Machine Learning Report on Assignment – 3 Multi-Layer Perceptron Classifier

By: Group 32
Krinal Patel (21CS60R39)
Sarvesh Gupta (21CS60R53)

Dataset: Optical Recognition of Handwritten Digits

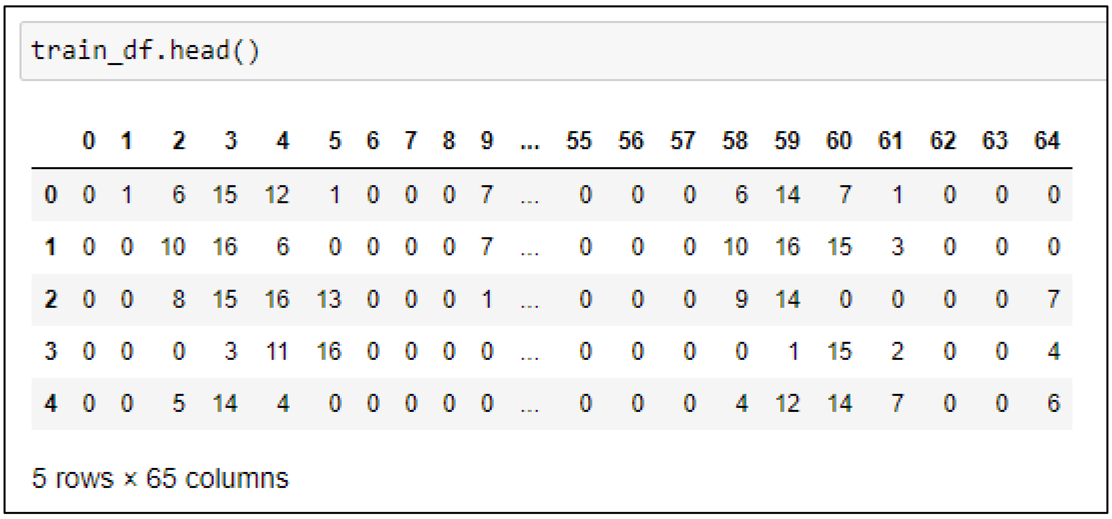
Details of Dataset:
We used pre-processing programs made available by NIST to extract normalized bitmaps of handwritten digits from a pre-printed form. From a total of 43 people, 30 contributed to the training set and different 13 to the test set. 32x32 bitmaps are divided into nonoverlapping blocks of 4x4 and the number of on pixels are counted in each block. This generates an input matrix of 8x8 where each element is an integer in the range 0..16. This reduces dimensionality and gives invariance to small distortions.

Link of Dataset:
https://archive.ics.uci.edu/ml/datasets/optical+recognition+of+handwritten+digits

Procedure:

Data Pre-Processing:
The Original Dataset is in the form of BITMAP, with some details of the dataset. And from that dataset file, a comma separated file is created, in which all the non-overlapping 4x4 blocks are treated as one entity, and that entity stores the value of number of 1's in that small part of the bitmap. This Comma Separated Dataset is Already provided to us along with original dataset. Therefore, we are directly using it.
So, we have 2 different datasets, one for training, and other for testing. Each dataset has 64 i nput attributes, and the last column tells us the actual value stored in that bitmap.

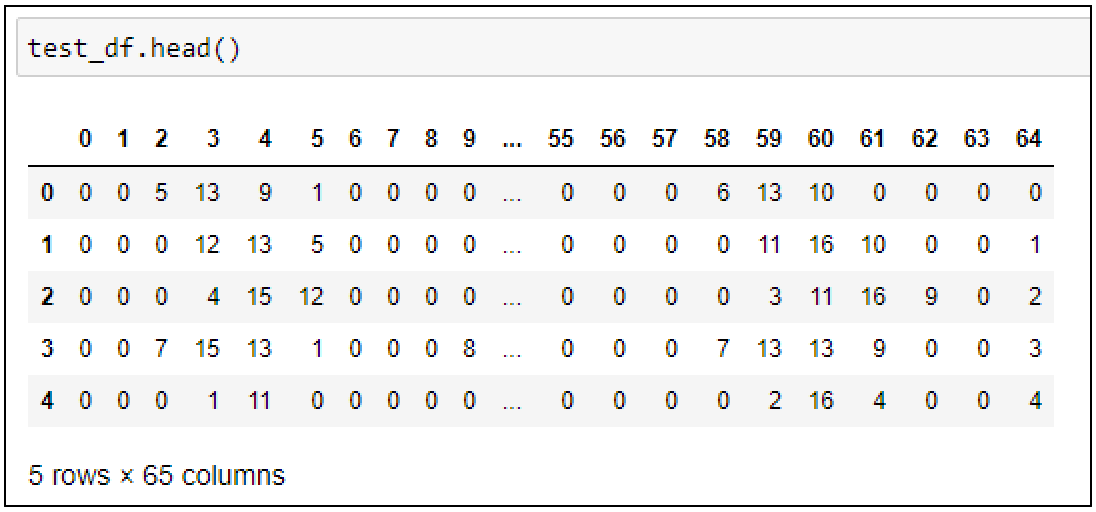
Look of Training and Testing Dataset:



```
train_df.head()
```

	0	1	2	3	4	5	6	7	8	9	...	55	56	57	58	59	60	61	62	63	64
0	0	1	6	15	12	1	0	0	0	7	...	0	0	0	6	14	7	1	0	0	0
1	0	0	10	16	6	0	0	0	0	7	...	0	0	0	10	16	15	3	0	0	0
2	0	0	8	15	16	13	0	0	0	1	...	0	0	0	9	14	0	0	0	0	7
3	0	0	0	3	11	16	0	0	0	0	...	0	0	0	0	1	15	2	0	0	4
4	0	0	5	14	4	0	0	0	0	0	...	0	0	0	4	12	14	7	0	0	6

5 rows × 65 columns

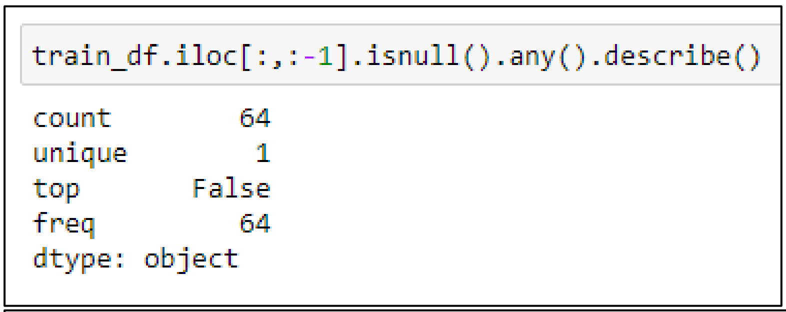


```
test_df.head()
```

	0	1	2	3	4	5	6	7	8	9	...	55	56	57	58	59	60	61	62	63	64
0	0	0	5	13	9	1	0	0	0	0	...	0	0	0	6	13	10	0	0	0	0
1	0	0	0	12	13	5	0	0	0	0	...	0	0	0	0	11	16	10	0	0	1
2	0	0	0	4	15	12	0	0	0	0	...	0	0	0	0	3	11	16	9	0	2
3	0	0	7	15	13	1	0	0	0	8	...	0	0	0	7	13	13	9	0	0	3
4	0	0	0	1	11	0	0	0	0	0	...	0	0	0	0	2	16	4	0	0	4

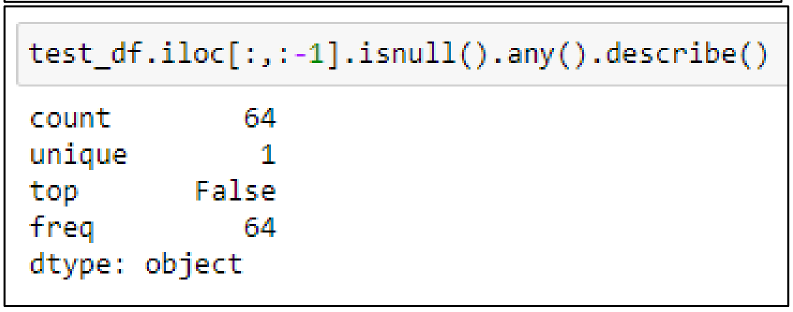
5 rows × 65 columns

Details of Both the Dataset, having 0 NULL Values:



```
train_df.iloc[:,:-1].isnull().any().describe()
```

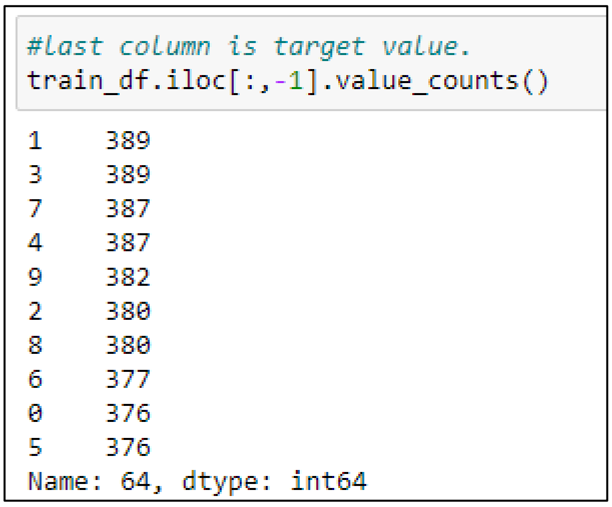
```
count         64
unique         1
top        False
freq          64
dtype: object
```



```
test_df.iloc[:,:-1].isnull().any().describe()
```

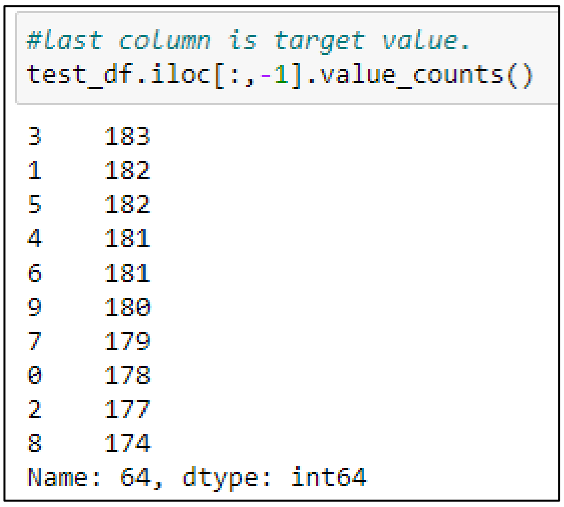
```
count         64
unique         1
top        False
freq          64
dtype: object
```

Count of Class of Each Category in Training and Test Dataset:



```
#last column is target value.
train_df.iloc[:,-1].value_counts()
```

```
1    389
3    389
7    387
4    387
9    382
2    380
8    380
6    377
0    376
5    376
Name: 64, dtype: int64
```



```
#last column is target value.
test_df.iloc[:,-1].value_counts()
```

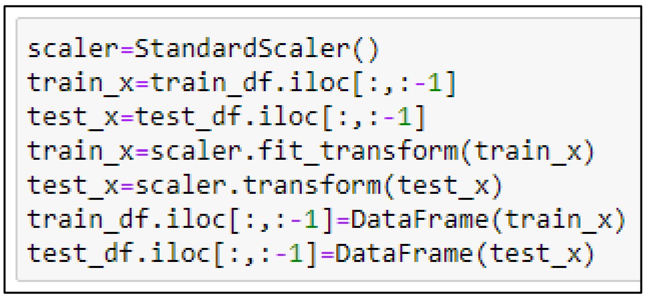
```
3    183
1    182
5    182
4    181
6    181
9    180
7    179
0    178
2    177
8    174
Name: 64, dtype: int64
```

Here we can see that output classes are balanced in training as well as testing dataset

Data Standardization:
Rescaling the distribution of values so that the mean of observed values is 0 and the standard deviation is 1 is the process of standardising a dataset. This is similar to removing the mean value from the data.

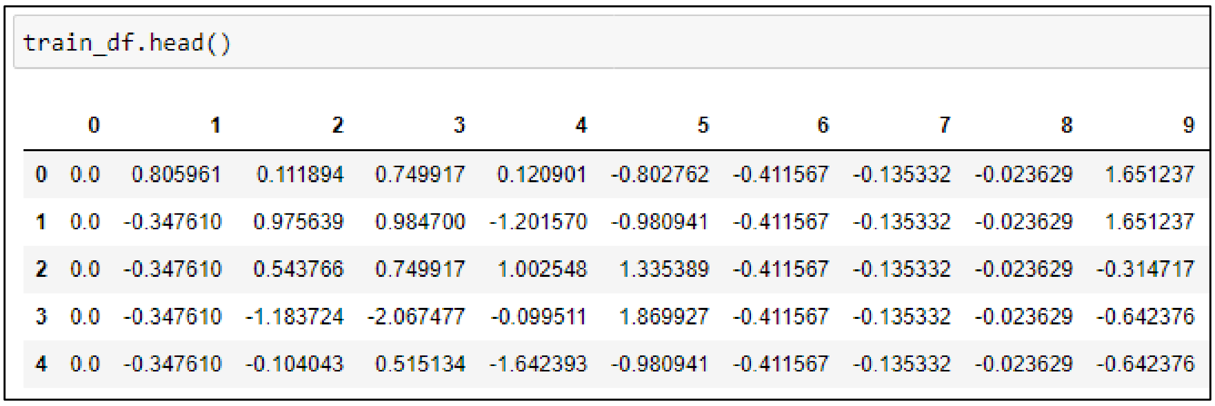
- MinMax Scaler Transform
- Standard Scaler Transform

After testing we found that Standard Scaler is providing better results, therefore we have used standard scaler in the code.



```
scaler=StandardScaler()
train_x=train_df.iloc[:,:-1]
test_x=test_df.iloc[:,:-1]
train_x=scaler.fit_transform(train_x)
test_x=scaler.transform(test_x)
train_df.iloc[:,:-1]=DataFrame(train_x)
test_df.iloc[:,:-1]=DataFrame(test_x)
```

Top 5 Rows After Standard Scaling:

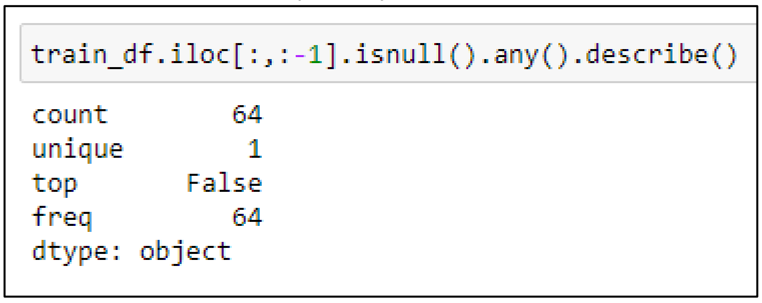


```
train_df.head()
```

	0	1	2	3	4	5	6	7	8	9
0	0.0	0.805961	0.111894	0.749917	0.120901	-0.802762	-0.411567	-0.135332	-0.023629	1.651237
1	0.0	-0.347610	0.975639	0.984700	-1.201570	-0.980941	-0.411567	-0.135332	-0.023629	1.651237
2	0.0	-0.347610	0.543766	0.749917	1.002548	1.335389	-0.411567	-0.135332	-0.023629	-0.314717
3	0.0	-0.347610	-1.183724	-2.067477	-0.099511	1.869927	-0.411567	-0.135332	-0.023629	-0.642376
4	0.0	-0.347610	-0.104043	0.515134	-1.642393	-0.980941	-0.411567	-0.135332	-0.023629	-0.642376

Q1 Find the number of nodes in input and output layer according to the dataset and justify it in the report. Specify and justify any other hyper parameter that is/are needed.

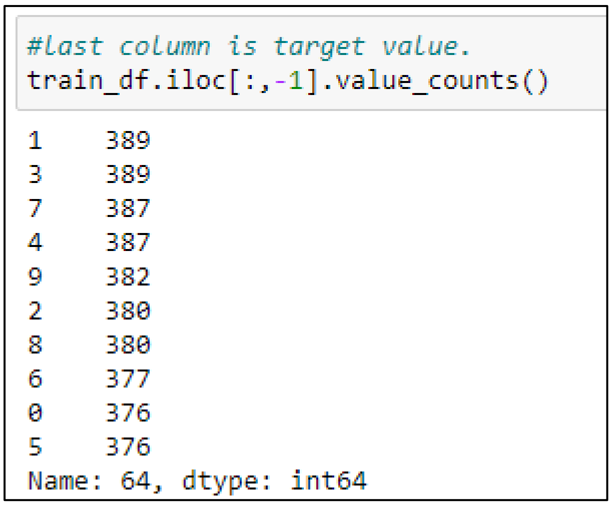
Number of Nodes in Input Layers: **64**



```
train_df.iloc[:,:-1].isnull().any().describe()
```

```
count         64
unique         1
top        False
freq          64
dtype: object
```

Number of Nodes in Output Layer: **10** (0 to 9)



```
#last column is target value.
train_df.iloc[:,-1].value_counts()
```

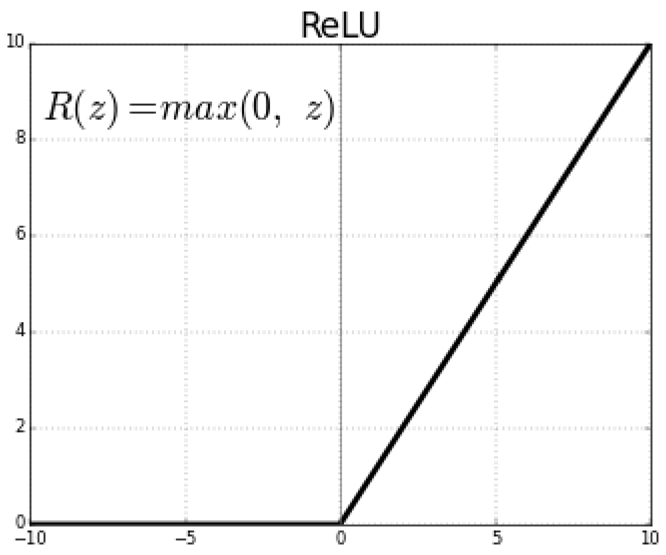
```
1    389
3    389
7    387
4    387
9    382
2    380
8    380
6    377
0    376
5    376
Name: 64, dtype: int64
```

This Dataset has 64 features (That are from column 0 to column 63 of the dataset), and those features are treated as nodes in input layer. And the last column of the Dataset has ac tual output, and that output is classified into 10 categories. And these 10 categories are nodes of output layer.

Hyper Parameters used in this assignment are:

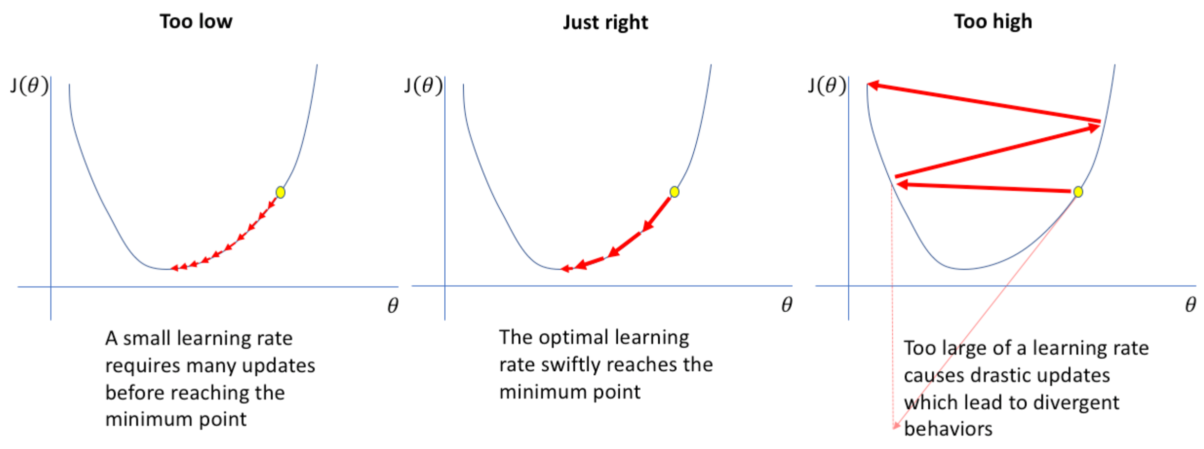
- Activation Function: In a neural network, an activation function specifies how the weighted sum of the input is turned into an output from a node or nodes in a layer.
    - Linear
    - ReLU
    - Tanh
    - Sigmoid
    - Softmax

  The Softmax function is used as activation function, in the output layer, that predict a multinomial probability distribution. Basically, it assigns probability between 0 and 1 to each output class and total sum of probability of each class is 1.
  To understand the complex task a non-linear function should be used in the hidden l ayers. Therefore, we have explored ReLU, tanh and Sigmoid functions and found out that ReLU is providing better accuracy.



```python
def get_model(layer_info):
    layers = []
    if(len(layer_info) == 2):
        layers.append(nn.Linear(layer_info[0],layer_info[1]))
        layers.append(nn.LogSoftmax(dim=1))
    elif(len(layer_info) == 3):
        layers.append(nn.Linear(layer_info[0],layer_info[1]))
        layers.append(nn.ReLU())
        layers.append(nn.Linear(layer_info[1],layer_info[2]))
        layers.append(nn.LogSoftmax(dim=1))
    elif(len(layer_info) == 4):
        layers.append(nn.Linear(layer_info[0],layer_info[1]))
        layers.append(nn.ReLU())
        layers.append(nn.Linear(layer_info[1],layer_info[2]))
        layers.append(nn.ReLU())
        layers.append(nn.Linear(layer_info[2],layer_info[3]))
        layers.append(nn.LogSoftmax(dim=1))
    model=nn.Sequential(*layers)
    return model
```

- Learning Rate: It controls how much to change the model, whenever an error is detected. All the Learning Rates given in question are taken under consideration.


Too low
Just right
Too high
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Reference: https://www.jeremyjordan.me/nn-learning-rate/

- Number of Hidden Layer: They are considered as given in question.
- Number of Nodes in Hidden Layers: They are considered as given in question.
- Epochs: Number of epochs are the number of time whole training dataset is shown to the network. Number of Epochs for our system is 1000 (More than that will take too much time because of system limitation).
- Batch Size: Mini batch size is the number of sub samples given to the network after which parameter update happens. We have selected 64 as batch size.

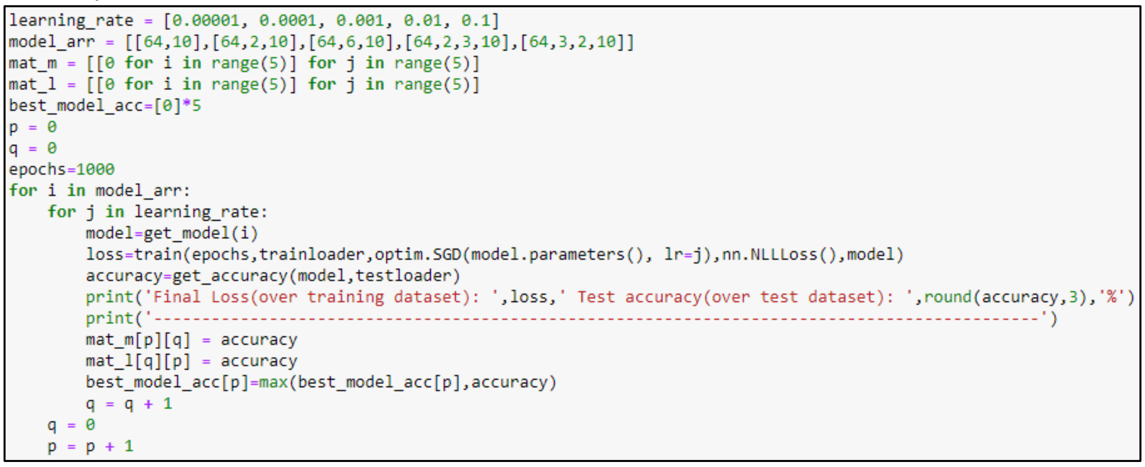
Neural Network Training:

Here given task is a multiclass classification problem. Thus, we have used LogSoftMax activation function in output layer. Furthermore, we have used SGD (stochastic gradient descent) optimizer. SGD optimizer calculate the error and backpropagate the error. This process is repeater for many iterations to achieve convergence. Here as we have used LogSoftMax activation function in output layer, it is recommended to use NLLLoss (negative log likelihood loss). Alternatively, you may use CrossEntropyLoss, which combines LogSoftMax and NLLLoss.

Krinal Patel (21CS60R39)
Sarvesh Gupta (21CS60R53)

Q2 Vary the number of hidden layers and number of nodes in each hidden layer. Consider the following architectures.
(a) 0 hidden layer
(b) 1 hidden layer with 2 nodes
(c) 1 hidden layer with 6 nodes
(d) 2 hidden layers with 2 and 3 nodes respectively
(e) 2 hidden layers with 3 and 2 nodes respectively
For each of the architectures, vary the learning rates as 0.1, 0.01, 0.001, 0.0001, 0.00001.

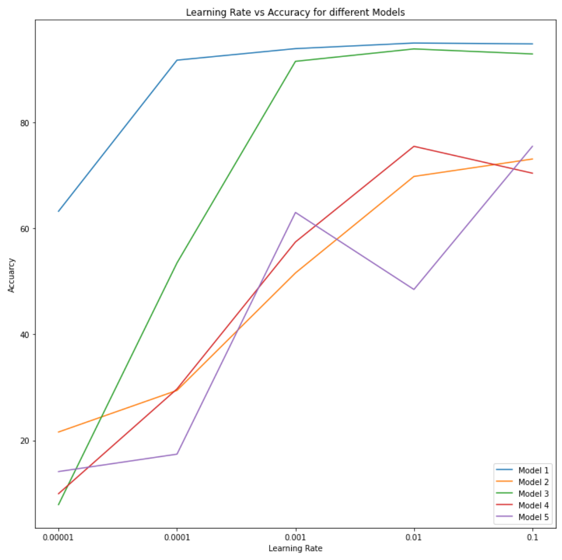
In this all the above architectures are implemented by varying to all the learning rates given in the question:



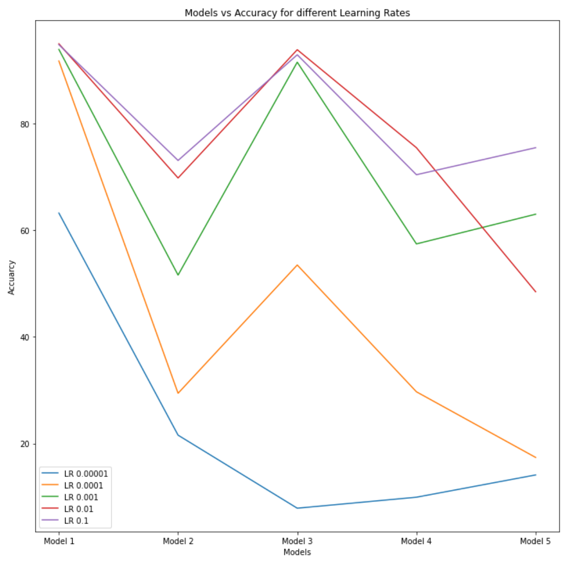
```
learning_rate = [0.00001, 0.0001, 0.001, 0.01, 0.1]
model_arr = [[64,10],[64,2,10],[64,6,10],[64,2,3,10],[64,3,2,10]]
mat_m = [[0 for i in range(5)] for j in range(5)]
mat_l = [[0 for i in range(5)] for j in range(5)]
best_model_acc=[0]*5
p = 0
q = 0
epochs=1000
for i in model_arr:
    for j in learning_rate:
        model=get_model(i)
        loss=train(epochs,trainloader,optim.SGD(model.parameters(), lr=j),nn.NLLLoss(),model)
        accuracy=get_accuracy(model,testloader)
        print('Final Loss(over training dataset): ',loss,' Test accuracy(over test dataset): ',round(accuracy,3),'%')
        print('-----------------------------------------------------------------------------------------')
        mat_m[p][q] = accuracy
        mat_l[q][p] = accuracy
        best_model_acc[p]=max(best_model_acc[p],accuracy)
        q = q + 1
    q = 0
    p = p + 1
```

Q3 Plot graph for the results in the previous parts with respect to accuracy.

Learning Rate VS Accuracy for different Models



Models VS Accuracy for different Learning Rate



Q4 Mention the architecture and hyper parameters of the best-found model in the report.
As from the Graph we can see that, Model 1 has better accuracy than all the others models, and for Learning Rates 0.1 is providing the better accuracy. And to be precise, Model 1(0 hidden layer) with Learning Rate of 0.01 has the accuracy as per our observation (Accuracy: 94.93%) (for 1000 epochs).

<u>Architecture and Hyper-parameter</u>:
64 Nodes for Input Layer
0 Hidden Layer
10 Nodes for Output Layer
Learning Rate: 0.01
Activation Function is LogSoftmax for Outer Layer
Epochs: 1000

There can be many reasons for the result to be like this, discuss each reason point by point:

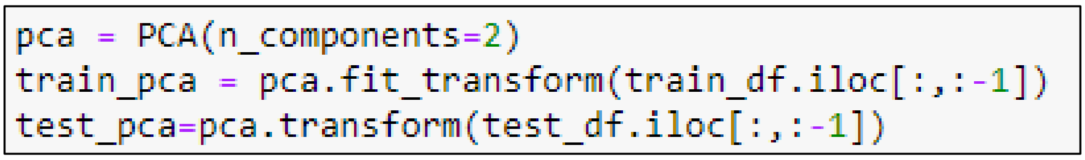
- <u>Hidden Layer and Number of Nodes in Hidden Layer</u>:
  Larger the number of hidden layers, longer it will take to converge, but will solve more complex problems, but here input layer has 64 nodes and hidden layers have very less no of nodes. Therefore, they might be giving poor results.
- <u>Learning Rate</u>:
  It controls how much to change the model, whenever an error is detected. Smaller learning rate can cause the convergence process to stuck. Furthermore, we have trained for 1000 epochs only. And larger learning rate converges faster. Thus, for 1000 epochs, Learning Rate 0.01 is providing better result.

Q5 Reduce the feature dimension of the data into a two-dimensional feature space using Principal Component Analysis (PCA). Plot the reduced dimensional data in a 2d plane. In the plot, all data points of a single class should have same colour and data points from different classes should have different colours.

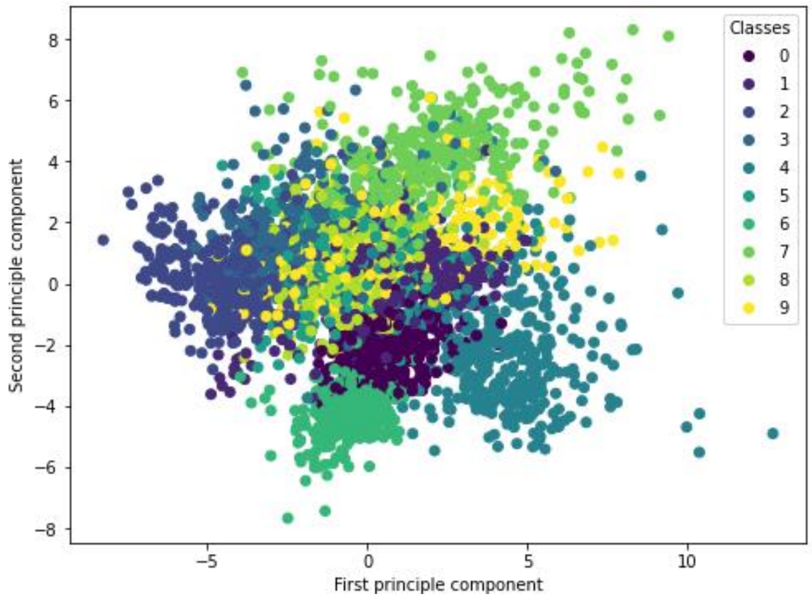
Principal Component Analysis (PCA) is a statistical process that turns a set of correlated variables into a set of uncorrelated variables via an orthogonal transformation. We know that, it is computationally expensive to run machine learning model on high dimensional data. Furthermore, it is also difficult to visualize the data in high dimension. Thus, PCA is useful in reducing the high-dimensional data to low dimensional data with retaining most of the information.

Here, we have already applied normalization on the data. Thus, we can directly use PCA. We have selected 2 principal components using PCA.

PCA Implementation in the Code:



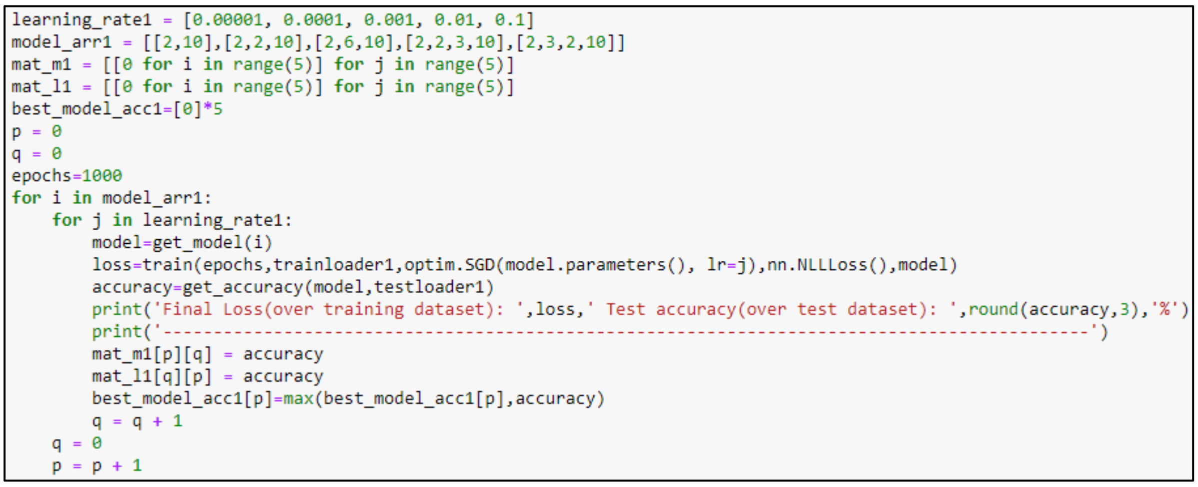
```
pca = PCA(n_components=2)
train_pca = pca.fit_transform(train_df.iloc[:,:-1])
test_pca=pca.transform(test_df.iloc[:,:-1])
```



Q6 Apply MLP of step 2 in the reduced feature space. Compare with classification output generated from step 2. you may use bet learning rate obtained from step 3

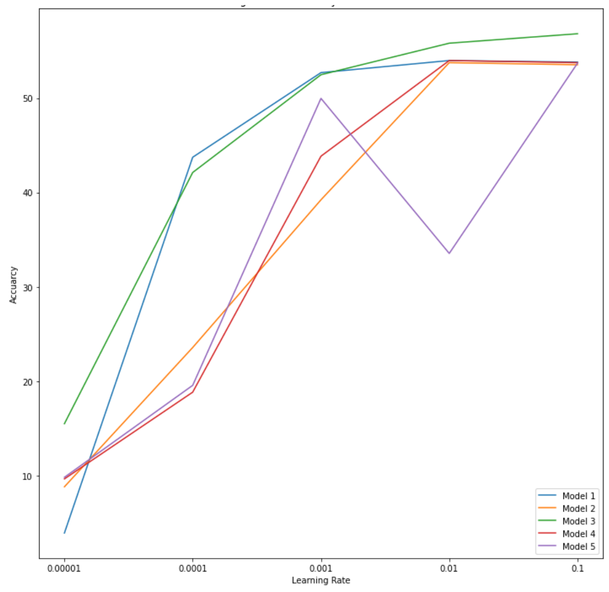
After applying PCA we have only two input features. Thus, input layer has only 2 nodes and output layers has 10 nodes.

In this all the above architectures are implemented by varying to all the learning rates given in the question 2:

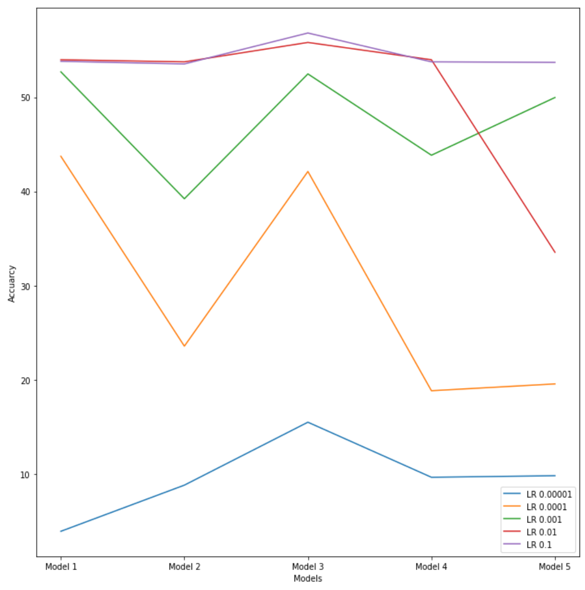


```
learning_rate1 = [0.00001, 0.0001, 0.001, 0.01, 0.1]
model_arr1 = [[2,10],[2,2,10],[2,6,10],[2,2,3,10],[2,3,2,10]]
mat_m1 = [[0 for i in range(5)] for j in range(5)]
mat_l1 = [[0 for i in range(5)] for j in range(5)]
best_model_acc1=[0]*5
p = 0
q = 0
epochs=1000
for i in model_arr1:
    for j in learning_rate1:
        model=get_model(i)
        loss=train(epochs,trainloader1,optim.SGD(model.parameters(), lr=j),nn.NLLLoss(),model)
        accuracy=get_accuracy(model,testloader1)
        print('Final Loss(over training dataset): ',loss,' Test accuracy(over test dataset): ',round(accuracy,3),'%')
        print('---------------------------------------------------------------------------------------------')
        mat_m1[p][q] = accuracy
        mat_l1[q][p] = accuracy
        best_model_acc1[p]=max(best_model_acc1[p],accuracy)
        q = q + 1
    q = 0
    p = p + 1
```

Learning Rate VS Accuracy for different Models



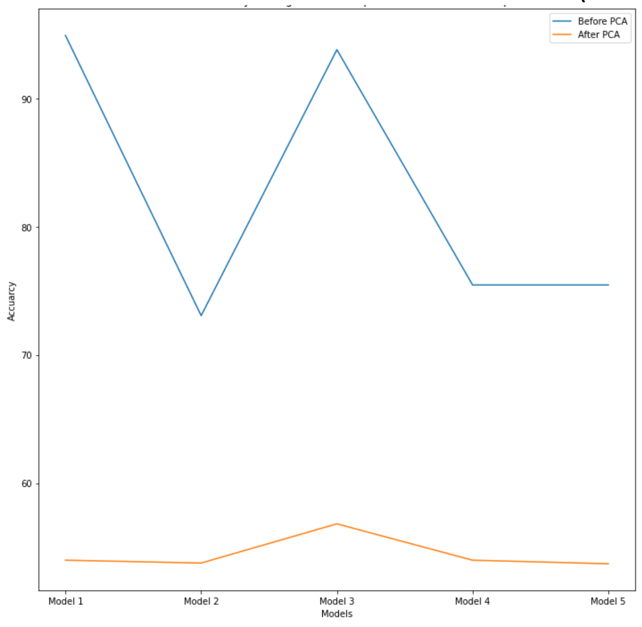
Model VS Accuracy for different Learning Rate



Here, we can see that accuracy is reduced after PCA, which is expected as we have reduced 64-dimensional data to 2-dimensional data.

Conclusion:

- Before PCA:
  - Model 1 (0 hidden layer) has best accuracy among all the models.
  - Learning Rate is 0.01
- After PCA:
  - Model 3 (1 hidden layer with 6 nodes) has best accuracy among all the models.
  - Learning Rate is 0.1
- Activation function for the outer layer should be Softmax, and for hidden layer ReLU is used activation function.
- Comparison between Best Accuracies of different models (Before PCA and After PCA)


Accuracy
Models
Model 1
Model 2
Model 3
Model 4
Model 5
Before PCA
After PCA