ECE324 Assignment 3 Qualitative Questions

Devansh Ranade
1004221399

Due: 3 October 2019

Part 1 - PyTorch and the SNC from Assignment 2

1. The tensor object which stores the weights is called `weight`, which is a variable of the `Linear` class within the `torch.nn` module.
   Similarly, the tensor object which stores the bias is called `bias`, which is a variable of the `Linear` class within the `torch.nn` module.

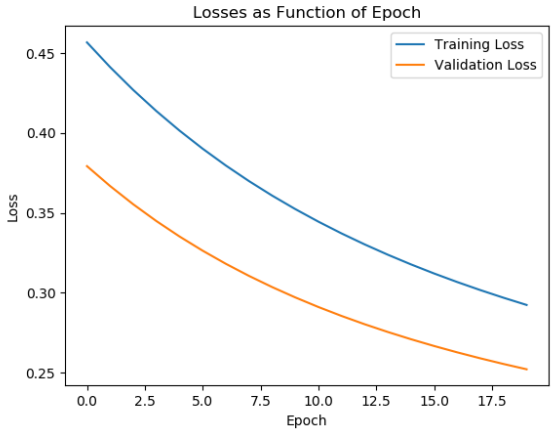
2. In PyTorch, each Tensor has an attribute called `grad` which stores its own gradients. By default, this is set to `None` when the object is created. However, when the object is passed as an input to a loss function, a call of `loss.backward()` will populate the `grad` attribute of each tensor with its own gradient. As such, the tensor object which stores the calculated gradients of the weights is called `grad` and can be found in the SNC class through `self.fc1.weight.grad` anytime after the `loss.backward()` function has been called. The same applies to the bias for the SNC, whose gradient is found through `self.fc1.bias.grad` anytime after the `loss.backward()` function has been called. Note that these gradients are reset to zero at the beginning of each training loop with `optimizer.zero_grad()`.

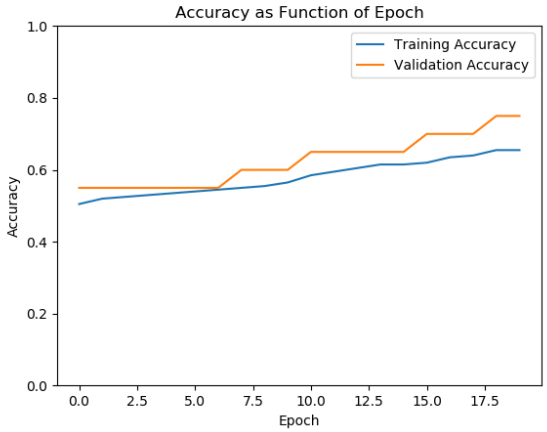
3. The gradients are calculated in the call of `loss.backward()`. This function is called to the loss object. This object knows the function being used to calculate loss. Thus, the gradients are known for each particular loss function and can be stored simply as a function for each type of loss. In this part, we used the `MSELoss` module. Each of the different loss modules in PyTorch would have their own functions stored which could easily compute gradients for each input.

4. See requested plots on next page

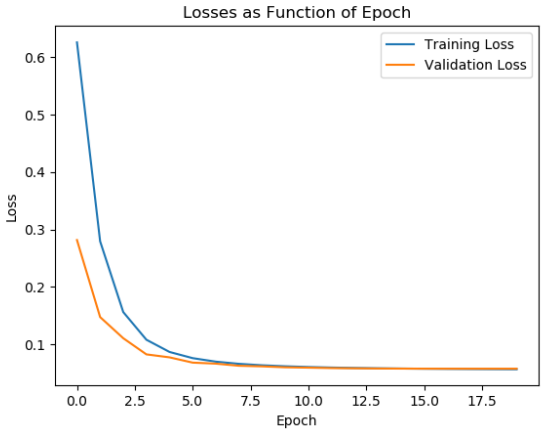
Continued...

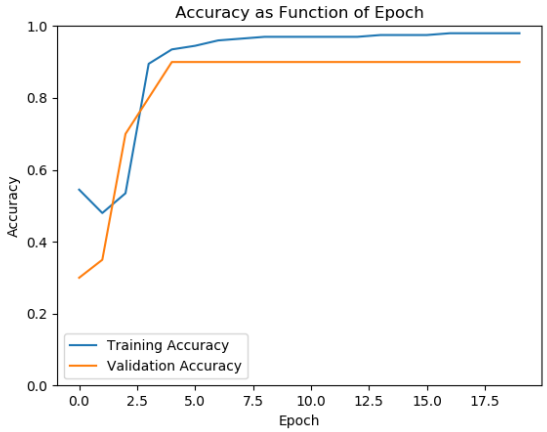
Too Low



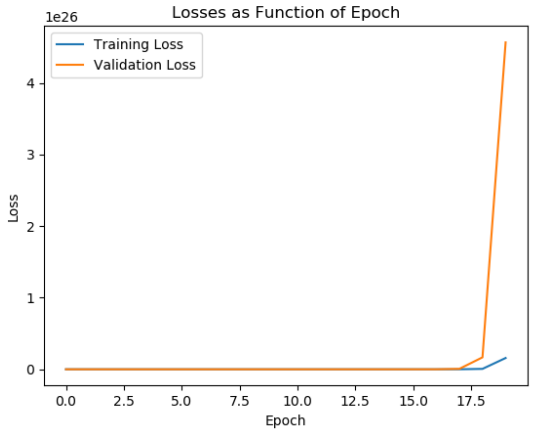


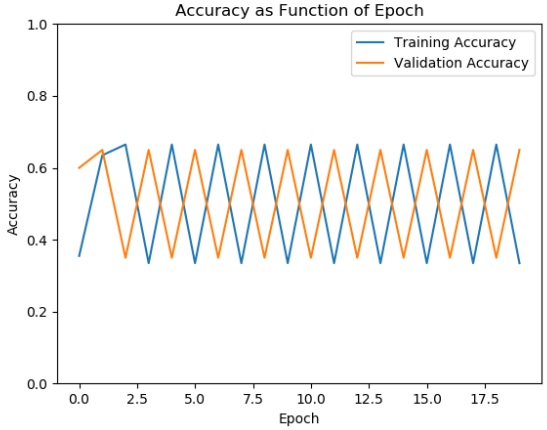
Good





Too High





Part 3 - Data Pre-processing and Visualization

3.2 Understanding the Dataset

1. There are 11687 high income earners in this dataset. There are 37155 low income earners in this dataset.

2. No. There are considerably more low income earners than high income earners (the ratio is more than 3:1). A classifier seeing this dataset is more likely to have biased predictions (more likely to predict low income earners). Specifically, a classifier which always predicts low income earners would be correct $\sim$75% of the time.

3.3 Cleaning

1. Before cleaning, there were 48842. After cleaning, there are 45222 data points. This means we removed 3620 data points in the cleaning process.

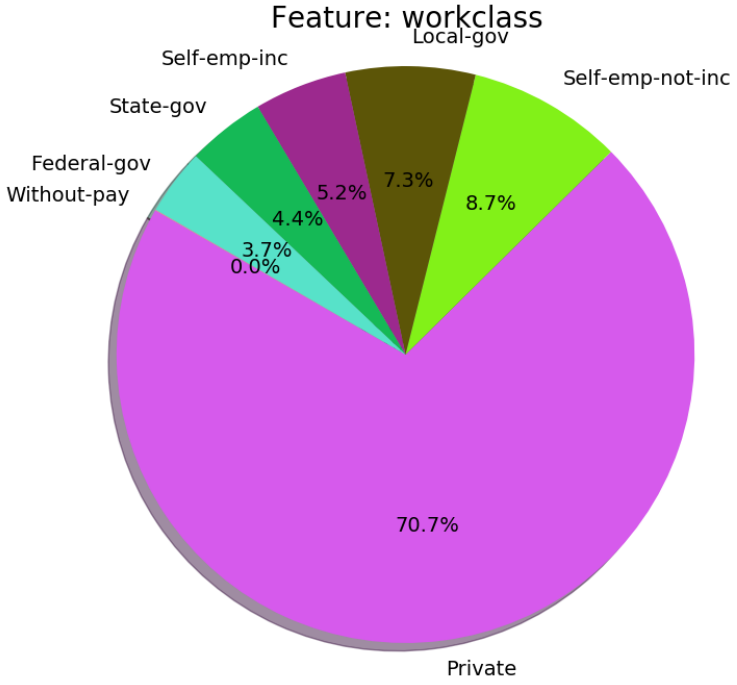
2. I think that is is a reasonable number of samples to throw out, but I don't think this is a reasonable method of choosing which ones to throw out. This process discarded almost 8% of data available, which is reasonable in terms of amount. However, if we analyze the raw data closely, there were almost 3000 samples where the missing information was 'workclass' or 'occupation'. This is comparable to the number of total samples removed. Further, there were no entries in either of these categories which indicated 'unemployed' for *any* of our original samples. As such, by removing all the samples for which we don't have data, we remove a large chunk of people who are likely unemployed. This could misrepresent the populus and cause us to train an inaccurate classifier.

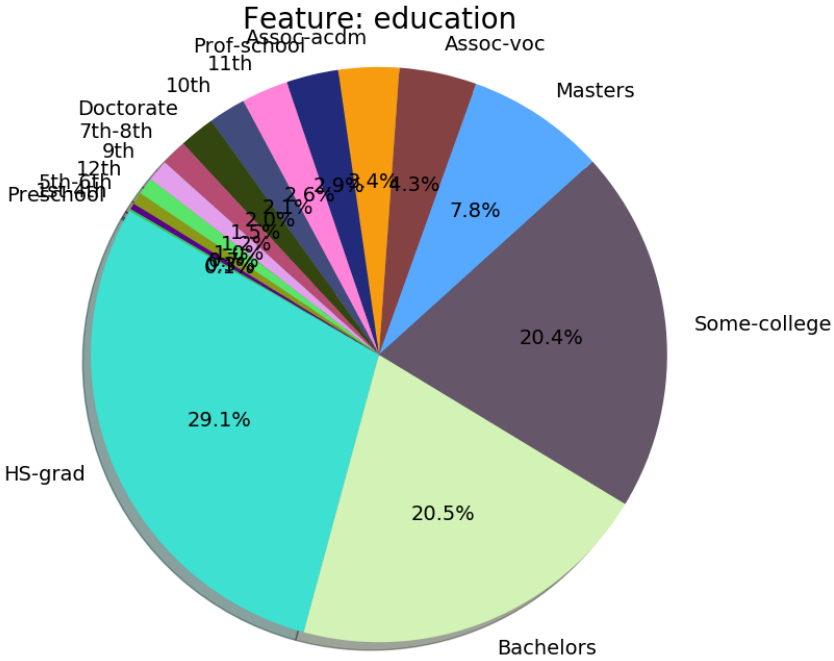
3.4 Balancing the Dataset

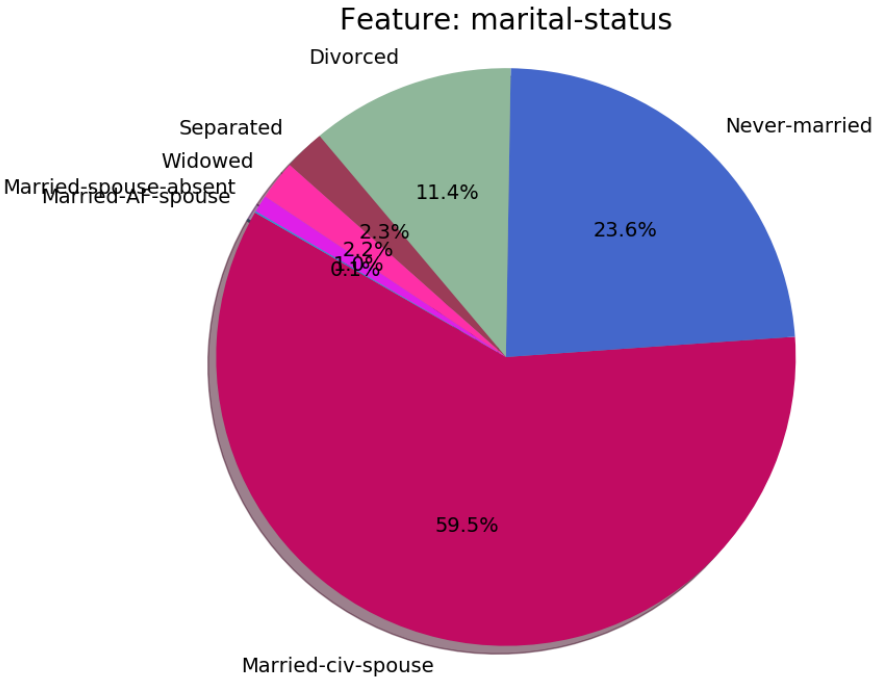
1. The minimum age of the individuals for the dataset is 17. The minimum number of hours worked per week for the dataset is 1. (Note that these numbers represent my chosen sample with `random_value=10`. It could be different from other students.)

Continued...

Categorical Pie Charts







2. Yes. The private sector is over-represented. Also, there is no option in the 'workclass' feature for unemployed. So, the unemployed are grossly under-represented. Those with higher education (Doctorate and above) are also under-represented. Also, married people are very over-represented. Similar claims can be made about other features as well (White people are over-represented, husbands are over-represented, etc.)

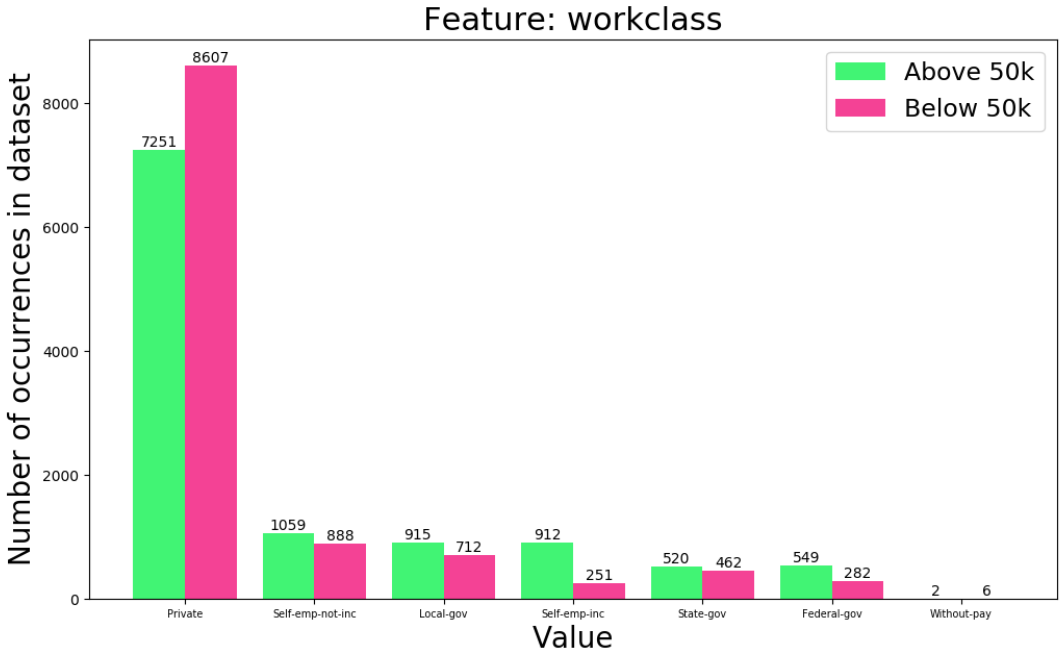
   It can also be argued that, for some of these misrepresentations, they are not equal in terms of samples per class, but they represent the percentage of the populous which belong to that class. While this can explain the low number of samples for some classes, the under-represented classes will still likely have lower per-class accuracy due to the lesser representation (and therefore fewer opportunities for training and weight updates).

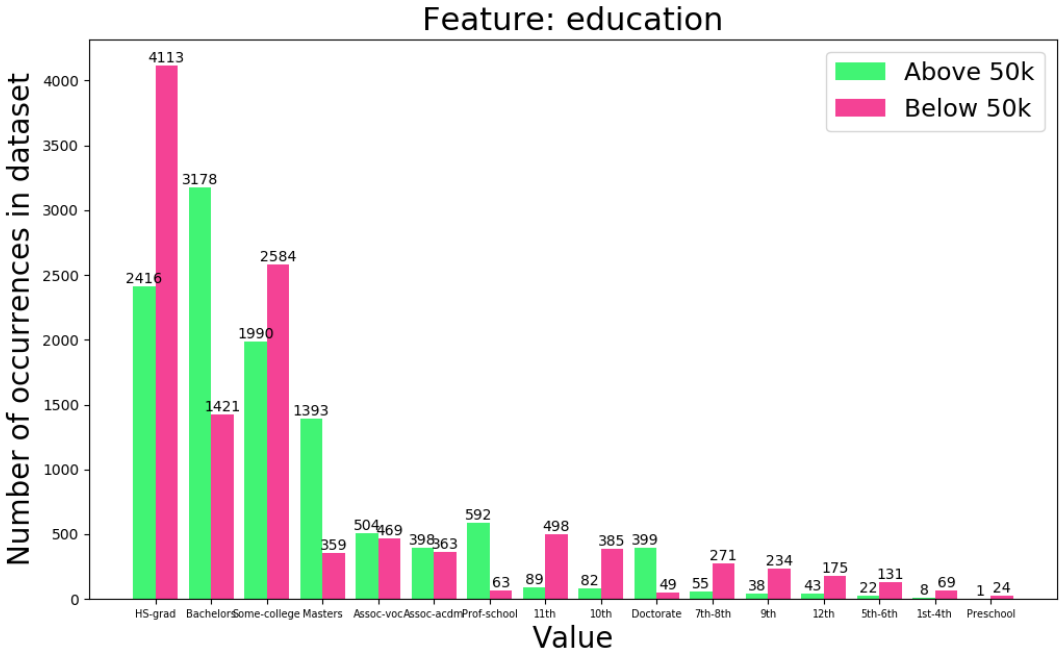
   I believe that this dataset will generalize decently to other races. While there is a decent amount of over-representation and under-representation within the dataset

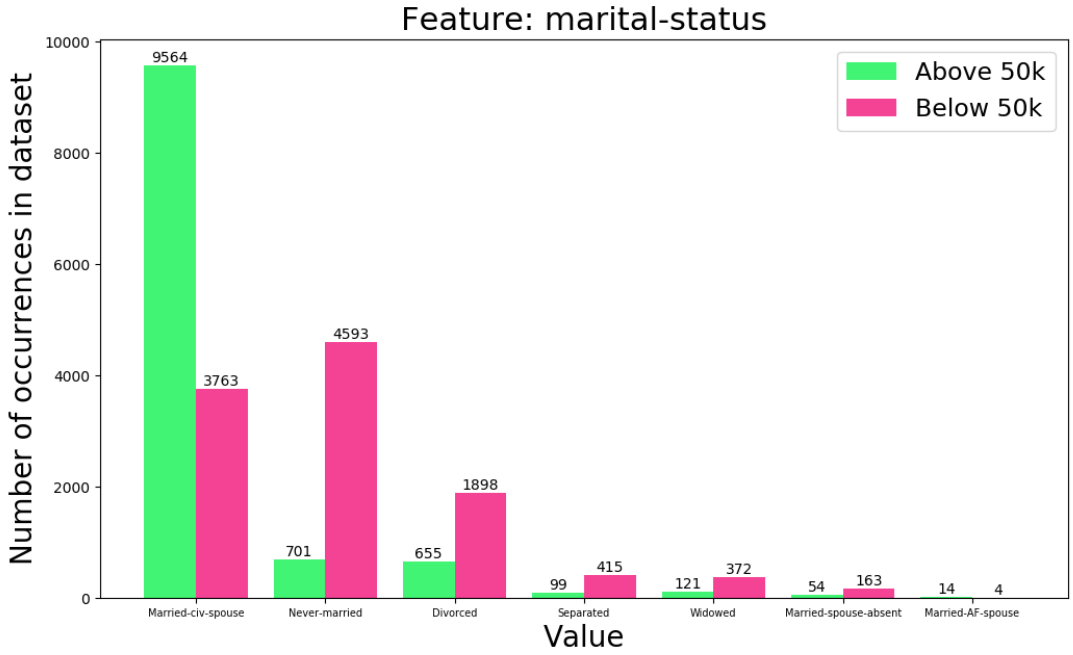
3. Some of the categorical labels are biased towards one class or another. For example, the 'workclass' feature offers various levels of distinction for public sector jobs (federal, local, state) but only gives one option to those working in the private sector. This could place unnecessarily high importance on the type of work done (which is not fairly represented) in classification. Also, there is potentially a case of inputs being repeated (put into the network with double importance) through the 'marital-status' and 'relationship' features. The majority of people will be ('Married-civ-spouse', 'Husband') or ('Married-civ-spouse', 'Wife'). These pieces of information (which largely convey the same information) are being fed into the network twice, which could cause incorrect classification.

Continued...

Categorical Bar Graphs







4. I think that Age, Education, Marital-Status will be the top three features to distinguish between high and low salary earners.

5. I would predict for this person (according to the education feature chart) '>50K' since about two-thirds of people ($\sim \frac{3000}{4500}$ with 'Bachelors' earn '>50K'

6. I would look take a weighted average of their education, marital status, and education.

3.6 Pre-processing

1. Since the weights and biases are numbers and have meaning, the classifier might mistakenly think that the numbers associated with the categorical inputs have some meaning. This could lead to slower training and increases the odds of overfitting (classifier is not able to generalize as well).

2. Since the network parameters will start initialized to some near-zero values, having normalized data allows the network to start in a better location. If the data is not normalized, the network has to spend a bunch of training time getting to the proper subset of the input space before it can start learning to generalize. Further, this practice will help reduce the chance of getting stuck in a local minima, since the network will have all directions to explore when it starts (instead of just towards the cluster of data points in the input space). Overall, while this practice doesn't make a difference in theory (given enough epochs), it allows for faster and better training in practice.

Part 4 - Model Training

4.1 DataSet

1. Shuffling data is a good practice to potential bias which exists in the ordering of data. For example, if one batch was full of high income earners and the next batch was full of low income earners, the parameters of our network might swing back and forth unnecessarily. This would happen during each epoch, since the batches would be fed in the same way each time. By shuffling data, we improve the balance of data inputs in any one mini-batch. This way, the mini-batch is a better approximation of

the entire dataset and SGD would still perform comparably to gradient descent while still maintaining its advantage (speed).

4.3 Model

1. I chose the output of the first layer to be size 30. This is a reasonable number between the input size ($\sim$100) and output size (1). Thirty is a number of neurons which allows some decisions to be made while decreasing the input space as a step towards eventually making it binary. We want to find a good balance between keeping all the information and making an uninformed decision, which I think can be achieved by choosing a layer 1 output size $\sim$ 15-50.

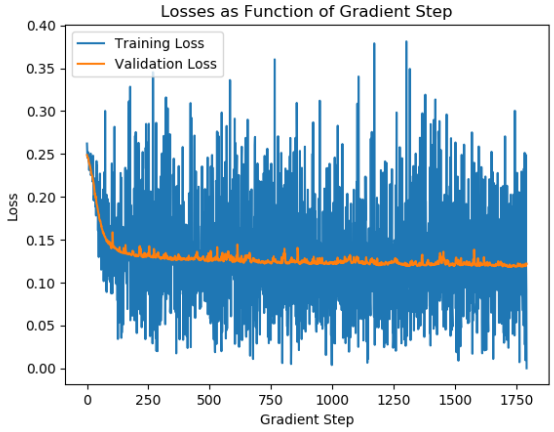
2. This output between zero and one represents the probability that an input belongs in a certain class. For this problem, an output of 1 will represent that the person is a high income earner. An output of 0 will represent that the person is a low income earlier. The output is simply a 'level of confidence' the network has that the particular input belongs to a certain class.

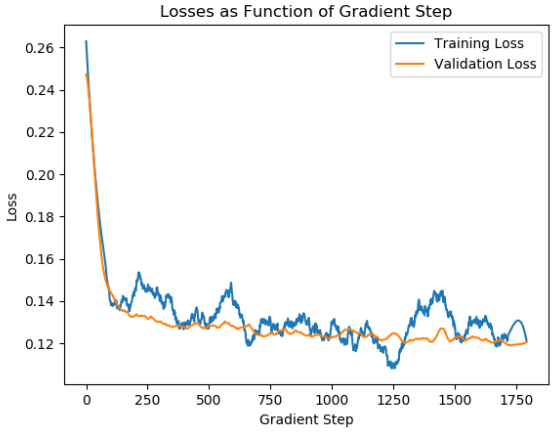
4.6 Validation

1. Please find below the plots of Loss and Accuracy as a Function of Gradient Step. For these graphs, the following set of hyper-parameters were used $\begin{cases} \text{Learning Rate} - & 0.1 \\ \text{Batch Size} - & 10 \\ \text{MLP Hidden Layer Size} - & 30 \end{cases}$

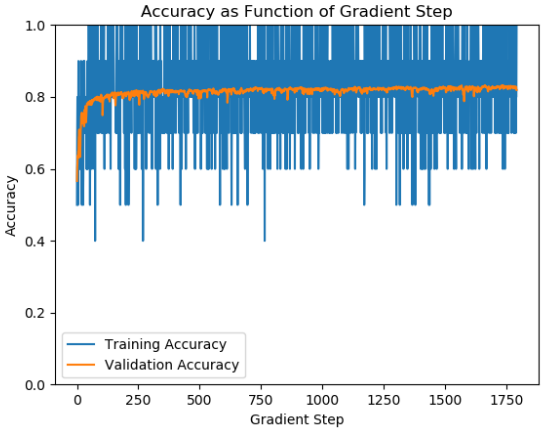
Continued...

Loss (Unsmoothed and Smoothed)





Accuracy (Unsmoothed and Smoothed)



Part 5 - Hyperparameters

5.1 Learning Rate

To be able to see a difference between the learning rates, only two epochs of training were done for each learning rate. (See the required graphs on the next page)

Learning Rate:	0.001	0.01	0.1	1	10	100	1000
Best Validation Accuracy	0.676	0.793	0.814	0.824	0.828	0.497	0.496

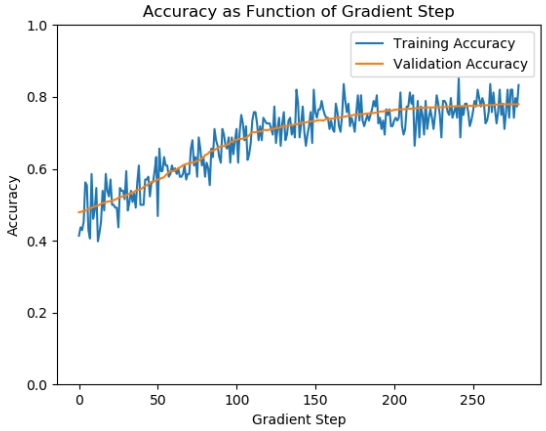
1. For me, most learning rates achieved similar performance (given enough epochs). The best learning rates were those $\sim$1-10, as they were able to peak very early in training (only two or three epochs). From the table above, a learning rate of 10 achieved the best results.

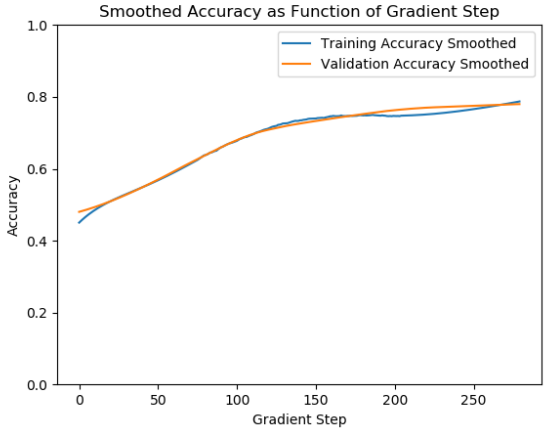
2. If the learning rate is too low, the parameter updates will be very small in magnitude. This means that training will be an unnecessarily slow process. Also, if the parameters get stuck in a local minima, it will be much harder to escape since the parameter updates will always be small in magnitude.

   On the other end of the spectrum, a learning rate too large can cause the model to diverge since it could overstep the minimum on each step. This back and forth will make it harder for the model to converge and get good accuracy. This can be seen in the table above, where the validation accuracy drastically drops (to the point where it's comparable to guessing) for learning rates $\geq$100.

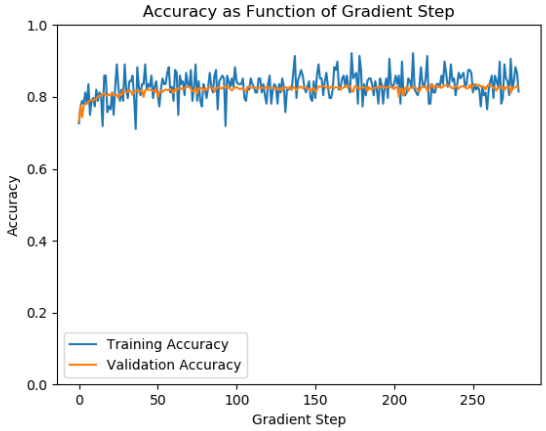
Continued...

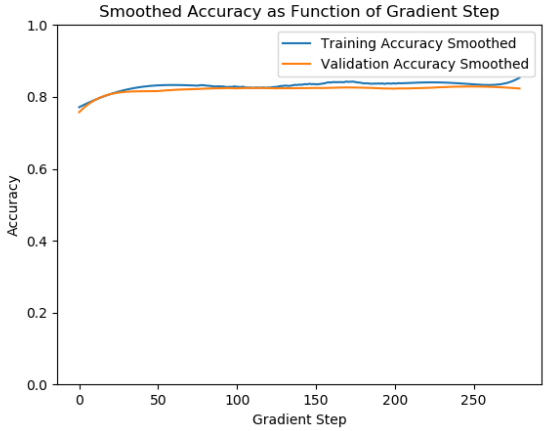
Learning Rate = 0.01



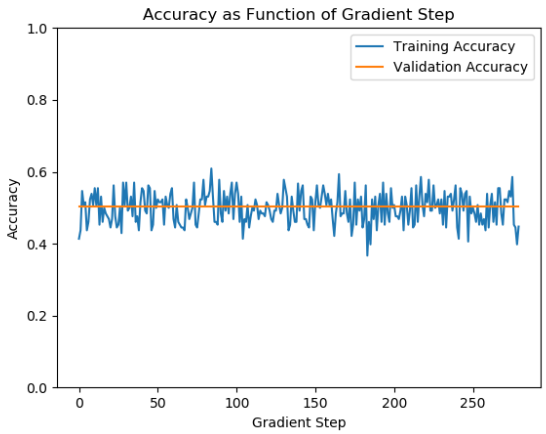


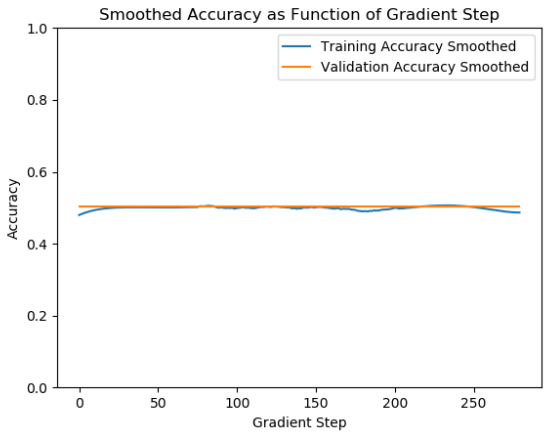
Learning Rate = 1





Learning Rate = 100





5.3 Batch Size

1. The highest validation accuracy happens with batch size 64.

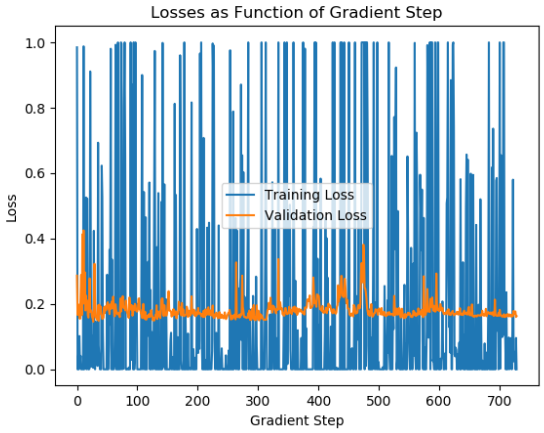
2. A batch size of 17932 performed best in terms of steps. This is because it includes all the information and calculates the full gradient. Each step is in the exact correct direction for each parameter to minimize the objective function.

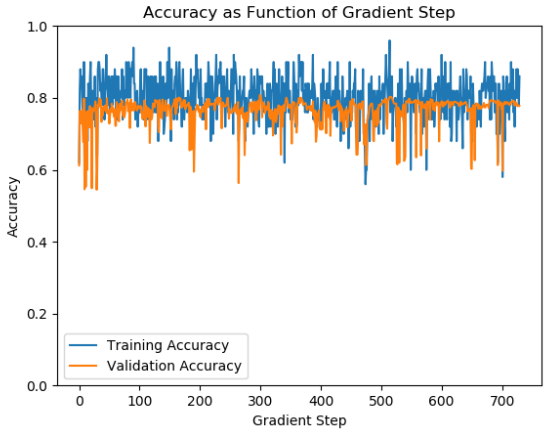
   In terms of time, however, batch size 64 performed best. This is due to the advantage of SGD requiring less computation per step while still decently approximating the whole dataset.

3. Batch size 1 (too small) allows too much fluctuation as the weights get updated and changed at every single outlier processed as well as normal data. Having a batch size of 17932 (too large) is effectively regular gradient descent. It does not allow for the time advantage of SGD due to randomness.

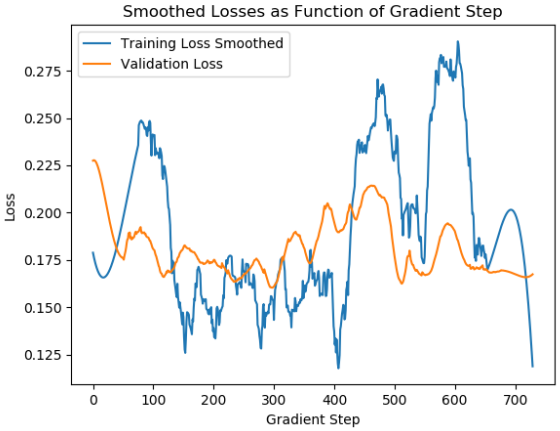
4. The biggest advantage of a small batch size is faster steps and reduced computation per step. Due to the randomness, small batch sizes can still approximate the dataset decently. One must be cautious against using a batch size which is too small though. This would cause lots of fluctuation and longer training time, reducing the advantages of SGD. In general, for decently sized datasets, the batch sizes should probably be around 50-100, or 1-2% of the size of the total dataset (∼100-300 for our particular example).

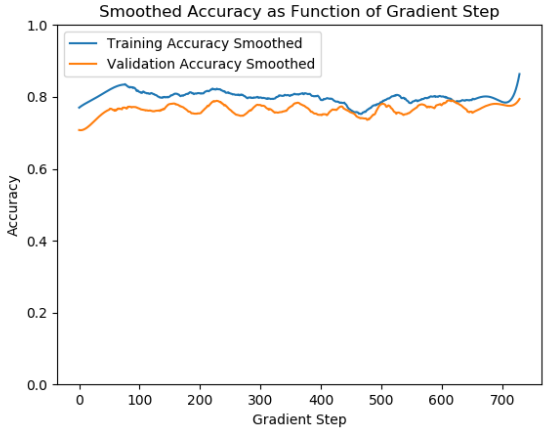
By Steps, Batch Size = 1



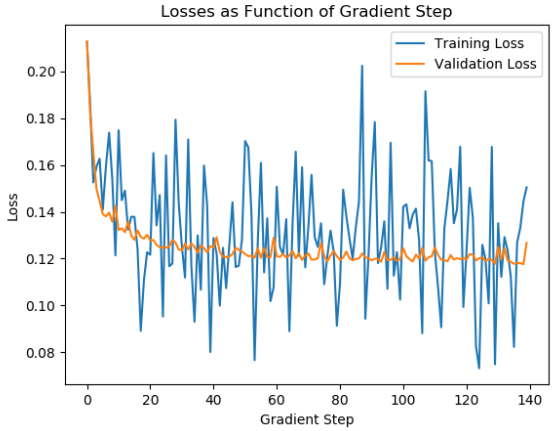


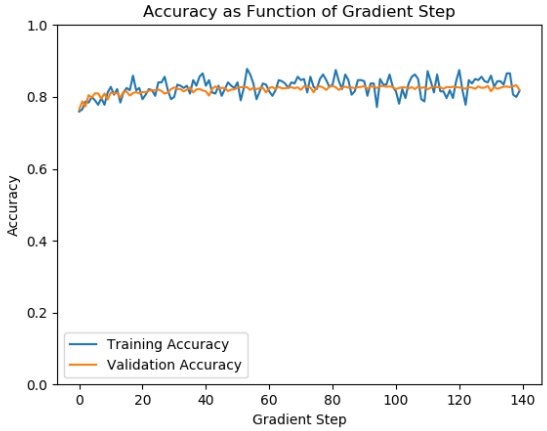
By Steps, Batch Size = 1 (Smoothed)



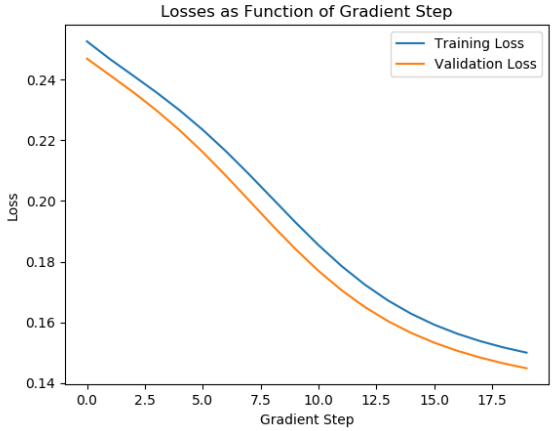


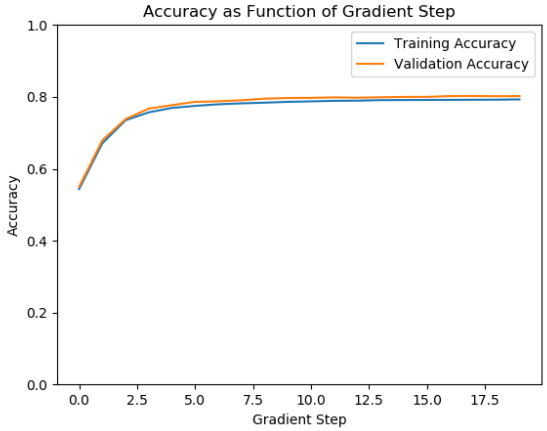
By Steps, Batch Size = 64





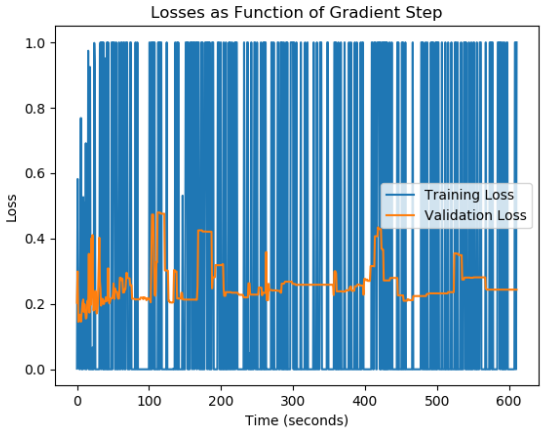
By Steps, Batch Size = 17932

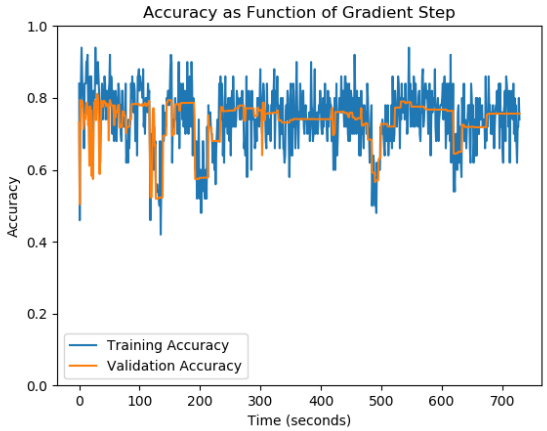




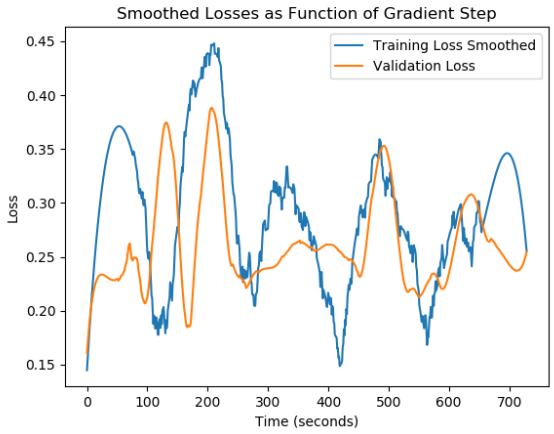
Below are the plots for varying batch size in terms of time spent training.

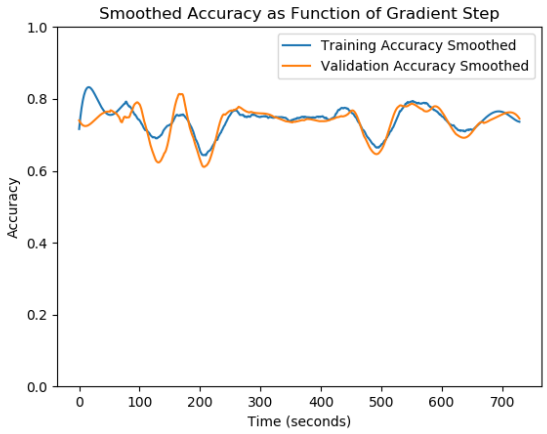
By Time, Batch Size = 1



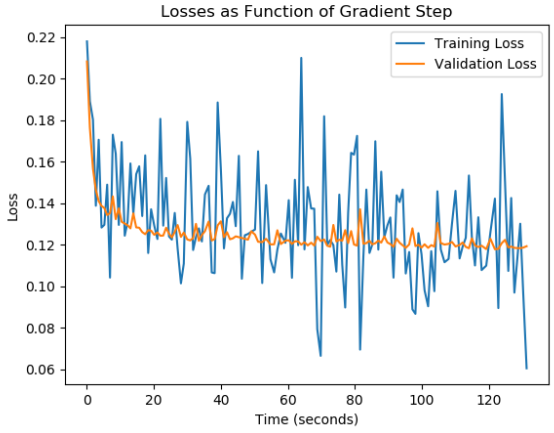


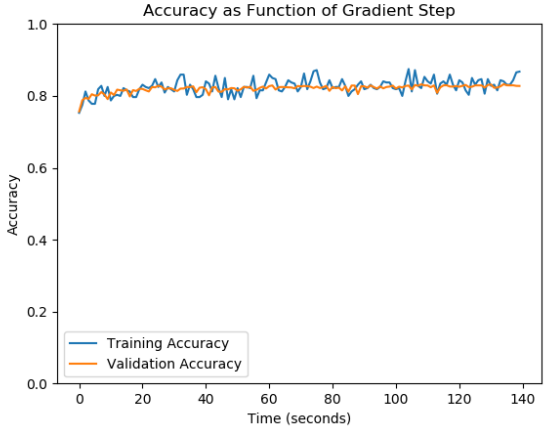
By Time, Batch Size = 1 (Smoothed)



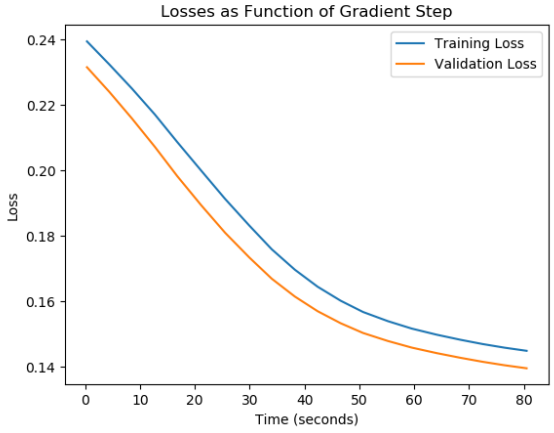


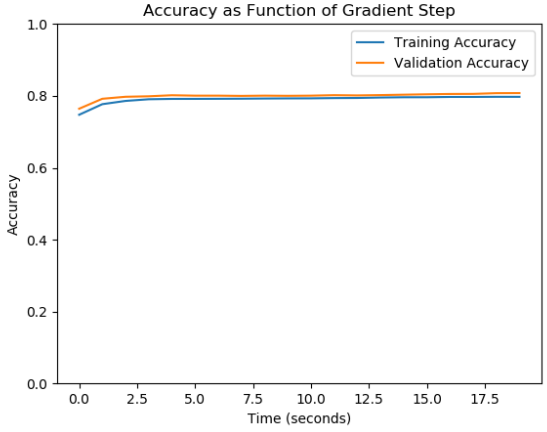
By Time, Batch Size = 64





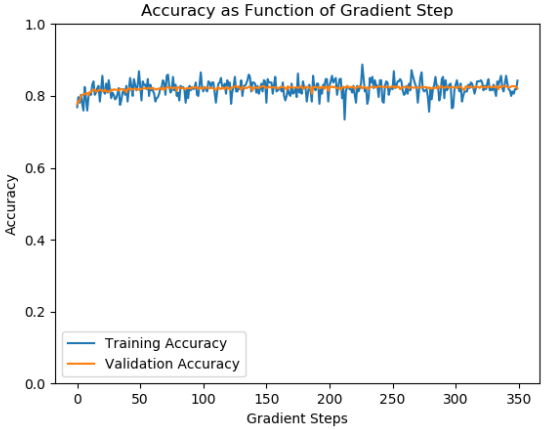
By Time, Batch Size = 17932

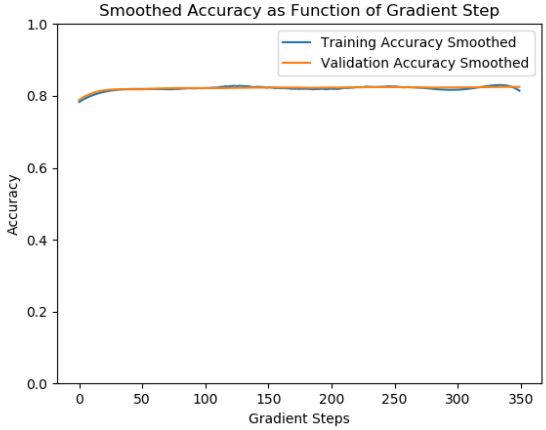




5.4 Under-fitting

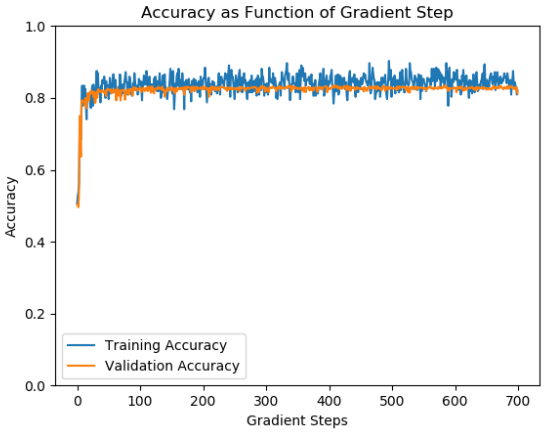
Below is the validation accuracy chart (regular and smoothed) with only a one layer network. This model still achieves validation accuracy >80% within the first few epochs. This is comparable with the best model trained so far which includes a hidden layer. This model is not underfitting. Due to the small number of input (only ∼100), we don't actually need a large network to make this classifier successful. Even a single hundred-to-one layer has over 100 parameters which can be optimized over the thousands of data we can train on. If the problem were considerably more complex (such as image processing), such a small network would likely underfit.

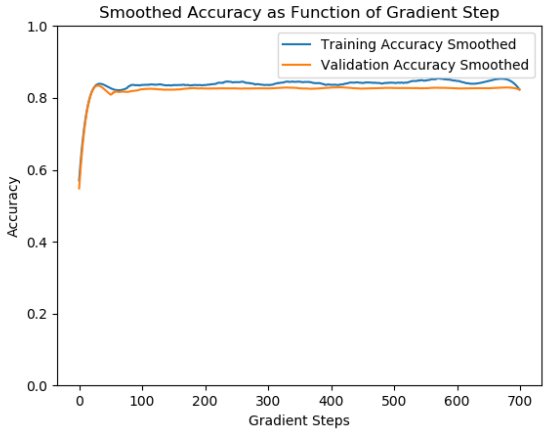




5.5 Over-fitting

Below is the validation accuracy chart (regular and smoothed) with a four layer network. This model achieves a validation accuracy of >80%. While this performance is comparable to the previous best model, it did take longer to converge (train) due to the additional parameters and model complexity. Based on the relationship between training accuracy and validation accuracy, this model is not overfitting.

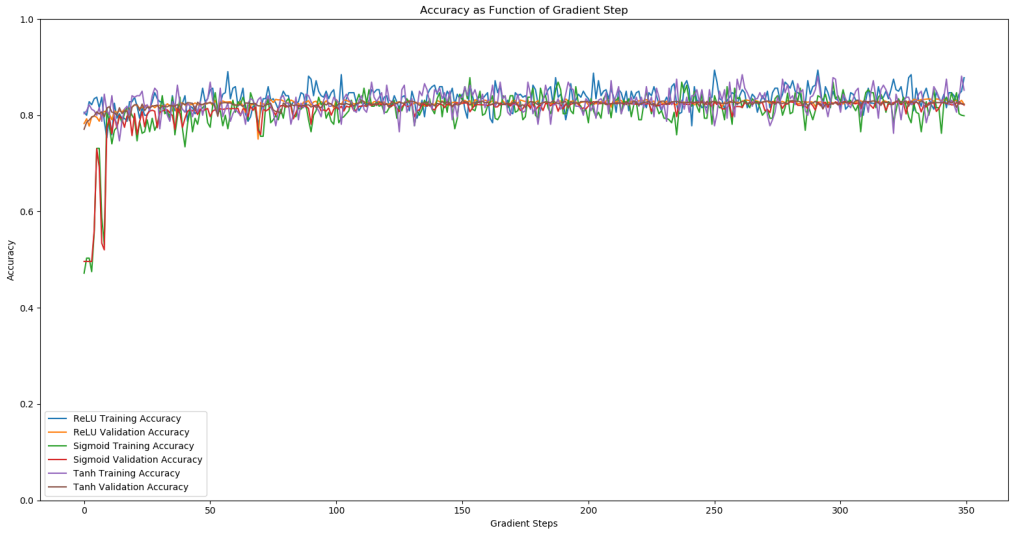


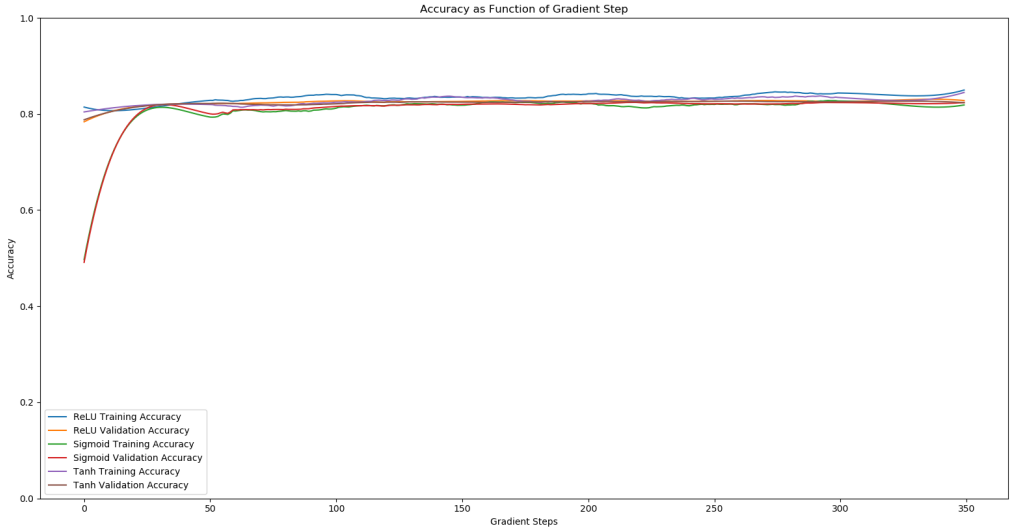


Continued...

5.6 Activation Function

1. From the graphs of accuracy per gradient step below, we notice that the Sigmoid activation function takes a much longer time to plateau than the other two activation functions. Tanh and ReLU, however, have similar performance and generally perform well. In general, however, the accuracies for the ReLU activation function are slightly higher than those for tanh.





2. In terms of time, see the table. The times were recorded while keeping all other hyperparameters constant. The Sigmoid function seems to run a bit faster than the other two functions. This seems counter-intuitive, since ReLU is so easy to compute and Tanh is considerably harder. It may be because PyTorch spent a lot of time optimizing for Sigmoid given it's wide use in logistic regression and multiclassification problems.

Activation Function	ReLU	Sigmoid	Tanh
Time (seconds)	322.22	307.79	322.51

5.7 Hyperparameter search

After a number of iterations, I found that the best results for my network happen with a ReLU activation function, batch size 50, learning rate 1, and one hidden layer of size 38. All of this was done using seed=10.

6 - Feedback

1. (a) In total, this assignment (start to finish) took me around 20 hours of dedicated, purposeful effort.

   (b) Figuring out expected inputs/outputs, optional parameters, etc etc for all the various classes and modules we had never seen before in lecture. For example, the DataLoader, Dataset, DataFrame, and other classes/objects. Also, creating all the different charts and training the network so many different times was quite tedious. I spent at least 5 hours (of my 20) just making plots and charts, waiting for training to finish for a subtask, etc.

   (c) I had learned these concepts a number of times before, but never actually had the chance to create a tangible ML project myself. While navigating the documentation of the modules was not fun, the assignment instructions were fairly well laid out.

   (d) Documentation for classes which were not discussed in lecture such as DataLoader, DataFrame, etc. I spent a lot of time on StackExchange figuring out how to get what I needed when using those classes.

   (e) The neat layout and starter code was helpful in the assignment. There was already an outline of the architecture, so I didn't have to make everything from scratch.