Assignment 1 (Due 5/5/2018)

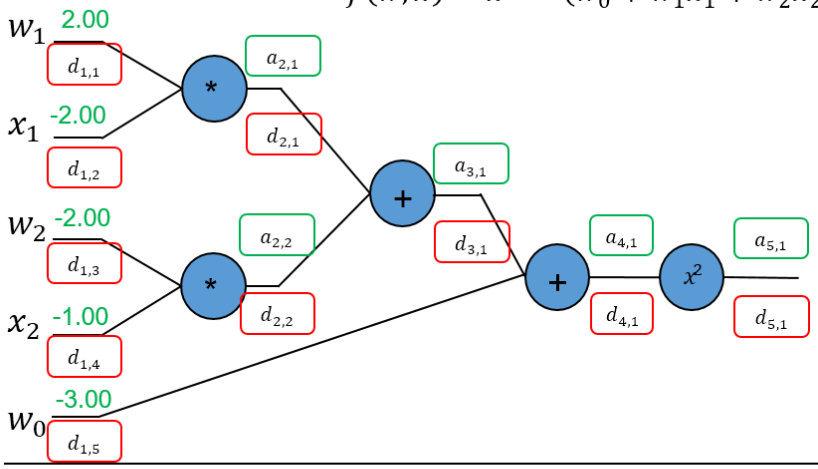
Understanding Neural Networks

Name	Eric (Kehan) Pan
Discussion partner	
Comments	For the first 4 problems I ran the code to 200 replicates. For problem 5 I ran all the 1000 replicates. 1000 runs reach 100.0%, and 200 reach 98.6%. First 4 problems I wasn't focusing on optimization, instead I focused on addressing the problem. Hope you forgive me for detailed explanation there; once I get the 100.0% methodology I understand the naive models don't have much room for improvement. I would suggest you go to my github for the full result. Jupyter notebooks are included there: https://github.com/pankh13/Design-NNet
Feedback	1. Working on the optimization is so interesting! I love designing our own neural network. I learned so many things there. 2. If some of the problems could be clearer it is going to be much better cause they seem ambiguous now.

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								

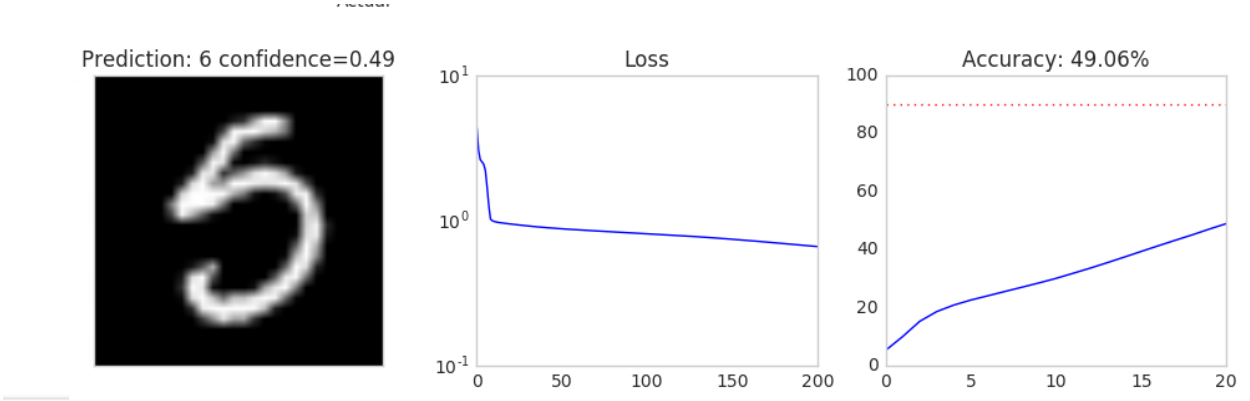
$\frac{\partial}{\partial x} x^2$	2x
$\frac{\partial}{\partial x} a^x$	$ln(a)a^x$

2. Neural Networks in code (12 points)

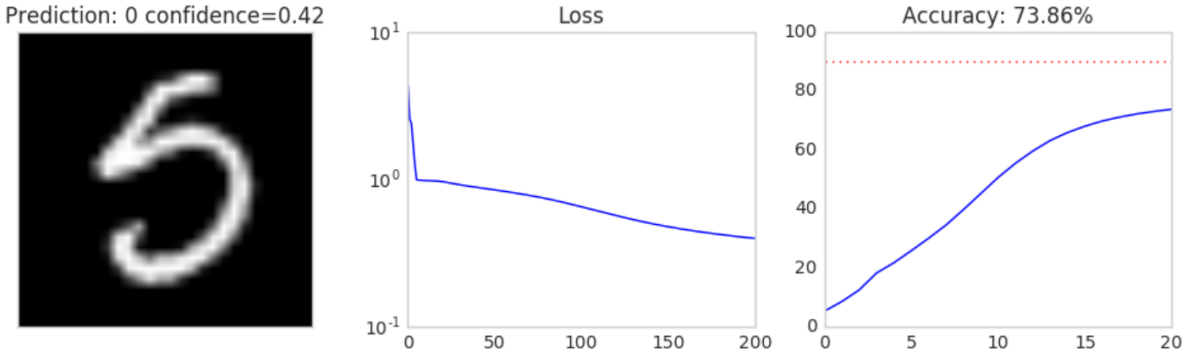
Using the provided example code from Lecture 3, explore the items below and demonstrate how to improve the example code by showing plots of the loss and accuracy curves. For each plot, show the curves for the first 200 iterations (You can stop training after 200 iterations for this part of the assignment).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

The default learning rate 1e-5:



Among my tests 2.5e-5 is the best.



When we have an appropriate learning rate, the loss function should, instead of going all the way down (too conservative), zigzag a little bit while going down in general. And the precision rate should go up with a little bit zigzag pattern. Of course, we also don't want too large vibration in the curve which means it's too aggressive.

However, in sigmoid if we set the learning rate too large it's possible that neurons get saturated and the loss function stuck. So we find the one with steepest descend.

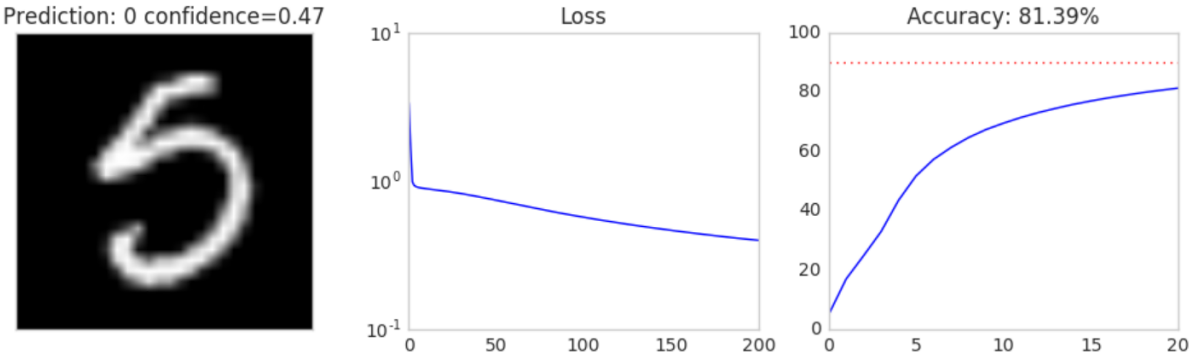
(2) Activation function: Try changing the sigmoid function to "1.0/(1.0 + np.e**-(k*x))", where k is another training parameter. Explain what k does. What is the effect of a small k on training versus a larger value for k? Is there an optimal k for a given learning rate? Use the default learning rate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

K scales the sigmoid function so that it has a different distribution.
$sigmoid(x)$ requires input to be preferably distributed in [-3,3] since its gradient has better value; while for $sigmoid^{kx}$) it is $\left[\frac{-3}{k}, \frac{3}{k}\right]$.

a) When k is large (for a certain initialization of course), the desired range of activation ($\left[\frac{-3}{k}, \frac{3}{k}\right]$) is small and there may be many neurons falling out of the range and get saturated easily; after saturated they updates slowly.
In this case, most neurons get saturated instantly; the neurons in the middle range gets updates quickly, then kind of plateau. So the loss function improves quickly and then plateaus.

b) While when k is small enough, that all the activations fall within the range $\left[\frac{-3}{k}, \frac{3}{k}\right]$, we need an optimal learning rate so that the learning process is good. What often happens is that the gradient is too large or small.

In our case, use np.var() check W1.X distribution we can see their variance is about 28, so it goes out of the range. We want the k smaller.
After a few experiments we see the 0.4 is about the best.



(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?

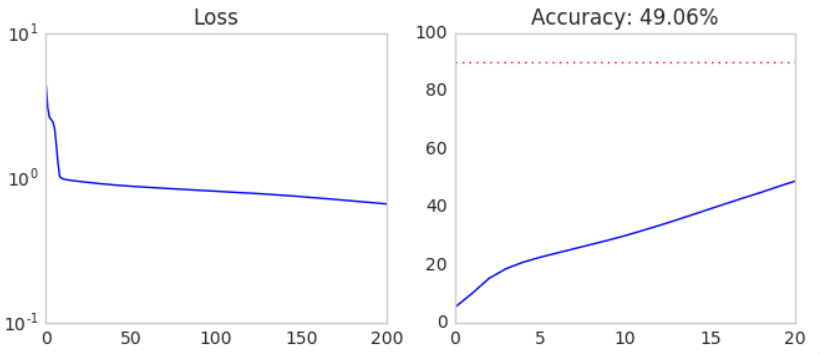
Try different distribution of initializing; e.g. normal distribution; However, there's no confirmation that which distribution is better, it's more like pure luck.

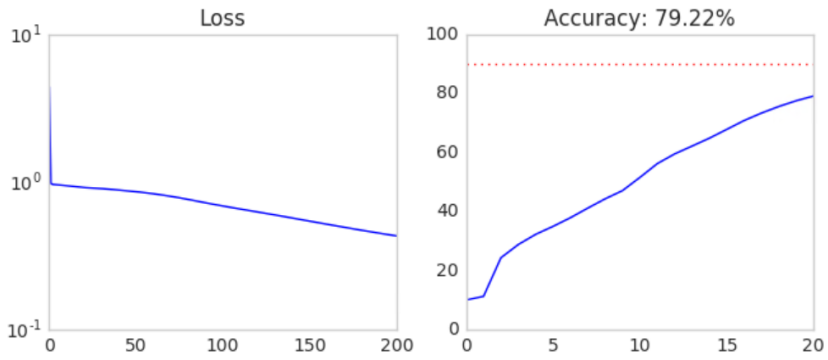
Use variance to fix the W1 distribution; (divide W1 by $\sqrt{n}$, n is the number of nodes in previous layer; or divide by $\sqrt{variace(W_i L_{i-1})}$, this requires one forward pass);

In this case I divided them by 5, 5, and 2.

Orthogonal initialization (discussed in details in part 5); this is the best one. Compared to the initial one we can see big improvement.

200 runs uniform vs orthogonal initialization





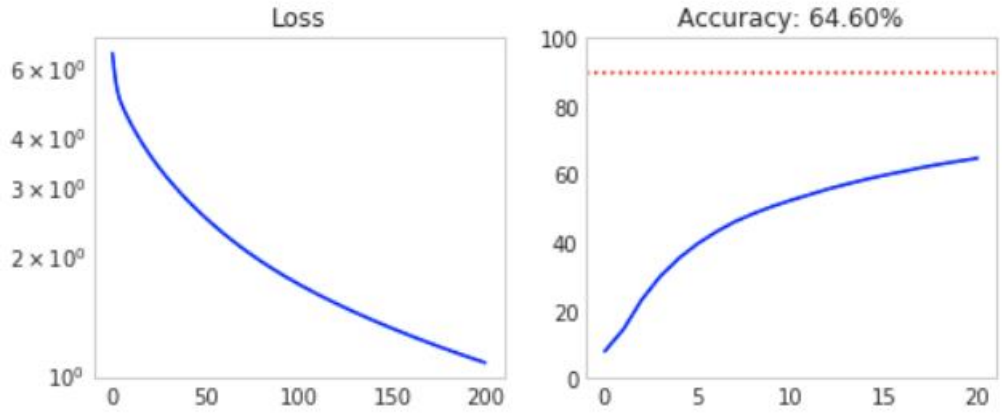
3. Optimization in code (16 points)

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

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

For activations, when they are close to the final result they are generally good. They should be distributed within the range where the gradient of activation functions has reasonable scale.
For gradient updates, check the magnitude plot. We want the three magnitudes to be of similar scale, and we can use different learning rate for different layer to address the problem.
Also we don't want strong pattern in gradient, which indicates too large learning rate (each time learn from the last picture).
For activations we can see the initialization in part 2 or 5.

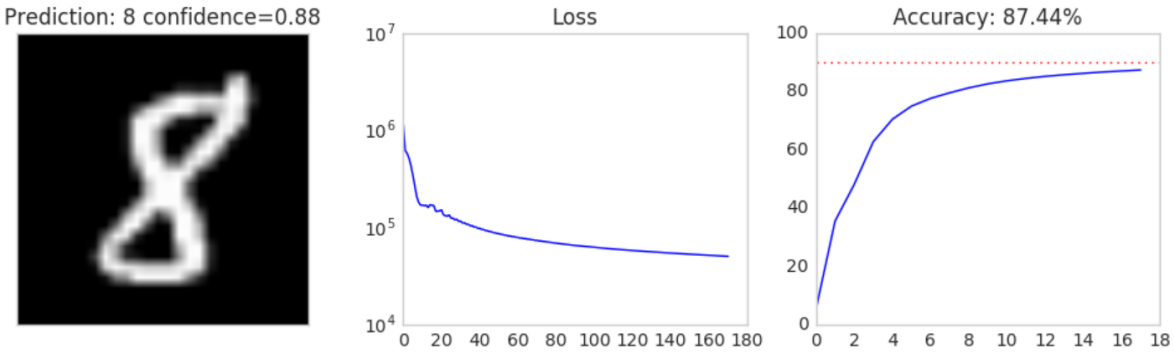
(2) **Tanh:** Implement tanh(x) instead of the sigmoid. Explain why tanh(x) may be better and show plots. Hint: what is the derivative of tanh(x)?



Tanh(x) function is potentially better for two reasons:

a) The output ranges within range (-1,1) which makes more sense and avoid biased gradient, whereas Sigmoid only output within (0,1). It addresses the zigzag issue due to non-zero mean activation function.
b) The gradients are steeper, thus stronger in training. It could enable faster training rate since the derivative of Tanh is generally larger than that of Sigmoid.

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


Prediction: 8 confidence=0.88
Loss
10^7
10^6
10^5
10^4
0 20 40 60 80 100 120 140 160 180
Accuracy: 87.44%
100
80
60
40
20
0
0 2 4 6 8 10 12 14 16 18

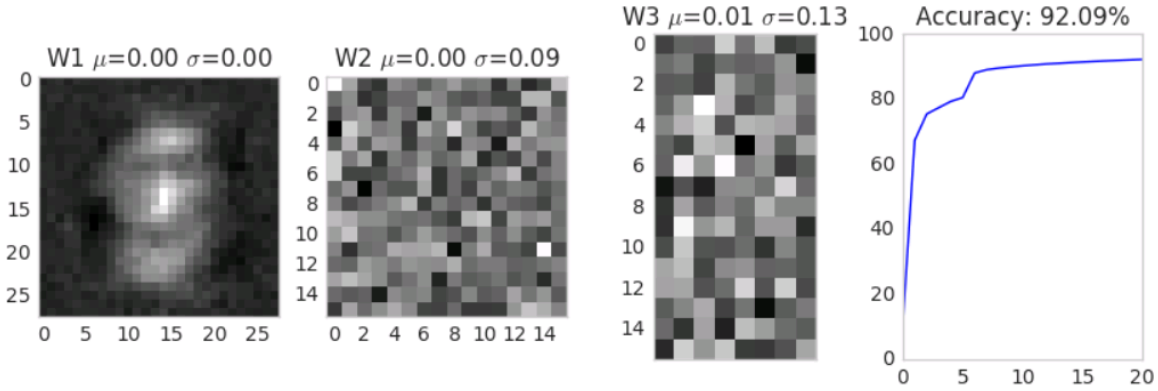
(4) **ReLU:** Implement rectified linear units and justify why they may be better. Show plots. Hint: what is the derivative of relu(x)? Did any of your neurons "die"? What do dead neurons look like in the visualizations? How can we "fix" dead neurons?

ReLU may be better for

a) No vanishing gradient. As long as it's positive it has gradient.
b) Fast calculation. Compared with exponential calculation.
c) Useless neurons die out. (this may have some negative effects though, if not properly initialized)

Yeah there're dead neurons. They are totally black.

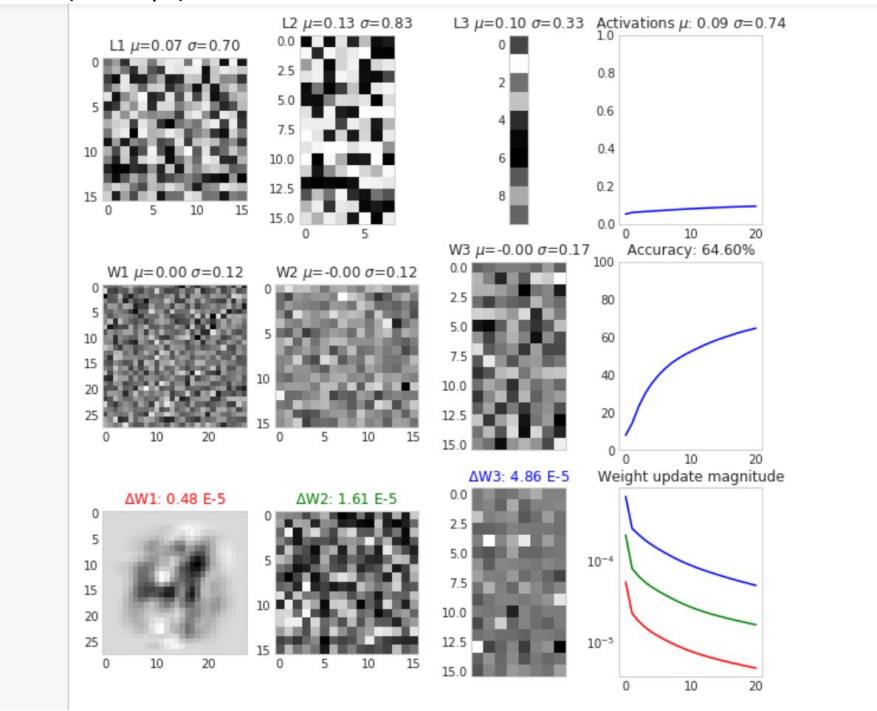
Fixing could be Leaky_ReLU, enabling the negative-valued to come back with small gradient. Also initialize the neurons to be positive could partly prevent it ahead of time.



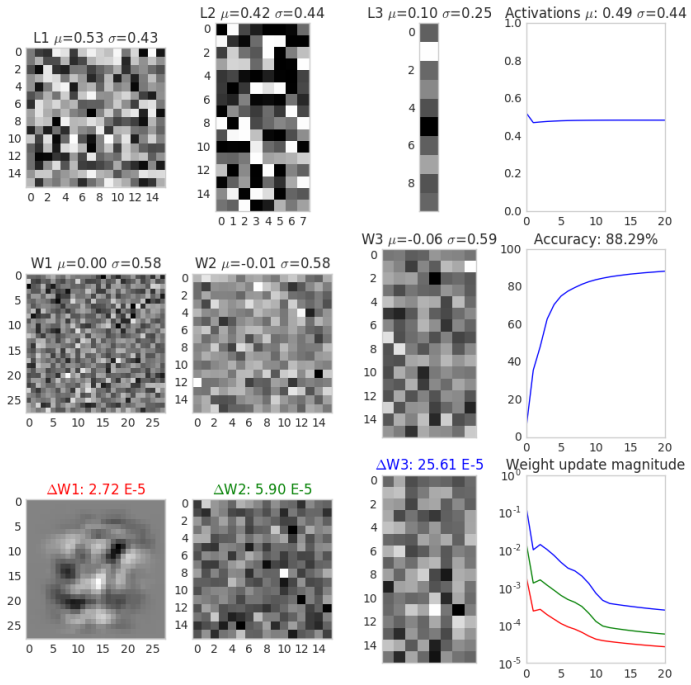
4. Understanding the weights (7 points)

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.

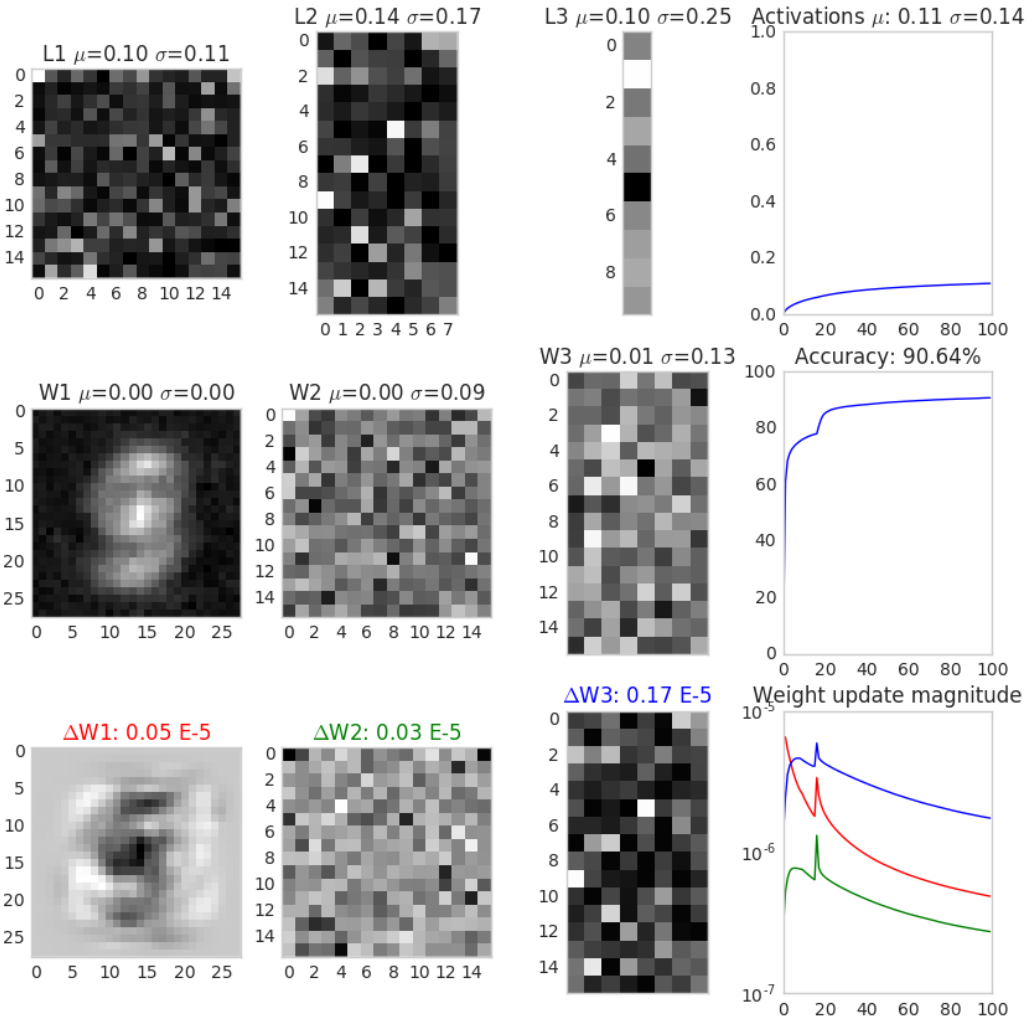
Tanh (200 reps)



Cross Entropy (200 reps)



ReLU (200 reps)



Activation means the average of $L = WX$. This is the input of activation function.
When weights are good they don't show strong patterns (like a certain digit); and they are generally grey.

Activations and weights, they themselves don't mean as much as other plots.

Weight update tells how the training process is going on; the magnitude tells the efficiency of training and can also reflect how aggressive/conservative the current training is. Also comparing the different layer weight update help set learning rate.

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

a) Different weight: ReLU have more neurons black (dead), while for the other two they are generally grey, meaning not much of them saturated.
b) Different activation: for ReLU much of the neurons have small values (black, in dead region) while the other two have black and white (in saturated region) both indicating the activations fall in vanishing gradient
c) Different converging speed: ReLU guarantees faster converging while the other two are slower;
d) For Tanh it is not converging fast but the plateau starts late;
e) The plots show their different distribution of weight.
f) Different of weight update magnitude across layers for ReLU is smallest; for the other two exponential based function the descent in gradient is significant.

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

Weight update gets smaller with the training progresses. Meaning in the later stage there is no much room for improvement.

The magnitudes are different depending on the depth of layer. Shallower layer has smaller updates and deep layers has large updates since it updates

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

For 'stuck' some neurons may die / saturate, and the loss function / accuracy plot may zigzag without much improvement or plateau. For sigmoid the weight picture may seem black and white while for ReLU is may seem much black. May also spot some signs of overfitting/ too large learning rate.

Final state the weight updates are really small and not improving; the accuracy is high and loss function low; the weights updates are small; the loss function / accuracy plot plateau;

(could you better specify what kind of stuck that is? A local optimal?)

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

For certain type of network some activation and weight settings are preferred; for most of the cases, ReLU (Leaky_ReLU) is preferred (refer to problem 5 for details). And for weight initialization the normalized orthogonal is preferred. (Discussed in part 5)

And the activations (calculated form weight) are updated with the update of weights. This partially depend on initialization. (initialization kind of determines the final stage of training)

For example, initializing in the meaningful range (discussed in 2) make the weight training more efficient.

See details in part 5.

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.

1. Activation Function & Loss Function

Activation function for each layer is set to Leaky_Relu, Leaky_Relu and Softmax independently and loss function cross entropy, since Leaky_Relu ensures fast convergence and prevents neural saturation or death, while Softmax-cross entropy for last layer is generally better for classification jobs.

2. Initialization

First thing to consider is initialization. As claimed by Mishkin & Matas (2015), initialization is very important in deep network models' training (especially when the network has ≥16 layers). It determines the speed of convergence, and also, at least to some degree, determines the final state of a trained model. In this part I compared the uniform random method (Glorot & Bengio, 2010), the popular Gaussian distribution initialization, and **Layer-sequential unit-variance** (LSUV) initialization method (Mishkin & Matas, 2015), and turns out the LSUV method works best.

This method first generates a Gaussian random matrix, then use SVD to decompose it into matrices with orthogonal columns; and normalize the matrix with the variance of weight (by doing one pass forward).

$$W_L = \frac{W_L}{\sqrt{Var(LL)}}$$

Note that, it only works for linear activation functions since only for linear activation functions the linear transformation changes the scale linearly.

The orthogonal initialization makes sure the columns are learning different info in the initial steps, while normalization scales it to the adequate range.

3. Momentum

Momentum can speed up the converging of deep nets by using $\sqrt{R}$-times fewer iterations to reach same level of accuracy, where R is the condition number of the curvature at the minimum (Glorot et al., 2010).

In this part I used **Classical Momentum** (CM) method, which is expressed:

$$v_{t+1} = \mu v_t + \epsilon \nabla f(\theta_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

and after several experiments the momentum factor μ is set to 0.75 for optimal performance. (actually I didn't run many experiments here, this number comes more from previous researches)

4. Learning Rate

Learning rate is normally set to a constant, which may not apply to this case where the layers have different functions and scales (also because of the gradient decay across layers).

Also, learning rate of the general model should change with the training iterates more (He et al., 2016), since the update of weight is different in different stage of training.

After running multiple experiments, the learning rate is set for each layer independently:

$$lr_1 = 1.2 * 10^{-5}$$

$$lr_2 = 1.2 * 10^{-5}$$

$$lr_3 = 1 * 10^{-6}$$

For the decay function, after observing the trend, a hypothesis is made that learning rate should increase and a more aggressive strategy is to be adopted in later stage of training. This is proved by experiment. Although this is not often the case, but an increasing learning rate works better here.

(I only experimented with linear decay functions, but I believe a non-linear decay function may perform even better. Since the accuracy already hits 100% I stopped experiments, but I left the space in code for more improvement.)

And the decay rate is set to $-2 * 10^{-4}$.(negative means increasing)

References

[1] Glorot, X., & Bengio, Y. (2010, March). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics (pp. 249-256).

[2] Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013, February). On the importance of initialization and momentum in deep learning. In International conference on machine learning (pp. 1139-1147).

[3] Mishkin, D., & Matas, J. (2015). All you need is a good init. arXiv preprint arXiv:1511.06422.

[4] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

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.

Random result:

In the random result gives about 21.57% accuracy rate, which makes sense.

It must be higher than 10% since the network can pick up some patterns unobservable for human being; and 20% makes sense.

We see the loss function is decreasing at first, however, when it reaches certain number of replications it is no longer improving. The most likely explanation is, my neural network doesn't have the capacity to remember the whole set; (although there's a slim possibility that only exists in theory: it's in a local minimum).

For some neural nets if we train it with more (like 10,000) replications it potentially can remember the whole training dataset (the degree of freedom is enough). In this case we don't want to train it with so many replications.

If we add more layer into the net or add more neurons in each layer, we may observe the random set converge fast as well; then we may have to reduce the number of layers or neurons.