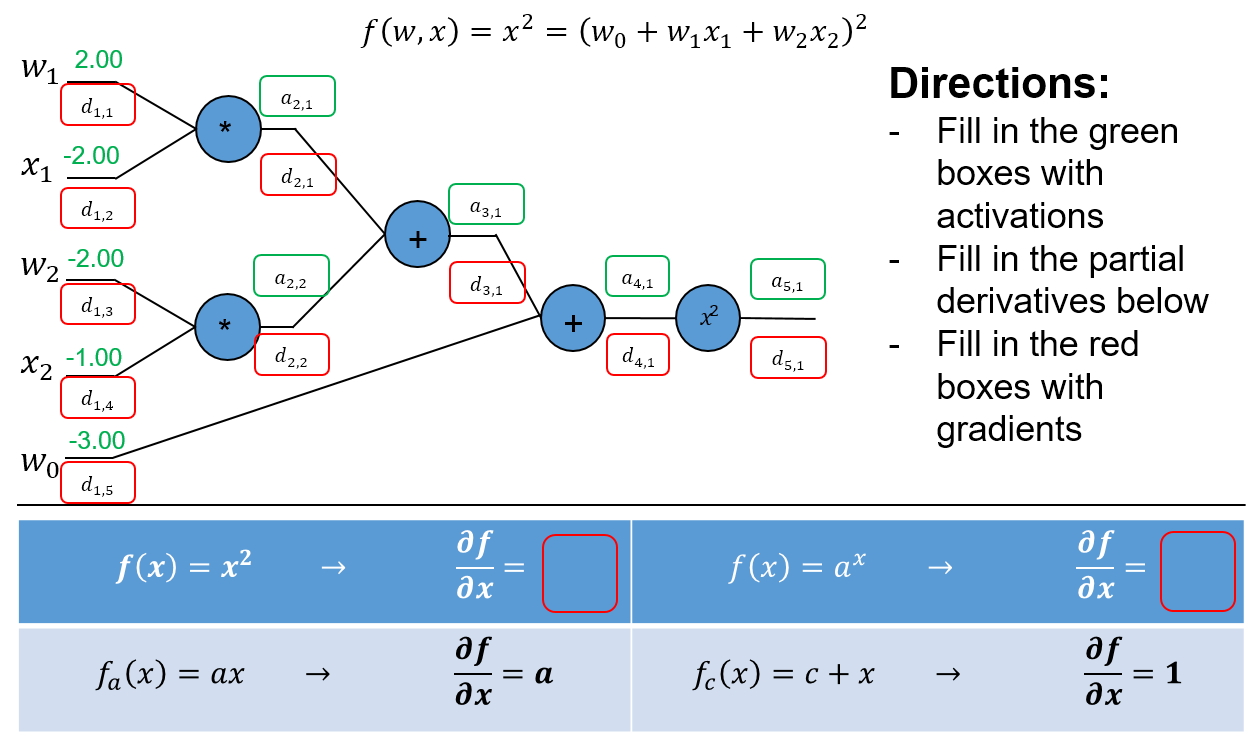
Assignment 1 (Due 4/26/2017)

Understanding Neural Networks

|  |  |
| --- | --- |
| **Name** |  |
| **Discussion partner** |  |
| **Comments** | Add comments for the grader here. E.g. How to run the code, or anything to note when grading the code. |
| **Feedback** | Note any feedback that you’d like to address in a future lecture. |

# 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?

From my experiments in testing different learning rates, I found that the ideal learning rate is between 0.0001 and 0.00001.

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.

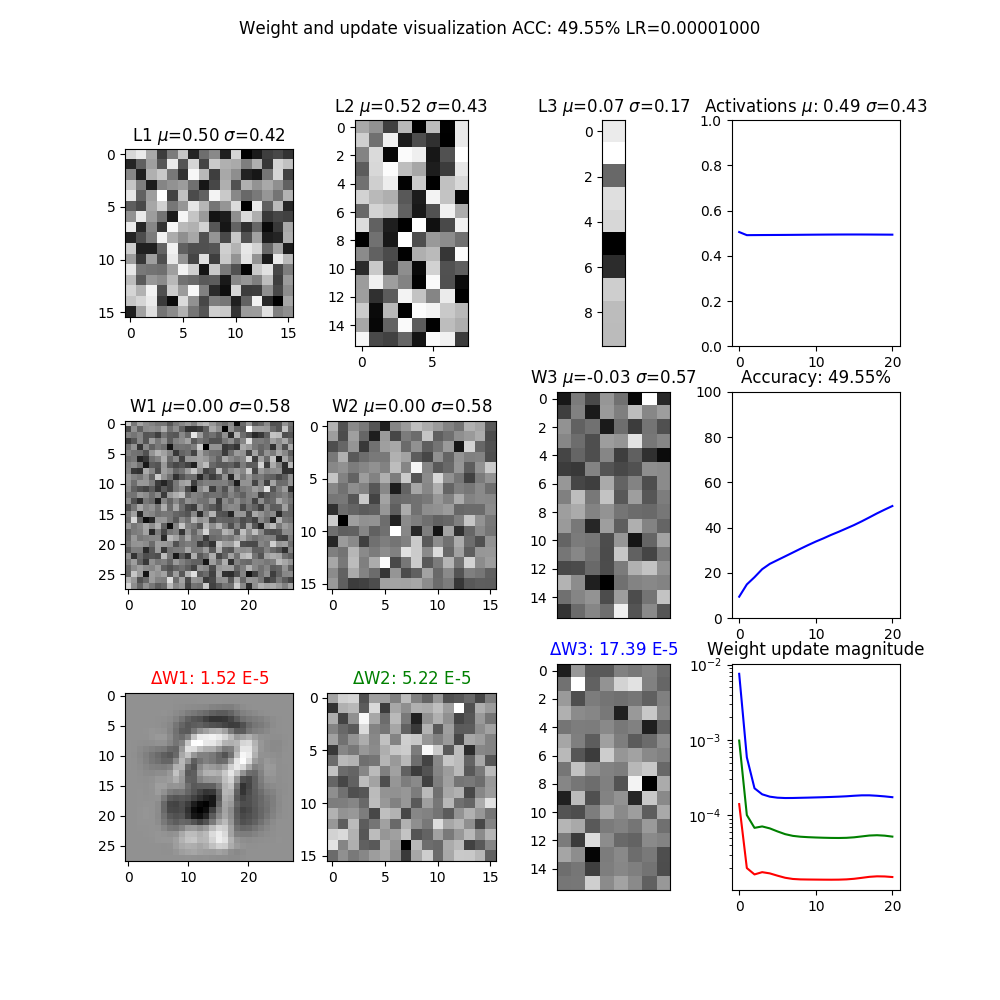
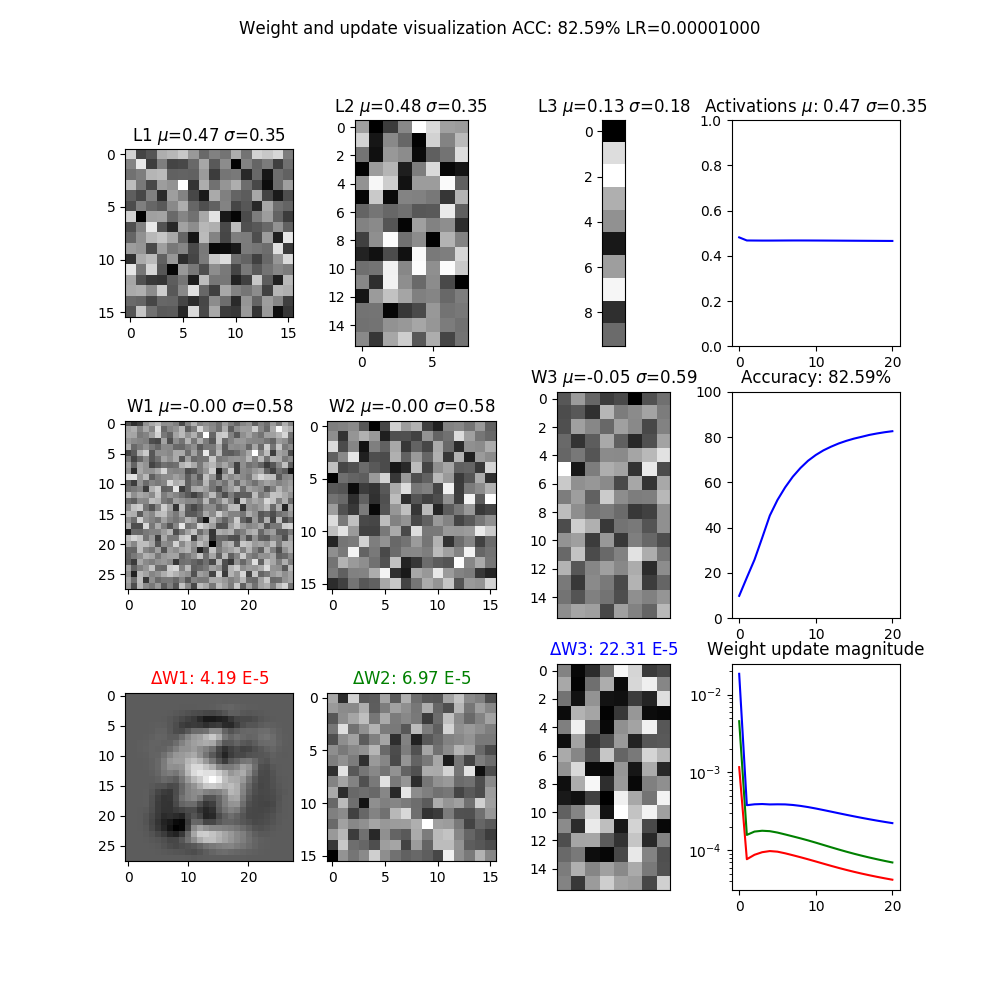
k is a constant by which the input value is multiplied. A large value of k (greater than 1) will translate to artificially large inputs for x, forcing y values closer to the extremes of the sigmoidal function and “narrowing” the function. A small value of k (between 0 and 1) will do the opposite, “widening”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

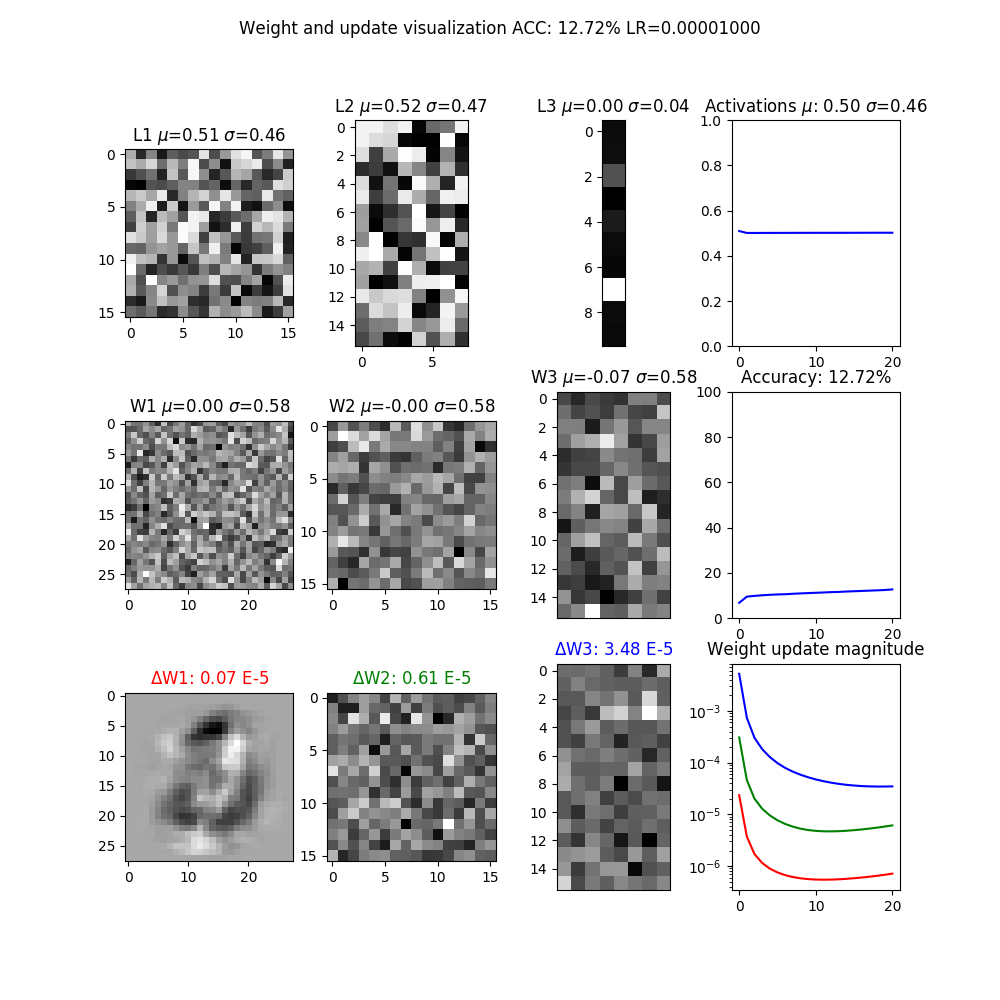
****

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.

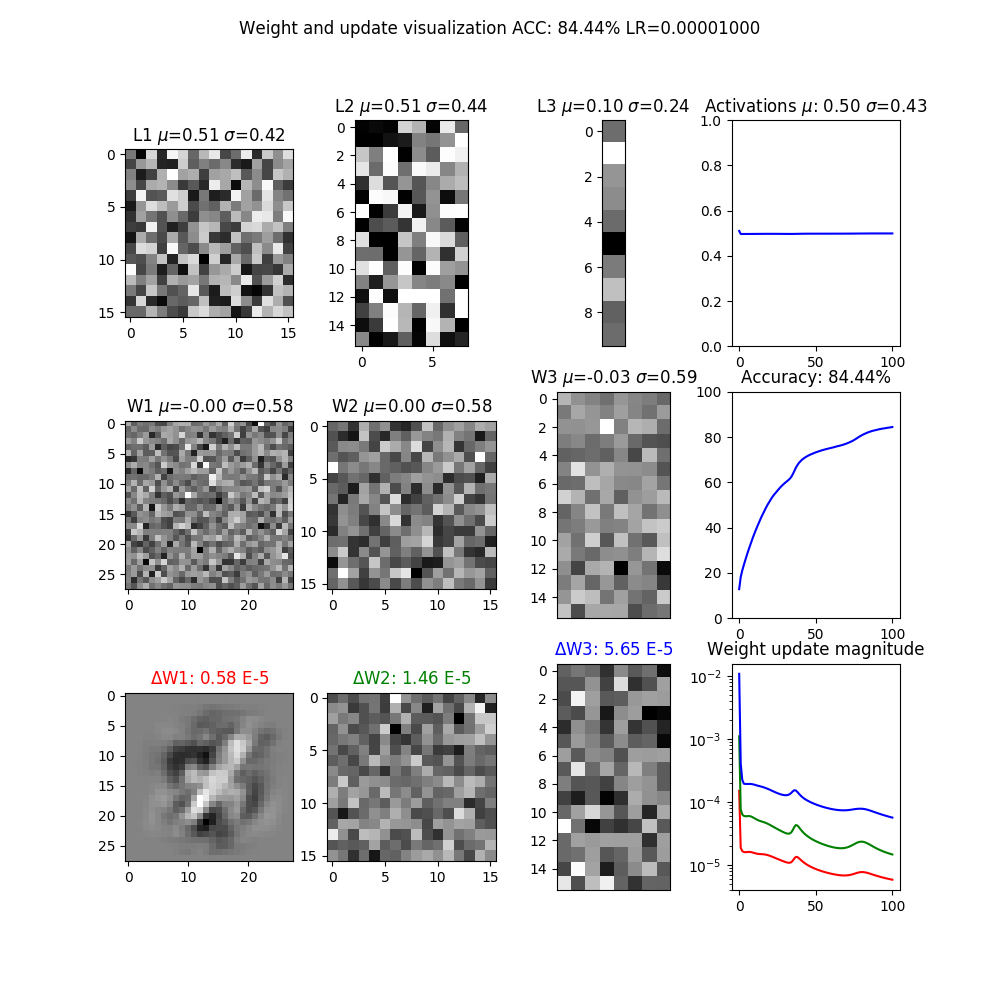
One possible fix is to sample from a standard normal distribution instead of a uniform distribution. This can be justified because all inputs were standardized to the range [0, 1] – therefore, we expect the weights to be centered around 0. Sampling with the standard normal distribution allows us to obtain 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 pushed up.

An 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).

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:

Lecture 4, Ellick chan

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

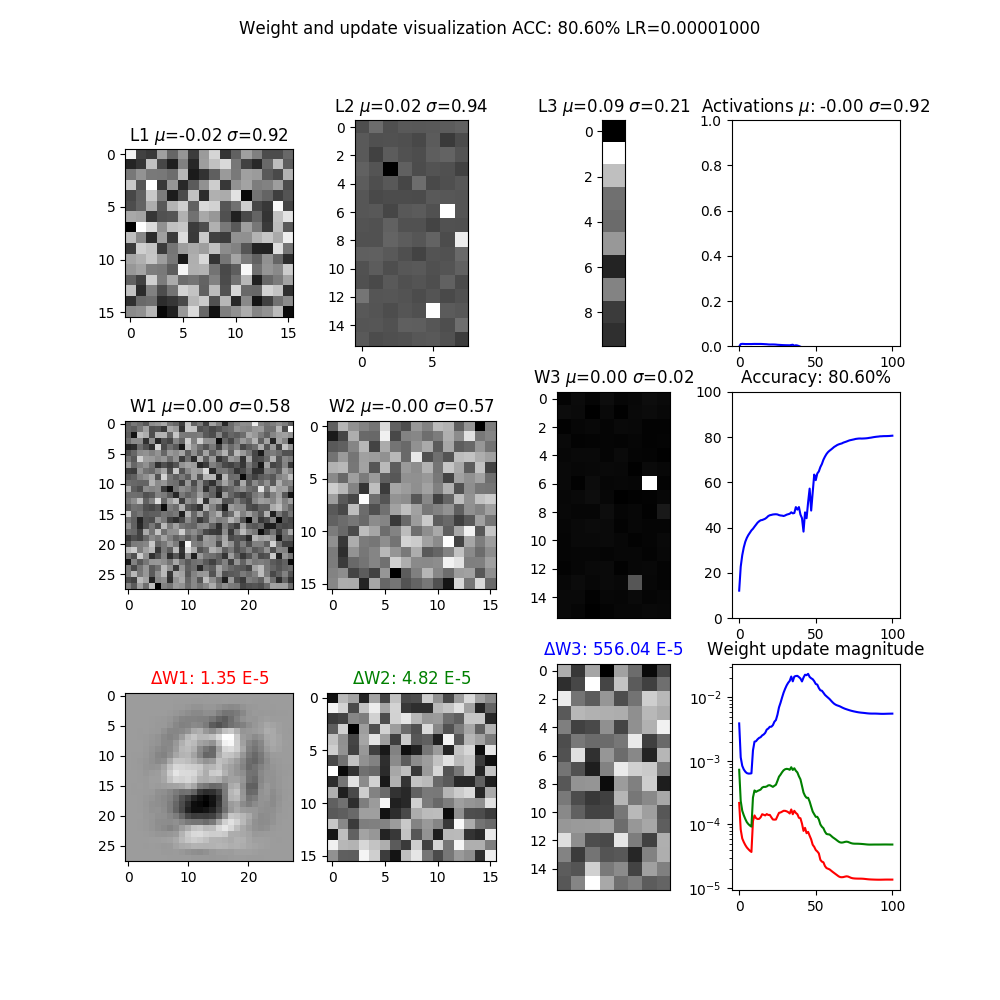
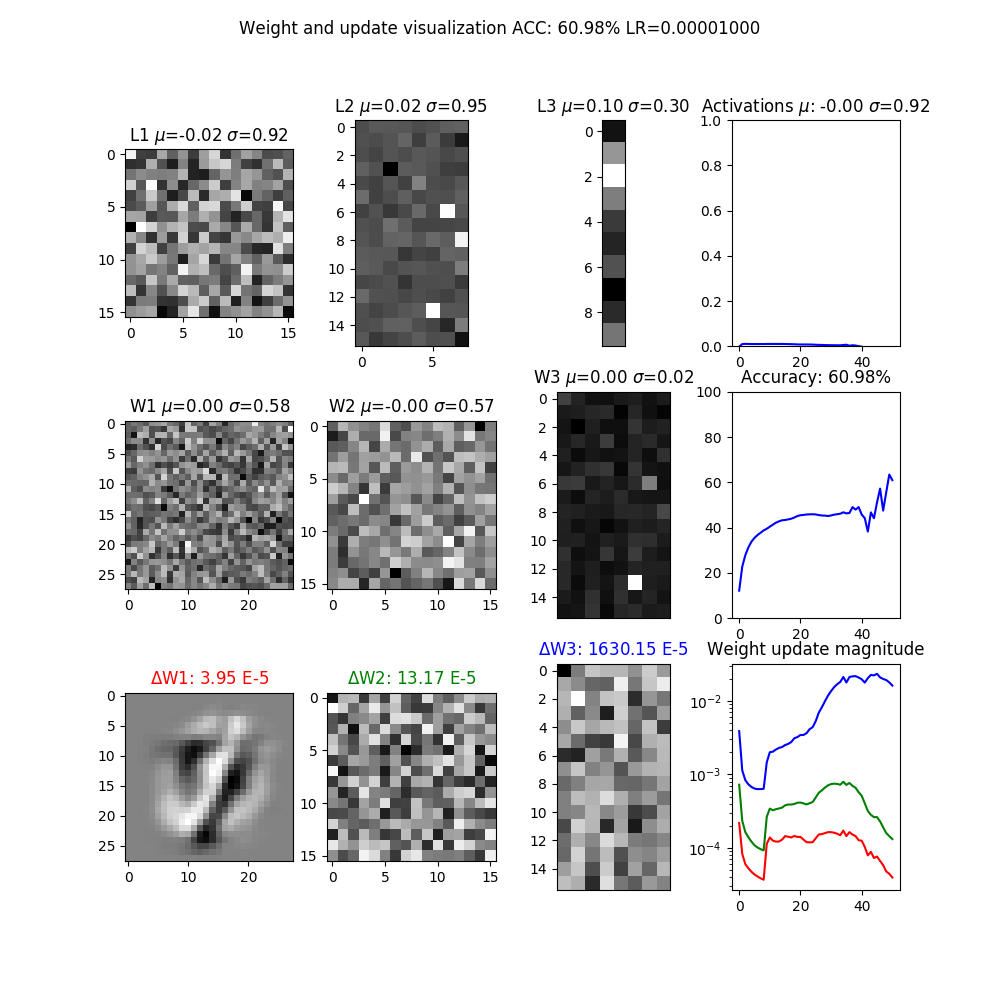
Tanh(x) solves the “zigzag” problem and the activation bias issue. Because the sigmoid function maps all inputs to the range [0, 1].

1. Activation bias: Sigmoid activations have a mean of 0.5. This bias can
2. “Zigzag” problem: Another consequence of the sigmoid’s [0, 1] range is that all the inputs to layers beyond the input layer will be positive. Because of this,

], this means that for. 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

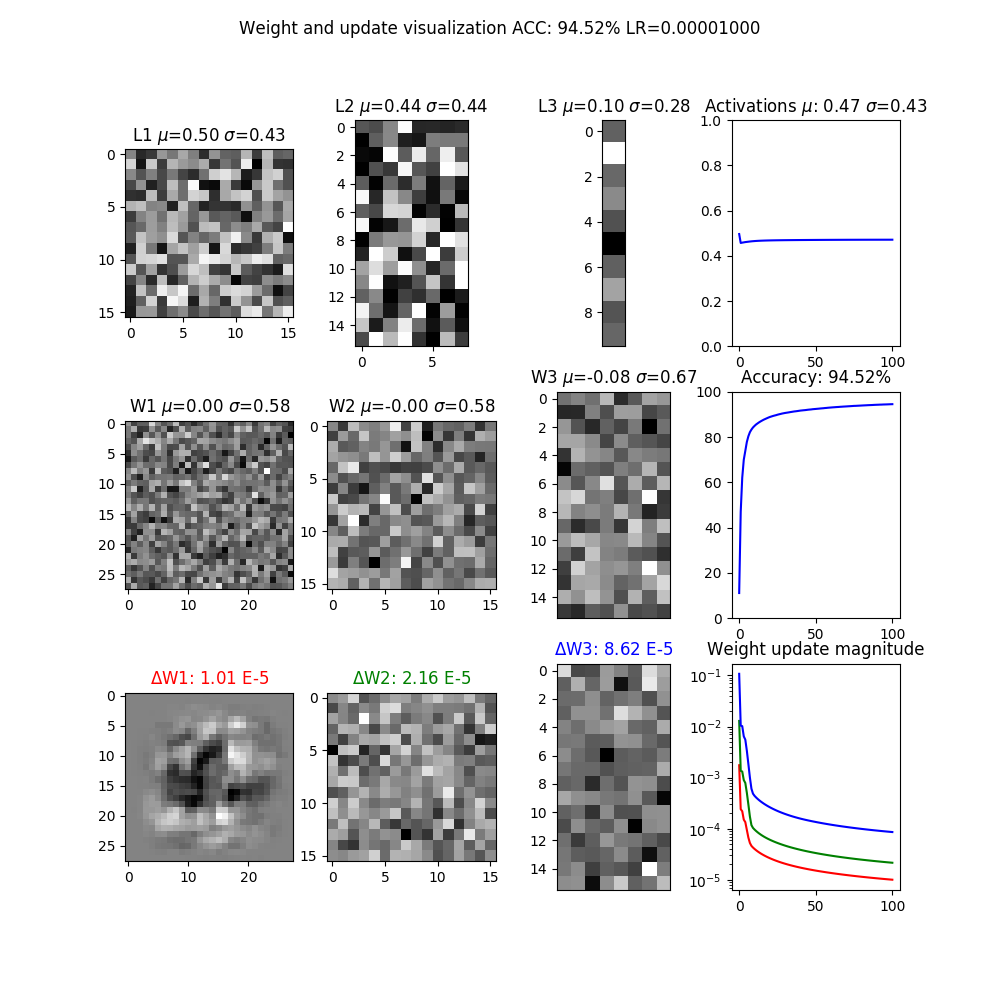
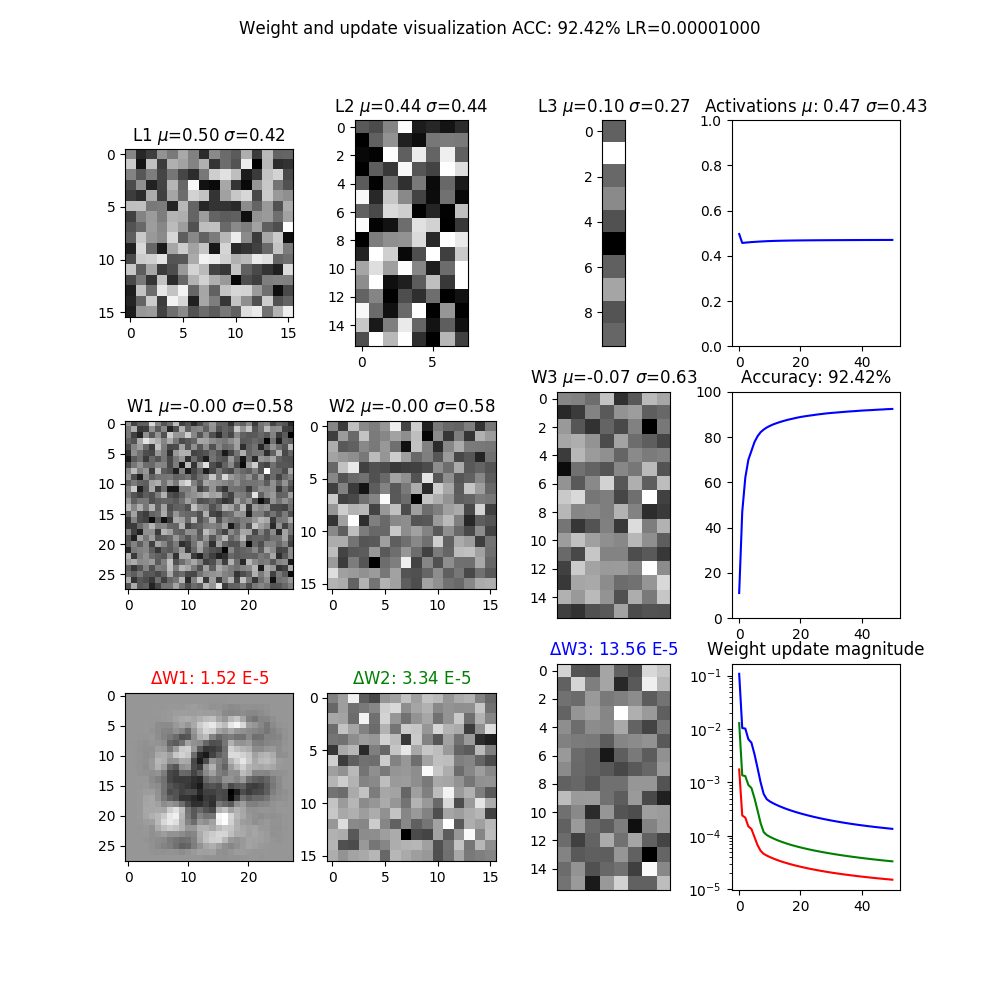
1000 epochs

500 epochs

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

Resources used:

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 does indeed speed up training by a large factor – an accuracy of around 90% is reached before the first 100 epochs.

****

1000 epochs

500 epochs

Activation: sigmoid(x)

**Cost: cross entropy**

Learning rate: 1e-5

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?

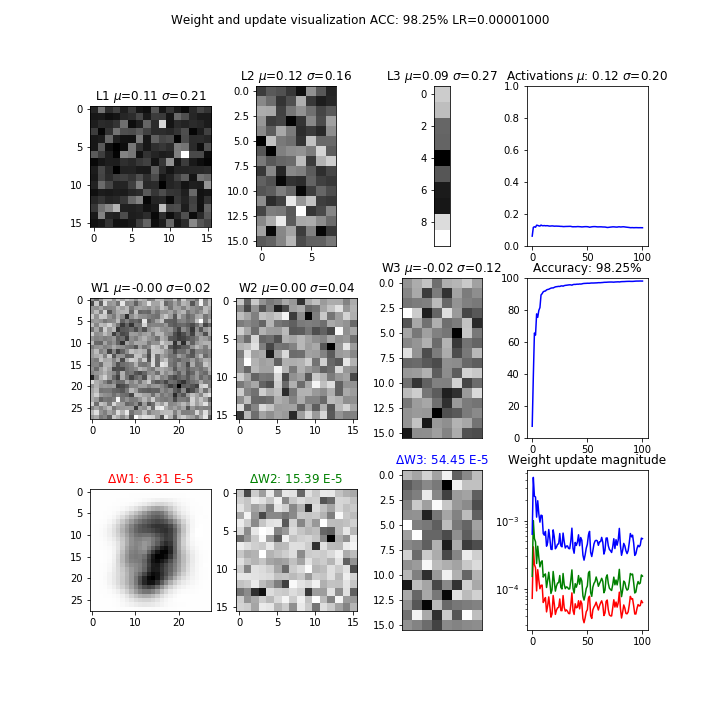
<https://stats.stackexchange.com/questions/273927/when-using-relu-is-it-normal-for-the-activations-to-go-up-at-each-layer/273933>

When implementing the cross entropy cost, 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. This is due to the fact that the previous activation functions we used mapped all values to between [0, 1] or [-1, 1], but ReLU allows large activations.

It is also worth noting that using ReLU units results in faster iterations. This is due to the simplicity of the ReLU activation compared to sigmoid and tanh, which both require exponential evaluations.

“Dead” neurons occur when a large gra.

We can fix “dead” neurons 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



**Activation: relu(x)**

Cost: MSE

Epoch: 1000

Learning rate: 1e-5

# 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?

# 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.

**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.

+