CSC311 Homework 2

Eric Zhu

12/10/2020

Question 1:

Part a:

The label given by the pair (1,0) is category , i.e., a negative example, so given our two of the data points are (-1, 1) and (3,1), which are both positive examples. Thus, it is impossible to have a positive half space as 1 lies in between -1 and 3.

Part b:

Assuming that we have no bias term our model becomes $z = w_1\psi_1(x) + w_2\psi_2(x)$. Since we want a positive half space, we need $z \geq 0$. Thus, we get the system of equations for the 3 training examples:

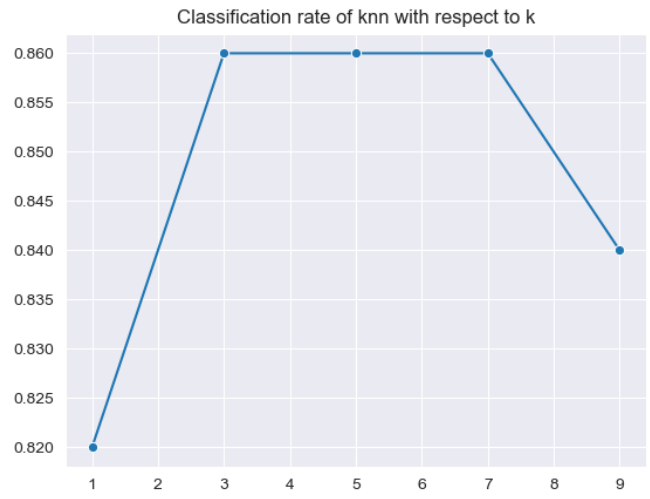
$$\begin{cases} -1w_1 + 1w_2 \geq 0 \\ 1w_1 + 1w_2 < 0 \\ 3w_1 + 9w_2 \geq 0 \end{cases}$$

Thus, we choose the pair of weights (w_1, w_2) to be (-2,1).

Question 2:

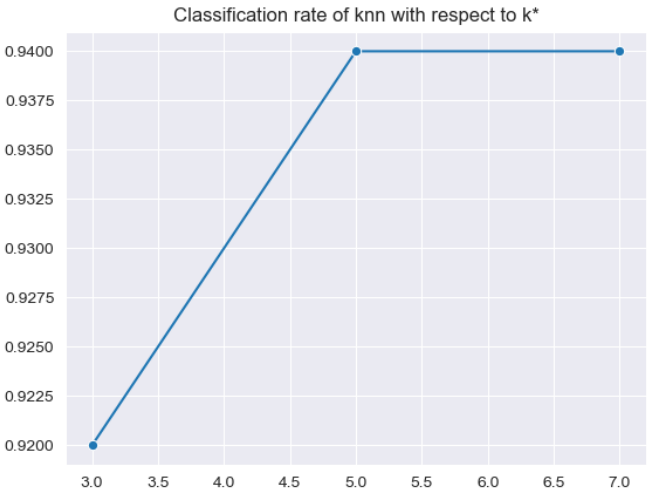
Question 2.1

Part a:



Part b:

We see from part a that the classification rate increases until $k = 3$, where it plateaus at 0.860, and then decreases after $k > 7$. Thus, we choose $k = 5$ because 5+2=7 and 5-2=3, so $k^* + 2$ and $k^* - 2$ should still have high classification rates.



These values of k, i.e., k^* perform quite a bit better than the validation performance, i.e, the classification rates on the test data set are at least 0.9, whereas before, the highest classification rate was 0.860.

Question 2.2

Part b:

For `mnist_train`, the best hyperparameters were:

- $\alpha = 0.01$
- `num_iterations` = 2000
- `weights` = [0,0,...,0] (all weights initialized to 0, and `weights` is $M+1$ dimensional, where M is M in the N by M training input matrix, i.e., `train_inputs`)

Calculating the statistics for averaged cross entropy loss and classification rate we get:

Classification rate:

- Train: 1.0
- Validation: 0.88
- Test: 0.92

Thus the classification errors are:

Classification error:

- Train: 0.0
- Validation: 0.12
- Test: 0.08

Averaged cross entropy loss:

- Train: 0.08103536433274461
- Validation: 0.21870556316704876
- Test: 0.20749746131225105

For `mnist_train_small`, the best hyperparameters were:

- $\alpha = 0.008$
- `num_iterations` = 200
- `weights` = [0,0,...,0] (all weights initialized to 0, and `weights` is $M+1$ dimensional, where M is M in the N by M training input matrix, i.e., `train_inputs`)

Calculating the statistics for averaged cross entropy loss and classification rate we get:

Classification rate:

- Train: 1.0
- Validation: 0.62
- Test: 0.76

Thus the classification errors are:

Classification errors:

- Train: 0.0
- Validation: 0.38
- Test: 0.24

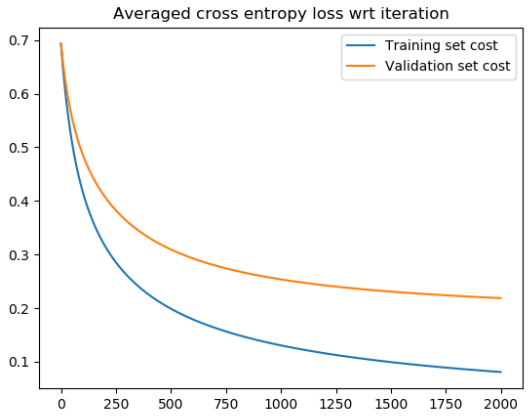
Averaged cross entropy loss:

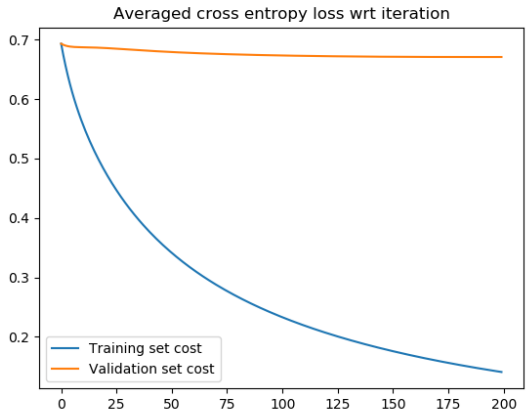
- Train: 0.14083241322039444
- Validation: 0.6707811141287476
- Test: 0.54085967273196

We set the `num_iterations` to 200 because we see that the validation curve starts to curve back up after ~200 iterations, so we need to early stop to help prevent overfitting.

Part c:

The first graph depicts the cross entropy loss on `mnist_train`, while the second is the cross entropy loss on `mnist_train_small`



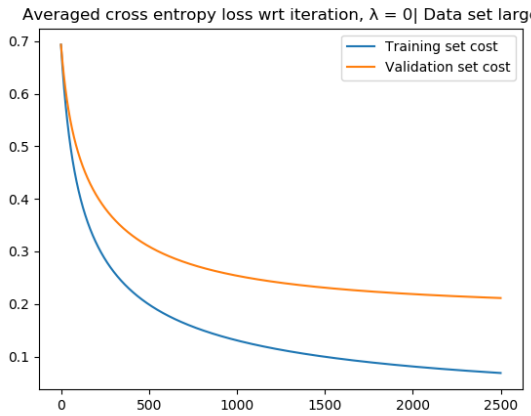


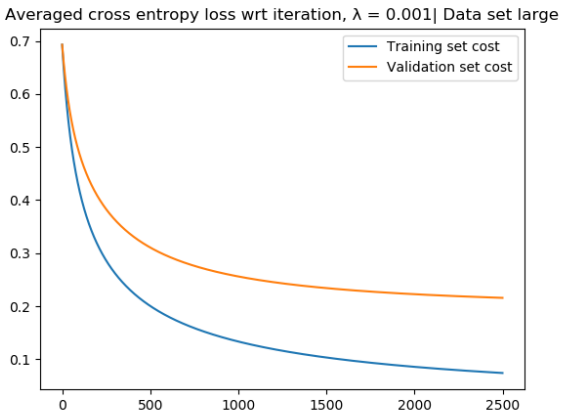
Question 2.3

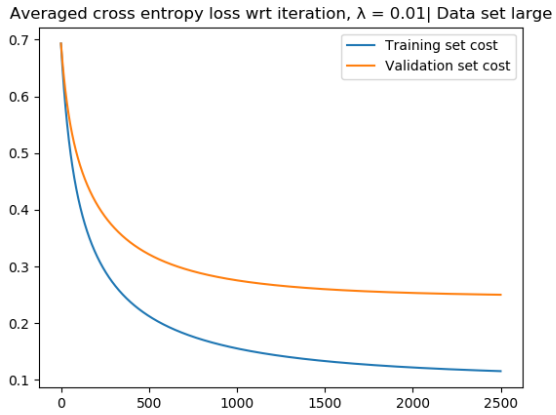
Part b:

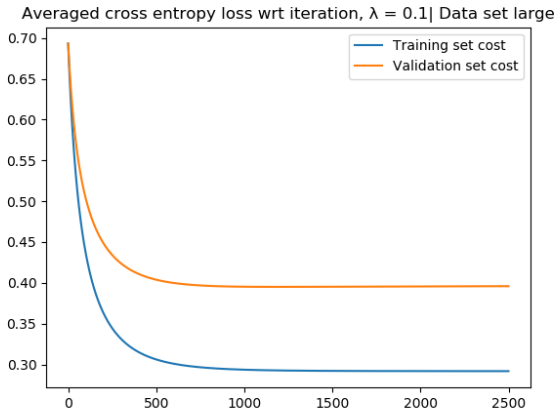
Plots for `mnist_train`:

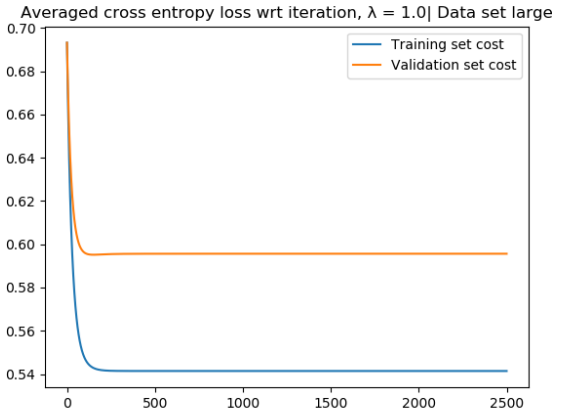
Note that the other hyperparameters (excluding `weight_regularization`,i.e., `lambda`) are: `learning_rate`=0.0075, `num_iterations`=2500











Averaged cross entropy loss over 5 reruns:

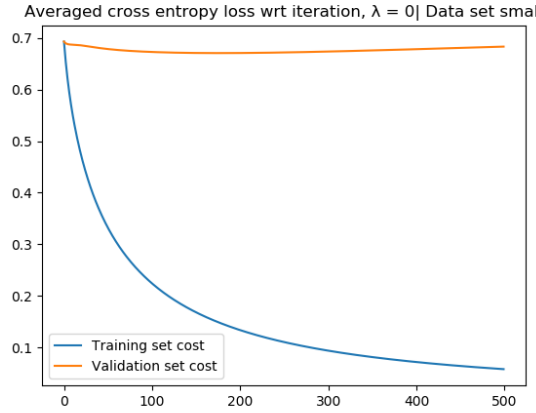
- `lambda` = 0.1: {Train: 0.06861143355669892, Validation: 0.2112125119289888}
- `lambda` = 0.001: {Train: 0.07397081461011173, Validation: 0.21568640117312735}
- `lambda` = 0.01: {Train: 0.11554561710304465, Validation: 0.25026372509009237}
- `lambda` = 0.1: {Train: 0.29181475048772954, Validation: 0.39588409918597556}
- `lambda` = 1.0: {Train: 0.5414379636917224, Validation: 0.5956277543217102}

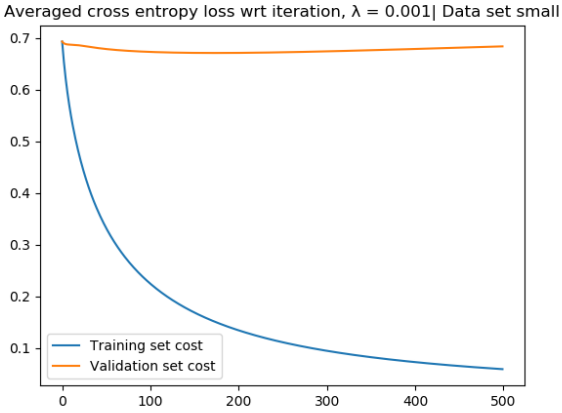
Averaged classification errors over 5 reruns:

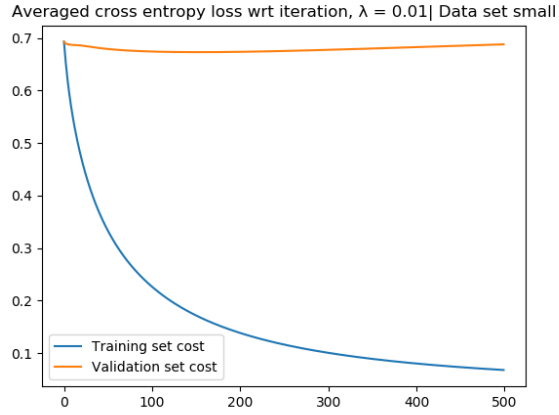
- `lambda` = 0.0: {Train: 0, Validation: 0.12}
- `lambda` = 0.001: {Train: 0, Validation: 0.12}
- `lambda` = 0.01: {Train: 0, Validation: 0.12}
- `lambda` = 0.1: {Train: 0, Validation: 0.1}
- `lambda` = 1.0: {Train: 0.05625, Validation: 0.16}

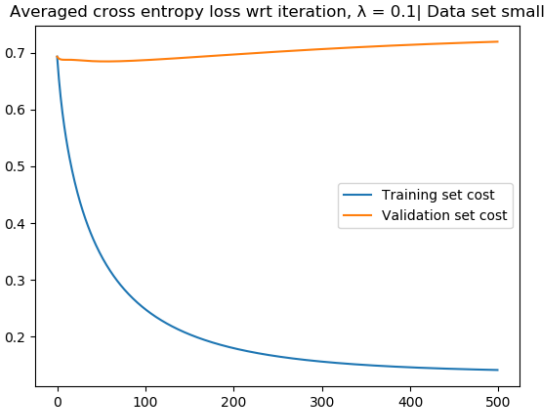
Plots for `mnist_train_small`:

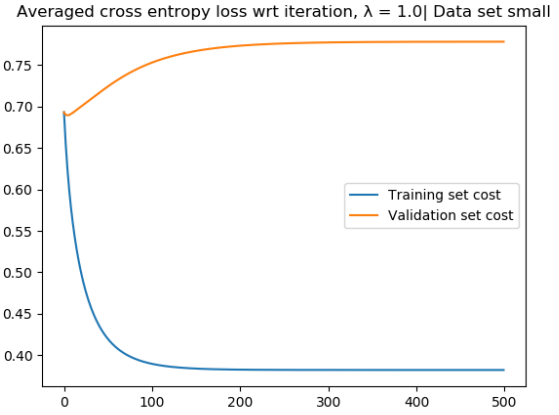
Note that the other hyperparameters (excluding `weight_regularization`,i.e., `lambda`) are: `learning_rate`=0.008, `num_iterations`=500











Averaged cross entropy loss over 5 reruns:

- `lambda` = 0.0: {Train: 0.057706515428323876, Validation: 0.6832976307984405}
- `lambda` = 0.001: {Train: 0.05875395688849485, Validation: 0.6837931346250221}
- `lambda` = 0.01: {Train: 0.06794147497472387, Validation: 0.6880974128678744}
- `lambda` = 0.1: {Train: 0.14119849980018578, Validation: 0.7194906170125598}
- `lambda` = 1.0: {Train: 0.38219295400780207, Validation: 0.7782305298753662}

Averaged classification errors over 5 reruns:

- `lambda` = 0.0: {Train: 0, Validation: 0.36}
- `lambda` = 0.001: {Train: 0, Validation: 0.36}
- `lambda` = 0.01: {Train: 0, Validation: 0.36}
- `lambda` = 0.1: {Train: 0, Validation: 0.36}
- `lambda` = 1.0: {Train: 0, Validation: 0.44}

Part c:

My testing did not conclude a specific shared trend between the two data sets for varying values of `lambda` nor a shared best value for `lambda`. The first data set saw an upwards trend with respect to cross entropy loss, however there was a upwards trend of classification rate (and conversely a downwards trend in classification error). The second data set saw an upwards trend in cross entropy loss as lambda increased, while the classification rate was monotonically decreasing (classification error was monotonically increasing). However regularization is meant to penalize terms that may be causing overfitting by decreasing their weights, although if `lambda` is too big, it can potentially cause the model to underfit as the model will lose some of its capacity to detect more granular features. We do see this a bit with the first data set (`mnist_train`), where the validation classification error was the smallest for `lambda`= 0.1, while for `lambda`= 1.0, we get the highest classification error. Conversely, since the model performed best on `mnist_train` when `lambda`= 0.1, we see that the penalizing effect of the regularization term does help the model generalize.

For `mnist_train` we see that the best `lambda` is 0.1, and the metrics from the test set are as follows:

- Cross entropy: 0.27131847
- Classification rate: 0.92

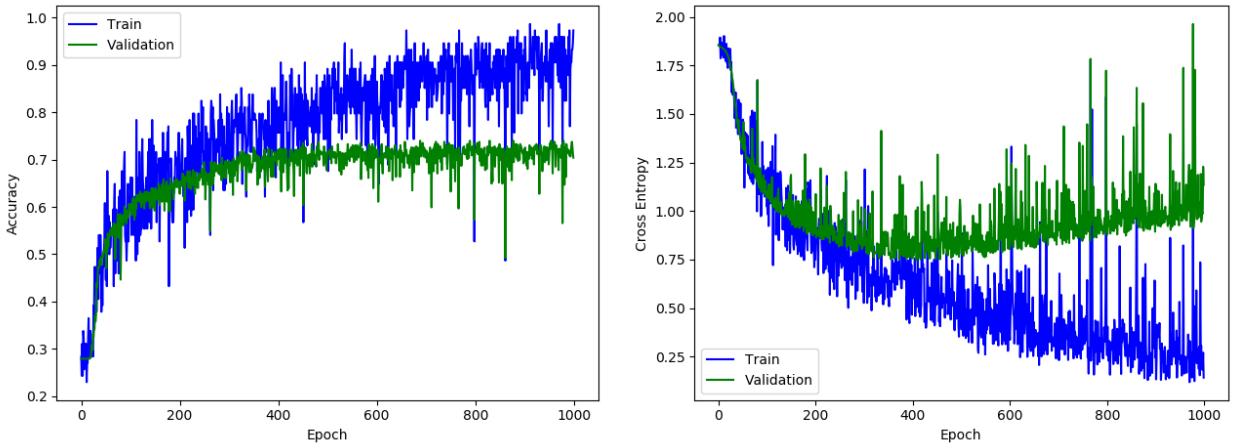
For `mnist_train_small` we see that the best `lambda` is 0.0, and the metrics from the test set are as follows:

- Cross entropy: 0.55924972
- Classification rate: 0.78

Question 3:

Part b:

Recall that the default hyperparameters were `num_hiddens`=[16,32], α=0.01, `num_epochs`=1000, `batch_size`=100.



Model statistics:

Cross entropy:

- Train: 0.29714
- Validation: 1.13450
- Test: 0.97885

Accuracy:

- Train: 0.88234
- Validation: 0.70406
- Test: 0.72727

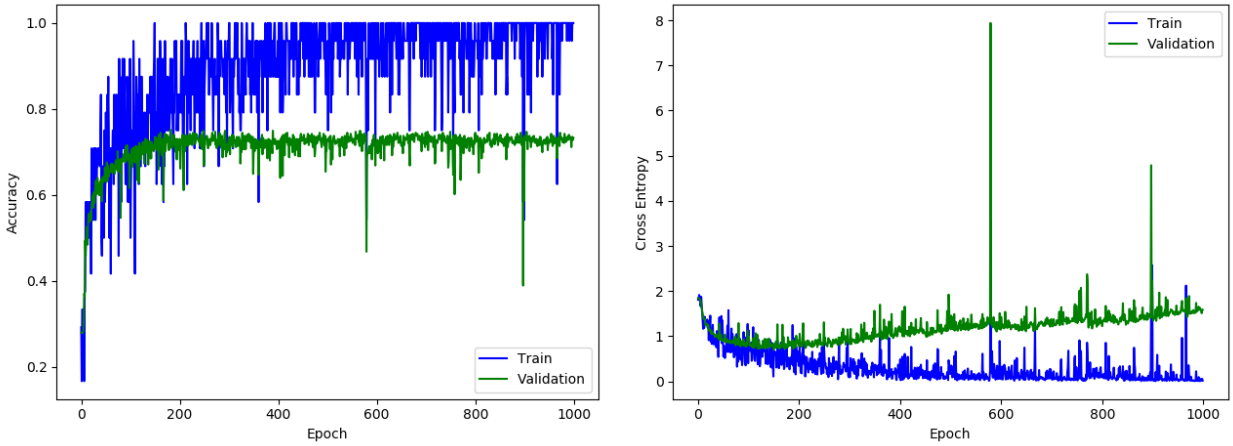
Comments on network's performance on training set vs validation set:

We see that the validation accuracy curve stops rising after about 400 epochs, while the training accuracy curve keeps on rising, while starts plateauing around 800 epochs (it doesn't fully plateau). With regards to the cross entropy loss plot, we see that after about epoch 300, the validation curve starts rising, i.e., the trend for is that with each successive epoch after about epoch 300, there is more cross entropy loss. In contrast, training curve's trend is to keep decreasing in cross entropy loss as we increase in epochs. So it seems by the validation cross entropy curve, that the model beings to overfit after about epoch 300 as the loss increases.

Part c:

Greatly increasing the learning rate α to a high value such as 1 causes problems with convergence. In extreme cases, such as 1, the model will not converge, i.e., there is significant "bouncing" around the loss function as we saw in the optimization tutorial. The cross entropy curve for large values of α resembles random noise distributed around some mean, which appears to be the validation curve that appears to become a line. With large α, there the averaged cross entropy is generally large (~1.7 for α=0.9 on the validation curve) and both the test accuracy and validation accuracy are low compared to sensible values of α, e.g., when α=0.9, both test and validation accuracy are around 0.3. With smaller values of α, such as 0.01, we see that the model takes longer to converge, and depending on the other hyperparameters, both the cross entropy and accuracy curves appear to be less steep.

For varying mini-batch sizes, we see that for large mini-batch sizes such as 100, the convergence of the model is generally have slower convergence than smaller mini-batch sizes such as 25. In other words, the trend is that larger batch sizes lead to slower convergence both with the cross entropy curve and accuracy (percent-correct). Although, we see more variance in the train accuracy values between batches when smaller batch sizes are used, and conversely, we see less variance in the cross entropy values. Below are the accuracy plot and cross entropy plot for default hyperparameters but with `batch_size`=25.



Clearly, when compared to the graphs of the training process using default hyperparameters, we see that a `batch_size`=25 results in far more variance in the accuracy plot, while there is little variance in the cross entropy plot. Both plots show quick convergence when compared to the default hyperparameters. It is also worth noting that a smaller `batch_size` (keeping all else constant) seems to allow for marginally higher accuracy.

Thus, to choose the hyperparameters we want, we will need to consider our goals. Using a larger learning rate (but still sensible), we will be able to converge faster, which allows us to train models faster. On the other hand, if we want more control over early stopping, we may want a smaller learning rate as we "step" slower through gradient descent, which gives us more options at which epochs we want to stop at. We may want to choose a larger batch size if we want to have less variance in the accuracy curve, converge slower (converge in more epochs), and have the weight updates be based on more data. On the other hand, we may want to have a small batch size if we want more variance in the accuracy curve, less variance in the cross entropy curve, and converge faster (in less epochs) but with more steps per epoch, and if we want to have the weight updates be based on smaller samples of examples.

Part d:

Increasing the hidden units in each layer of the MLP increases the capacity of the model, and can potentially help it learn more "features" from the examples. However, that may cause the model to overfit since the training examples may have unique features not present in the validation or test, i.e., the model cannot generalize well. I tested various values for the hidden units, but in general I discovered that the more hidden units you use, the smaller the batch size you need in order to help avoid overfitting (and a low classification rate). The more hidden units that we use in the model, the faster it seems to converge, i.e., the loss decreases faster, so when using more hidden units in each layer, I found it beneficial to early stop the training. Additionally, adding more capacity does help, for example, my highest test accuracy was achieved using `num_hiddens`=[24,48] with `batch_size`=12, α=0.0075 and 500 epochs (to early stop the training). The test accuracy was 0.78961, which is quite a bit higher than the 0.727 test accuracy obtained with default hyperparameters. So it does seem empirically that using a small batch size with more hidden units helps the model to generalize. Finally, with more hidden units, the model converges much quicker as the training accuracy and validation accuracy rises much faster and the cross entropy loss drops much quicker (the curve is much more steep) compared to a model that uses less hidden units. However, as I mentioned earlier, with more capacity, we are more likely to overfit the model, so if the epoch hyperparameter is not well tuned, then the training cross entropy curve will continue to drop while the validation cross entropy curve will drop quickly then quickly rise again.

Part e:

The following are 8 plots of the examples that the neural network (trained with default hyperparameters) were not confident on. I set the threshold to a probability of 0.4, so those that had a probability of 0.4 or less were considered to be examples that the NN were not confident on. These examples came from the test set. Ultimately, the NN was capable of classifying 25% of them correctly despite low confidence, however this is probably not a hard number, i.e., this percentage will almost certainly change with different reruns, hyperparameters, etc....

Honestly, these examples are pretty ambiguous to even a human (like me!) because human emotions are rather complex, often consisting of multiple emotions; in fact, in my personal experience, it is extremely rare that I experience one emotion at some particular point in time. For example, the 7th example is labeled as category 4, happy, yet, to me, I see happiness and maybe even a bit of forlorn, which could be classified as sadness. So the classification of the 7th example as "neutral" is totally plausible. Thus, I believe that the labels we are assigning these people are not "good" labels as they do not capture many of the other emotions that these people in the examples are feeling, even if the labels are indeed their primary emotions. It makes "sense" that the model can only sometimes classify these low confidence examples correctly as the model can

only perform as well as we teach them to, which is a function of the labels we give them. In other words, if the labels are even possibly ambiguous to a human, it will probably lead to a low confidence classification for the model too, and the top socring class may or may not be the target category.