Project-2 Report, MLP, Deep Learning with CNN

Karthik Satya Raghuram Bylapudi
(1225577730)

Introduction:

The project has 2 parts.

Part A-
This involves classifying the given data into two classes. The data is split into train, validation and test. We are asked to run this data on a 2-Nh-1 MLP where Nh varies between 4,8,10,12,14. There is freedom given regarding what activation function to use. The Error metrics used is MSE (Mean square error). There is an additional condition which says we stop the training once we observe that validation error is shooting up. Finally we will be plotting the train, test and validation plots and observe their trends.

Part B-
This involves training a CNN using the keras module. There is a description of what layers need to be used. Our main task here is to observe the changes in accuracy due to the changes we do in hyperparameters (such as learning rate, kernel_size, etc) The data set we run this over is the MNIST digit recognition dataset which contains 28x28 sized images.

Method:

Part A:
Reference: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

I have implemented a 2-Nh-1 MLP network taking the above website as my reference:

The coding language chosen is python and the packages used are seaborn, numpy, pandas and matplotlib.

Step1 (preprocess the data to get into the required format):
The data provided is in .txt files. This data is extracted and converted to a numpy array. THen we add the labels column. Then we split the data into train and validation where we give 1500 samples from each class to train and 500 for validation. We also perform normalisation over this data by subtracting the mean from the data points and dividing by their standard deviation. The dataset itself has only 2 features so there is no feature reduction step required.

Step 2: (Create the model)

As we have seen in the case of MLP, there are two major steps involved in this - Forward propagation and backward propagation. The backprop step involves updating the weights so as to reduce the error obtained. I have used the sigmoid activation in the hidden to output layer. The weights are randomly initialised using np.random. The array hidden_layer_nodes helps us to modify the number of nodes in the hidden layer. We have also plotted the two classes here. And we observe that class 2 has 3 distinct clusters whereas class 1 is randomly distributed. The graph shows that the points are highly clustered in this lower dimension space which makes them good data to see how the MLP model handles it.

Fprop -
The forward propagation step from input to hidden is just doing dot product between input and weights and adding the bias to them. For the hidden to output layer we do the same. This gives us a simple linear equation output. After this we run the outputs through the sigmoid activation function which introduces the nonlinearity into the model.

Backprop -
The complexity of an MLP model is due to the back propagation step.

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

This equation is what we use to get the error gradient for the output to the hidden layer. For the activation function of sigmoid - We can essentially break down the equation into the following - PD1 = prediction - label, PD2 = prediction * (1 - prediction) and PD3 = output of hidden layer. The total gradient is - PD1 * PD2 * PD3.
But getting the gradient of the hidden to input layer is a lot more complex since the gradient is affected by both output values.

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

pd(Error)/pd(w1) = pd(error)/pd(output_hidden) *
pd(output_hidden)/pd(pre_activation_hidden_output) *
pd(pre_activation_hidden_output)/pd(w1)

pre-activation_hidden_output = input * weight + bias
Output_hidden = Sigmoid(pre-activation_hidden_output)
Error = (1/(2*number_of_samples))*(output-pred)^2

$$\frac{\partial E_{total}}{\partial w_1} = \left(\sum_o \frac{\partial E_{total}}{\partial out_o} * \frac{\partial out_o}{\partial net_o} * \frac{\partial net_o}{\partial out_{h1}}\right) * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

All of this is done in the partial differentiation step. Note that some of the values in the partial differentiation were already previously calculated so we use the concept of memoization to escape calculating them again. This is why we can run extremely big neural networks.
Error -
We are using the mean square error to calculate our loss which is
(½)*(pred-label)^2*(1/samples)

Accuracy -
The accuracy is nothing but (number of samples wrongly classified) / (total samples)

Plots -
This function plots the train, validation and test losses over the epochs.

Train_network -
We finally define the whole network and how it runs using the train_network function:

- Perform forward propagation over the given data
- Obtain error with respect to the train data.
- perform backprop to update the weights based on the error gradient obtained and learning rate
- observe the train loss
- Do a test on validation set to observe the accuracy change. If the validation loss stops decreasing you can stop training
- Observe the loss and accuracy in the test.

Once we have all the components defined in our MLP class. We can start training and testing our model.
The last part of code for part A creates a model with our required number of hidden nodes and trains the model over 2500 iterations.

Part B:

This is a very straightforward implementation of a basic CNN architecture model using keras. The model takes in inputs of 28x28 sized images. The feature set mapping is 32,16. There are 3 fully connected layers 128,64,10. Activation function for output is softmax and the number of classes is 10. The general activation function for the hidden layers is ReLu and finally Convolution is done initially on 3x3 kernels with stride 2 and we perform max pooling over the convolution output.

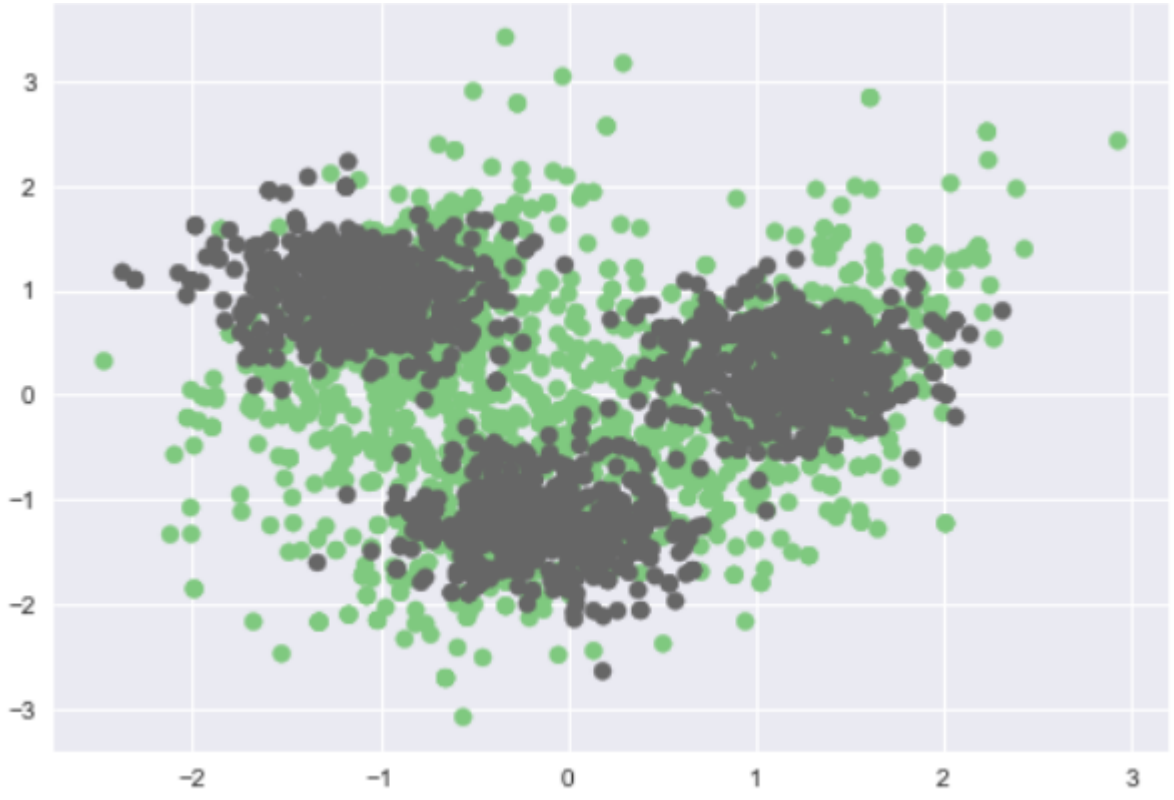
The major task on this is to observe the changes due to changing the hyper parameters learning rate, kernel_size, feature set mappings, number of neurons in the fully connected layers. We have written a function which takes the parameters of kernel size, learning rate, feature set mappings, number of neurons in the fully connected layers and creates a CNN

model using keras and returns the model. We then created multiple models by changing the hyper parameters and trained them on the mnist dataset. We observe the accuracy of change because of 1 parameter and kept the other parameters to the default value.

Results and Observations:

Part A:

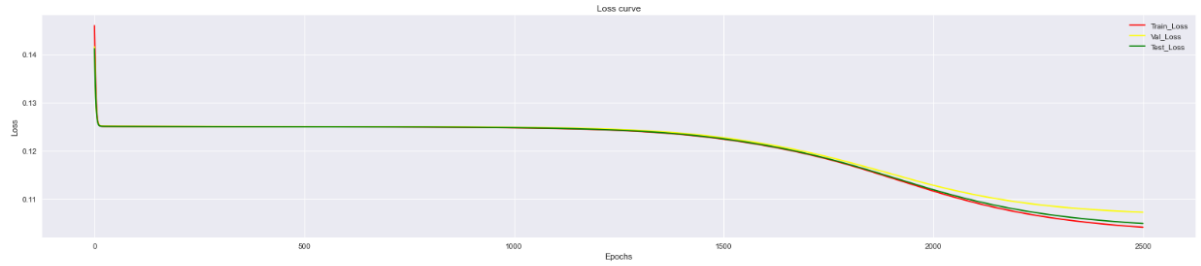
The plot below shows how the class distribution of the data points is being done for both the classes. We observe that there is a lot of overlapping. This makes the data extremely hard to classify using the conventional means of linear classification. We cannot be sure on what degree of freedom is required to get a good accuracy. So this makes this dataset a good candidate to be trained using the Multi Layer Perceptron model.



Train Data spread

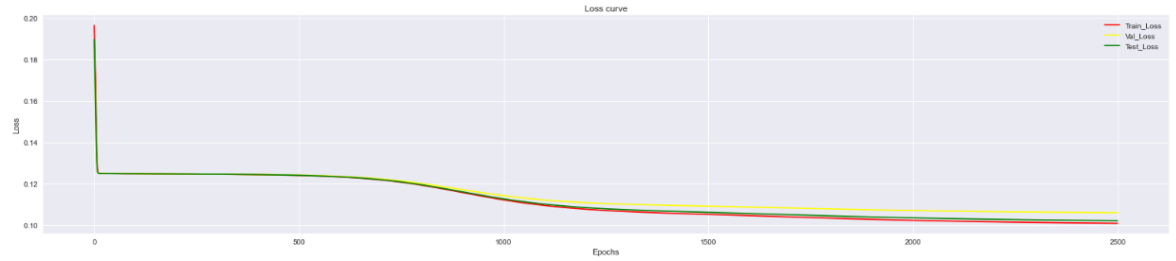
Now our next observation is regarding the change of error/loss with respect to the number of hidden nodes we use in the hidden layer.

```
hidden nodes = 4
Epoch 2500, Train Loss: 0.10408867084297585, Val Loss: 0.10721182131697746, Test Loss: 0.10486497292341618,Train Accuracy: 0.68
23333333333333, validation Accuracy: 0.66, Test Accuracy: 0.67675
```

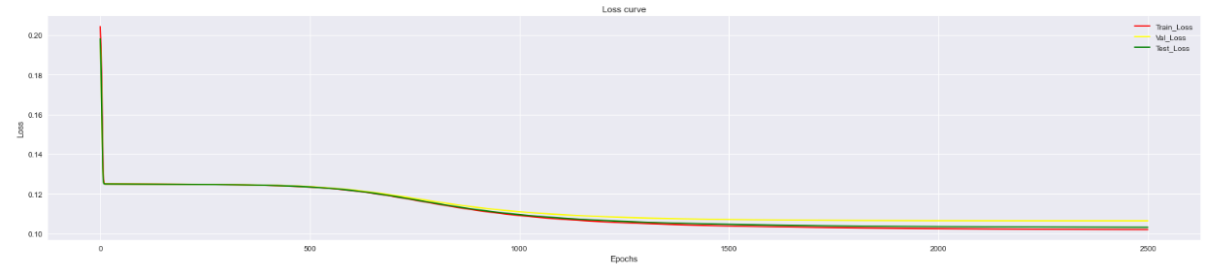


With hidden nodes = 4, our general assumption would be it might be too less to capture the nonlinearity possessed by the data due to the extreme overlapping. That is one of the reasons why it took much longer number of epochs to converge (>2500)

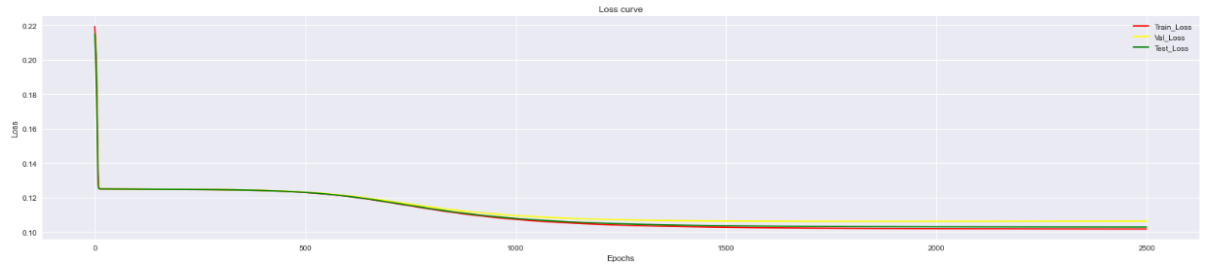
```
hidden nodes = 8
Epoch 2500, Train Loss: 0.10098438262662643, Val Loss: 0.10607766362526946, Test Loss: 0.10225646868787532,Train Accuracy: 0.69
76666666666667, validation Accuracy: 0.667, Test Accuracy: 0.69
```



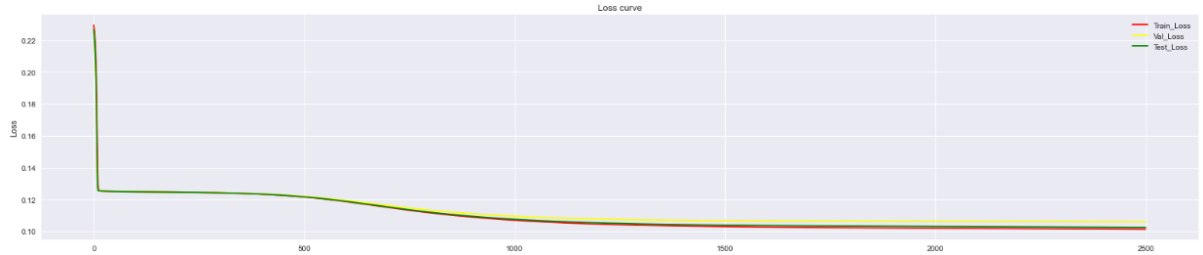

```
hidden nodes = 10
Epoch 2500, Train Loss: 0.10203077790328796, Val Loss: 0.10641860678555698, Test Loss: 0.10312728850245226,Train Accuracy: 0.69
23333333333334, validation Accuracy: 0.67, Test Accuracy: 0.68675
```




```
hidden nodes = 12
Epoch 2500, Train Loss: 0.10176867963755104, Val Loss: 0.10630542871481315, Test Loss: 0.10290267562005323,Train Accuracy: 0.69
66666666666667, validation Accuracy: 0.67, Test Accuracy: 0.69
```



```
hidden nodes = 14
Epoch 2500, Train Loss: 0.10127924847259516, Val Loss: 0.10602423485322293, Test Loss: 0.10246437398389563,Train Accuracy: 0.69
76666666666667, validation Accuracy: 0.677, Test Accuracy: 0.6925
```



If we observe the plots for hidden node count of 8,10,12 and 14 we observe that there is essentially not much difference over the point of convergence. This means that 8 nodes is a good node count for the given data set to be run over. We should also note that since we are not using L1norm or L2norm during weight update, if we run for too long, the data is going to overfit and would give a poor test accuracy. This is partially being prevented by the early stopping condition that we created saying stop when the validation loss starts increasing. But if we create too many hidden nodes, the huge number of parameters(weights) can memorise the data instead of doing testing over it. Our best accuracy is obtained at hidden nodes 8 and 14.

Part B:

The accuracy of the model with the default parameters is 0.98580002784729.

The default learning rate is 0.001. After decreasing the learning rate to 0.0001 we got the accuracy of 0.9894999861717224. There is a slight increase in accuracy after decreasing the learning rate. After increasing the learning rate to 0.01 we got the accuracy of 0.965499997138977 which is less than the accuracy for default values. This shows that the minima is being overshot at the case where the learning rate is 0.01. But with a lower learning rate we can observe that the accuracy improved, this shows that the gradient slope is very steep and having a smaller learning rate controls the loss minima fall a lot more better than a higher learning rate.

The default kernel size is 3*3. After increasing the kernel size to 5*5 we got the accuracy of 0.9905999898910522 which is more than the accuracy we got with 3*3. After decreasing the kernel size to 2*2 we got the accuracy of 0.9884999990463257 which is slightly more than the accuracy we got in 2*2 but is less than the 5*5 accuracy. Our observation is that we get the highest accuracy out of all the changes when kernel size is increased. This means there are good higher order features and larger kernels are able to capture the intricacies of it. Also HIgher size kernels are good for generalisation and reducing the influence of noise on the model. The reduction from 3*3 to 2*2 isn't making any considerable change in accuracy which in turn shows that there isn't much difference in features being captured by 3x3 and 2x2 kernels.

The default learning feature mapping is 16 in hidden layer 1 and 32 in hidden layer 2. After changing the feature mapping to 8 for hidden layer-1 and 16 for hidden layer-2 we got the accuracy of 0.9860000014305115. After changing the feature mapping to 64 for hidden

layer-1 and 128 for hidden layer-2 we got the accuracy of 0.989300012588501. For both the scenarios the accuracy has a very slight increase when compared to the default accuracy. This shows that the images we have don't contain too many complex features which is true since they are numerical digits with very low image resolution. So even fewer feature mapping space aka channels are able to capture all the required features for optimal classification of the digits.

The default number of neurons in the fully connected layer is 128 in layer 1 and 64 in layer 2. After decreasing the number of fully connected layers to 4 and 2 for layer 1 and layer 2 we got the accuracy of 0.6674000024795532. This shows a very significant decrease in accuracy compared to the original model.This can be due to the significant loss of information. The output of the dense layer is of length 21632 which means it has learned a rich set of features from the input data. However, reducing it to just 4 neurons in the next layer means you are compressing a large amount of information into a very small representation. This made the model lose the ability of classifying the images which is a case of underfitting due to lack of parameters.

Conclusion:

This Project about Convolution neural networks and Multi layer perceptron makes us understand the intricacies of hyperparameter tuning and how the change in hyperparameters is influencing the output accuracies. The MLP model takes us through different steps involved in a neural network in a detailed way. We are exposed to the mathematics behind back propagation and we are also able to do the partial differentiation and gradient calculation for the given 2-Nh-1 model. The impact of activation function and loss function is analysed and studied. We can also conclude that the modification of the number of hidden nodes will have an impact on the degree of freedom for the classification boundary. Having too few hidden nodes will cause the model to converge a lot slowly and to a worse value and having too many hidden nodes can cause overfitting.
From the CNN model we understand the impact of hyperparameters namely learning rate, kernel size, number of neurons in fully connected layer and feature mapping in hidden layer 1 and 2. A lower learning rate is a generally better way to reach optimal decision boundary than keeping the learning rate high. A bigger kernel size can capture the intricacies and reduce noise due to the lower resolution. We should also take care about the nodes in the fully connected layer as keeping too few would result in features being lost. With regards to the feature map it should be large enough to capture the complex structure of the image. Although for simple images having a smaller feature map might be better as it could result in faster processing. These are my takeaways from the project.