MSIA 432 Deep Learning
Spring 2020

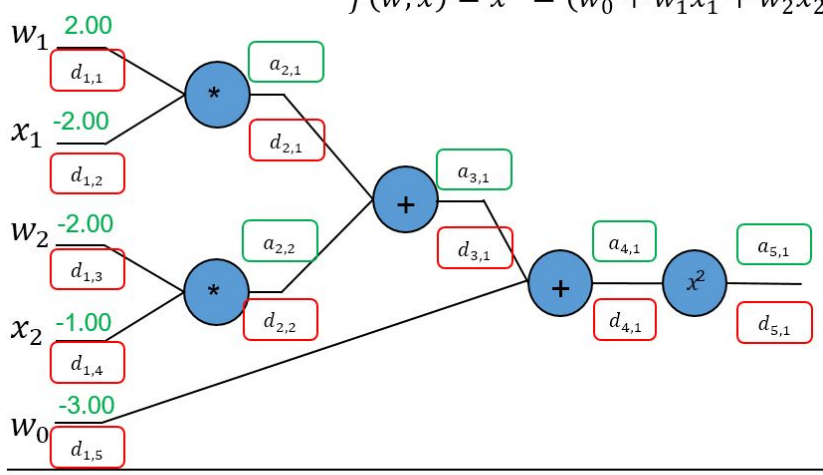
Assignment 1 (Due 5/6/2020)
Understanding Neural Networks

Name	Kristiyan Dimitrov
Discussion partner	Catherine Yang
Comments	1) I have commented out all code & functions which are irrelevant except for the ones for the final, "best run through" 2) There are gifs in the doc so I recommend you go through it on Google Docs at THIS LINK
Feedback	I'd love to hear Ellick's comment on how he would interpret some of these inputs and in general more about how he would tackle some of the challenges I faced or the mistakes I may have made below.

1. Neural Networks on paper (5 points)

Fill in the blanks below:

$$f(w,x) = x^2 = (w_0 + w_1x_1 + w_2x_2)^2$$



Directions:

- Fill in the green boxes with activations
- Fill in the partial derivatives below
- Fill in the red boxes with gradients

$f(x) = x^2 \rightarrow \frac{\partial f}{\partial x} =$	$f(x) = a^x \rightarrow \frac{\partial f}{\partial x} =$
$f_a(x) = ax \rightarrow \frac{\partial f}{\partial x} = a$	$f_c(x) = c + x \rightarrow \frac{\partial f}{\partial x} = 1$

$d_{1,1}$	20	$a_{2,1}$	- 4	$a_{3,1}$	- 2	$a_{4,1}$	- 5	$a_{5,1}$	25
$d_{1,2}$	- 20	$a_{2,2}$	2	$d_{3,1}$	-10	$d_{4,1}$	- 10	$d_{5,1}$	1
$d_{1,3}$	10	$d_{2,1}$	- 10						
$d_{1,4}$	20	$d_{2,2}$	- 10						
$d_{1,5}$	- 10								

$\frac{\partial}{\partial x}x^2$	2 * x
$\frac{\partial}{\partial x}a^x$	ln(a) * a^x

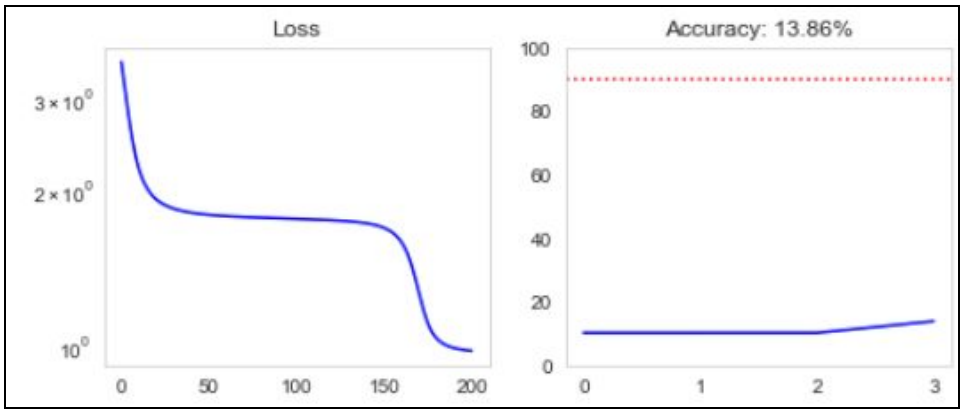
2. Neural Networks in code (12 points)

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.

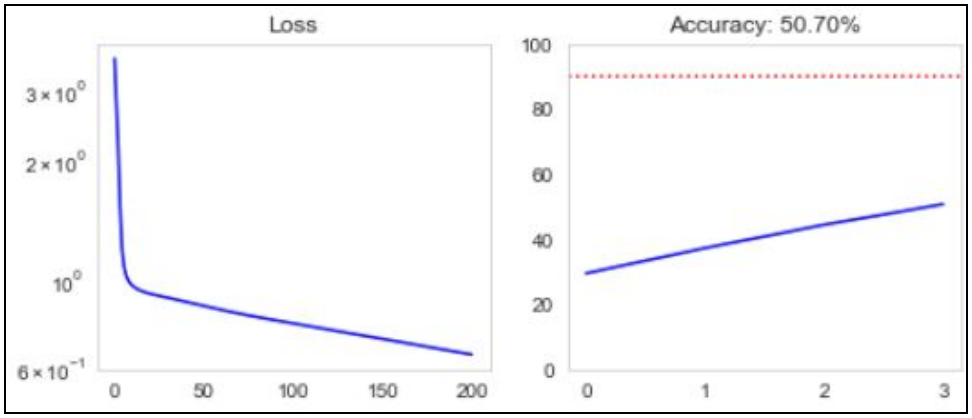
(1) **Learning rate:** Adjust the learning rate variable (lr) to try to achieve the "fastest" possible training rate. Show your loss and accuracy curves. What should you generally look for in the visualizations to ensure a "good" learning rate?

I tested 5 different learning rates. They are the base value of 1e-5 times 1/10, 1/2, 1, 2, 10.
Loss & Accuracy plots:

Learning rate = 1e-5 * 1/10

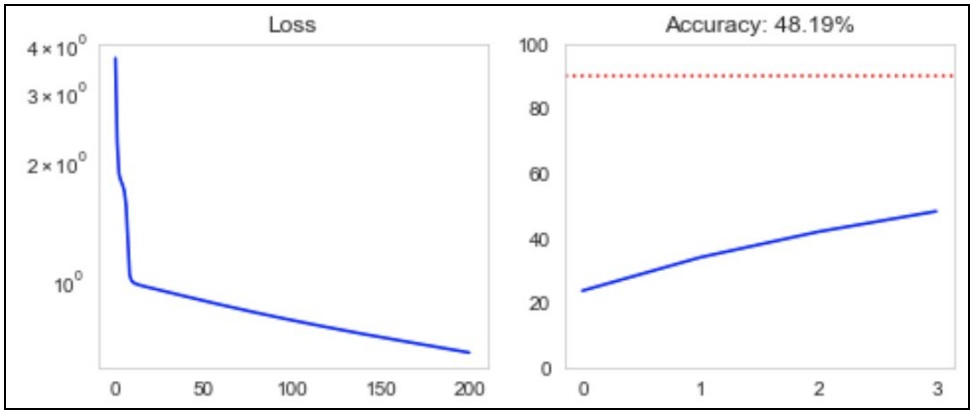


Learning rate = 1e-5 * 1/2

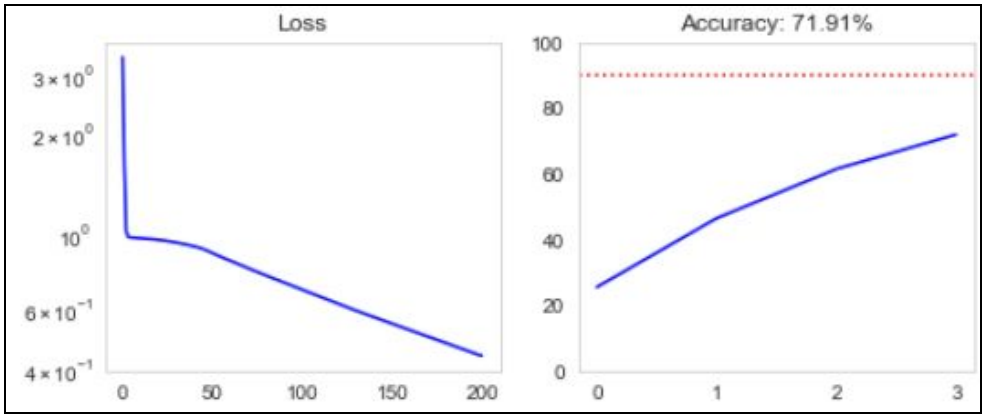


Learning rate = 1e-5

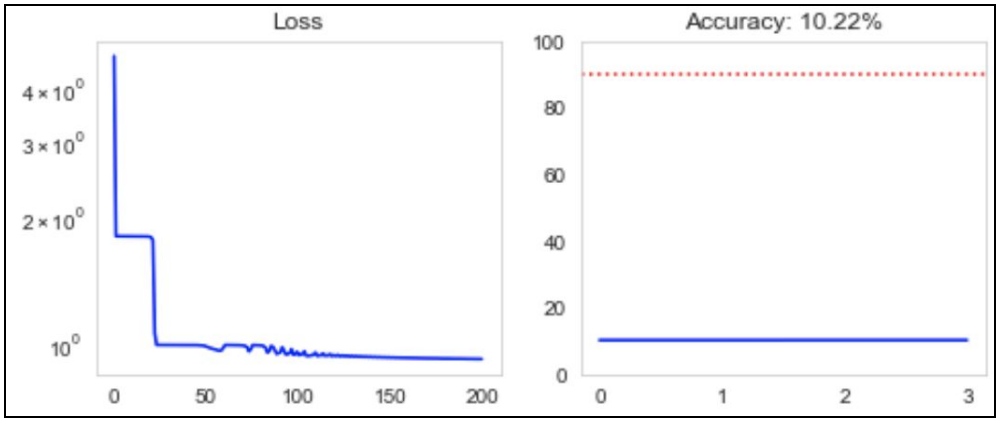
MSIA 432 Deep Learning
Spring 2020



Learning rate = 1e-5 * 2



Learning rate = 1e-5 * 10

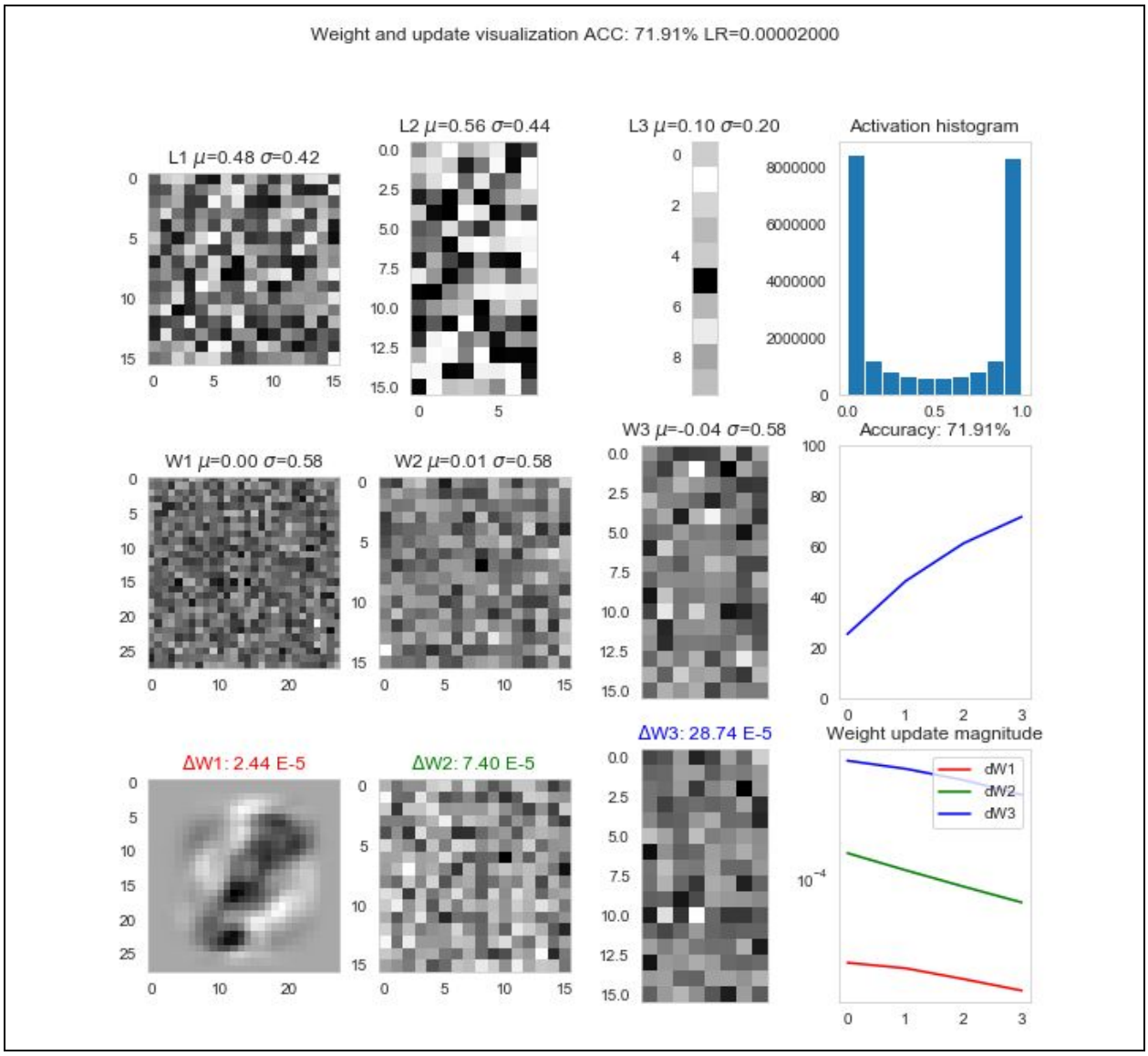


Observations:

- Both the very high loss and the small loss produced very poor accuracies. Therefore, a more 'goldilocks' value is best.

- A very low learning rate has a loss curve which decreases more slowly. This is reasonable, after all the learning rate determines the size of the steps we take along the surface of the loss function.
- Within the 'goldilocks' values (1/2 , 1, and 2) I see that a higher learning rate makes the loss curve decrease faster
- A very high learning rate, however, does not produce the same decrease in loss and plateaus early

Let's look at the visualizations for the best learning rate (1e-5 * 2):



It's important to keep in mind that all these visualizations show the *average* activation of the corresponding node.

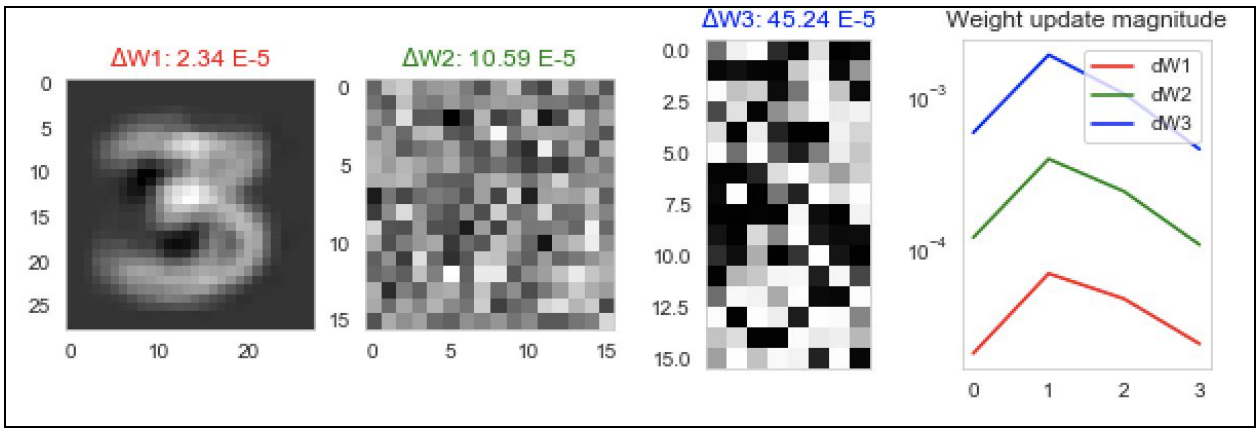
Therefore, for example, the rightmost graph of the top row shows us the mean activation of each output node i.e. what the prediction of the NN is on average. Since the MNIST dataset is fairly well balanced,

we'd hope to get a fairly monotone prediction in that output layer i.e. the NN is not failing to predict for some of the digits i.e. some of the output nodes are not "dead" i.e. not predicted, regardless of the input.

Similarly, if we look at the second row of the initialized weights, we would hope that there is no "pocket" of weights which have been initialized unusually high.

Finally, I would look at the bottom row of delta W to evaluate if the NN has converged or not. Ideally, there wouldn't be any bright pixels close to the borders, because the digits generally tend to be in the middle of the image.

I investigated what happened with a high learning rate. It looks like the NN got stuck in a bad local minimum where it's simply guessing only 3, regardless of input.



(2) 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.

I tested 5 different values for k: 1/10, 1/2, 1, 2, 10, while keeping the default learning rate: 1e-5
Note that I modified the sigmoid(), forward_pass() & backward_pass() function to correctly include the k parameter:

```
# Define basic sigmoid activation
def sigmoid(x, k): return 1.0/(1.0 + np.e**-(k*x))
```

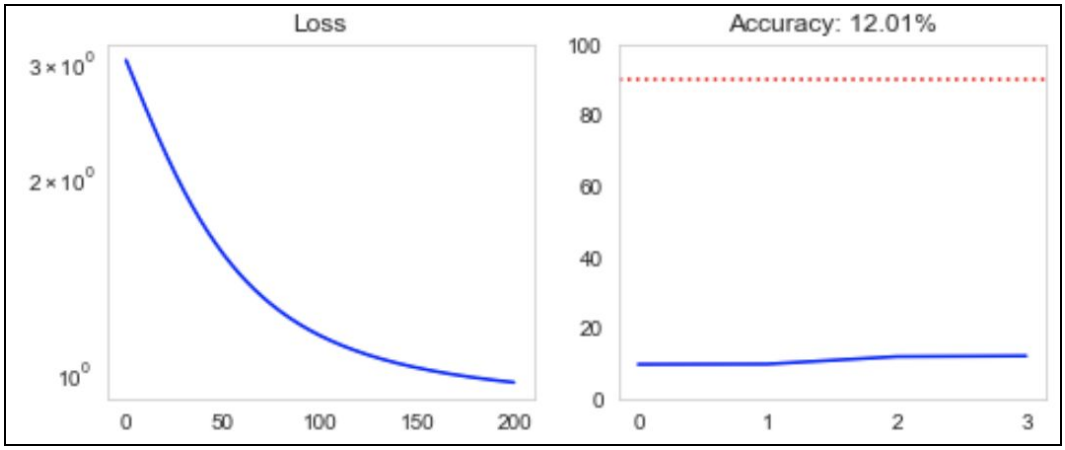
```python
def forward_pass(X, W1, W2, W3, k): # Adding
    L1 = sigmoid(W1.dot(X), k)
    L2 = sigmoid(W2.dot(L1), k)
    L3 = sigmoid(W3.dot(L2), k)
    return L1, L2, L3

def backward_pass(L1, L2, L3, W1, W2, W3, k):
    dW3 = (L3 - T) * L3*(1 - L3) * k
    dW2 = W3.T.dot(dW3)*(L2*(1-L2)) * k
    dW1 = W2.T.dot(dW2)*(L1*(1-L1)) * k
    return dW1, dW2, dW3
```

So, in the main loop of my code, I've also included this configuration parameter

```python
# Problem 2, Part (2) - PARAMETERS
k = 1

for i in range(200):
    L1, L2, L3     = forward_pass(X, W1, W2, W3, k)
    dW1, dW2, dW3 = backward_pass(L1, L2, L3, W1, W2, W3, k)
    W1, W2, W3     = update_weights(lr, W1, W2, W3, dW1, dW2, dW3, X, L1, L2)
```

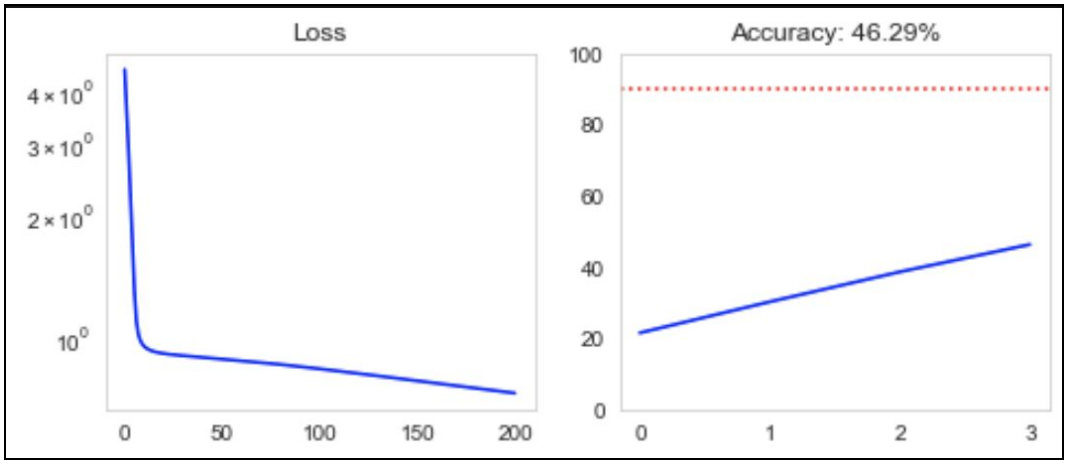
Loss & Accuracy plots:

k=1/10

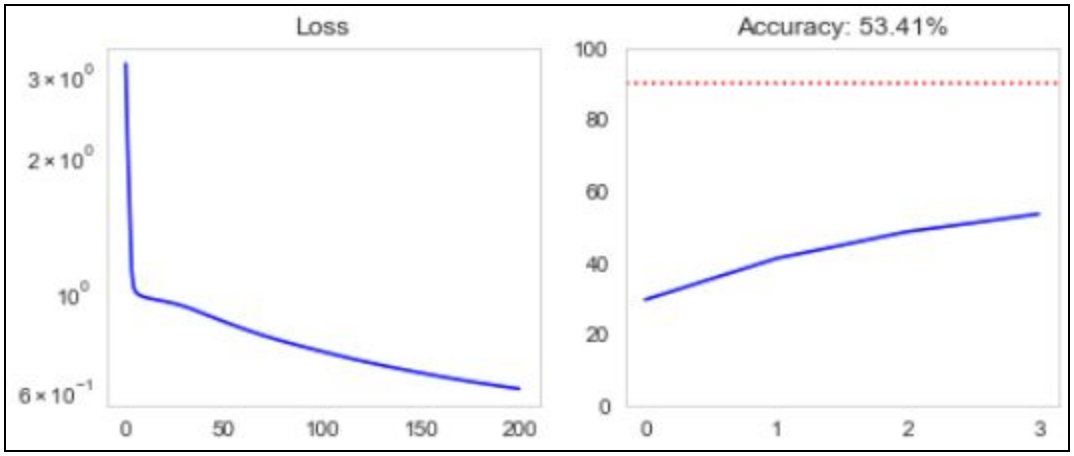


k=1/2

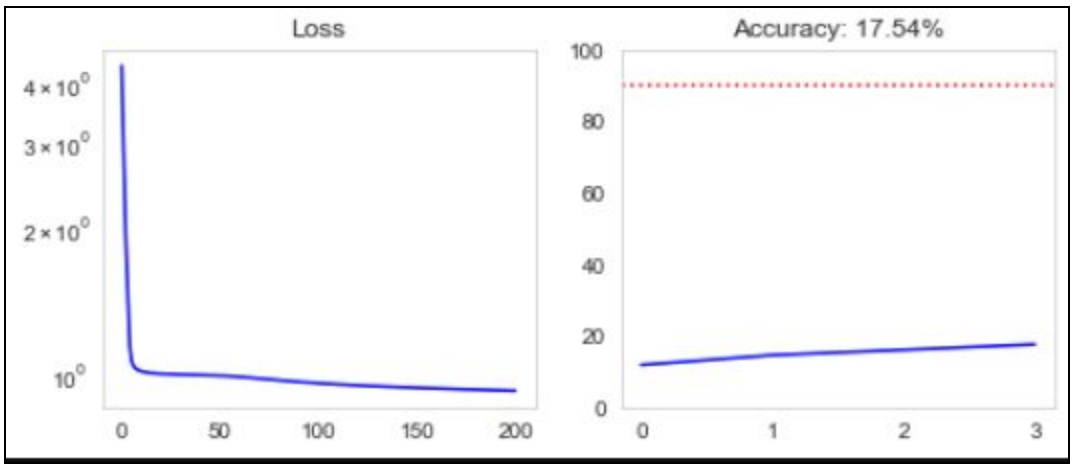
MSIA 432 Deep Learning
Spring 2020



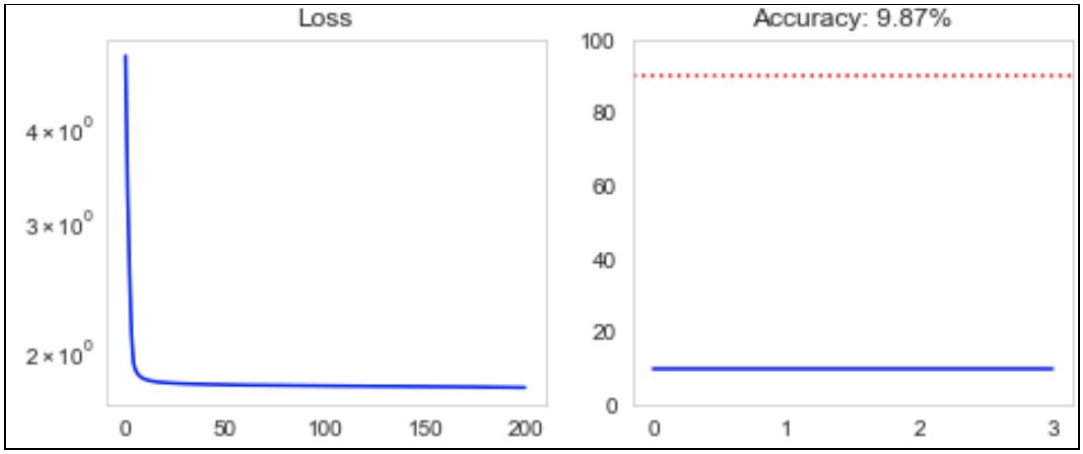
k=1



k=2

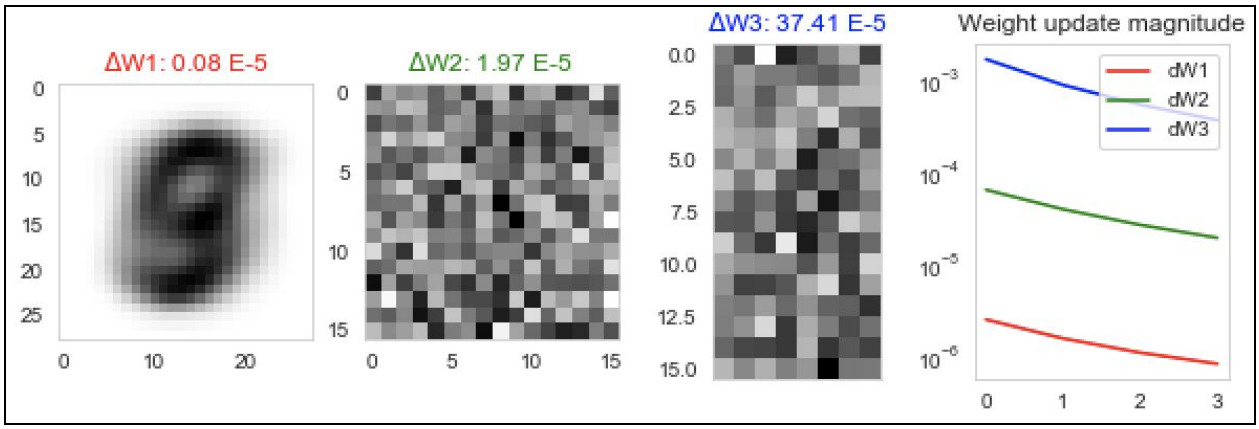


k=10



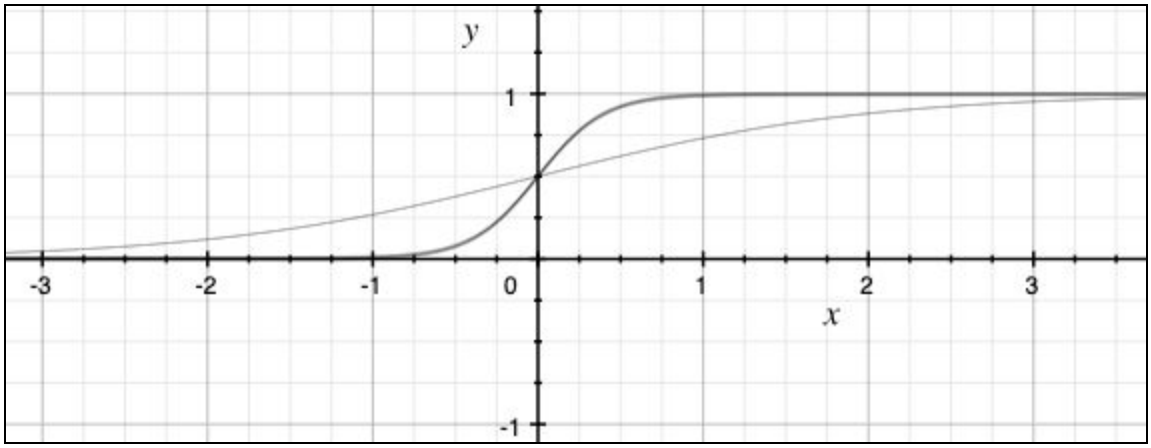
Just as with the learning rate, a more "in the middle" value appears to be better. Note: For k=10 I got an overflow warning, which is concerning. In fact, I observe that the accuracy is less than 10%, i.e. the NN wasn't able to perform better than random guess. The loss & accuracy graphs suggest that this won't improve.

When k=1/10, I see that the NN just guesses 9 every time. I think this is a local minimum of the cost function it has gotten stuck in.



In terms of investigation, I did try k=5, but the results were the same as with k=10 i.e. Accuracy ~10%.

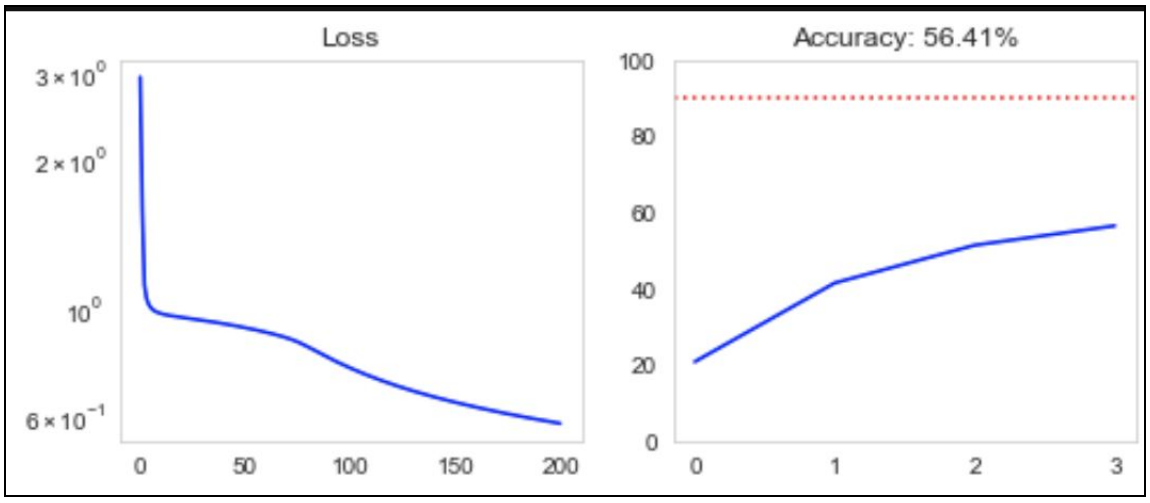
In general, the influence of k is:
If x > 0, then as k --> infinity, the value of the activation function --> 1
if x < 0, then as k --> infinity, the value of the activation function --> 0
In other words, the higher the k, the more our sigmoid neuron behaves like a perceptron.
i.e. the sigmoid curve looks more and more like a step function
This is problematic. The main reason we use a sigmoid function instead of a perceptron is that we need a continuous loss function for our gradient descent. With perceptrons, we run into discontinuity problems.
Here is a graph for k = 1 and k = 5 (the thicker line is k=5)



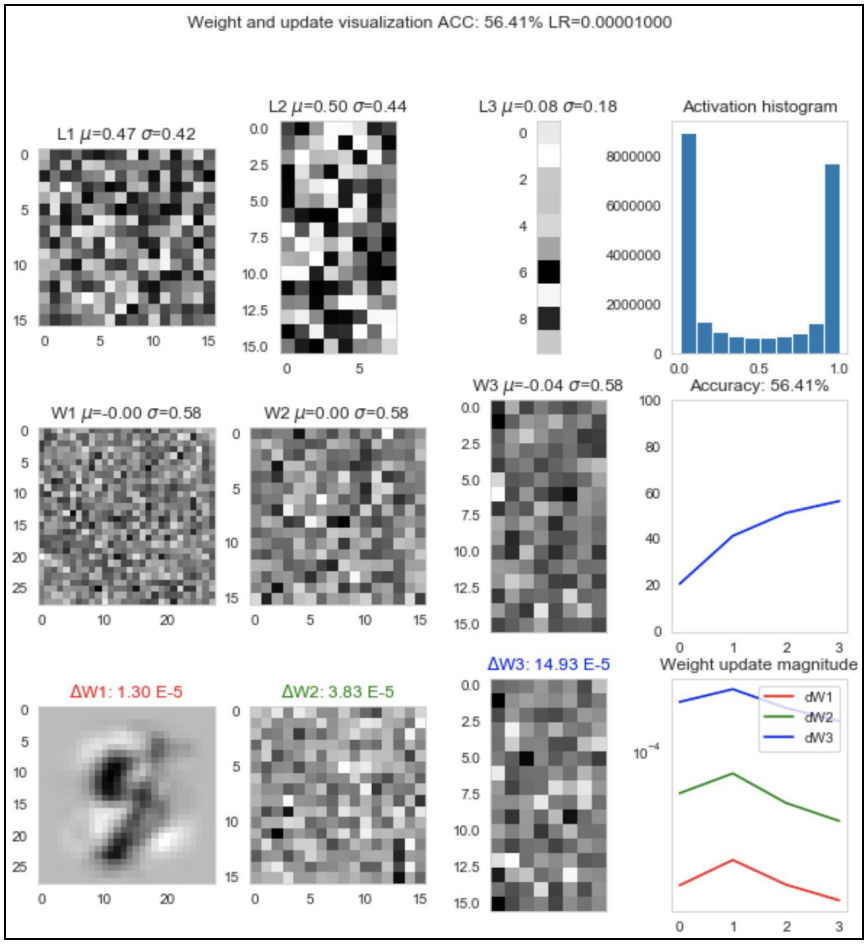
(3) Initialization: In the sample code, the weights W1, W2, W3 are initialized uniformly from -1 to 1. Experiment with various kinds of initialization and report your findings. Justify why your proposed initialization is better than the default initialization. Show up to 5 plots. Hint: how do the visualizations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

All the above experiments were done with a random initialization of all weights from a uniform distribution U[0, 1] and scaled to the [-1, 1] range.

I repeated the training one more time (with base learning rate and k=1) to have a reference point.



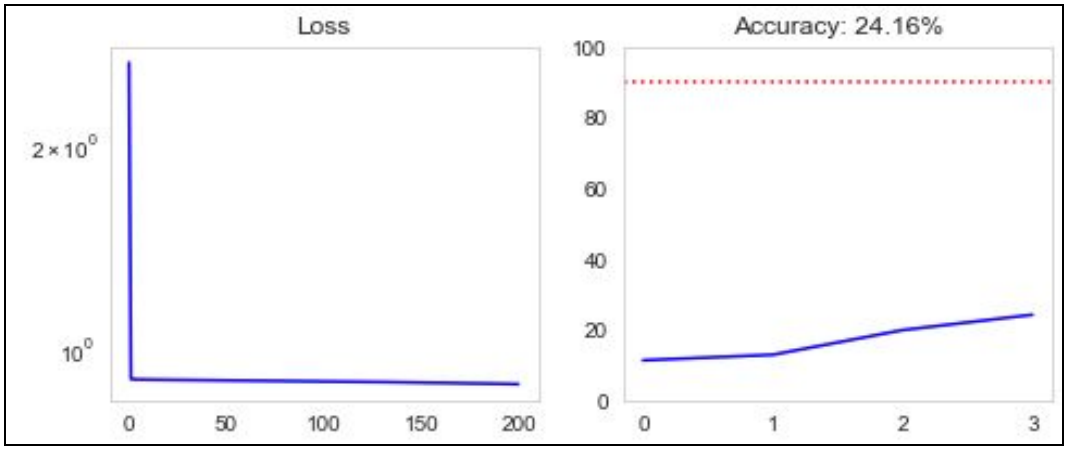
Perhaps with more than 200 epochs it could have achieved higher accuracy, but it doesn't seem like it would reach 90% accuracy. The problem with the uniform initialization of the weights is that very high (close to 1) or very low (close to -1) values for a given weight will give a value for the derivative of the sigmoid which is very small. As a result, the gradient for a specific node might "die" i.e. turn black and become irrelevant.

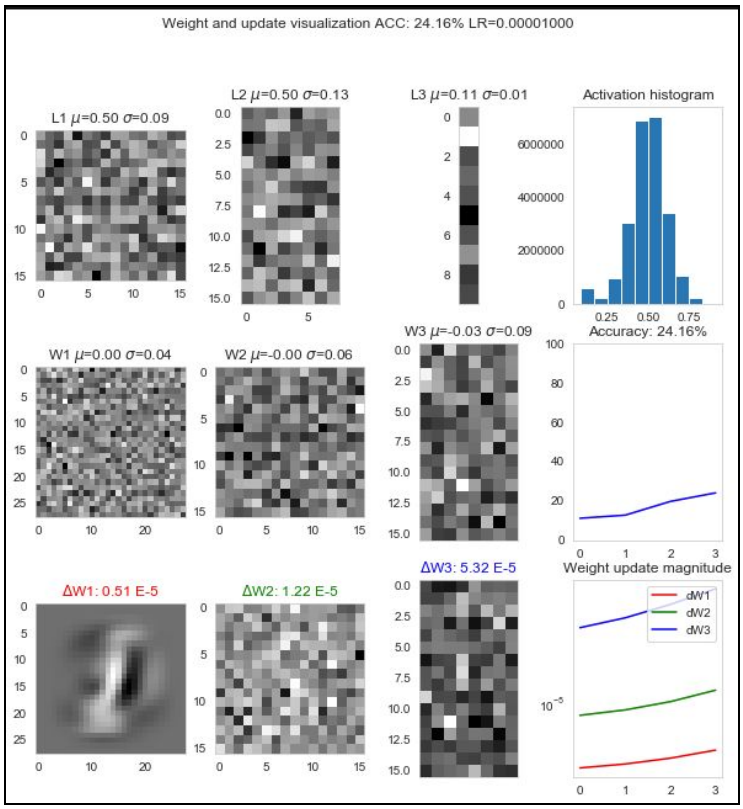


As a next step, I tried sampling from a normal Distribution N[0, 1/ sqrt(n)] Why divide by sqrt(n), where n is the size of the layer? Because, as discussed in class 4, the variance for our sigmoid will "overshoot" i.e. be larger than ideal by a factor of sqrt(n). By dividing by the same value we are correcting for this.

```
# # Initialize from Normal Distribution with N(0, 1/sqrt(n)) where n is the number of the input vector
W1 = np.random.randn(784, 256).astype('float32').T * 1/np.sqrt(784)
W2 = np.random.randn(256, 128).astype('float32').T * 1/np.sqrt(256)
W3 = np.random.randn(128,  10).astype('float32').T * 1/np.sqrt(128)
```

These are the results:





The activation histogram suggests the initialization worked. But the NN is making predictions of mostly "1's". Perhaps this initialization would work better with a different activation function or a larger learning rate i.e. we might not be allowing the NN to "express itself" with such a low learning rate i.e. it needs more iterations or larger steps.

Finally, I found an article, which recommends **He Initialization** (Link). It's basically the same as the previous one, but using 2 instead of 1

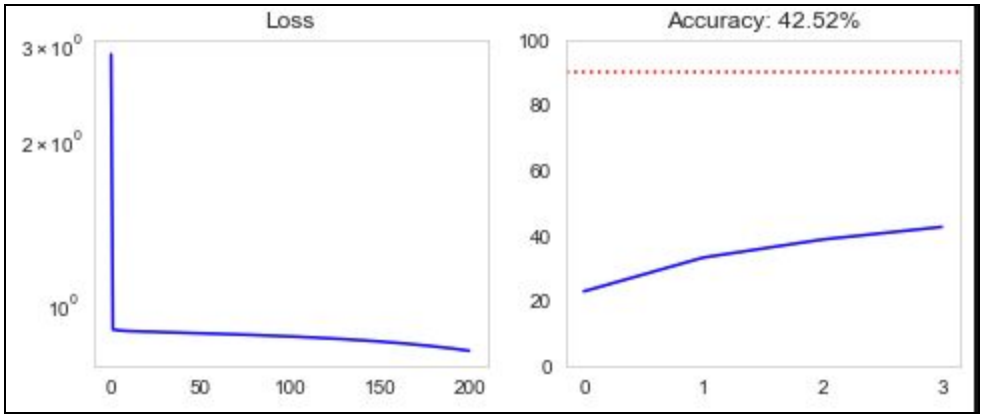
$$\sqrt{\frac{2}{size^{[l-1]}}}$$

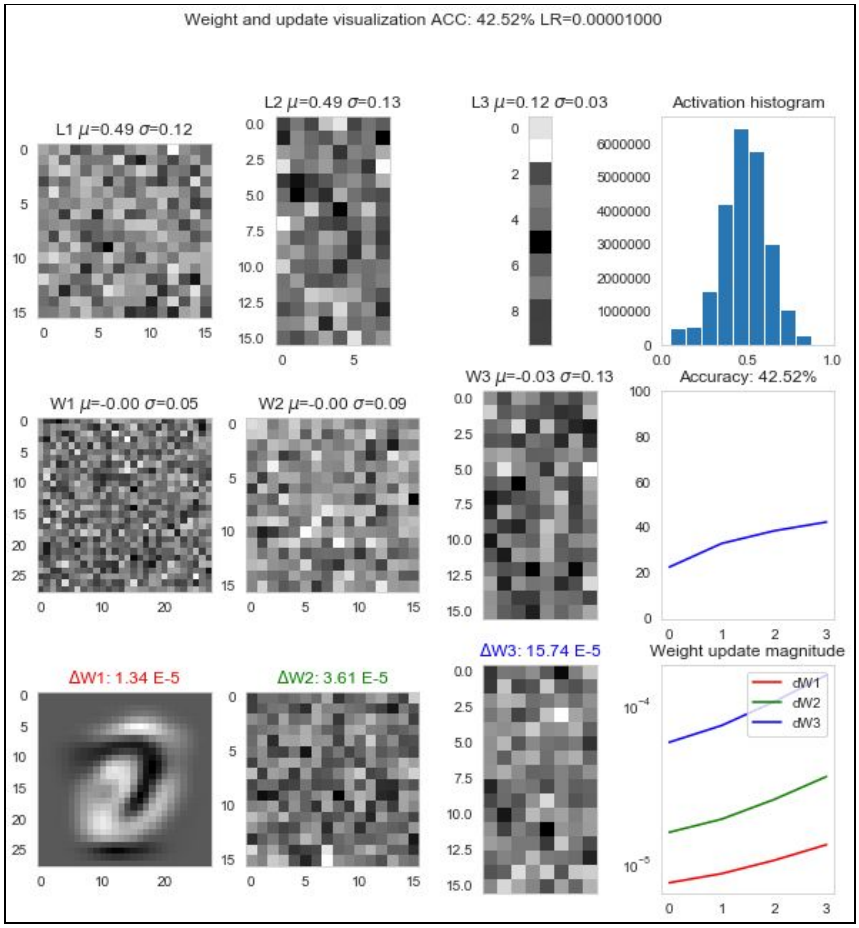
$$W^{[l]} = np.random.randn(size\_l, size\_l\text{-}1) * np.sqrt(2/size\_l\text{-}1)$$

This is how I coded it:

```
# # Initialize with He Initialization i.e. with N(0, 2/sqrt(n)): https:
W1 = np.random.randn(784, 256).astype('float32').T * np.sqrt(2/784)
W2 = np.random.randn(256, 128).astype('float32').T * np.sqrt(2/256)
W3 = np.random.randn(128,  10).astype('float32').T * np.sqrt(2/128)
```

And these are the results:





The Accuracy is higher than with the previous initialization and the gradients are moving in the right direction (up) so I would expect the NN to improve its performance if I increase the number of epochs or the learning rate.

3. Optimization in code (16 points)

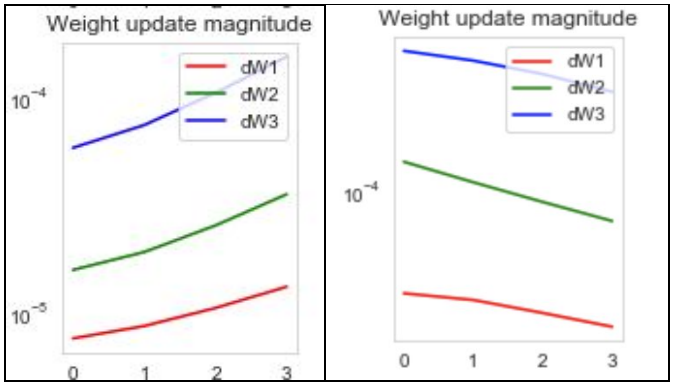
Using the example code from lecture 3, demonstrate your understanding of the principles in lecture 4 by doing the following (submit your code for these in the notebook):

(1) **Activations and Gradients:** Examine the activations and gradients visualized during training. Justify why 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.

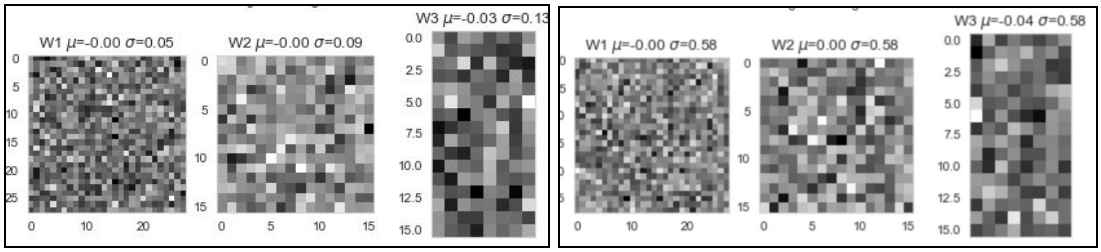
For example, in the last example with He initialization of the weights, I observe that the standard deviations of the weights and activations are fairly low (sigma equal to .12, .13, .03 for the first row of activations and .05, .09, and .13 for the second row of weights.

More specifically, they are fairly low, if I compare with the standard deviations from the best learning rate rate (1e-5 * 2): those are .42, .44, and .2 for the Activations and .58, .58, .58 for the weights. This means that in the first instance with the He initialization standard deviations, there will be fewer cases where the gradient contribution of the sigmoid function will be very small. As a result, our Gradients shouldn't shrink as much. In fact, for He Initialization, the Gradient appears to be increasing.

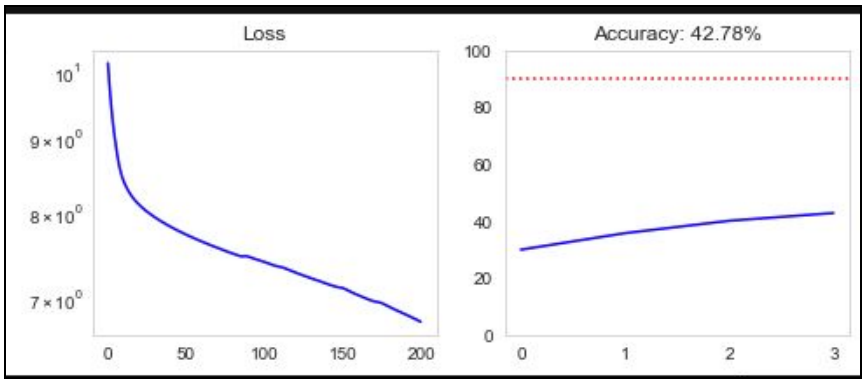


Gradient for He Init. Gradient for Uniform Init.

All this translates to fewer "dead" i.e. completely black nodes with He Initialization as opposed to the Uniform Initialization. Overall, the He Init. images are a little "brighter"

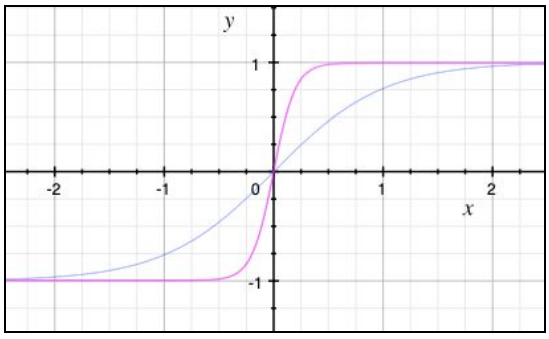


Weights for He Init. Weights for Uniform Init.



(2) **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)?

Here is what the graph of tanh looks like; Note that the blue one is with k = 1 and the magenta one with k = 5



I've created the function in Python and modified the forward pass and backward pass functions to reflect the switch to tanh (note the change in the backward_pass, which now uses the derivative of tanh (1-tanh2(x)). Note that I will revert back to the default weight initialization so I can more clearly see the impact of using tanh over sigmoid

```
def tanh(x, k=1):
    exponent = np.e**(2*k*x)
    return (exponent - 1) / (exponent + 1)
```

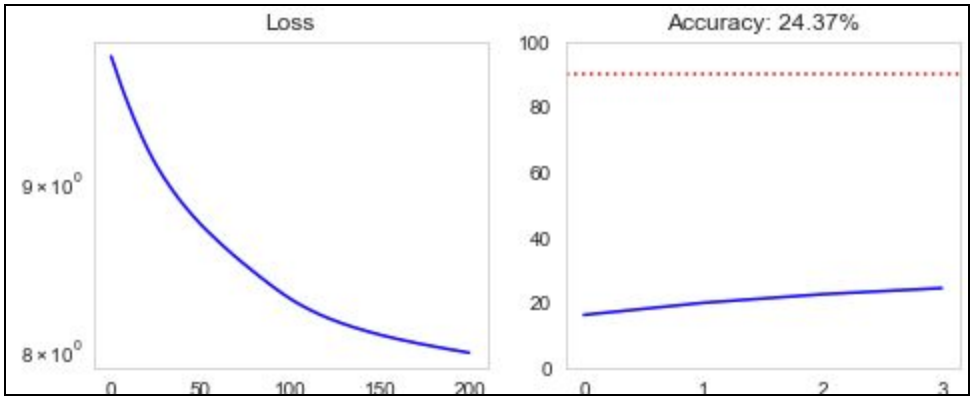
```
# Forward Pass Function for TANH activation function
def forward_pass(X, W1, W2, W3, k):
    L1 = tanh(W1.dot(X), k)
    L2 = tanh(W2.dot(L1), k)
    L3 = tanh(W3.dot(L2), k)
    return L1, L2, L3
```

```
# Backward Pass Function for TANH activation function
def backward_pass(L1, L2, L3, W1, W2, W3, k):
    dW3 = (L3 - T) * L3*(1-L3**2) * k # NOTE!: This line
    dW2 = W3.T.dot(dW3)*(1-L2**2) * k # NOTE!: This line
    dW1 = W2.T.dot(dW2)*(1-L1**2) * k # NOTE!: This line
    return dW1, dW2, dW3
```

The issue with the sigmoid function is that it is biased and not centered around 0. Tanh, alleviates that issue, because the output of tanh is in [-1, 1]. It doesn't, however, prevent the gradient from vanishing, because for input values with large absolute absolute values, the derivative (slope) of tanh is close to 0.

Note that for the below results I had to decrease the learning rate to 1e-6.

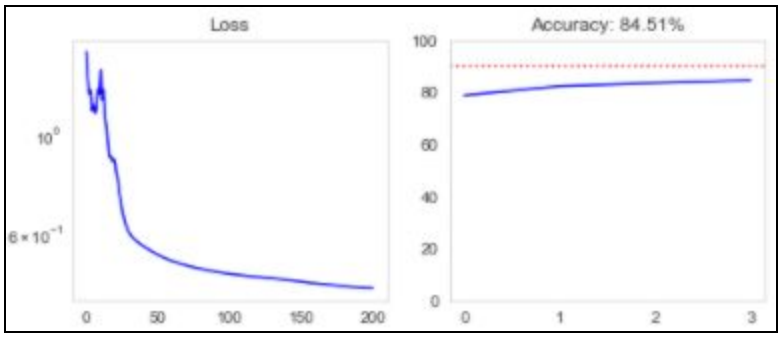
The results are quite poor, although still better than the very first graph in this homework where I had an accuracy of 13.86%:



So, tanh managed to, relatively, improve the results dramatically. Next, I tried to use tanh with the normal initialization of the weights

```
# # Initialize from Normal Distribution with N(0, 1/sqrt(n)) where n i
W1 = np.random.randn(784, 256).astype('float32').T * 1/np.sqrt(784)
W2 = np.random.randn(256, 128).astype('float32').T * 1/np.sqrt(256)
W3 = np.random.randn(128,  10).astype('float32').T * 1/np.sqrt(128)
```

The results were the best so far!



(3) **Cross Entropy:** Implement cross entropy. Show plots of how "Cross-entropy" improves training.

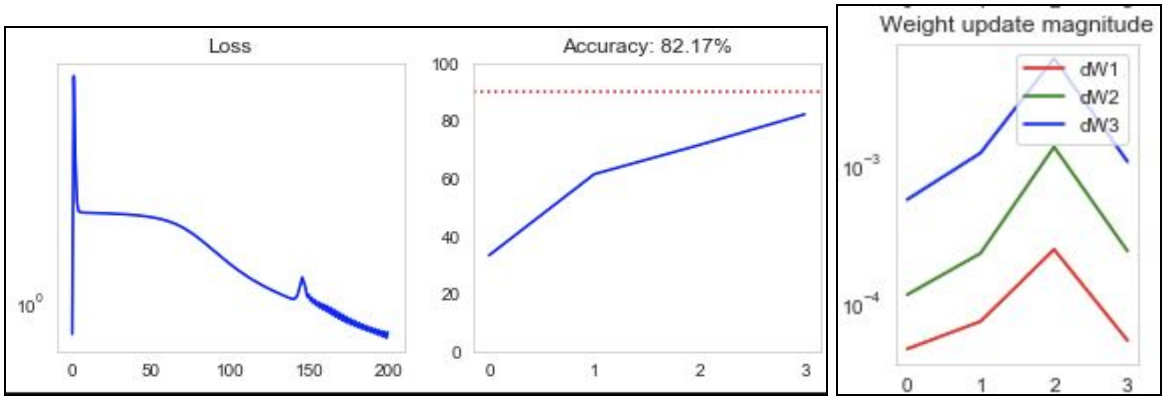
To implement cross-entropy as a loss function I had to do 2 things. First, I had to modify the calculate_loss() function:

```
# Cross-Entropy (Deviance?) loss function (NON BINARY CASE)
def calculate_loss():
    global losses
    loss =  -np.sum(np.nan_to_num(T*np.log(L3)))/len(T.T)
    losses.append(loss)
    #print("[%04d] MSE Loss: %0.6f" % (i, loss))
```

And second, I had to modify the gradient calculation to reflect the new derivative. This demonstrates the main benefit of cross-entropy, because we are not multiplying by the derivative of the sigmoidal function.

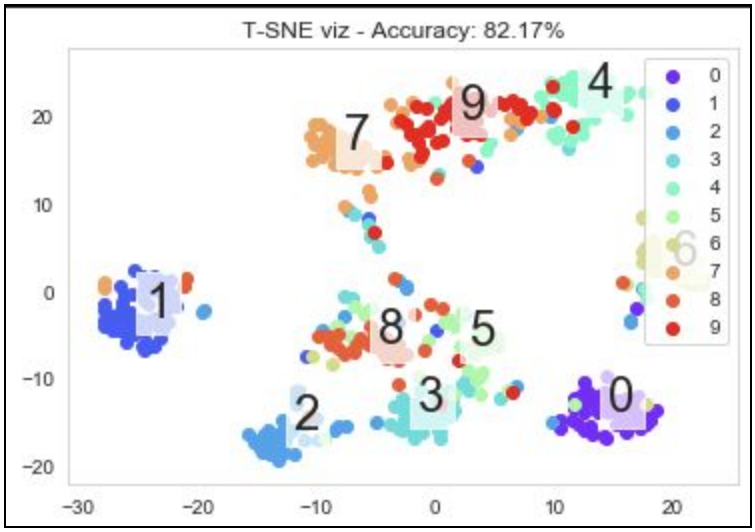
```python
# Backward Pass Function for SIGMOID activation function with Cross_Entropy as the loss function
def backward_pass(L1, L2, L3, W1, W2, W3, k):
    dW3 = (L3 - T) * k
    dW2 = W3.T.dot(dW3)*(L2*(1-L2)) * k
    dW1 = W2.T.dot(dW2)*(L1*(1-L1)) * k
    return dW1, dW2, dW3
```

The results are below and note I had to switch back to using sigmoidal activation with learning rate 1e-5:



I observe that the gradients are increasing. This makes sense, since the nature of the cross_entropy function is that the more mistakes the NN makes, the higher the gradient will be. So, when it begins initially and makes a lot of mistakes, it will increase the gradient. As it starts making fewer mistakes, the gradient will not increase as much (and maybe even decrease?).

The T-SNE plot also shows decent separation between the classes:



(4) **ReLU/6:** 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? Also implement ReLU6 and compare.

This is how I implemented the ReLU function and it's derivative:

MSIA 432 Deep Learning
Spring 2020

```python
# ReLU Activation function
def ReLU(x, k=1):
    return np.maximum(0.01, x*k)

# ReLU Derivative function
def reluDerivative(x):
    matrix = x
    matrix[matrix<=0] = 0
#     matrix[matrix>0] = 1
    return matrix
```

Note that originally the line "matrix[matrix>0] = 1" was not commented out. In that scenario, however, my output was completely nonsensical. Upon further reflection I decided that I was ignoring the "ds" in the derivative as specified in the slides from class. The effect of commenting out that line is that whatever values in "s" are positive will remain so, while whatever values are negative will become 0 (or .01 in my case).

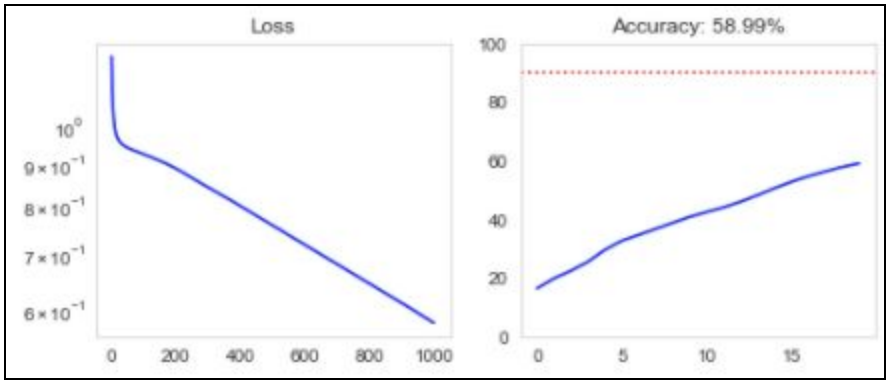
- $f(s) = \max(0, s)$
- $$\frac{\partial}{\partial s} f = \begin{cases} 0, & s \leq 0 \\ 1 * ds, & s > 0 \end{cases}$$

Here is how I modified my Forward Pass & Backward Pass functions:

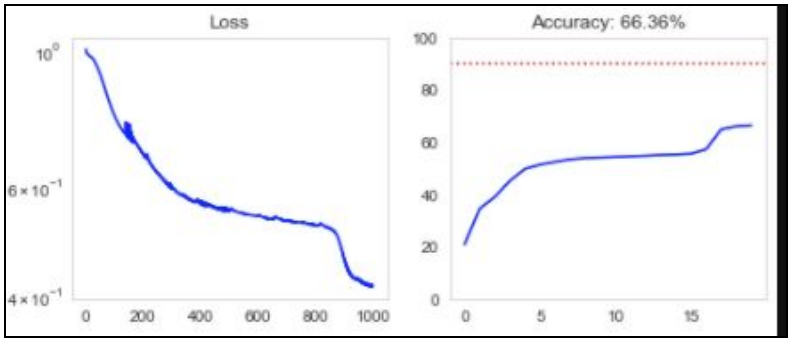
```python
# Forward Pass Function for ReLU Activation
def forward_pass(X, W1, W2, W3, k):
    L1 = ReLU(W1.dot(X), k)
    L2 = ReLU(W2.dot(L1), k)
    L3 = ReLU(W3.dot(L2), k)
    return L1, L2, L3
```

```python
# Backward Pass Function for ReLU Activation function
def backward_pass(L1, L2, L3, W1, W2, W3, k):
    dW3 = (L3 - T) * reluDerivative(L3) * k
    dW2 = W3.T.dot(dW3) * reluDerivative(L2) * k
    dW1 = W2.T.dot(dW2) * reluDerivative(L1) * k
    return dW1, dW2, dW3
```

Finally I used 1000 iterations and this is the result I got:



This is what the results look like after 1000 iterations when the clipping of the ReLU derivative is .0001 instead of .01:




```python
# ReLU Activation function
def ReLU(x, k=1):
    return np.maximum(0.0001, x*k)

# ReLU Derivative function
def reluDerivative(x):
    matrix = x
    matrix[matrix<=0] = 0
#     matrix[matrix>0] = 1
    return matrix
```

I conclude that the more conservative clipping led to better results.

Now I move on to the ReLU6 function:

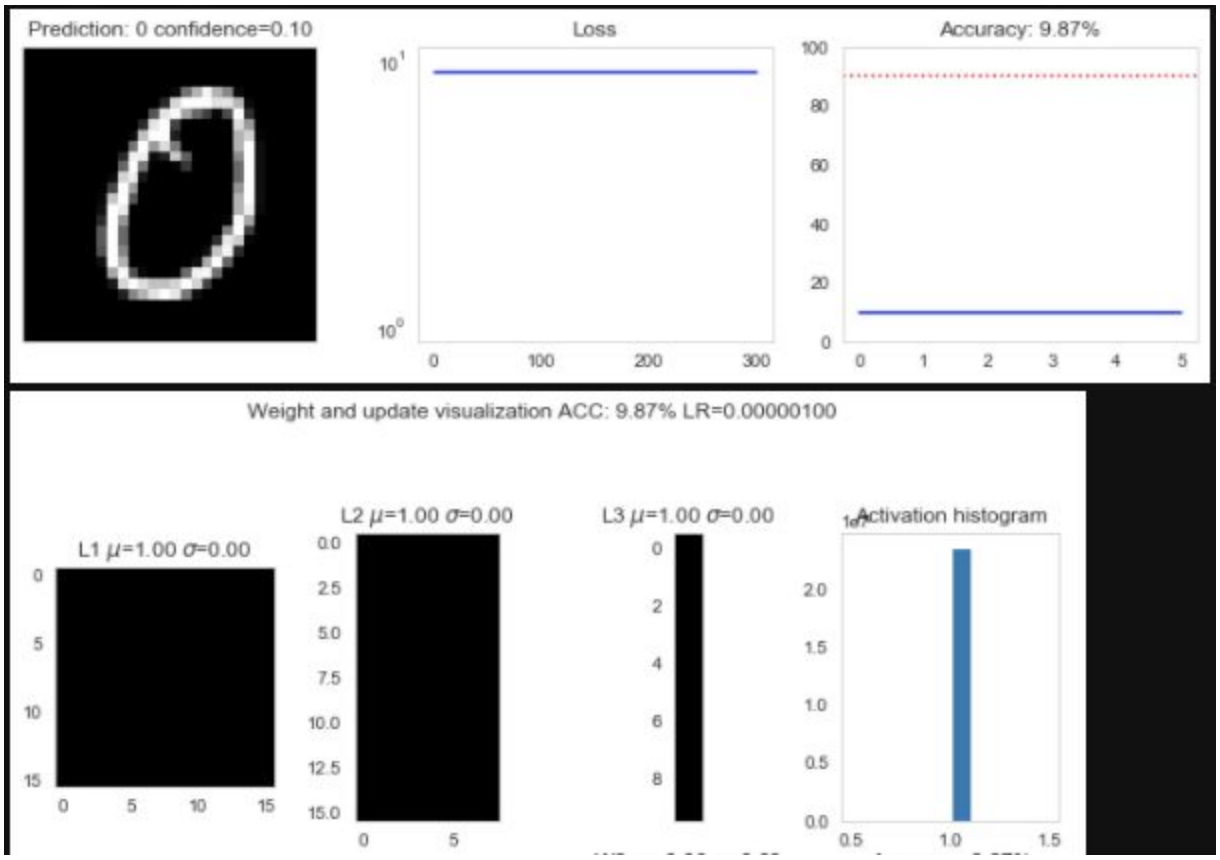
```python
# ReLU_6 activation function
def ReLU_6(x, k=1):
    return np.minimum(np.maximum(0.01, x*k), 6)

# ReLU Derivative function
def ReLU_6_Derivative(x):
    matrix = x
    matrix[np.where((matrix<=0) & (matrix >=6))] = 0
    matrix[np.where(matrix != 0)] = 1
    return matrix
```

I've modified my forward pass & backward pass functions accordingly:

```python
# Forward Pass Function for ReLU_6 Activation
def forward_pass(X, W1, W2, W3, k):
    L1 = ReLU_6(W1.dot(X), k)
    L2 = ReLU_6(W2.dot(L1), k)
    L3 = ReLU_6(W3.dot(L2), k)
    return L1, L2, L3
```

```python
# Backward Pass Function for ReLU_6 Activation function
def backward_pass(L1, L2, L3, W1, W2, W3, k):
    dW3 = (L3 - T) * ReLU_6_Derivative(L3) * k
    dW2 = W3.T.dot(dW3) * ReLU_6_Derivative(L2) * k
    dW1 = W2.T.dot(dW2) * ReLU_6_Derivative(L1) * k
    return dW1, dW2, dW3
```

Unfortunately, I didn't manage to get this one to produce meaningful results:



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.

The first column of images shows the average values of the activations, weights, and changes in weights (from top to down) in the first hidden layer of 256 nodes. Similarly, the second column of images shows the activations, weights, and changes in weights for the second hidden layer. Finally, we see the same values for the output layer.

I will make specific observations for each scenario below

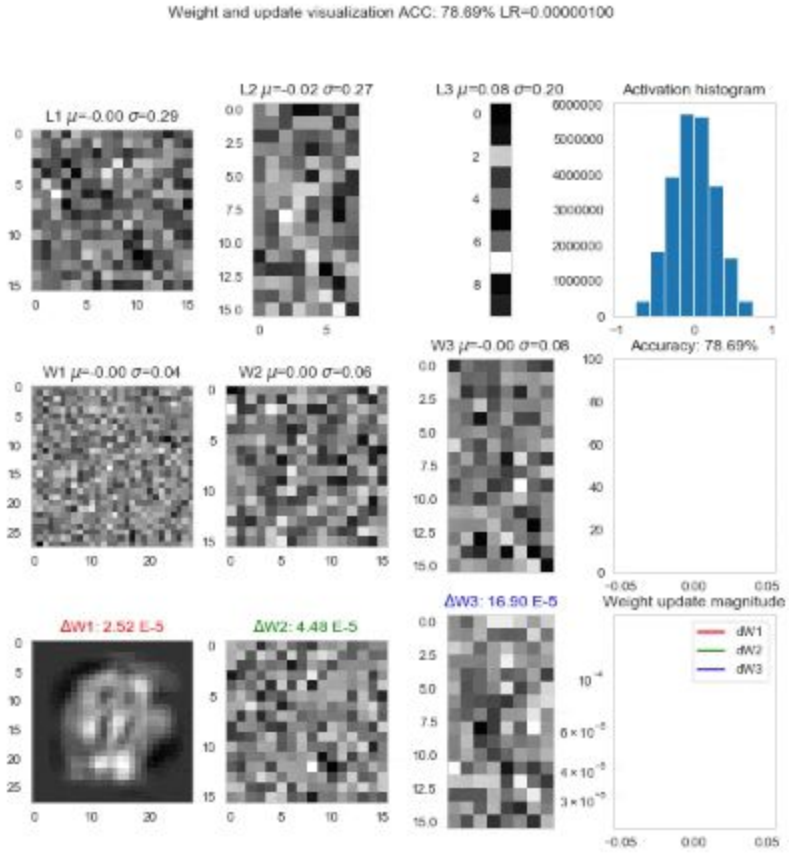
- How do the visualizations/plots differ for Tanh, ReLU and cross entropy?

Note that the below are gifs. The Google Docs version of this file can be found here

MSIA 432 Deep Learning
Spring 2020

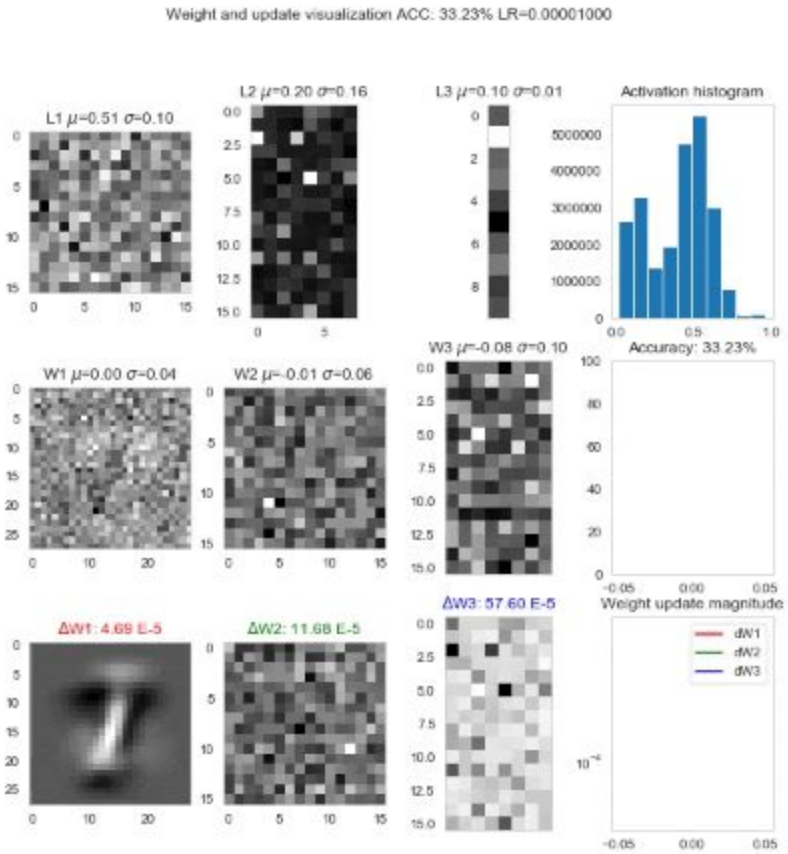
Below are for Tanh with weights initialization from N[0, 1/ sqrt(n)]
I think it's nice to see how the delta W's appear to be identifying a 3 with subsequent iterations
Also, I note that the weighst all appear to NOT change over the iterations. I suspect this might be due to the magnitude of the deltaW's (they are of the order 10^-5) and the relatively large weights, which are of order 10^-1 in absolute value.

Finally, I look at the first row, 3rd image with the 10 activations for the 10 classes. It's nice to see that with more iterations, the square become more monotone in their color. In other words, the model doesn't have, on average, a very different activation for each class. This is also related to the high accuracy i.e. the model predicts well across the board.



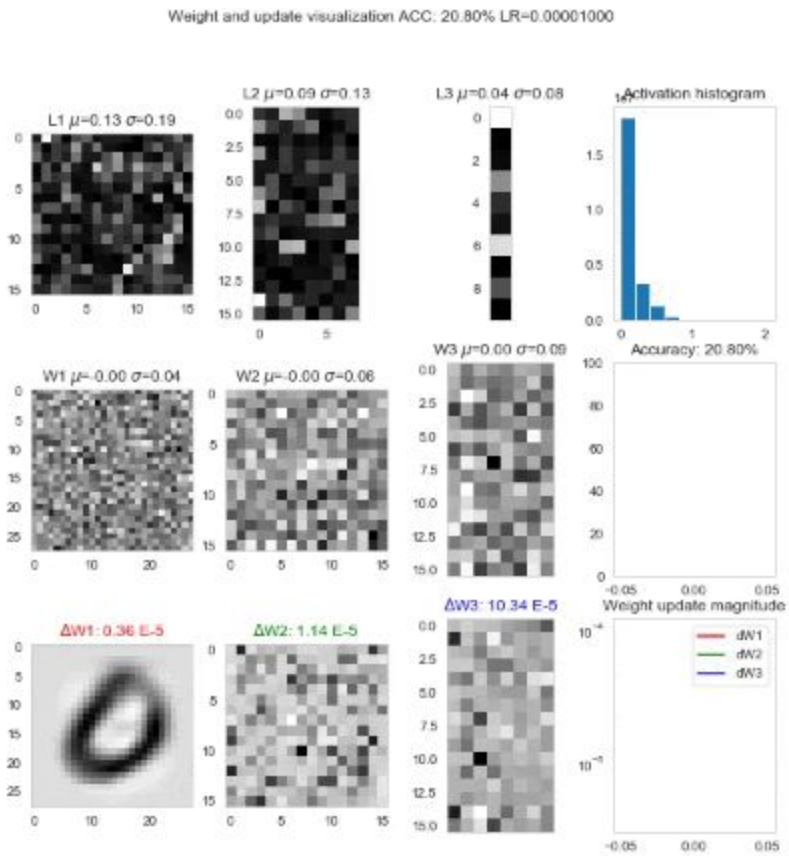
Below are the Cross_Entropy images as a gif.
Note how the gradient changes (3rd row) get brighter and brighter with successive iterations. This is precisely due to the use of Cross Entropy as a loss function
Below are the Cross_Entropy images as a gif.

MSIA 432 Deep Learning
Spring 2020



Finally, here is the gif for the ReLU with clipping at .0001 and 1000 iterations

```
# ReLU Activation function
def ReLU(x, k=1):
    return np.maximum(0.0001, x*k)
```



I observe what ReLU is famous for - it incentivizes the network to use a sparse set of it's nodes. That's why there are a few brighter nodes in the first row of images and most of them are quite dark.
Similarly, we can see how the images in the second row gradually get darker.

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

In each of the three scenarios above, it appears that in general the updates to the magnitude increase with subsequent iterations. This is good, we don't want our gradient to be vanishing, otherwise, our training will converge before the NN can make good predictions.

Moreover, the magnitude of change is higher the deeper down the layers we go i.e. the output layer has the largest changes in all three scenarios, then the second hidden layer has the second highest changes in magnitude and finally, the first hidden layer changes the least. This is expected, as our gradient gradually decreases as we "propagate backwards".

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

The graphs showing the loss function should flatten out; the magnitude of weight change should become very small and therefore our weights & activations should more or less stop changing. A plateau in the accuracy curve would also suggest the same. All this would be happening before we see some uniformity in the output layer's colors i.e. the NN is not able to make proper predictions for all classes in

the data and is "stuck" in a local minimum or just has a vanished gradient and can't make more improvements.

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

I've already established that the tanh definitely prefers the random Normal initialization of the weights as opposed to the uniform initialization, because the accuracy after only 200 iterations was 24.37% vs 84.51%.

I'm not exactly sure what this question is asking. I've noted above that the weights change with more training but that change is very small w.r.t. the initial values of the weights. As a result, the images of the weights appear to not change.

And of course, the way we initialize the weights influences how the weights & activations will change. That's why if we initialize with uniform values, we get more values closer to 1 or -1, where the gradient of our sigmoid and tanh functions is close to 0. It turns out to be a better idea to initialize from a normal distribution.

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 (including T-SNE plots and confusion matrices). Submit your final code, but comment out the lines that you aren't using, e.g. tanh.
Extra credit: Implement a cosine learning rate.

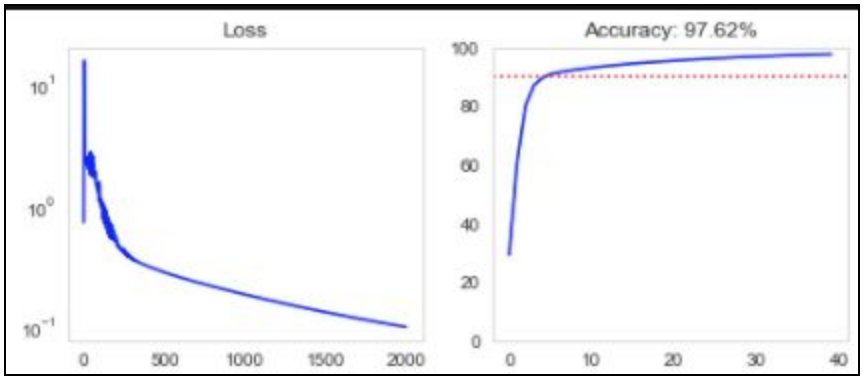
I will train with 2000 iterations for a network with the parameters that performed best during my investigation: k = 1, learning rage = 1e-5 * 2, He Initialization of weights, Cross-Entropy Loss function, Sigmoid activation function. I can't be sure that this combination is the best, because I've only tested these parameters in isolation while keeping the other ones constant. However, it's the only reasonable thing to do. GridSearch would take far too long.

The final result is 97.62% accuracy. Of course, it would probably make sense to do stochastic gradient descent with mini batches, regularization, and some cross-validation, to see how well this would generalize to new data.

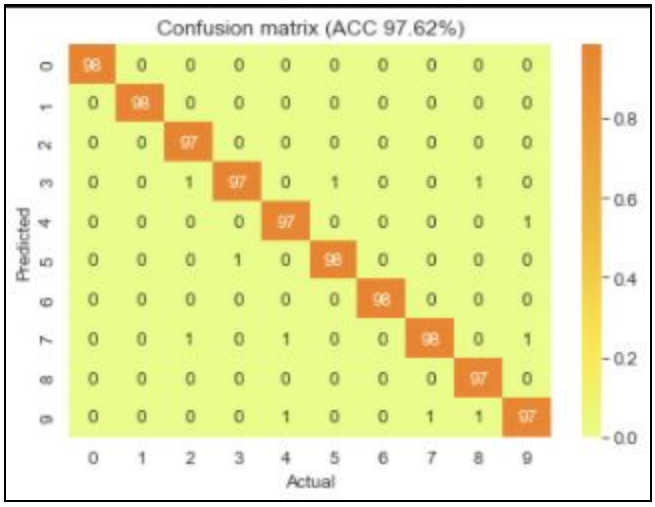
Here are the detailed results:

Final Loss & Accuracy curves

MSIA 432 Deep Learning
Spring 2020

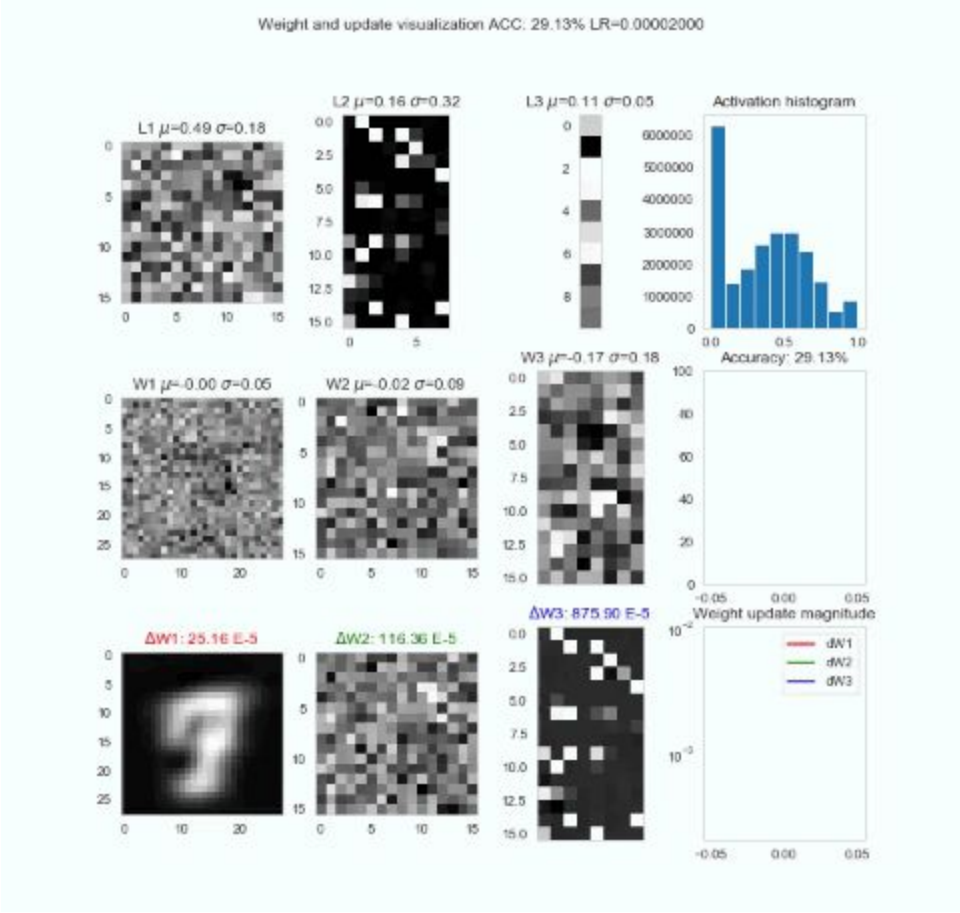


Final Confusion Matrix:

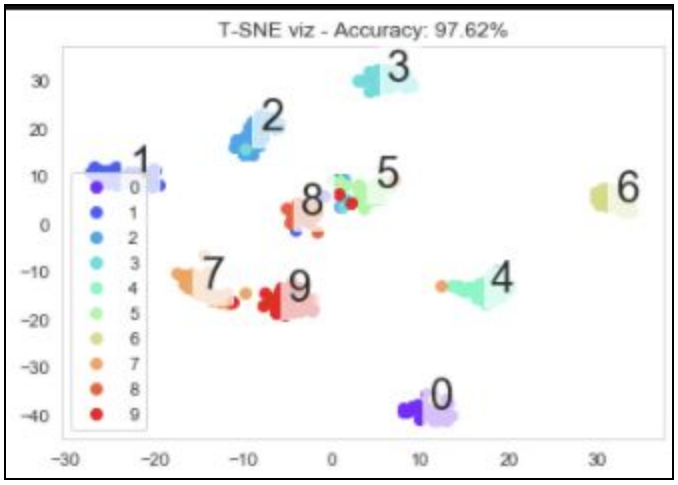


MSIA 432 Deep Learning
Spring 2020

Evolution of activations, weights, and magnitudes of weights changes:



Finally, the T-SNE plot, which shows nice separation between the classes:



The Jupyter Notebook should be in the zip file with the docx & pdf version of this file.
I've left detailed comments throughout this homework on why I think different things work and why I think they work the way they do.

MSIA 432 Deep Learning
Spring 2020