COL774 ASSIGNMENT 3B

Raval Vedant Sanjay

2017CS10366

1 General Neural Network Architecture Model

I made a program to implement the general neural network architecture model as asked in the problem statement. Now, since the number of inputs, outputs etc would be different for all the different units and layers, so this might be tempting to implement the parameters as a 3D list. But actually that implementation would be very costly and implementing the same by using a numpy 3D matrix resulted in the 4 fold performance improvement! I also implemented a SGD as done in the first assignment, with the general stopping criterion being the average cost after an epoch to be below a particular value, and the number of epochs below a certain bound. This condition was giving a monotonous update and thus, helped the gradient descend proceed in a more regular manner!

2 Experimenting with different architectures

The Common stopping criterion for this sub-problem was taken as:

- The minimum cost value= 0.1
- The maximum number of epochs= 200

Note that here, I didn't make use of a stopping criterion which would be involving of the difference between the consecutive costs and taking a threshold etc and this is because there are many plateaus, local minimas etc seen in the overall plot and so keeping a condition on the difference between consequtive costs would only make the program get more prone to report the local minima as the final answer, no matter how low the threshold is kept! And this value of minimum number of epochs is kept at this value since the model converged at a decently good value for the complex architectures at 200 epochs and keeping the smaller archtectures running for a huge number of epochs was not a good idea since anyways they weren't going to give any improvement. Also, the minimum cost value is kept a moderately high value in order to prevent overfitting

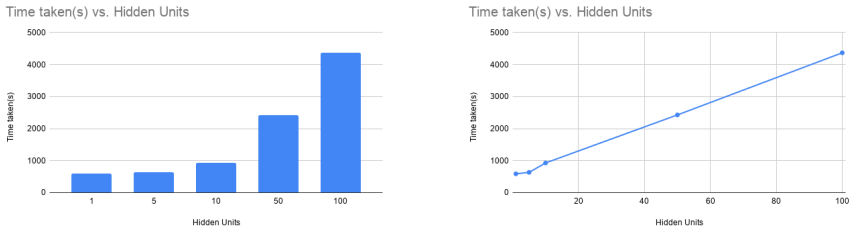
The table for the different values for this sub-part is given below:

Hidden Units	Train Accuracy	Test Accuracy	Final Cost	Time (sec)
1	0.03846	0.03846	0.4803	589.54
5	0.03846	0.03846	0.4803	635.35
10	0.1108	0.1077	0.4689	928.28
50	0.9192	0.7600	0.1011	2428.67
100	0.9134	0.7689	0.1015	4368.52

From the above table, we can see that the model does not train much for the architectures with very few hidden units. Infact, in the case of 5 and 10 hidden units, it was observed that the cost got stuck in the region near to the overall cost of 0.48 and wasn't able to improve much despite of running the code for 200 epochs. While, for the case of 10 layers, it learned pretty well as compared to the other two but still the rate of learning per epoch was very low for such a small neural network architecture and that's why it was only at a cost of 0.46 in the end, inspite of managing to get out of the low slope regions around 0.48!

Also for the complex architectures like 50 and 100, a significant improvement was found in terms of their performance on the training as well as the test set. It was also observed that the complex the network is, the more it will progress towards a lower optimal cost in one epoch. So it was easier for the complicated networks to get away with the low slope regions in the way as compared to the simple networks.

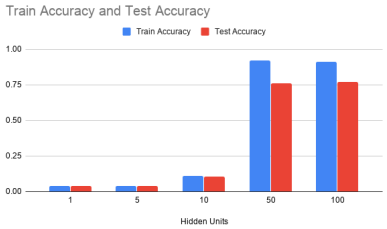
The required plots are given below:



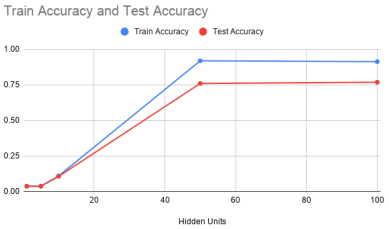
(a) Column Chart

(b) Line plot

Figure 1: Plots for the time taken



(a) Column Chart



(b) Line plot

Figure 2: Plots for the Training and Test Accuracies

3 Adaptive Learning Rate

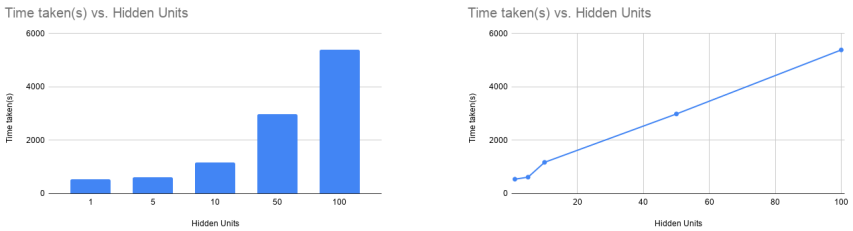
In this case, I chose to have the same stopping criterion as in the previous case to have a direct comparision between the given adaptive learning rate and the constant learning rate of 0.1 as considered in the previous sub-part. That was because of the fact that the new adaptive rate would get equal to the previous in just 25 epochs and then it will start decreasing and start going down and down towards zero, and so having too many epochs to run would eventually halt everything because of very low learning rate.

It was observed that in this case, the learning was quicker than the previous case in the beginning but then because of the increasing denominator, eventually this model learns slower than previous model. The Corresponding table is:

Hidden Units	Train Accuracy	Test Accuracy	Final Cost	Time (sec)
1	0.03846	0.03846	0.4803	543.37
5	0.03846	0.03846	0.4803	619.74
10	0.03846	0.03846	0.4803	1175.22
50	0.29215	0.25214	0.4205	2988.25
100	0.46615	0.40769	0.3822	5388.11

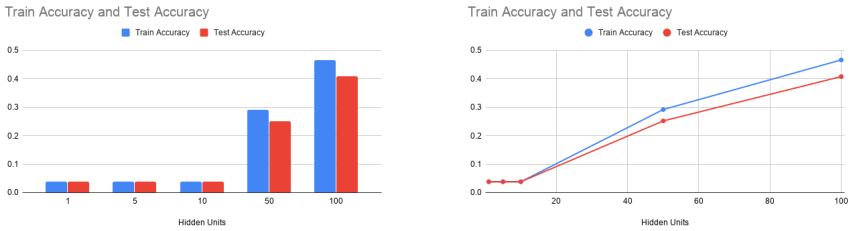
Thus, from this table we can see that the adaptive learning would take more time than the other case because of the rapidly decreasing learning rate. Also, we can see that in the given scenario, a monotonously decreasing learning rate wouldn't work because of the nature of the cost curve, consisting of many local minimas or saddle points etc). Also, the accuracies obtained for the different cases is lower than the ones obtained with the constant learning rate (for the same upper bound of the number of iterations). Thus this shows that the adaptive learning actually slows down the learning.

The required plots for the same are added below:



(a) Column Chart (b) Line plot

Figure 3: Plots for the time taken



(a) Column Chart (b) Line plot

Figure 4: Plots for the Training and Test Accuracies

4 RELU activation for the hidden layers

I implemented ReLU with an adaptive learning rate as asked. Now, when I experimented the ReLU and Sigmoid units with the asked network, then there was an issue that because of the adaptive learning getting too low very soon. So even after running the models for more than 300 epochs (Taking nearly 8700 secs for both the models), the training and test accuracies for the Sigmoid model were **0.03846** and the training and test accuracies for the ReLU model were **0.206** and **0.166** respectively. The final cost for the Sigmoid model didn't even go beyond **0.479** and for the ReLU model, it was nearly getting very slowly decremented to reach **0.486**. And since, after 300 epochs the learning rates would be very low, so it would be practically impossible for these models to show a proper convergence beyond this epoch! Though here, the time taken for the ReLU model was lower than the time taken for the Sigmoid model, and it obtained better results as well. So, if we compare the results of this adaptive learning with the ones obtained in the **(b)** part, then those results were faster and better than the results obtained from the adaptive learning

So, in order to have a better comparision of the performance, I decided to run the non-adaptive learning for the given architecture on both the ReLU and the sigmoid units. It was observed that in the beginning of the learning, the Sigmoid model learns faster than the ReLU model i.e the cost for the sigmoid model reduces quickly with the number of epochs as compared to the ReLU model. But it seemed that the ReLU units have an advantage over the Sigmoid ones in terms of the overall comparision. When the entire algorithms were run for a maximum of 200 epochs, the following results were obtained:

Hidden Units	Train Accuracy	Test Accuracy	Final Cost	Time (sec)
Sigmoid	0.4017	0.3400	0.3900	5754.93
ReLU	0.4126	0.3386	0.4123	5714.88

Now, if we compare the above performance with the earlier case of running the Sigmoid units on the same parameters and for the same maximum number of epochs, it can be seen that the single layer showed a better performance as compared to the multi layer case. The algorithm took a longer time and more iterations to learn in the latter case. It took the latter network architecture to take more iterations to come out of the local minimas and the different plateaus. Though, in order to get a clearer view of the performance for the Sigmoid and the ReLU units, I ran the algorithms for a maximum of 500 epochs and obtained the following results:

Hidden Units	Train Accuracy	Test Accuracy	Final Cost	Time (sec)
Sigmoid	0.9007	0.7262	0.1011	11210.76
ReLU	0.6200	0.5115	0.3093	13737.64

So, here we can see that the ReLU units learn much slower as compared to the Sigmoid units. Also, note that the Sigmoid units converged at around 395 epochs, while the ReLU units couldn't converge even after these many iterations! So, from all these experiments it seems that the single layer architectures perform better than the more complicated architectures because of slow learning, over-fitting etc. Also, the ReLU units might be working better than the Sigmoid units for the Adaptive learning case and lower number of epochs, but in the long run the sigmoid network seems to be the more efficient one!

5 MLP Classifier

So here, we would be passing the One-hot encoding of the output as asked in the problem statement to get the Binary Cross Entropy loss as required. The parameters of the classifier were set as per what were used in the previous models. I used an inverse scaling learning rate with the initial rate set as 0.5 and tried to run the algorithm on it, but apparently the model just diverged to **inf** loss in the first epoch itself and so this set of parameters seemed to give a very worse result for the chosen classifier!

Then, I tried to change the initial learning rate to some lower value and it seemed to be working better for the initial learning rate to be set as 0.2. When there was an upper bound on the maximum number of epochs to be 200 (with tol as 10^{-6}), then the final loss was 1.456, but the training and test accuracies were obtained as 0.5907 and 0.551 respectively, taking only 106 secs to finish it's execution. Thus, the sklearn classifier gave a very fast and a much better result as compared to the self programmed implementations! Then in order to see where the corresponding set of parameters would converge, I tried keeping a bound on the maximum number of epochs as 10000 (with the convergence criteria as 10^{-4}) and the solver stopped at around 1600 epoch, giving a message "Training loss did not improve more than tol=0.000100 for 10 consecutive epochs", with the training and test accuracies as **0.749** and **0.705** respectively, the final loss as 0.9407, and the total time of 824 seconds.

So if we compare these results to the different tables in the previous section, we can find that the performance of the sklearn's classifier was far better than the performance of the scratch implementation, in terms of speed as well as final accuracies!