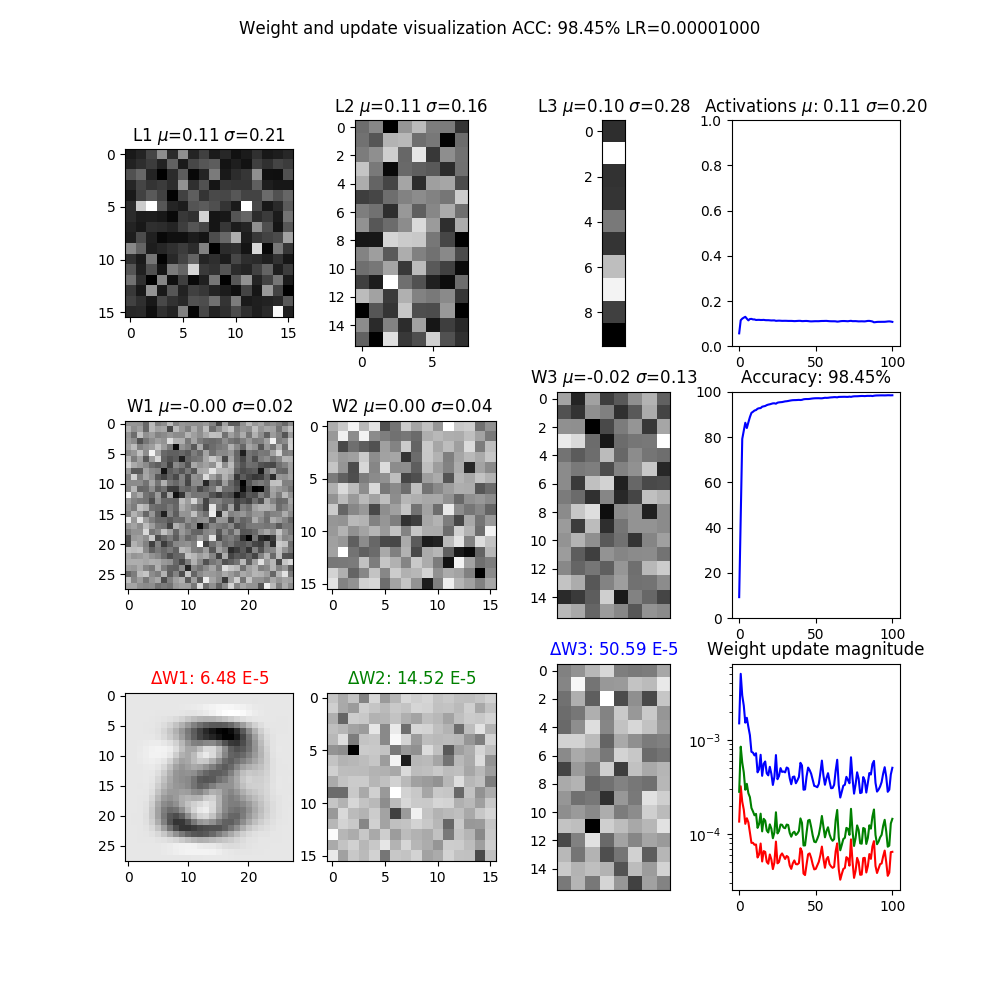
# 5. Putting it all together (10 points)

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.

My final model had the following parameters:



It achieves an accuracy of 98.56% at 1000 epochs. The main improvement can be seen in the quality and speed of the network. The network also converges very quickly, getting to nearly its full accuracy at around 100 epochs – due to both the ReLU activation and cross entropy cost.

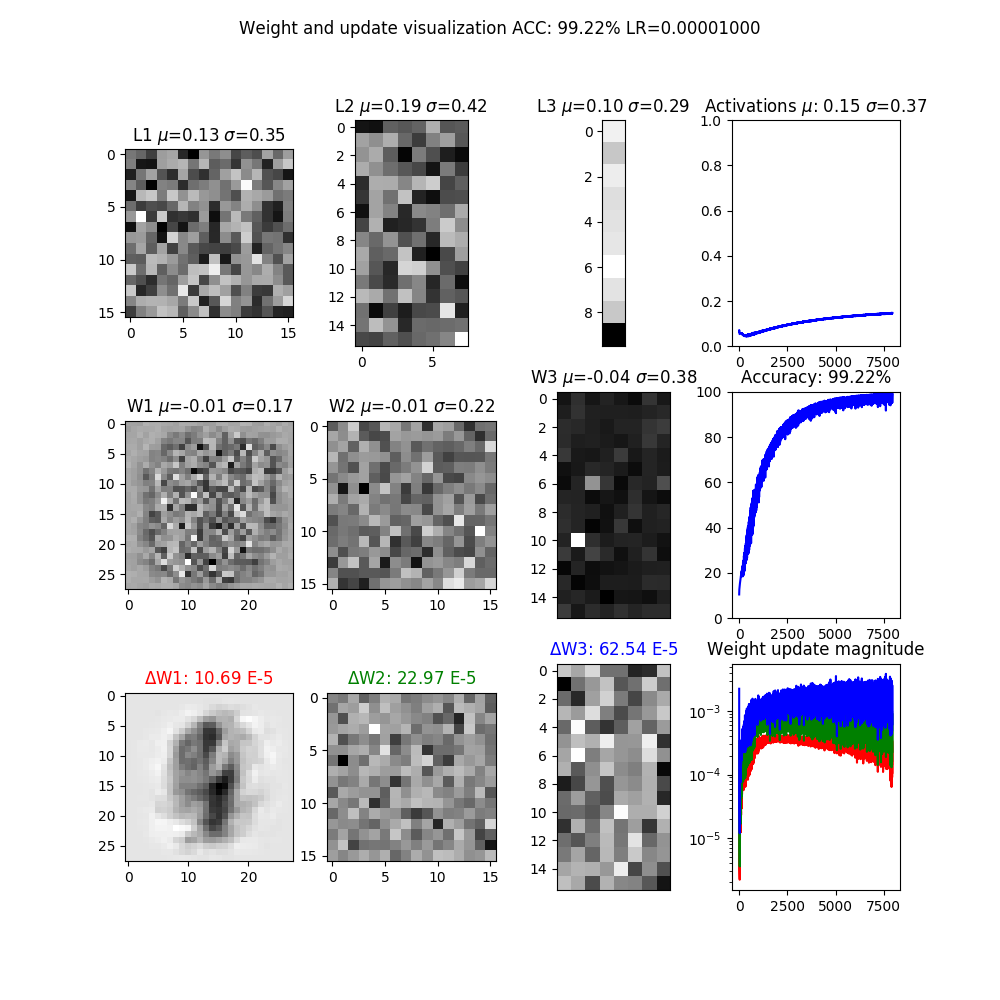




****

**Extra credit:** Recall the discussion of random labels in class and how neural networks may have enough capacity to remember the whole training set in the weights. Uncomment the lines for random labels. Explain if your model may have enough capacity for overfitting. Suggest ideas for fixing this problem.

I trained my final network configuration on random labels for around 20 hours. Here is the result at 75,000 epochs:



My network has completely overfit and attained an accuracy on the training set of 99%. This network absolutely has the capacity to memorize the entire training set. An interesting observation is that this network trained, the weight magnitudes actually grew and exhibited crazy oscillations as training continued. This is the exact opposite of properly labeled networks – usually we see an inverse relationship between accuracy and weight update magnitude.

If the network didn’t have enough capacity to memorize the entire dataset, a solution would be to simply increase the number of neurons.