ECE324 Assignment 3
Benjamin Cheng – 1004838045

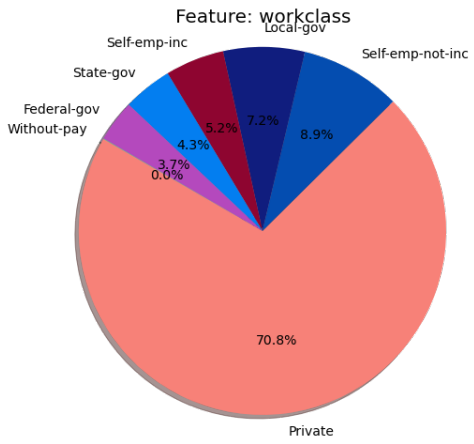
Section 3.2

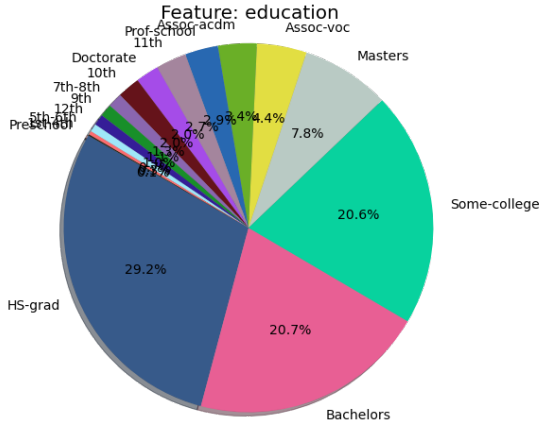
1. There are 37155 low income earners and 11687 high income earners.
2. Not balanced. The network will tend to favour low income earners and in this case could get 76% accuracy by solely guessing low income.

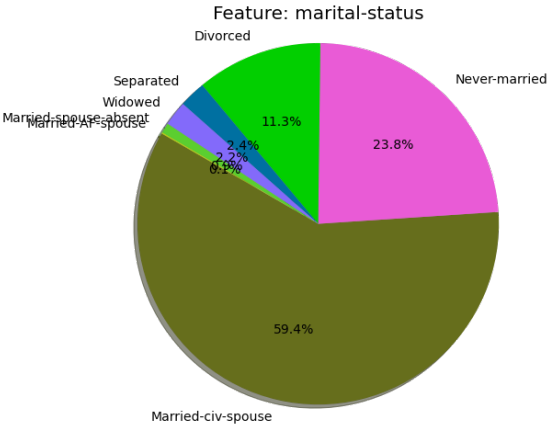
Section 3.3

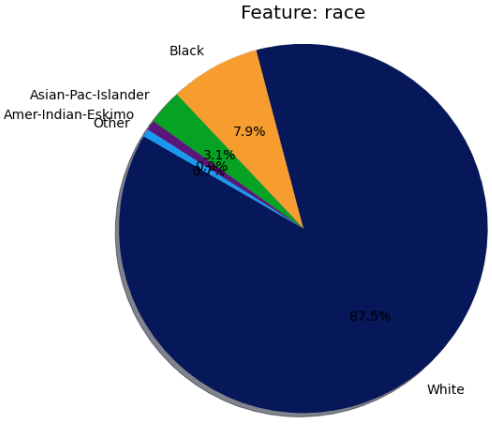
1. 3620 rows were removed, leaving 45222.
2. Although it's about 7% of the dataset, we still have 45222 left which is quite a lot.

Section 3.5





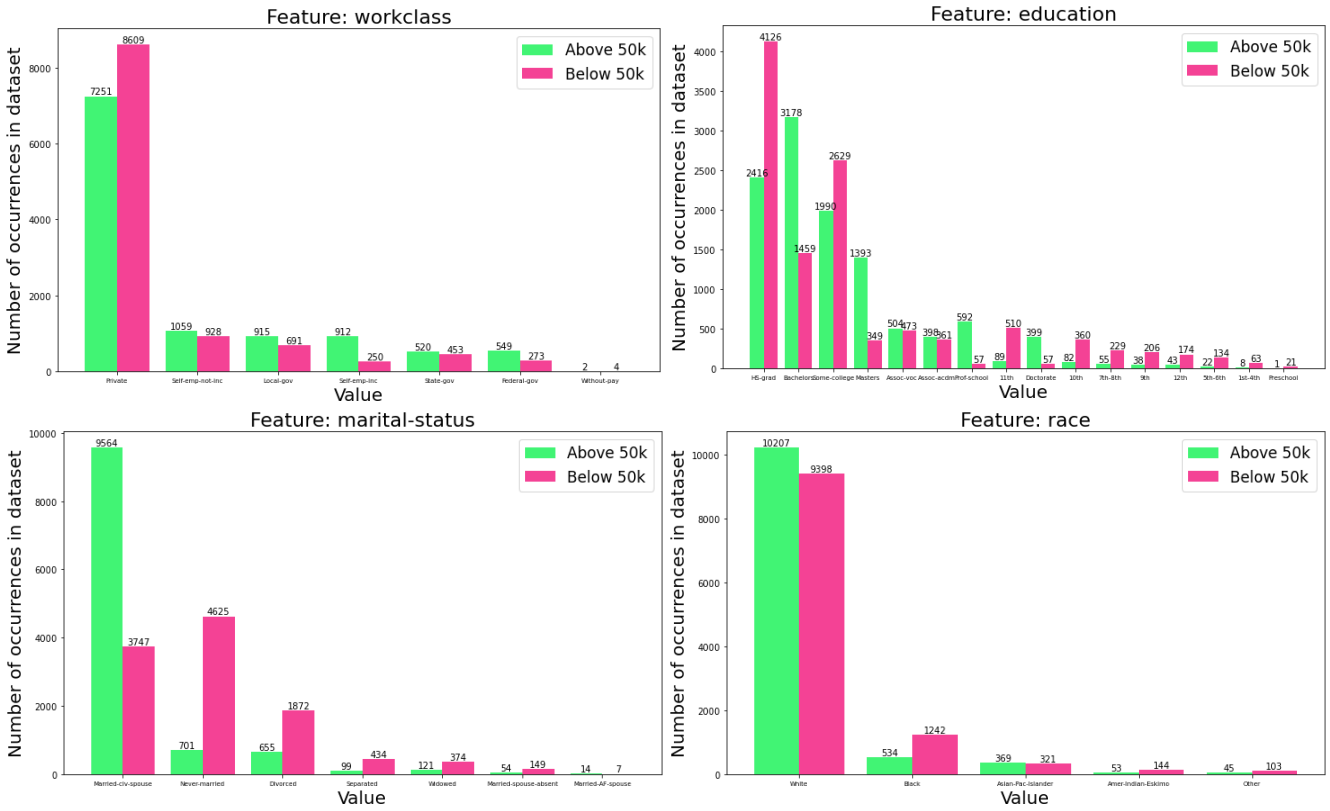




1. The minimum age is 17 and the minimum hours worked per week is 1 hr.

2. It is clear from the 4 features shown above that some groups are over/under-represented. Namely married-civ-spouse is way more common than the other marital statuses, and there are many more privately employeed workers. In terms of race, a *very* large percentage is white, meaning that the results trained on this dataset won't reflect very well for other races.

3. The dataset is predominately male (70%). As well its very heavily married couples (either Husband or Wife).



4. Looking at marital status, it appears that married-civ-spouse is the only maritial-status heavily skewed to above 50K, while the rest with significant numbers skew towards below 50k. Workclass isn't too much of an indicator, except for self-employed which is a indicator

5. Based on the education feature bar charts, a HS-grad has 37% of earning above 50k. For Bachelors, this increases to 68.6%. These percentages are taken by dividing the number of people earning 50k with HS-grad or Bachelors over the total number of people with HS-grad or Bachelors respectively.

Section 3.6

1. Using an integer representation for categorical data automatically biases some values more than others. Although the network can learn to discount these via weights, we don't have the network to bias anything automatically.
2. Using unnormalized continuous data can also introduce biases. Things like age are typically much larger than things like number of children, and we want to weight these things equally unless otherwise specified.

Section 4.2

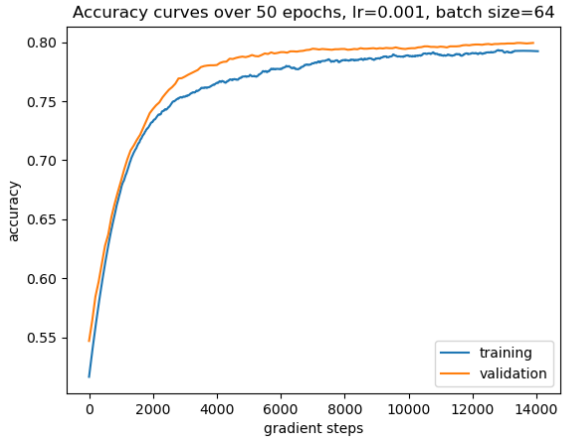
1. Shuffling the data ensures that we aren't optimizing into an local minima prematurely. If the income order was sorted then we would be optimizing for say, earners of less than $50k. We could be optimizing into a situation were we output only this class, reaching 100% accuracy, before encountering high income earners. With activation functions like ReLU, the gradient is equal to zero when the output is zero so we may end up getting stuck even.

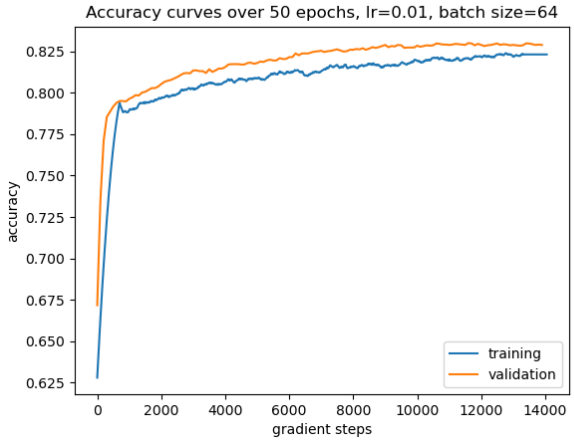
Section 4.3

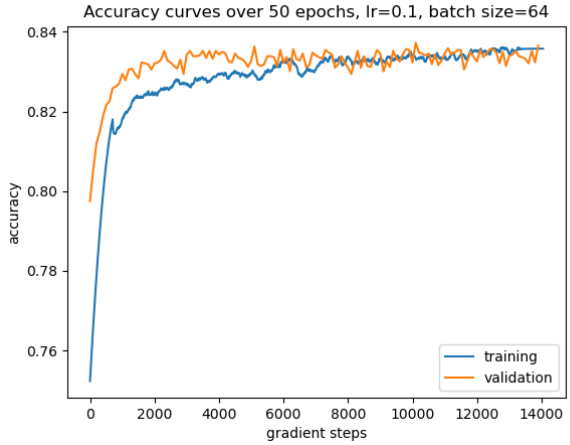
1. I chose 64 as my hidden layer size because there are 14 categories of data and I felt a multiplier of 4 should be enough to give parameters to learn each of the categories relation to income. I rounded up to the nearest power of 2. The output layer should be size 1 we are doing binary classification.
2. The sigmoid activation function limits values to between 0 and 1 in a fashion similar to probability. Very large numbers are limited to 1 and very small numbers are limited to 0. 0 would mean that there's 100% chance its a low income earner and 1 would mean there's 100% chance its a high income earner.

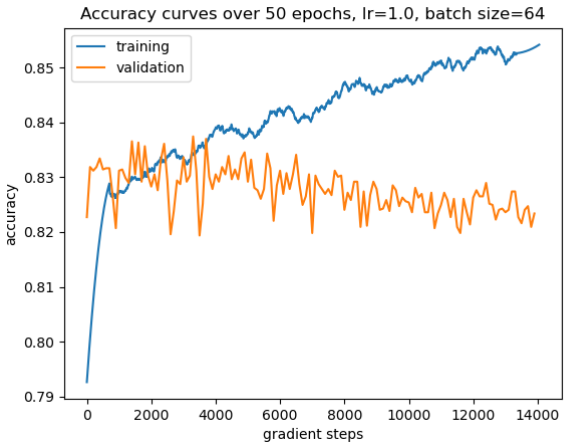
Section 5.1

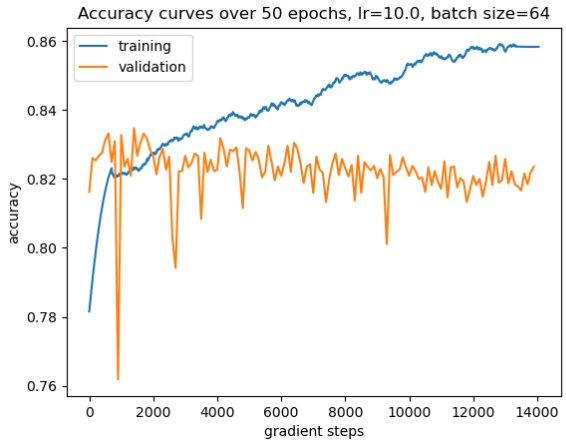
Learning Rate	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}	10^{2}	10^{3}
Valdation Accuracy	0.800	0.830	0.837	0.837	0.835	0.494	0.506

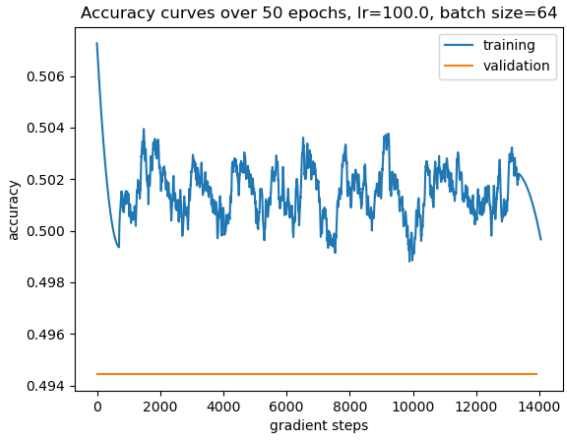


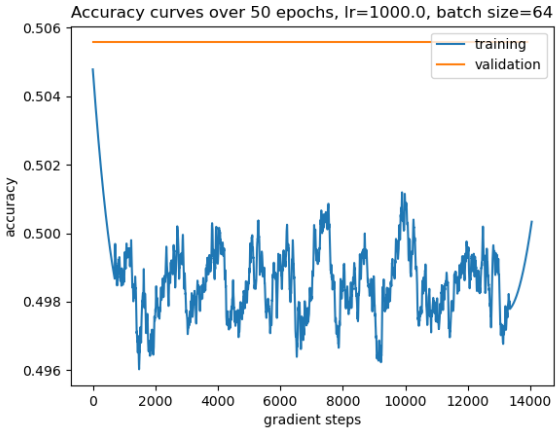








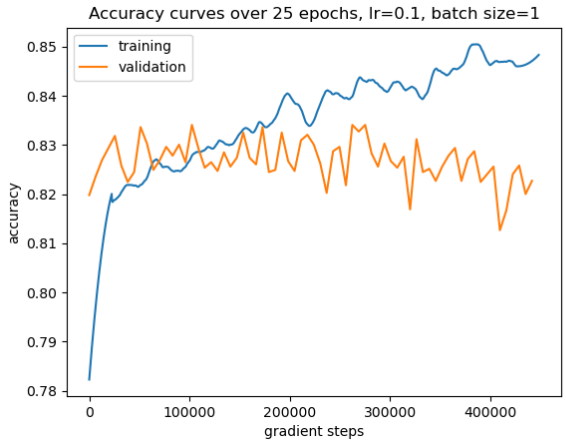


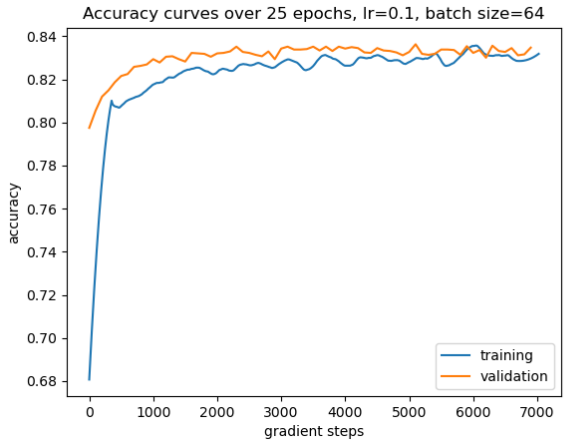


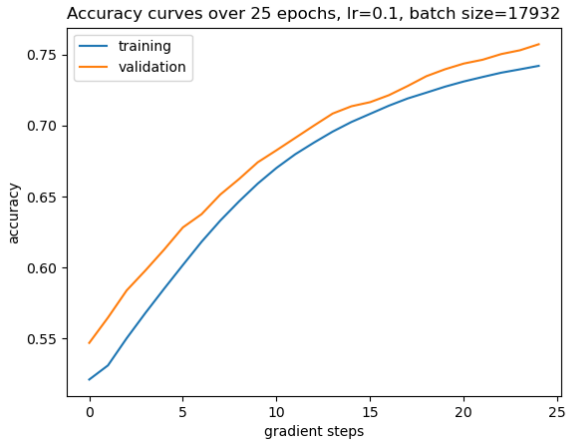
1. The learning rate of 0.1 worked the best. Although 1 also gave the same validation accuracy, the accuracy curve shown in the graph appears to diverge.
2. If the learning rate is too low, the training requires more epochs to reach the same epoch, or it may get stuck in a local minima. If the learning rate is too high, learning doesn't occur since it oscillates around the optimum.

Section 5.2
Note that the learning rate plots above all use 50 epochs to ensure we aren't going to miss anything.

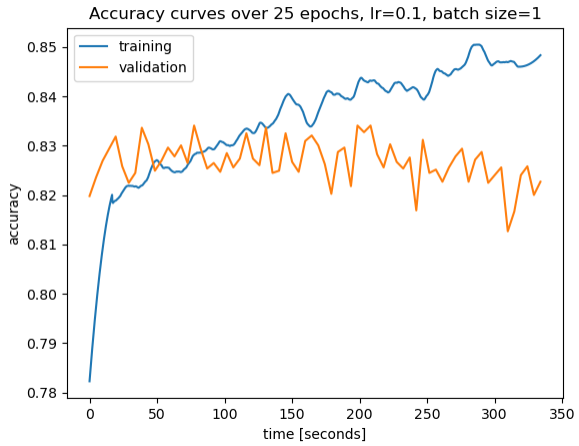
Section 5.3
The following plots are with respect to gradient steps:

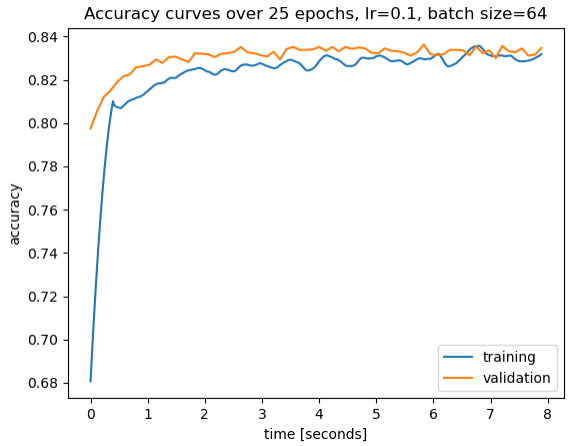


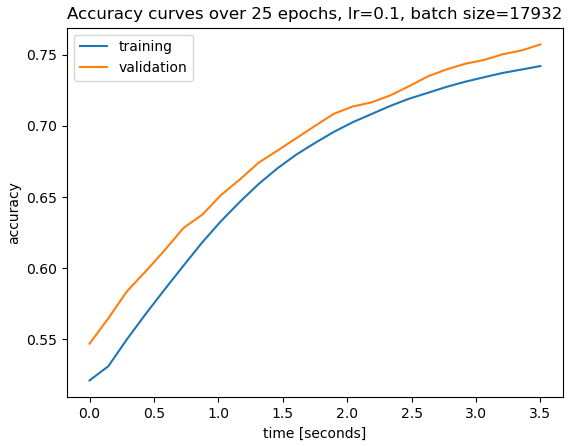




The following plots are with respect to time:







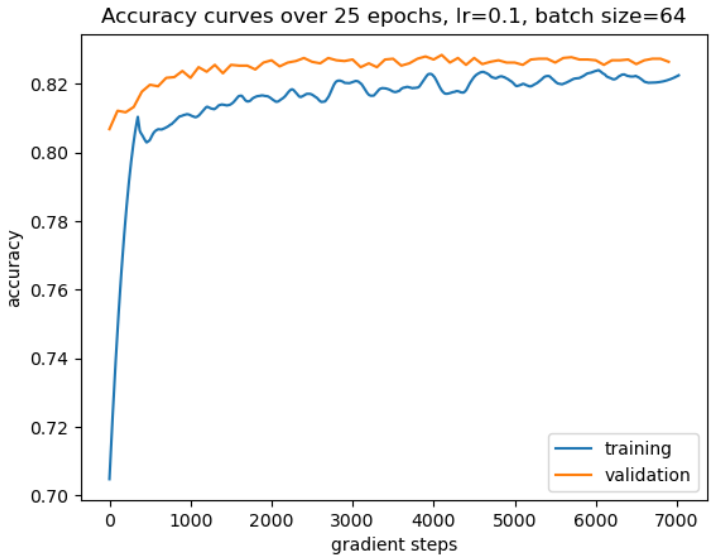
1. A batch size of 64 gave the most accuracy at 0.837.
2. A batch size of 64 reached a high accuracy the fastest in terms of both time and steps.
3. If the batch size is too low it takes a very long time to train and the model does not appear to converge, probably due to stochastic nature each iteration “pulls” in a different direction. If the batch size it too high, we lose the stochastic properties and the model learns more consistently but slower.

4. Using a small batch size decreases the effect of vectorization and can thus make computation much slower. It also makes the training more sensitive to outliers, meaning one outlier can affect the networks weights a lot. When your training data is good but in limited supply, this effect isn't a concern so the small batch size works well.

A large batch size slows down the gradient descent process since we are averaging the gradient over this large batch. However it is not very sensitive to outliers, making it very stable (i.e. accuracy doesn't jump around).

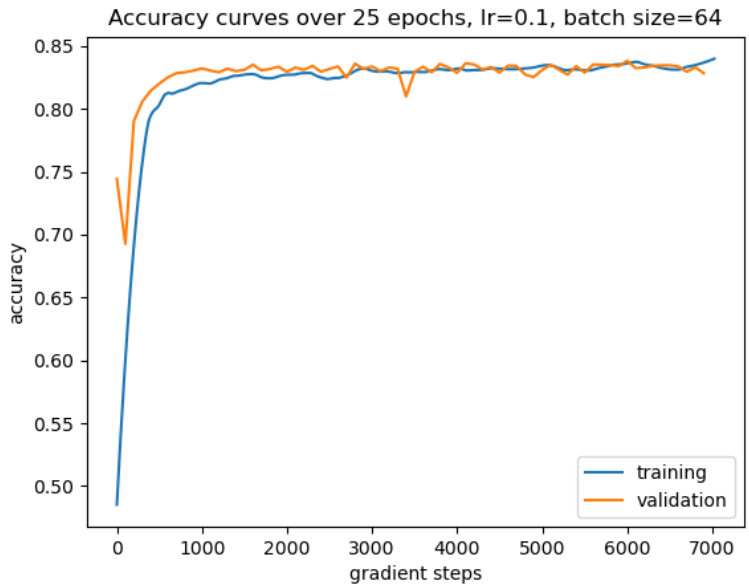
In general, the batch size should be much less than your entire dataset but closer to 1. However it shouldn't be too low, so around 32 or 64 is generally good depending on the size of your dataset.

Section 5.4



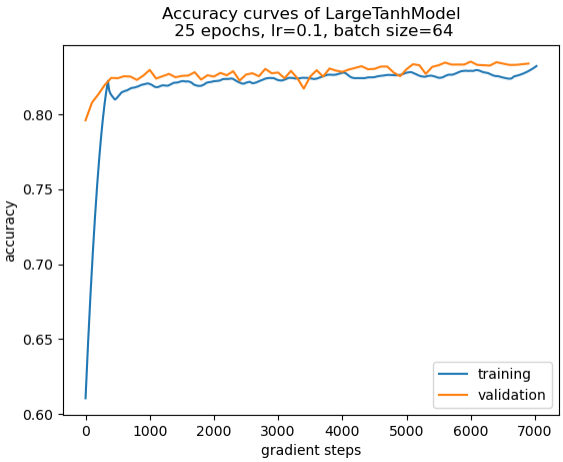
The validation accuracy achieved by this model is 0.829. This is slightly worse than the 0.837 achieved by the model with 64 neurons in the hidden layer. I don't believe this model is underfitting since the training and validation accuracies are still approximately similar to each other. It's also not an appreciable amount worse than the reference model.

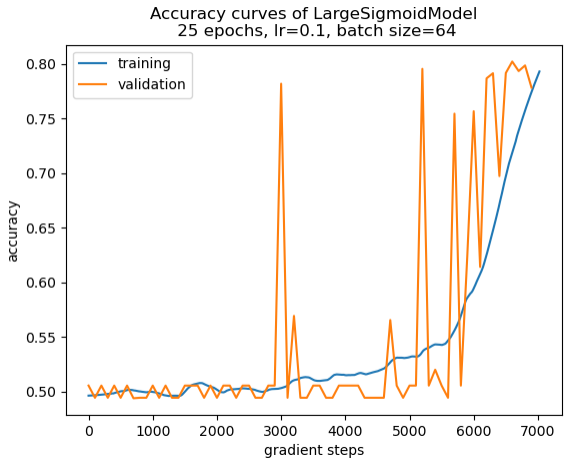
Section 5.5

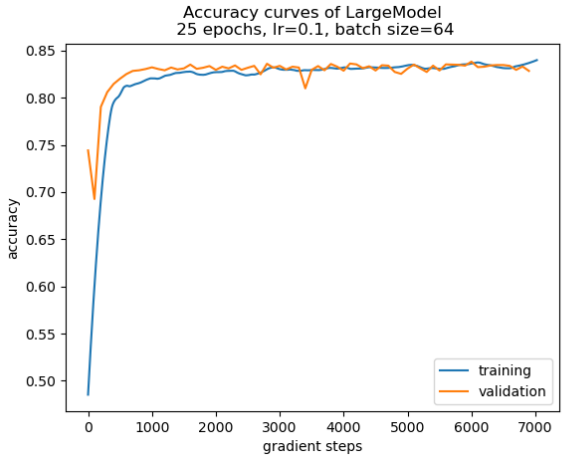


This model was able to produce a validation accuracy of 0.838. This is very slightly better than the 0.837 achived by the reference model. I wouldn't say this is overfitting because as shown in the accuracy curves, the validation accuracy is approximately inline with the training.

Section 5.6







1. The three plots are shown above. Note that LargeModel is the same as the “overfitting” model from section 5.5, which used the ReLU activation function. Tanh appears to produce similar results to ReLU, although accuracy trends a bit lower than the latter. Sigmoid appears to “learn” a lot slower than the other two, only reaching 80% accuracy near the end of training. Sigmoid also appears to be a bit unstable for validation, with spikes up to 80% accuracy very early, while the training is still at around 50%.

2.

Activation Function	**Wall Time**
ReLU	9.84s
Sigmoid	9.95s
Tanh	10.8s

The tanh function appears to be slower than the other two by an appreciable margin. Sigmoid and ReLU were quite close but sigmoid appears to be a little slower than ReLU. This makes sense as ReLU is the simplest to evaluate.

Section 5.7
I achieved 83.8% accuracy using a MLP with 2 layers and a hidden layer size of 64. It was trained for 75 epochs, evaluated every 64 steps, with a batch size of 32. The learning rate was set to 0.05 and the seed was set to 0.

Section 6
a) I worked on the assignment over the course of a few days, spending a total of about 3.5 hours.
b) I always have an issue with these types of assignments where I complete the coding portion a few days before the writeup. It becomes challenging when going back to write up the questions since it takes a while to recall what I was doing.
c) I haven't used pandas before so figuring that out was interesting.
d) The instructions sometimes contradict each other (i.e. section 3.6 parts 2 and 3 both tell you to use the OneHotEncoder)
e) The ability to use Jupyter Notebooks actually makes prototyping these assignments easier. I typically define a hyperparameter dictionary that gets passed around through function calls so I just run one cell to train with a different hyperparameters. If I was using python files directly it would also need to rerun all the dataprocessing we did.