Stat 6910, Section 002
Deep Learning: Theory and Applications
Spring 2019

Homework 4

Jared Hansen

Due: 8:00 AM, Monday 03/04/2019

A-number: `A01439768`

e-mail: `jrdhansen@gmail.com`

Homework IV

STAT 6910/7810-002 - Spring semester 2019

Due: Friday, March 1, 2019 - 5:00 PM

Please put all relevant files & solutions into a single folder titled `<lastname and initials>_assignment4` and then zip that folder into a single zip file titled `<lastname and initials>_assignment4.zip`, e.g. for a student named Tom Marvolo Riddle, `riddletm_assignment4.zip`. Include a single PDF titled `<lastname and initals>_assignment4.pdf` and any Python scripts specified. Any requested plots should be sufficiently labeled for full points.

Unless otherwise stated, programming assignments should use built-in functions in Python, Tensorflow, and PyTorch. In general, you may use the scipy stack ; however, exercises are designed to emphasize the nuances of machine learning and deep learning algorithms - if a function exists that trivially solves an entire problem, please consult with the TA before using it.

Problem 1

1. It can be difficult at first to remember the respective roles of the ys and the as for cross-entropy. It's easy to get confused about whether the right form is $-[yln a + (1-y)ln(1-a)]$ or $-[aln y + (1-a)ln(1-y)]$. What happens to the second of these expressions when y=0 or 1? Does this problem afflict the first expression? Why or why not? (Nielsen book, chapter 3)

- First, let's recall that $ln(1) = 0$ and $ln(0)$ is undefined. More exactly, $\Bigg[\lim_{x\to\infty}\big[ln(x)\big]\Bigg] \to -\infty$, so for practical purposes we might say that $\big[ln(0) \text{``} = \text{''} -\infty\big]$, understanding that we mean the limit of this expression approaches $-\infty$.
- <u>Second expression when $y = 0$:</u> $-\Bigg[aln(0) + (1-a)ln(1-0)\Bigg] = -\Bigg[a(-\infty) + (1-a)(0)\Bigg] = \infty$
  A cost function that can take on the value ∞ (is undefined) for a valid input in (predicted) response is not permissible, and wouldn't be helpful in solving an optimization problem.
- <u>Second expression when $y = 1$:</u> $-\Bigg[aln(1) + (1-1)ln(1-a)\Bigg] = -\Bigg[0 + (1-a)ln(1-1)\Bigg] = \infty$
  Again, a cost function that can take on the value ∞ (is undefined) for a valid input in (predicted) response is not permissible, and wouldn't be helpful in solving an optimization problem.
- <u>First expression when $y = 0$:</u> $-\Bigg[(0)ln(a) + (1-0)ln(1-a)\Bigg] = -\Bigg[0 + ln(1-a)\Bigg] = -ln(1-a)$
  This gives a positive, real value of the cost function as long as $0 \le a < 1$. (Having a positive, real value is a necessary characteristic of outputs of a cost function.)
- <u>First expression when $y = 1$:</u> $-\Bigg[(1)ln(a) + (1-1)ln(1-a)\Bigg] = -\Bigg[ln(a) + 0\Bigg] = -ln(a)$
  Again, this gives a positive, real value of the cost function as long as $0 < a \le 1$. (Having a positive, real value is a necessary characteristic of outputs of a cost function.)
- As we can see, the first expression does not have the same problem of realizing not-allowed cost function values (undefined, or $-\infty$), <u>so long as</u> we put restrictions on a. If we have activations a such that $0 < a < 1$, the cross-entropy cost function can handle (predicted) responses of $y = 0$ and $y = 1$. The second expression cannot do this, regardless of restrictions we might put on a.
- The fact that activations must be bounded by $(0, 1)$ explains why the sigmoid activation function (at least in the output layer) goes so nicely with the cross-entropy cost function. The sigmoid activation function will produce activations such that $0 < a < 1$.

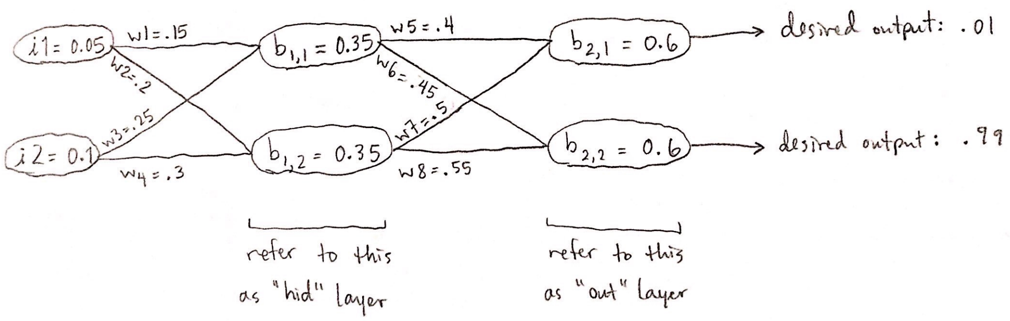
2. Show that the cross-entropy is still minimized when $\sigma(z) = y$ for all training inputs (i.e. even when $y \in (0,1)$). When this is the case the cross-entropy has the value: $C = -\frac{1}{n}\sum_x [ylny + (1-y)ln(1-y)]$ (Nielsen book, chapter 3)

- We will show that this is the true for one training example $\boldsymbol{x}$, and then generalize it to a training data set made up of many $\boldsymbol{x}_i$.
- First let's take the derivative (gradient) of the cross-entropy cost function (for a single training input) wrt a:
$$\left[C := -\big[ylna + (1-y)ln(1-a)\big]\right] \longrightarrow \frac{dC}{da} = -\left[\frac{y}{a} - \frac{1-y}{1-a}\right] = -\left[\frac{(1-a)}{(1-a)}\frac{y}{a} - \frac{1-y}{1-a}\frac{(a)}{(a)}\right] = -\left[\frac{y - ay - a + ay}{a - a^2}\right] = -\left[\frac{y-a}{a-a^2}\right] = \frac{dC}{da}$$
- Now let's set $\frac{dC}{da} = 0$ and solve:
$$\left[\frac{dC}{da} = 0\right] \longrightarrow -\left[\frac{y-a}{a-a^2}\right] = 0 \longrightarrow -\frac{(a-a^2)}{(1)}\left[\frac{y-a}{a-a^2}\right] = 0\frac{(a-a^2)}{(1)} \longrightarrow -\Big[y-a\Big] = 0 \longrightarrow (-1)\Big[-\big[y-a\big]\Big] = 0(-1) \longrightarrow \Big[y-a\Big] = 0 \longrightarrow \boxed{y=a}$$ is a critical point of this function.
- Now let's use the second derivative (Hessian) of the cross-entropy function to show that the solution $y = a$ is specifically a minimum, and not just a generic critical point.
First, let's find the Hessian of C:
$$\frac{dC}{da} = -\left[\frac{y-a}{a-a^2}\right] \longrightarrow \frac{d^2C}{da^2} = \frac{d}{da}\left[\frac{dC}{da}\right] \longrightarrow \frac{d}{da}\left[-\left(\frac{y-a}{a-a^2}\right)\right] = \left[\frac{(a-a^2)(1) - (a-y)(1-2a)}{(a-a^2)^2}\right]$$
$$= \left[\frac{a - a^2 - a + 2a^2 + y - 2ay}{(a-a^2)^2}\right] = \left[\frac{a^2 + y - 2ay}{(a-a^2)^2}\right] = \frac{d^2C}{da^2}$$
- We said $\sigma(z) = y$ in the prompt; since $\big(\sigma(z) = a\big) = y \longrightarrow$ we will evaluate $\frac{d^2C}{da^2}$ at $a = y$, giving:
$$\left[\frac{y^2 + y - 2y^2}{(y-y^2)^2}\right] \longrightarrow \left[\frac{-y^2+y}{(y-y^2)(y-y^2)}\right] \longrightarrow \left[\frac{y(-y+1)}{y(1-y)y(1-y)}\right] \longrightarrow \left[\frac{(-y+1)}{y(1-y)^2}\right]$$
Let's examine the components of this expression:
  - Numerator: $(-y+1)$ is **positive** since it is specified in the prompt that $y \in (0,1)$. Even when y is very close to 1, it still isn't equal to 1, making this quantity greater than 0.
  - Denominator, first term: y is **positive** since it is given that $y \in (0,1)$.
  - Denominator, second term: $(1-y)^2$ is **positive** since we know that $(1-y)$ is positive to begin with, and will be squared, keeping it a quantity greater than 0.

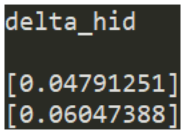
  Because all of these terms are positive, we know that both the numerator and denominator of the second derivative (Hessian) are positive $\implies \nabla^2_{CrossEntropy}$ for a single $\boldsymbol{x}$ is PD (positive definite).
  Since $\nabla^2_{CrossEntropy}$ evaluated at $a = y$ is PD, this implies that the cross-entropy cost function is convex, which then implies that the critical point $a = y$ is a minimizer of the cost function for a single training example.
- We've shown that the cross-entropy cost function is minimized for a single training example when $\big(\sigma(z) = a\big) = y$. We must now show that this is true when summing over all training examples $\boldsymbol{x}_i$. To do this, I will employ some simple logic to generalize. Let's say that $minCE1$ is the minimized cross-entropy cost function for training example 1, $minCE2$ for training examle 2, ..., up to $minCEn$ for the n^{th} training example. We know that the overall cost is just the (scaled by $\frac{1}{n}$) sum of each of these summands $minCEi$. Logically, if we minimize each of these summands $minCEi$ as individual quantities, their (scaled) sum will also be minimized. Therefore, the overall cost of the cross-entropy function is minimized for an entire training set when we let $\big(\sigma(z) = a\big) = y$, showing the desired result.

3. Given the network in Figure 3 below, calculate the derivatives of the cost with respect to the weights and the biases and the backpropagation error equations (i.e. δ^l for each layer l) for the first iteration using the cross-entropy cost function. Initial weights are colored in red, initial biases are colored in orange, the training inputs and desired outputs are in blue. This problem aims to optimize the weights and biases through backpropagation to make the network output the desired results. More specifically, given inputs 0.05 and 0.10, the neural network is supposed to output 0.01 and 0.99 after many iterations.

- I am going to slightly re-draw the network so it looks like what we've been using in the notes:

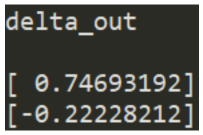


**** <u>NOTE:</u>** I used Python to perform these calculations. I've included my code in the zip file, named *hw4_prob_1_3.py.*
It is very thoroughly commented, and explains how I did the calculations. Since the comments explain how I obtained these calculations, I'll simply be reporting my answers here.
<u>See the code for mathematical explanations commented directly above each line</u>.



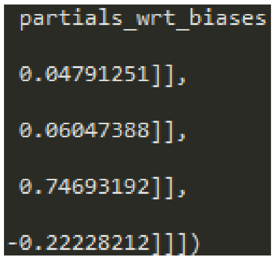
```
delta_hid

[0.04791251]
[0.06047388]
```

Figure 1: Here are the δ^{hid} values, given in order of the spatial position of the neurons in the drawing



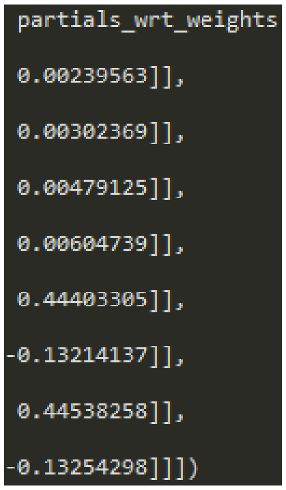
```
delta_out

[ 0.74693192]
[-0.22228212]
```

Figure 2: Here are the δ^{out} values, given in order of the spatial position of the neurons in the drawing



```
 partials_wrt_biases

 0.04791251]],

 0.06047388]],

 0.74693192]],

-0.22228212]]])
```

Figure 3: Here are the $\frac{\partial C}{\partial b_{i,j}}$ values given in this order: $\frac{\partial C}{\partial b_{1,1}}, \frac{\partial C}{\partial b_{1,2}}, \frac{\partial C}{\partial b_{2,1}}, \frac{\partial C}{\partial b_{2,2}}$ (**see drawing for labels**)



```
 partials_wrt_weights

 0.00239563]],

 0.00302369]],

 0.00479125]],

 0.00604739]],

 0.44403305]],

-0.13214137]],

 0.44538258]],

-0.13254298]]])
```

Figure 4: Here are the $\frac{\partial C}{\partial w_k}$ values given in this order: $\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \ldots, \frac{\partial C}{\partial w_8}$ (**see drawing for labels**)

Problem 2

1. Backpropagation with a single modified neuron (Nielsen book, chapter 2)
   Suppose we modify a single neuron in a feedforward network so that the output from the neuron is given by $f(\sum_j w_j x_j + b)$, where f is some function other than the sigmoid. How should we modify the backpropagation algorithm in this case?

   Let's say we've modified the activation function from being σ to being f for the k^{th} neuron in the m^{th} layer. All we need to do is replace $\sigma'(z_k^m)$ with $f'(z_k^m)$ everywhere that $\sigma'(z_k^m)$ appears in the backpropogation algorithm, as well as replacing a_k^m with $f(z_k^m)$. See below.

   - 1. **Input x:** set the activation $\boldsymbol{a}^1$ for the input layer
   - 2. **Feed forward:** for each $l = 2, 3, \ldots, L$, compute $\boldsymbol{z}^l = w^l \boldsymbol{a}^{l-1} + \boldsymbol{b}^l$ and $\boldsymbol{a}^l = \sigma(\boldsymbol{z}^l)$. For the k^{th} neuron in the m^{th} layer, we'll compute c.
   - 3. **Output error δ^L:** compute $\boldsymbol{\delta^L} = \nabla_{\boldsymbol{a}^L} C \odot \sigma'(\boldsymbol{z}^L)$. If it happens to be that the neuron we're modifying is in the output layer, modify the $\sigma'(\boldsymbol{z}^L)$ vector such that we'd have:
     $$\sigma'(\boldsymbol{z}^L) = \begin{bmatrix} \sigma'(z_1^L) \\ \vdots \\ f'(z_k^L) \\ \vdots \\ \sigma'(z_j^L) \end{bmatrix}$$
   - 4. **Backpropogate the error:** for each $l = L-1, L-2, \ldots, 2$
     compute $\boldsymbol{\delta^l} = \left(\left(w^{l+1}\right)^T \boldsymbol{\delta}^{l+1} \right) \odot \sigma'(\boldsymbol{z}^L)$. If it happens to be that the neuron we're modifying is one of the hidden layers (layer m), we'd modify the $\sigma'(\boldsymbol{z}^m)$ vector such that we'd have:
     $$\sigma'(\boldsymbol{z}^m) = \begin{bmatrix} \sigma'(z_1^m) \\ \vdots \\ f'(z_k^m) \\ \vdots \\ \sigma'(z_j^m) \end{bmatrix}$$
   - 5. **Output:** the cost function gradient is $\dfrac{\partial C}{\partial w_{j,k}^l} = \boldsymbol{a}^{l-1} \delta_j^l$ and $\dfrac{\partial C}{\partial b_j^l} = \delta_j^l$

2. Backpropagation with softmax and the log-likelihood cost (Nielsen book, chapter 3)
To apply the backpropagation algorithm for a network containing sigmoid layers to a network with a softmax layer, we need to figure out an expression for the error $\delta_j^L = \partial C/\partial z_j^L$ in the final layer. Show that a suitable expression is: $\delta_j^L = a_j^L - y_j$.

- First, let's write down the softmax activation function: $a_j^L = \dfrac{exp(z_j^L)}{\sum_k exp(z_k^L)}$
- Next, let's write down the log-likelihood cost function: $C = -ln(a_y^L)$ where the y^{th} neuron in the output layer corresponds to the correct label (assuming we're doing classification).
- Now let's slightly rewrite the cost in terms of a_j^L and take the partial derivative of C wrt z_j^L:
$$\left[C = -ln\left(\frac{exp(z_j^L)}{\sum_k exp(z_k^L)}\right)\right] \to \left[C = -ln(exp(z_j^L)) + ln\left(\textstyle\sum_k exp(z_k^L)\right)\right] \to \left[C = -z_j^L + ln\left(\textstyle\sum_k exp(z_k^L)\right)\right]$$
$$\to \left[\frac{\partial C}{\partial z_j^L} = -1 + \frac{exp(z_j^L)}{\sum_k exp(z_k^L)}\right] \to \left[\frac{\partial C}{\partial z_j^L} = a_j^L - 1\right] \to \left[\frac{\partial C}{\partial z_j^L} = a_j^L - y_j\right] \to \boxed{\frac{\partial C}{\partial z_j^L} = \delta_j^L = \left(a_j^L - y_j\right)}$$
We replace the 1 with y_j since we are saying that the j^{th} neuron in the output layer should ideally have an activation of 1 since that is the correct classification. (For all other neurons in the output layer, $y_i = 0$ for $i \neq j$.)

3. Where does the softmax name come from? (Nielsen book, chapter 3)

- As suggested in Nielsen, let's slightly rewrite the softmax function to be $a_j^l = \frac{exp(cz_j^l)}{\sum_k exp(cz_k^l)}$. Then we ask the question: what happens as $c \to \infty$?
- Let's say we have a vector $z = [1, 2, 3]$. We know that $\lim_{c\to\infty} exp(cz_j^l) \to \infty$ so long as z_j^l is positive. (Here we have all positive z_j.)
- For $c = 1$, we'd have:
  $a_1 = \frac{exp(1)}{exp(1) + exp(2) + exp(3)} = 0.0900$
  $a_2 = \frac{exp(2)}{exp(1) + exp(2) + exp(3)} = 0.2447$
  $a_3 = \frac{exp(3)}{exp(1) + exp(2) + exp(3)} = 0.6652$
  We can see that this is somewhat like a max function, placing by far the most weight on the highest value of $z = 3$.
- Now let's see what happens for $c = 10$:
  $a_1 = \frac{exp(1 * 10)}{exp(1 * 10) + exp(2 * 10) + exp(3 * 10)} = 2.06 \times 10^{-9}$
  $a_2 = \frac{exp(2 * 10)}{exp(1 * 10) + exp(2 * 10) + exp(3 * 10)} = 0.000045$
  $a_3 = \frac{exp(3 * 10)}{exp(1 * 10) + exp(2 * 10) + exp(3 * 10)} = 0.999955$
  We can see that this is almost exactly a max function, placing almost all of the most weight on the highest value of $z = 3$.
- Using $c = 1$ (aka the original softmax activation function) does a couple of things for us:
  - It doesn't exclusively give information about the most activated neuron, but may give information about other neurons.
  - Since the most activated neuron will only be very large if weighted sum z at that neuron is much higher than the others, this helps us to adjust weights and biases better. This is true, because the cost function will be penalized more, since the activation doesn't automatically get close to 1 unless the weights and biases are adjusted somewhat close to ideal values.
  - The softmax function with $c = 1$ is differentiable, while a max function is not. This version of the function gives the best combination of qualities of both a max function, as well as a continuous cost function, "softening" maxes and giving some weight to output values other than just the max: **this is where the name "softmax" comes from**. It also allows us to interpret outputs as probabilities since all of the output neuron activations will sum to 1. Overall, this has some very nice properties; a very cool activation function.

4. Show that with the log-likelihood cost and the softmax output layer,

$$\frac{\partial C}{\partial w^L_{jk}} = a^{(L-1)}_k (a^L_j - y_j).$$

- First, let's write down the softmax activation function: $a^L_j = \dfrac{exp(z^L_j)}{\sum_k exp(z^L_k)}$
- Next, let's write down the log-likelihood cost function: $C = -ln(a^L_y)$ where the y^{th} neuron in the output layer corresponds to the correct label (assuming we're doing classification).
- Now let's give the definition of $z^L_j = \sum_i \Big((w_{ij})(a^{L-1}_i) + b_j\Big)$ and take the derivative wrt w^L_{jk} giving: $\left[\dfrac{\partial z^L_j}{w^L_{jk}} = a^{L-1}_k\right]$ where $i = k$ (used for chain rule in taking partial of the second term of the cost function below).
- Now let's slightly rewrite the cost in terms of a^L_j and take the partial derivative of C wrt w^L_{jk}:
  $\left[C = -ln\Big(\dfrac{exp(z^L_j)}{\sum_k exp(z^L_k)}\Big)\right] \rightarrow \left[C = -ln(exp(z^L_j)) + ln\big(\sum_k exp(z^L_k)\big)\right] \rightarrow \left[C = -z^L_j + ln\big(\sum_k exp(z^L_k)\big)\right]$
  $\rightarrow \left[\dfrac{\partial C}{\partial w^L_{jk}} = -(a^{L-1}_k) + \Big(\dfrac{exp(z^L_j)}{\sum_k exp(z^L_k)}\Big)(a^{L-1}_k)\right] \rightarrow \left[\dfrac{\partial C}{\partial w^L_{jk}} = -(a^{L-1}_k) + (a^L_j)(a^{L-1}_k)\right]$
  $\rightarrow \left[\dfrac{\partial C}{\partial w^L_{jk}} = (a^{L-1}_k)(a^L_j - 1)\right] \rightarrow \boxed{\dfrac{\partial C}{\partial w^L_{jk}} = (a^{L-1}_k)(a^L_j - y_j)}$
  We replace the 1 with y_j since we are saying that the j^{th} neuron in the output layer should ideally have an activation of 1 since that is the correct classification. (For all other neurons in the output layer, $y_i = 0$ for $i \neq j$.)

Problem 3

1. Using either Tensorflow or PyTorch, design a single neural network for classifying the MNIST dataset. The neural network must have 2 hidden layers. In other words, your final neural network will have 4 layers total: the input layer, 2 hidden layers, and then the output layer. You will need to select the exact number of nodes in the hidden layers by tuning. However, you shouldn't choose less than 30 nodes or more than 100 nodes in each of the hidden layers. Use dropout and L2 regularization on the weights when training the network. Describe your entire design procedure. In particular, make sure to report the following:

   (a) The dropout rate (i.e. the percentage of nodes dropped out each time), the activation functions in each layer, cost function, weight initialization strategy, and stopping criterion. These do not need to be tuned but you should provide some justification for each choice.

   (b) Learning rate, regularization parameter, mini-batch size, and the number of nodes in each of the hidden layers. Each of these should be tuned in some fashion using a validation data set. Describe your tuning procedure for these parameters.

   (c) The final test error. To get full credit for this problem, you will need to obtain a test accuracy greater than 98% as this was the accuracy obtained using a single hidden layer with regularization. Partial credit will be awarded for designs that do not perform as well.

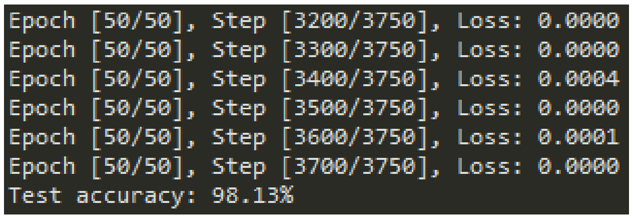
   (d) Include all of your code.

(a)
- Dropout rate: **0.0** (this is empirically what I found to give the highest validation accuracy, which is why I used it). I wasn't expecting this since we talked about how dropout gives better results, but I guess that wasn't the case with this combination of data and specific architecture choices.
- Activation functions: I exclusively used **sigmoid activation functions** throughout the network. I did this because it was what we used for the last assignment in classifying the MNIST data, and it seemed to work pretty well.
- Cost function: I used the **cross-entropy** cost function. This was a natural choice, pairing it with the sigmoid activation functions to avoid saturation and learning slowdown.
- Weight initialization strategy: I used the **default** (didn't manually specify this). I believe the default weight initialization is **Xavier uniform**. I figure that those who write these packages are much better at doing this than I am, so if I don't have to create my own strategy I'd be wise to use theirs.
- Stopping criterion: rather than tuning this specifically, I just chose an arbitrary number of epochs so I'd have one less thing to worry about. I initially tuned hyperparameters using fewer epochs (4 - 8) to save on computation time, and then ramped up to **50 epochs** in order to surpass the required 98% minimum accuracy necessary for full credit.

(b) Each of these should be tuned in some fashion using a validation data set. Describe your tuning procedure for these parameters.
- Learning rate: **0.05**. See below for tuning procedure.
- L2 regularization parameter: **0.0**. See below for tuning procedure.
- Mini-batch size: **16**. There wasn't much tuning involved with this. For the last assignment I used a batch size of 20, and that seemed to give ideal computational time. This go around I adjusted it down to 16 since Dr. Moon mentioned that a batch size of 2^n where $n \in \mathbb{N}$ is ideal. Since 16 is closest to 20, that's what I went with. I also empirically observed that this gave faster training time than other mini-batch sizes.
- Number of nodes in hidden layer 1: **100**. See below for tuning procedure.
- Number of nodes in hidden layer 2: **100**. See below for tuning procedure.
- General tuning procedure: to say the least, my tuning was hackish and was done largely by hand rather than using a more sophisticated method since I'm not familar with any packages

for doing this. Essentially, I just did a rough grid search, one pair of hyperparameters at a time.
First, I split the original MNIST training set (60,000 images) into a new training set of 50,000 images and a validation set of 10,000 images. I'd fit on the new training set (50k images) and validate on the 10k images. Once I'd come up with ideal validation accuracy with the parameters given above, I trained using these parameters on all 60,000 of the original training set images to obtain the 98.13% test accuracy shown below.
I started out with the minimum number of nodes in the hidden layers (30), and then decided that having more nodes would "give more pathways" for classifying training examples. As such, I maxed out both layers at 100 nodes. This also gave better validation accuracy, which is why I changed them in the first place.
Then I worked with learning rate and the L2 parameter. I surprisingly found that any L2 parameter > 0.0 gave worse results than 0.0, so I left it there. A learning rate of 0.05 is what gave the highest validation accuracy once I'd tuned the other parameters.

(c) The final test error. To get full credit for this problem, you will need to obtain a test accuracy greater than 98% as this was the accuracy obtained using a single hidden layer with regularization. Partial credit will be awarded for designs that do not perform as well.



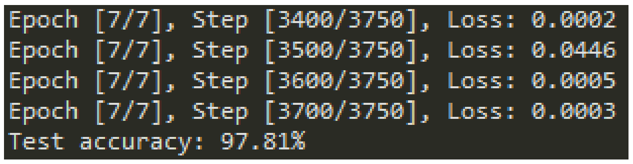
```
Epoch [50/50], Step [3200/3750], Loss: 0.0000
Epoch [50/50], Step [3300/3750], Loss: 0.0000
Epoch [50/50], Step [3400/3750], Loss: 0.0004
Epoch [50/50], Step [3500/3750], Loss: 0.0000
Epoch [50/50], Step [3600/3750], Loss: 0.0001
Epoch [50/50], Step [3700/3750], Loss: 0.0000
Test accuracy: 98.13%
```

Figure 5: I achieved 98.13% test accuracy with the architecture and parameters described above, tuning with the method described above.

(d) Include all of your code.
See *DL_hw4_prob3_FINAL.py*

2. Starting with the network you trained in the previous problem, replace L2 regularization with L1 regularization and tune the regularization parameter as well as the learning rate. Use two initialization strategies: 1) initialize with the weights obtained using L2 regularization and 2) initialize randomly. Which initialization strategy worked the best? Based on your results, which regularization worked best on this data?

   - I implemented both of the weight initialization strategies requested, also implementing L1 regularization with this. The strategy that uses the weights learned in problem 3.1 performed far better, giving 97.81% accuracy, while the other strategy only achieved around 93% accuracy when trained for an equal number of epochs.



```
Epoch [7/7], Step [3400/3750], Loss: 0.0002
Epoch [7/7], Step [3500/3750], Loss: 0.0446
Epoch [7/7], Step [3600/3750], Loss: 0.0005
Epoch [7/7], Step [3700/3750], Loss: 0.0003
Test accuracy: 97.81%
```

Figure 6: I achieved 97.81% test accuracy with L1 regularization and initialization using the weights trained in the Problem 3.1 network.

   - A finding worth noting: as with problem 3.1, I ended up dropping the regularization parameter (this case for L1 instead of L2) down to 0.0 to achieve the best validation accuracy before fitting for test accuracy with both weight initialization strategies.
   - Based on my results, I can't really say which regularization (L1 or L2) worked best on this data. I'd have to add a third option, which is that no regularization gave the best results. I did my best to find both an L1 and an L2 regularization parameter that gave better results than just 0.0, but try as I might, 0.0 outperformed everything else. Even very minimal regularization values, like 1.0×10^{-6} didn't do as well as 0.0.
     My guess is that this may be because the architecture of the network is still fairly simple relative to very deep and complex networks, so maybe regularization doesn't behave the same way (increasing accuracy) here as it would there.
   - See *DL_hw4_prob3_FINAL.py* for my code for this problem. It is clearly labeled, going in the same order as the problem (under the code for 3.1):
     - First, the network whose weights are initialized with the learned weights from problem 3.1 has parameters tuned and then is tested.
     - Then the network whose weights are randomly initialized has parameters tuned and then is tested.

Problem 4

Did you fill out the midterm evaluation? A link will be posted in the assignment. A yes answer gets you about 2 points. No will give you zero. Lying may result in -2 pts for everyone.

- **Yes**, I did the midterm evaluation linked in the assignment page on Canvas.