MSIA 432 Deep Learning
Spring 2019

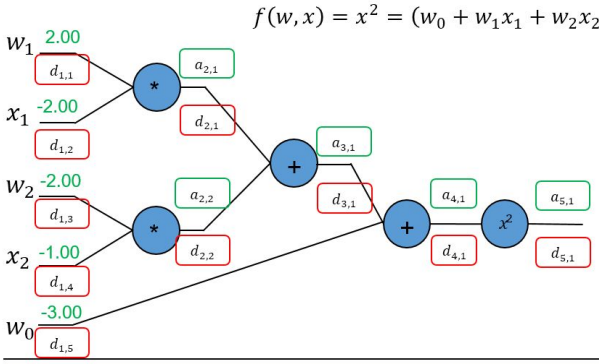
Assignment 1 (Due 5/8/2019)
Understanding Neural Networks

Name	Naomi Kaduwela
Discussion partner	Shiva
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:

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



Directions:

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

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

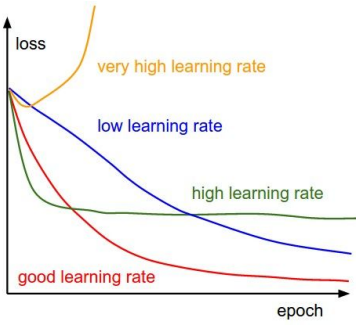
$d_{1,1}$	-10*-2= 20	$a_{2,1}$	-2*2=-4	$a_{3,1}$	-4+2=-2	$a_{4,1}$	-2+-3=-5	$a_{5,1}$	-5^2 = 25
$d_{1,2}$	-10*2 = -20	$a_{2,2}$	-2*-1=2	$d_{3,1}$	1*-10 =-10	$d_{4,1}$	2x = 2*-5=-10	$d_{5,1}$	1
$d_{1,3}$	-10*-1= 10	$d_{2,1}$	1*-10=-10						
$d_{1,4}$	-10*-2= 20	$d_{2,2}$	1*-10=-10			$\frac{\partial}{\partial x}x^2$	2x		
$d_{1,5}$	1*-10= -10					$\frac{\partial}{\partial x}a^x$	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.

2.1. 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?

- Learning rate configurable parameter indicating how much the amount weights should be updated during training. i.e. it's like a step size.
- It is often small between 0-1. Start at .01 or .1 lr = 0.1 is common, meaning that weights in the network are updated 0.1 * (estimated weight error) or 10% of the estimated weight error each time the weights are updated.
- **Larger learning** rate will learn faster, but if they are to big they can cause oscillations during epochs that may never converge because they keep jumping over the minima
- **A good learning rate** one that results in a decreasing loss curve and increasing learning rate that would level off somewhat quickly because it's taking optimal steps in the right direction without being too big or too small - see figure attached.



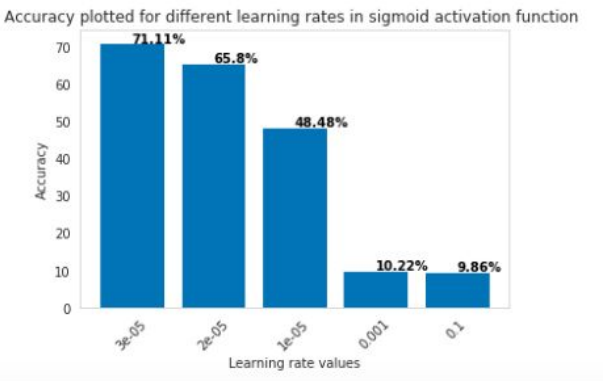
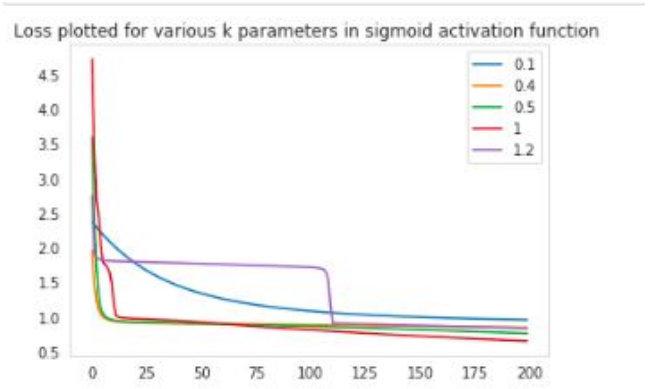


Figure: Optimal Learning Rate = 3e-5. ***Accuracy = 71.11%***

2.2 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.

- **Activation function** of a node defines the output of that node ("neuron") given the input set. That output is then used for the next node's input, until convergence on weights.
- **K training parameter:** represents the slope of the sigmoid function, and thus impacts the learning rate of the function.
  - **Small K:** As the slope decrease, the sigmoidal function becomes more linear and the loss decreases are steep, but learning is slower as it's taking small steps towards the minimum.
  - **Large K:** As the slope increases, the sigmoidal function becomes more step function like and the same is seen in the loss curve. We have to be careful here because the function could jump over the local minimum if the step size is too big.
  - **Optimal K:**
    - k= 1 looks the best with regard to the losses decreasing exponentially the fastest and the highest accuracy.



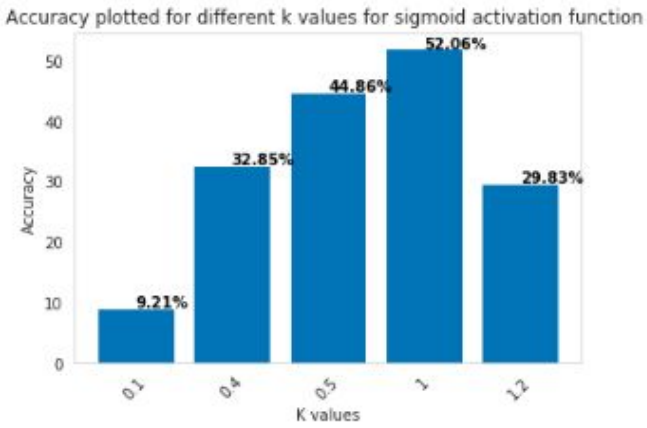


Figure: We can see the loss is best & accuracy highest for k=1. ***Accuracy = 52.06%***

2.3 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?

- **Distributions:** Different initializations for Sigmoid showing the normal dist is best
  - Initialize with 0 centered **Normal Dist** is better than **uniform dist** because there are more points closer to 0 so it converges faster.
- **Weights:** as it's mean centered and values are close to 0
  - **Initializing with 0** weights**:** If all the weights are initialized with 0 , the derivative with respect to loss function is same for every w in W[l], thus all weights have same value in subsequent iterations. This makes hidden units symmetric and continues for all the n iterations i.e. setting weights to 0 does not make it better than a linear model.
  - **Initializing with high terms:** maps values to 1 where the gradient changes slowly and takes more time to learn
  - **Initializing with low terms:** most values will be close to 0 and the gradient will change slowly and take more time to learn
  - I**nitialize with random weights:** works well because all the weights randomly from a univariate "Gaussian" (Normal) distribution having mean 0 and variance 1 and multiply them by a negative power of 10 to make them small.

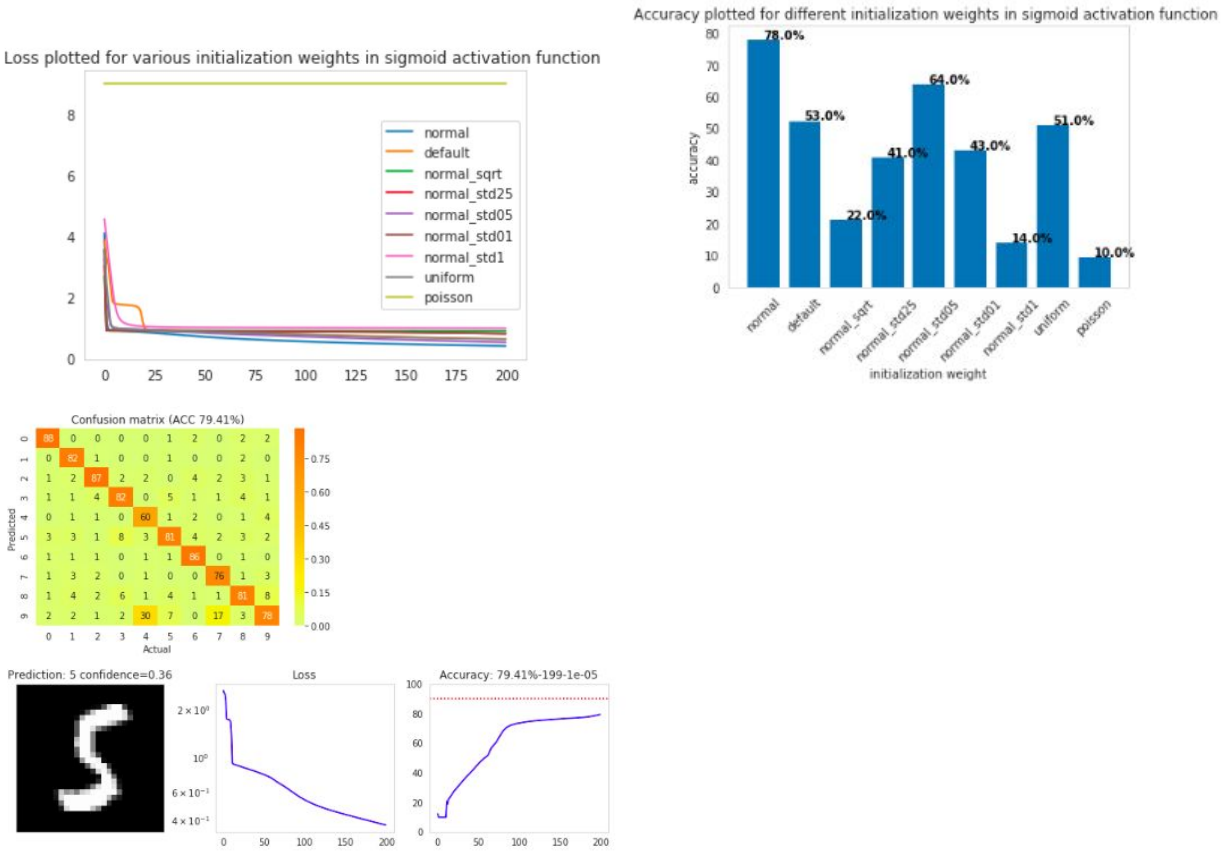


Figure: Best Initialization with normal dist. (k=1 & lr = 1e-5). ***Accuracy = 78% (top)***
Best Initialization with random. (k=1 & lr = 1e-5) beat normal! ***Accuracy = 78.41% (bottom)***

Testing results of all weights = ['default', **'normal' (Best!)**, 'normal_sqrt', 'normal_std25', 'normal_std05', 'normal_std01', 'normal_std1', 'uniform', 'poisson']

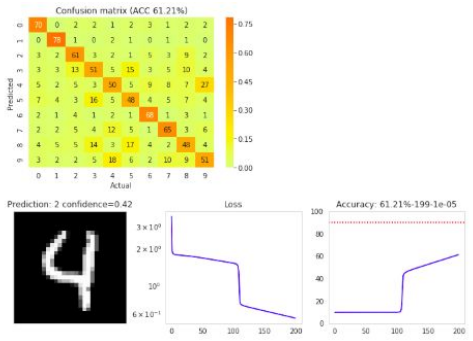


Figure: Sigmoid with k=1 & default random weights accuracy = 61.21%

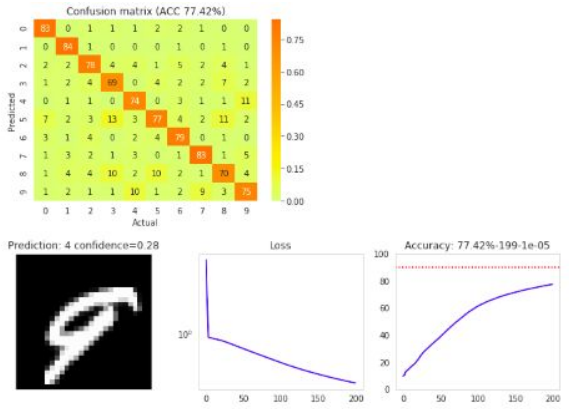


Figure: Sigmoid with k=1 & normal weights accuracy = 77.42% (**Best!)**

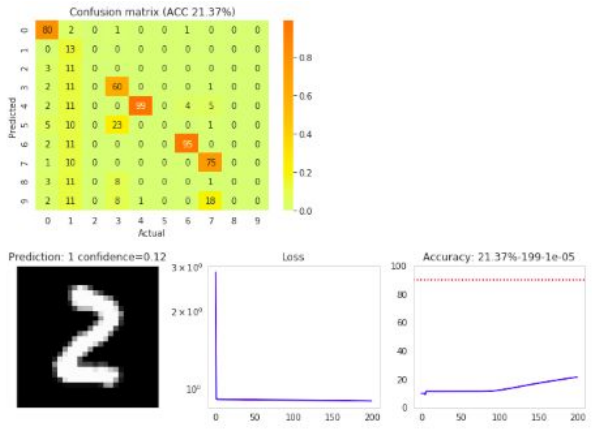


Figure: Sigmoid with k=1 & 'normal_sqrt' weights accuracy = 21.37%

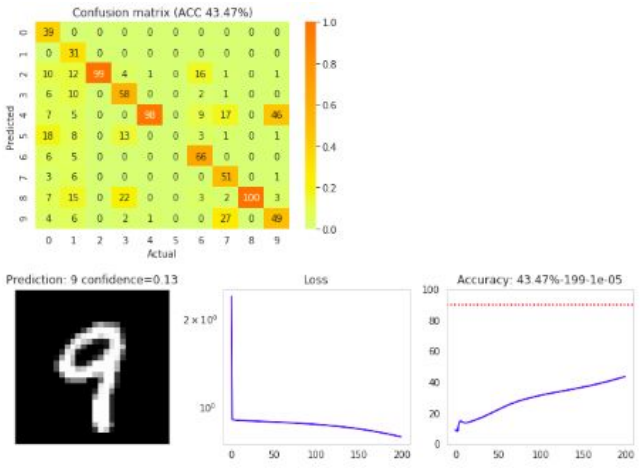


Figure: Sigmoid with k=1 & 'normal_std25' weights accuracy = 43.47%

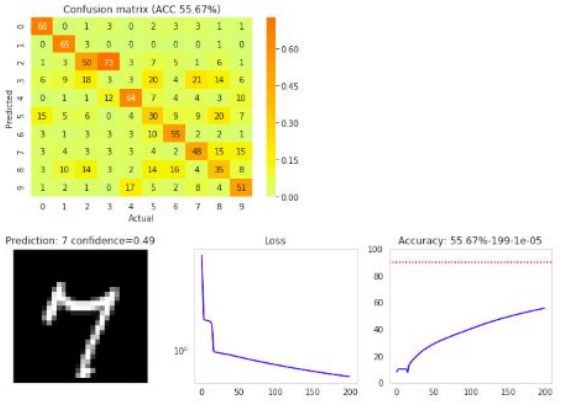


Figure: Sigmoid with k=1 & 'normal_std05' weights accuracy = 55.67%

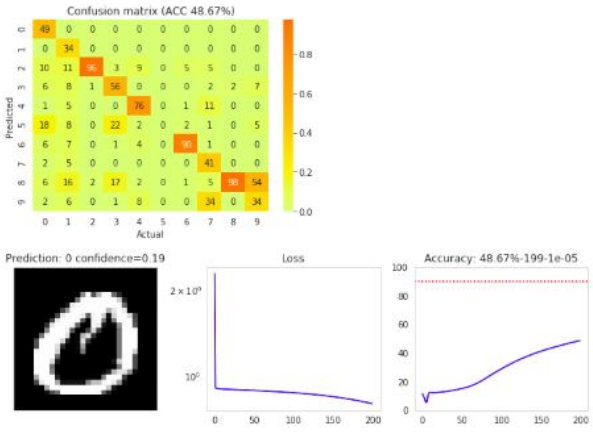


Figure: Sigmoid with k=1 & 'normal_std01' weights accuracy = 48.67%

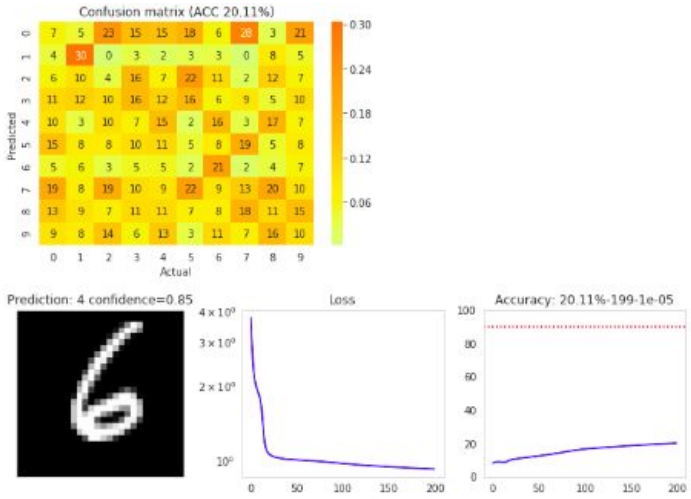


Figure: Sigmoid with k=1 & 'normal_std1' weights accuracy = 20.11%

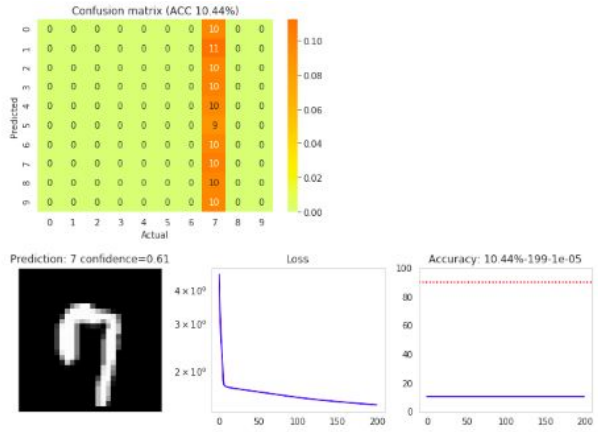


Figure: Sigmoid with k=1 & uniform weights accuracy = 10.44%

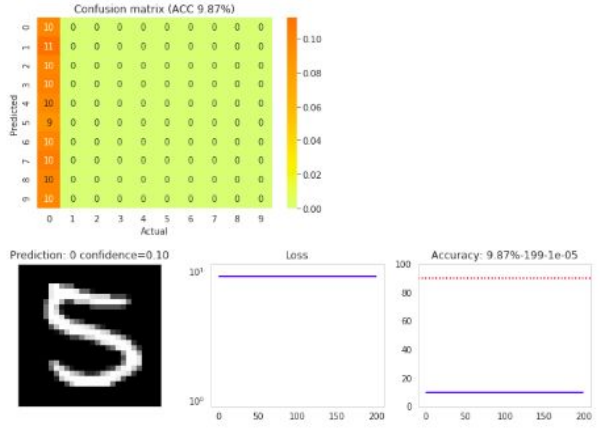


Figure: Sigmoid with k=1 & poisson weights accuracy = 9.87%

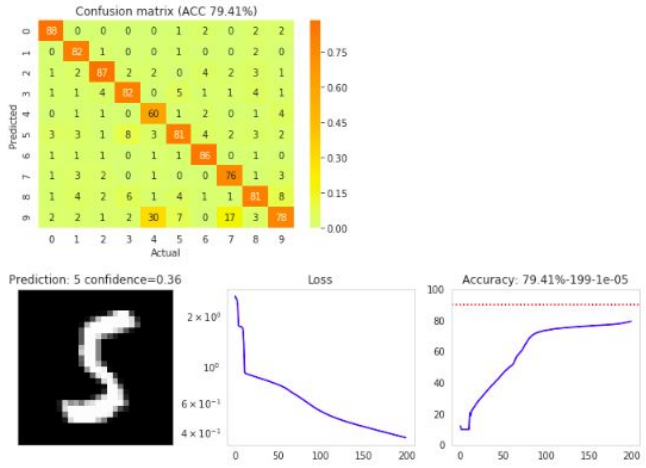


Figure: Sigmoid with k=1 & random weights (sqrt(2/<lower layer size>)). ***Accuracy = 79.41%***

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.

The Activation matrix plot is mean activation across all the images, and so we hope that it is mostly gray & some white in color because this means the neurons have enough value to be weighted. Black neurons are dead neurons, which have negative impacts to training and are not being effectively utilized. Too many whites means everything is activating, which isn't helpful either when trying to determine a definite classification class.

Sigmoid has a disadvantage of vanishing gradient, for very high or very low values of x there is almost no change to the prediction.

Since we are using a sigmoid activation function (which ranges from 0-1), we want the mean to be around .5 and would like a large standard deviation which would mean that the neurons are more active. As for the weights, we're looking for zero-centered. If they are not zero centered, gradients will be all positive or negative and then there is a higher likelihood for zig-zag of weight changes between steps leading to a decreased chance of convergence.

When we switch sigmoid from poisson to normal distribution we see the accuracy greatly increase, while the activation plots change from mostly black to shades of gray.

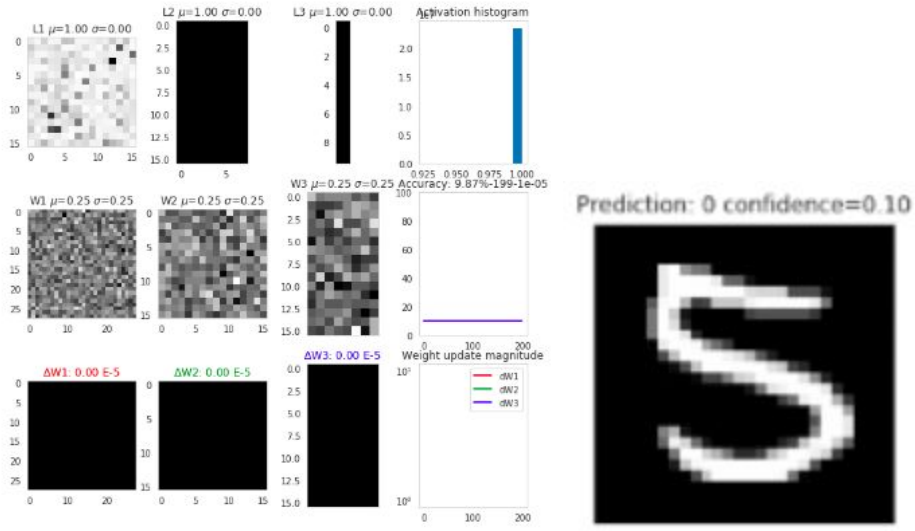


Figure: Sigmoid with k=1 & poisson weights **accuracy = 9.87%.**
In addition to extremely low accuracy, we see many the gradients are black - i.e. they are not activating at all - and are dead neurons. We also see only the first layer is all white, and then the proceeding layers are black, due to vanishing gradient.

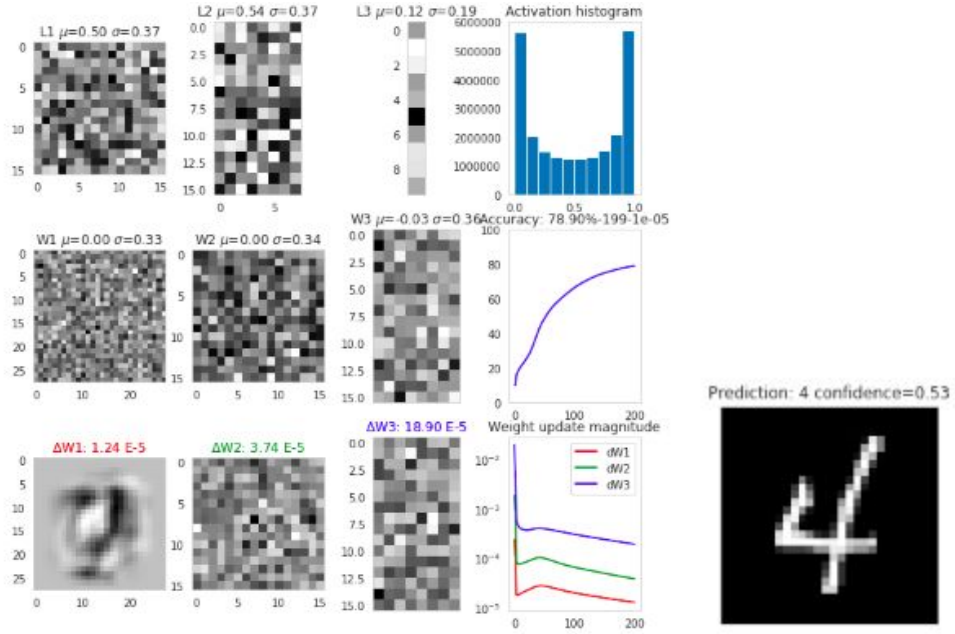


Figure: Sigmoid with k=1 & normal weights **accuracy = 78.90%.**
Here we see that the accuracy has improved, and the activation and gradient plots with it as they are now much more gray, leading to less dead neurons, and each layer of activation has some white and gray neurons.

(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)?

- **Tanh** is better than sigmoid its outputs are mean centered. Tanh's output range is -1-1, whereas sigmoid output range is 0-1. This will minimize inefficient zig zagging that arise from gradient values being all positive or all negative which makes optimization difficult, as Tanh will have positive, negative, and neutral values. The large range also enables more room to tweek the weights
- **The derivative** of sigmoid falls within 1-1/4 range. The derivative of tanh falls within the 0-1 range. Tanh also suffers from vanishing gradient.

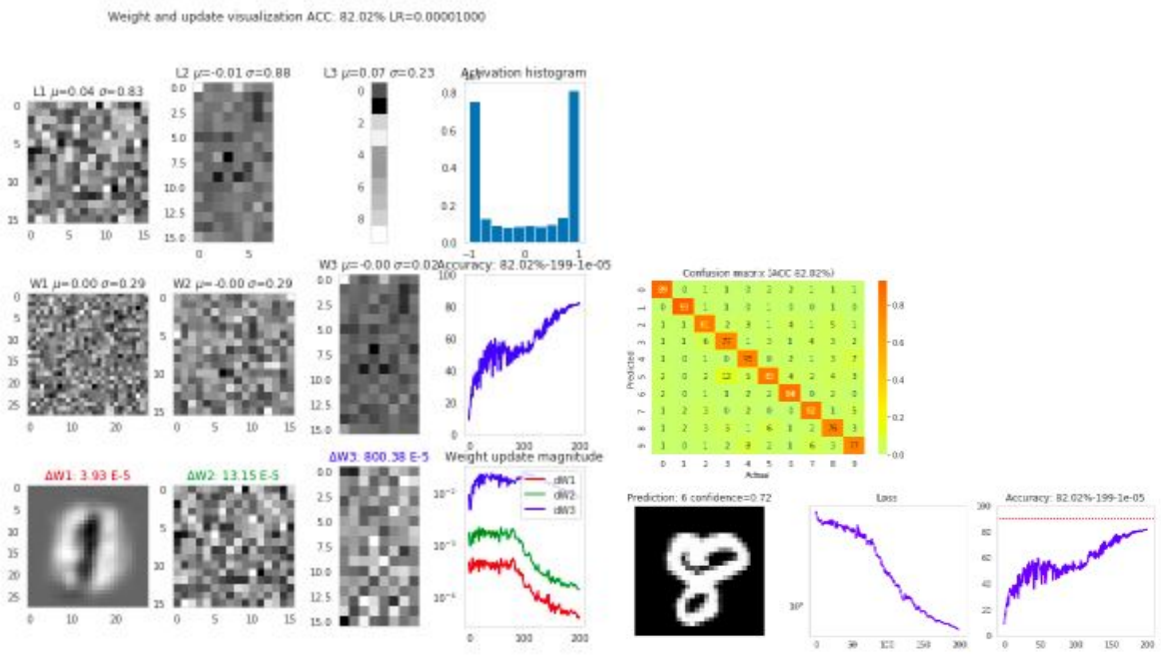


Figure: Tanh with k=1 & normal weights. Here we see a lot more gray than the sigmoid activation layers and gradient plots & accuracy has increased. ***Accuracy = 82.02%***

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

- **Cross Entropy** is another loss function we can aim to optimize loss on that is better for classification, and we can see the accuracy improved. MSE doesn't punish misclassifications enough but is the right loss for regression, where the distance between two values that can be predicted is small.

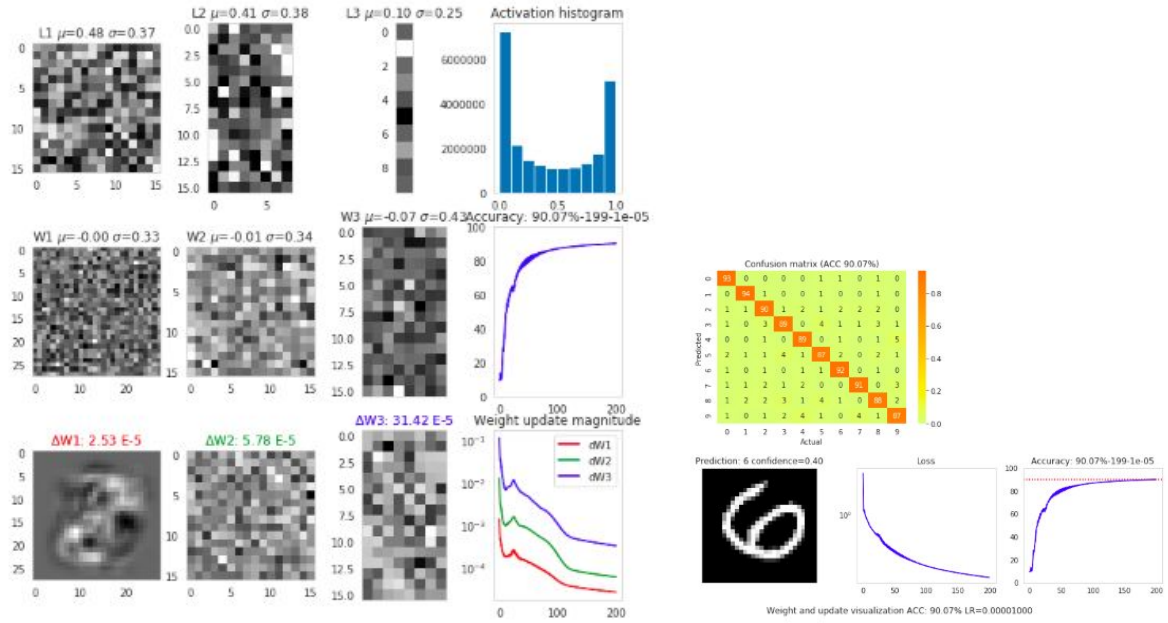


Figure: Sigmoid with cross entropy & k=1, learning rate = 1e-5, normal dist. ***Accuracy = 90.07%***

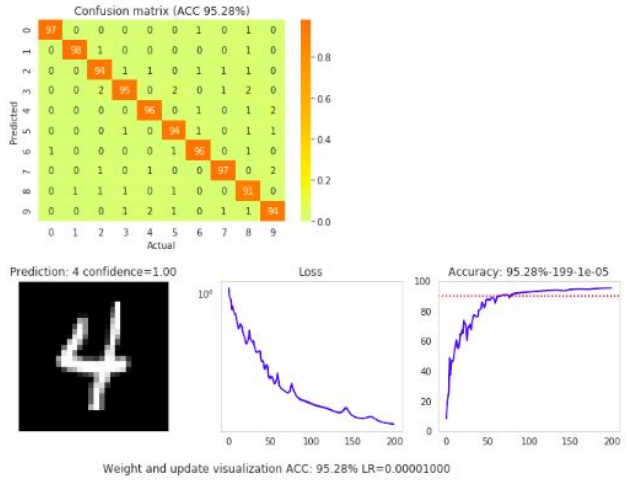


Figure: Relu6: loss vs Relu6: cross entropy & learning rate = 1e-5, normal dist. ***Accuracy =***

(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.

- **ReLU** improves drastically as it is computationally more efficient so that it can converge quickly and avoids both diminishing gradients and exploding gradients.

- **Derivative of Relu is 1**, when the input is greater than is 0, which is good because then the gradient will not vanish, which is a risk for sigmoid and tanh.
- However, when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn.
- Some of the neurons did die - as we can see they are black throughout the layers. This occurs when the weights update in such a way that cause the neuron to output zero and die as the information is no longer passed and updated. This is due to the ReLU function as any value when x<0 has a value of 0 and the function gradient is also 0.
- In order to prevent a ReLU neuron from dying, we can give a small positive gradient for negative inputs.

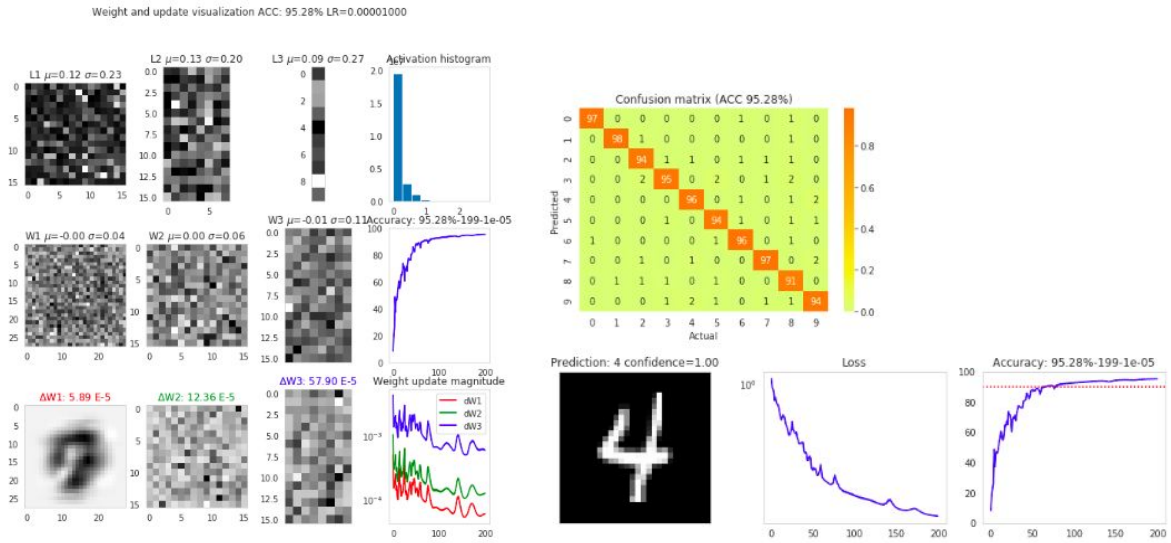


Figure: Relu with regular loss learning rate = 1e-5, normal dist. ***Accuracy = 95.28%***

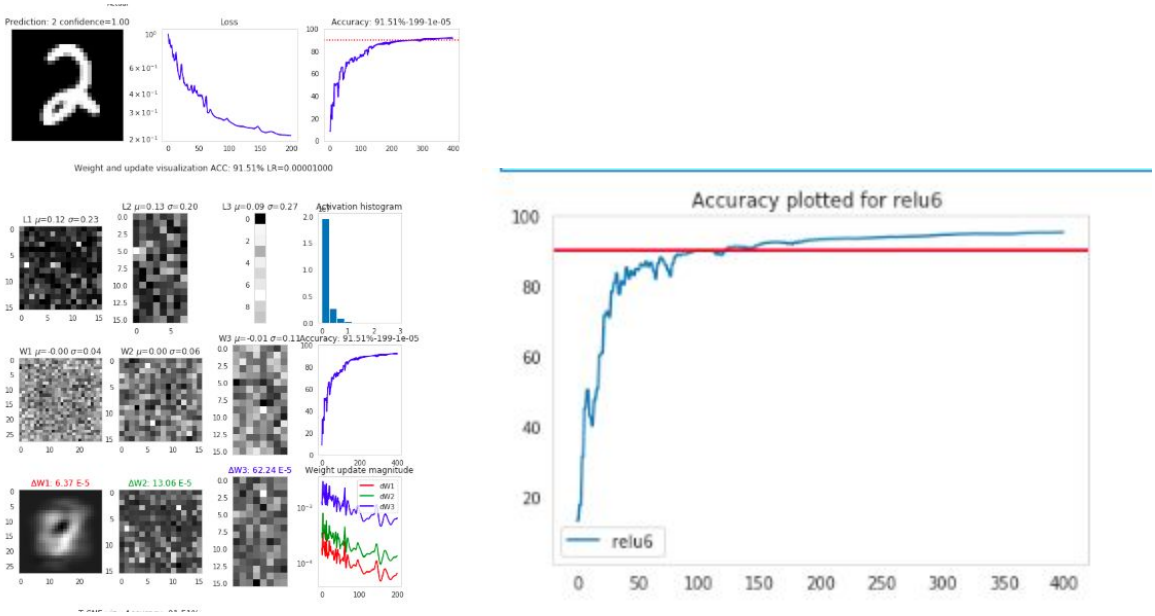


Figure: Relu6 with learning rate = 1e-5, normal dist. ***Accuracy = 91.51%***

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?

- The slope of the loss curves differ. Sigmoid looks like a smooth, exponential curve going downward, rapidly decreasing at the beginning and then leveling off. Tanh is jagged and almost convex looking that starts a little slow (due to the quadratic cost function) then starts to descend quickly. Relu6 looks more efficient than Tanh as the slope is more direct and linear. These loss curves generally correlate with the accuracy scores seen.
- As mentioned in the previous problem, the activation and gradient plots also differ across the 3 activation functions: with a good sigmoid having a lot more black and white than Tanh and relu.

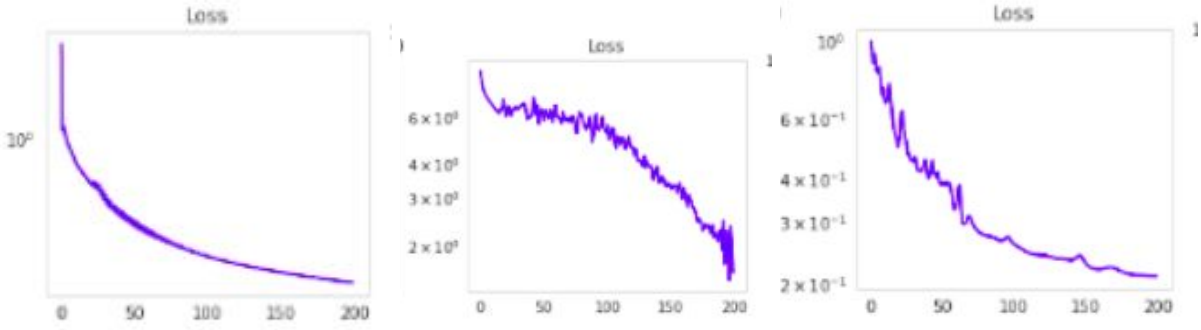


Figure: Loss curves for Sigmoid (with cross entropy), Tanh, and Relu6, in respective order

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

- During the periods of greatest gradient descent, the weight changes are greater in total magnitude as the neural network is training with new weights during and after the descent. This could occur during the beginning, middle, or end of training - depending on the function.
- With sigmoid, there is a steep descent at first, then an increase from a dip, then a graduate decline that levels off.
- Tanh takes longer to update the weights
- Whereas relu is continuously updating weights and improving with the steepest slope over the entire time

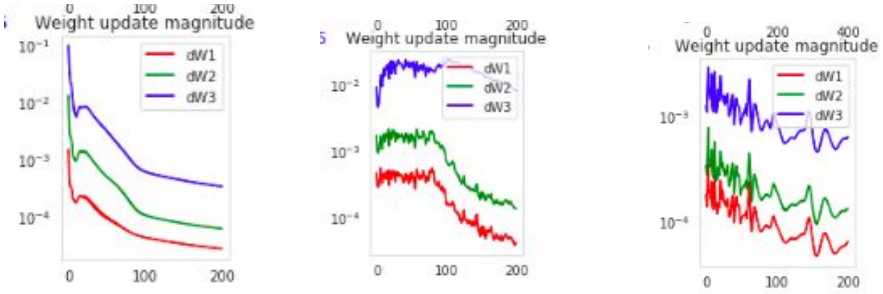


Figure: Weight update magnitude for Sigmoid (with cross entropy), Tanh, and Relu6, in order

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

- Constant black neurons in activation matrix plot means the neuron has died and is no longer sending information. This could potentially be fixed by ensuring the data is 0 centered.
- If the loss function does not decrease over time that could mean we are stuck at a local minima. This can be fixed by possibly increasing the step size so the gradient descent is not stuck in a local minima.
- At the final trained state, we should look for high accuracy plots, decreasing loss plots, and plots that look at the mean across all of the images should be gray as opposed to all black or all white.

- 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 function and associated corresponding optimal weighting do make a difference. The network prefers cross entropy - as it penalizes better for classification than MSE - and normal distributions - as mean centering around 0 helps optimize convergence. Optimal learning rate also needs to be tuned to ensure it does not get stuck in a local minima.

- While all of these make a difference, there is always an asymptote that can be estimated from the first few hundred epochs of training, and there will be less improvement as the training reaches infinity.

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.

The final model is a ReLU, which is not surprising as this is one of the most efficient activation functions that is better than tanh and sigmoid in such a way that it is not susceptible to vanishing gradients.

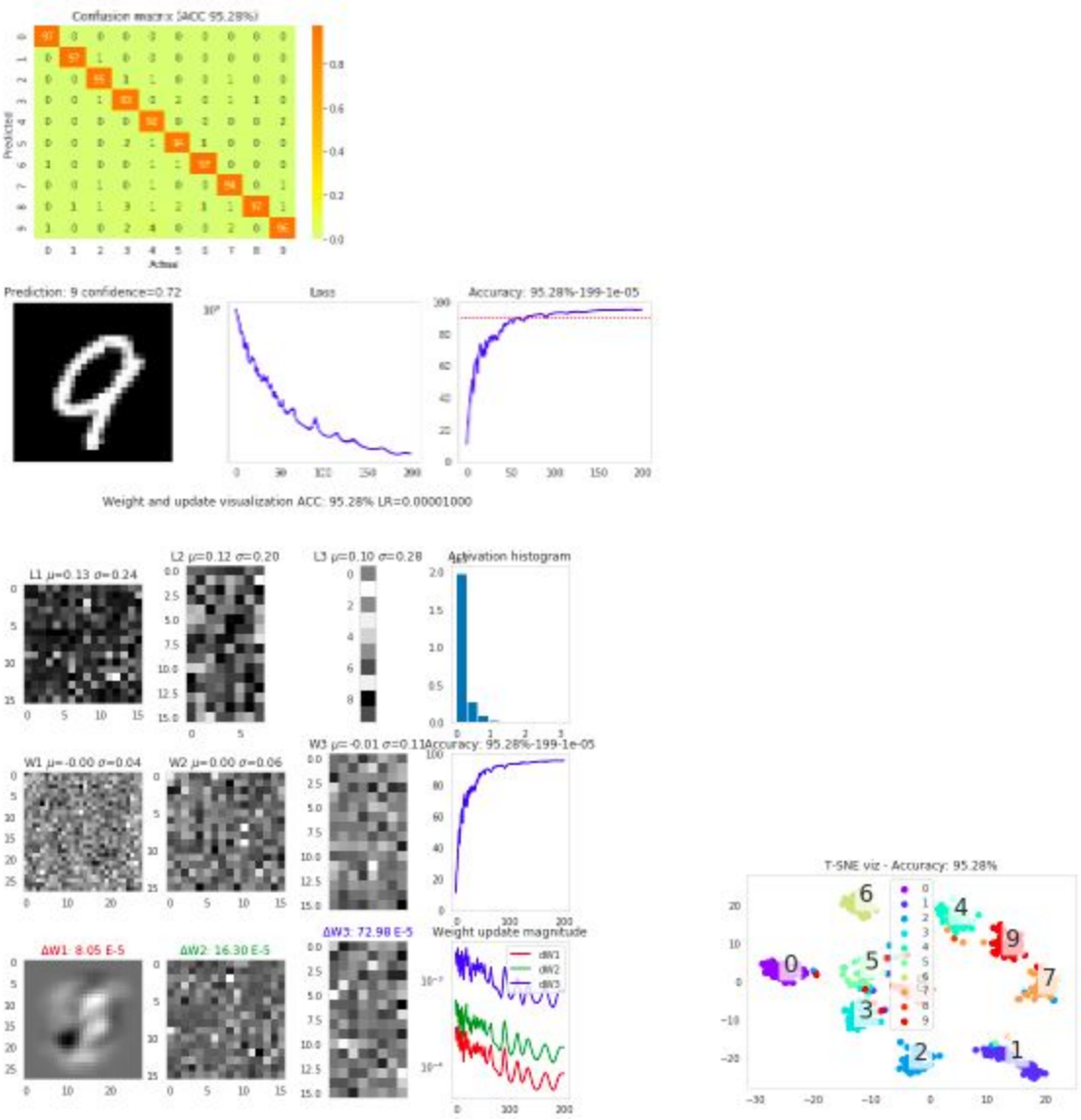


Figure: Overall best model with relu, lr = 1e-5. ***Accuracy = 95.28%***

Extra credit: Implement a [cosine learning rate]().

Learning Rate = lr_min + 0.5 * (lr_max - lr_min) * (1 + np.cos((i / (Epochs/cycles)) * np.pi))

MSIA 432 Deep Learning
Spring 2019