q1

April 3, 2020

1 Question 1

2 Perform PCA over all the images in the dataset.

3 Importing Librarys

[15]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
from PIL import Image
from mpl_toolkits.mplot3d import Axes3D
import glob
from IPython.display import Image, display
```

4 Code for reconstruction of Images after applying PCA

[2]:
```
def reconstruct_images(x_pca):
    count = 1
    base_dir = "result/"
    for i in x_pca:
        reshaped_i = i.reshape((64,64))
        unint_reshaped_i = reshaped_i.astype(np.uint8)
        i = Image.fromarray(unint_reshaped_i,'L')
        i.rotate(180)
        path = base_dir+str(count)+".jpg"
        i.save(path,"JPEG")
        count = count+1
    return
```

5 Loading data

[3]:
```
path = 'G:\second_sem\SMAI\Assignment_3\q1\A3\dataset'
images = []
count = 0
classes = []
for f in os.listdir(path):
    label = f[1:f.find("_")]
    images.append(np.asarray(Image.open(path +'/'+f).convert('L').resize((64,␣
 ↪64))).flatten())
    classes.append(int(label))
```

[4]:
```
xtrain = np.array(images)
```

6 Calculating the mean matrix from the original matrix and subtracting the mean matrix from the original matrix and forming the covariance matrix from the resultant matrix. After that corresponding eigen values and eigen vectors are calculated.

[5]:
```
x_mean = np.mean(xtrain, axis =0)
x_center = xtrain - x_mean
x_cov = np.cov(x_center.T)
x_eigenvalues, x_eigenvectors = np.linalg.eig(x_cov)
```

7 Sorting the eigen values and their corrsponding eigen vector in decreasing order.

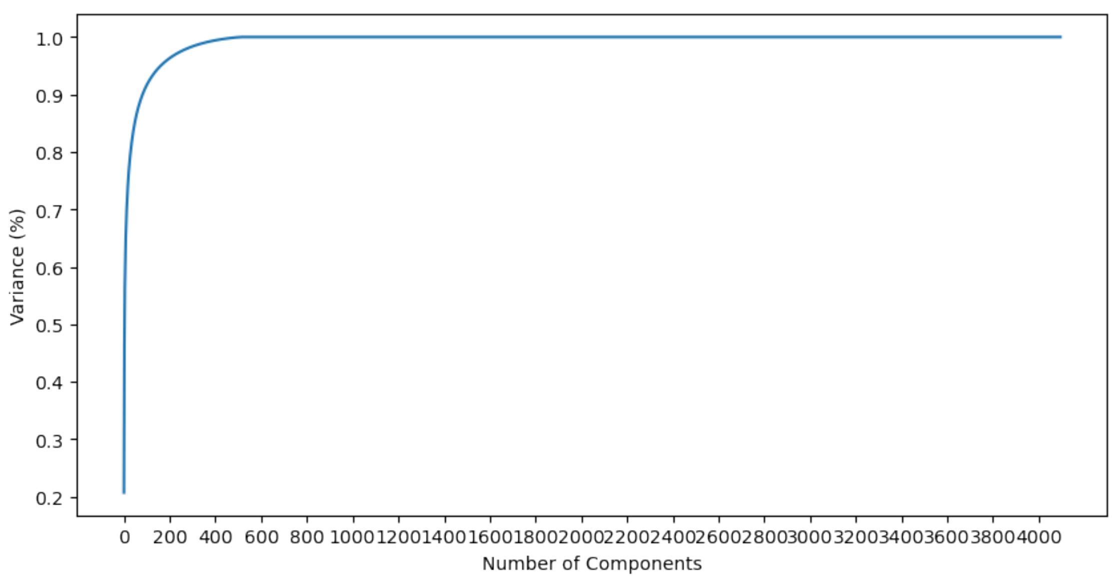
[6]:
```
indexes = x_eigenvalues.argsort()[::-1]
eigenvalues = x_eigenvalues[indexes]
eigenvectors = x_eigenvectors[:,indexes]
```

8 Plotting the graph between varaince error and the number of components And selecting the number of principal component where the error < 20%.

[8]:
```
var = []
k = 0
n_comp = 0
flag = 0
for i in eigenvalues:
    error = (np.abs(i)/np.sum(eigenvalues))
```

```
    k = k+1
    ratio = np.sum(eigenvalues[:k])/np.sum(eigenvalues[:])
    if(ratio > 0.89 and flag == 0):
        n_comp = k
        flag = 1
    var.append(error)
plt.figure()
plt.rcParams['figure.figsize'] = [10, 5]
plt.rcParams['figure.dpi'] = 100
plt.plot(np.cumsum(var))
plt.xlabel('Number of Components')
plt.ylabel('Variance (%)')
plt.xticks(np.arange(0, 4096+1, 200.0))
plt.show()
```

```
C:\Users\Indranil\Anaconda3\lib\site-packages\numpy\core\numeric.py:538:
ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```



9 The number of Principal component is 74 which is calculated in previous block.

[9]:
```
n_comp
```

[9]: 74

[10]:
```
n_components = n_comp
```

10 Reconstruction of dataset using the principal components which is 74 by the dot product of the transpose of the selected eigen vectors and and the transpos of the original dataset

[11]:
```
red_eigenvec = eigenvectors[:,:n_components]
x_pca = red_eigenvec.T.dot(xtrain.T)
```

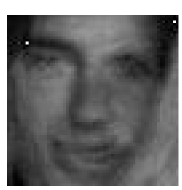
11 Reconstructing the images using the principal components

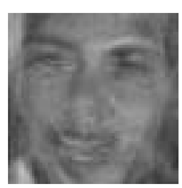
[12]:
```
new_eigvec = eigenvectors[:,:n_components]
z=new_eigvec.T.dot(xtrain.T)
re_xtrain =new_eigvec.dot(z)
reconstruct_images(re_xtrain.T)
```

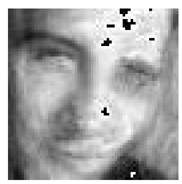
```
C:\Users\Indranil\Anaconda3\lib\site-packages\ipykernel_launcher.py:6:
ComplexWarning: Casting complex values to real discards the imaginary part
```

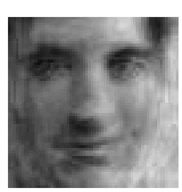
12 The reconstructed images using the pcincipal components

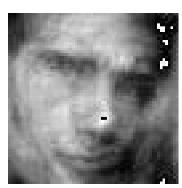
[20]:
```
for imageName in glob.glob('./result/*.JPG'): #assuming JPG
    display(Image(filename=imageName))
```

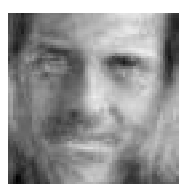






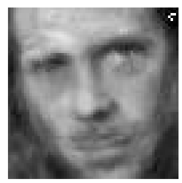


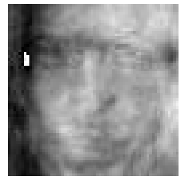


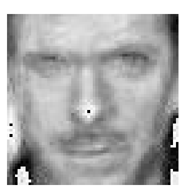


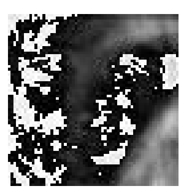


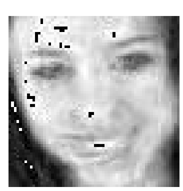


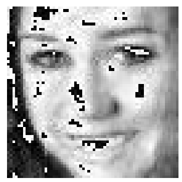



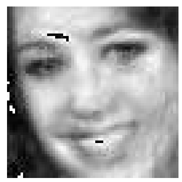


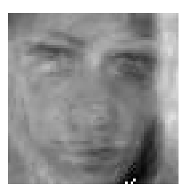






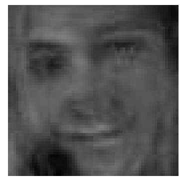


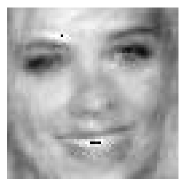


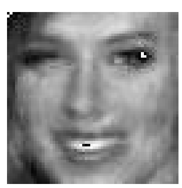


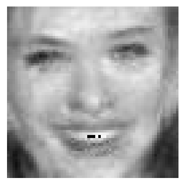


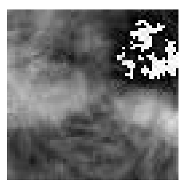


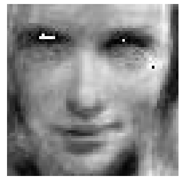



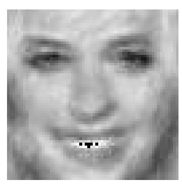


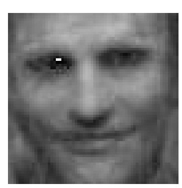


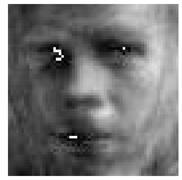


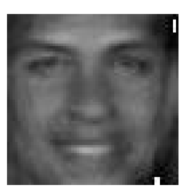


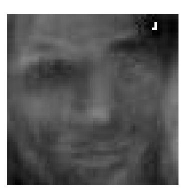


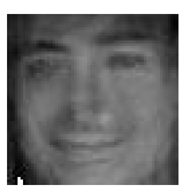


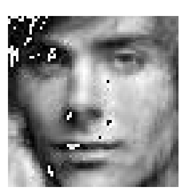


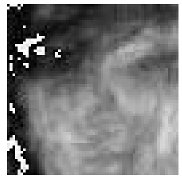


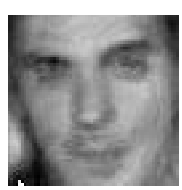


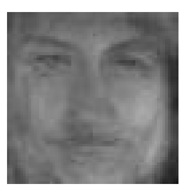






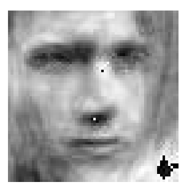


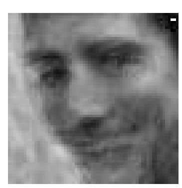








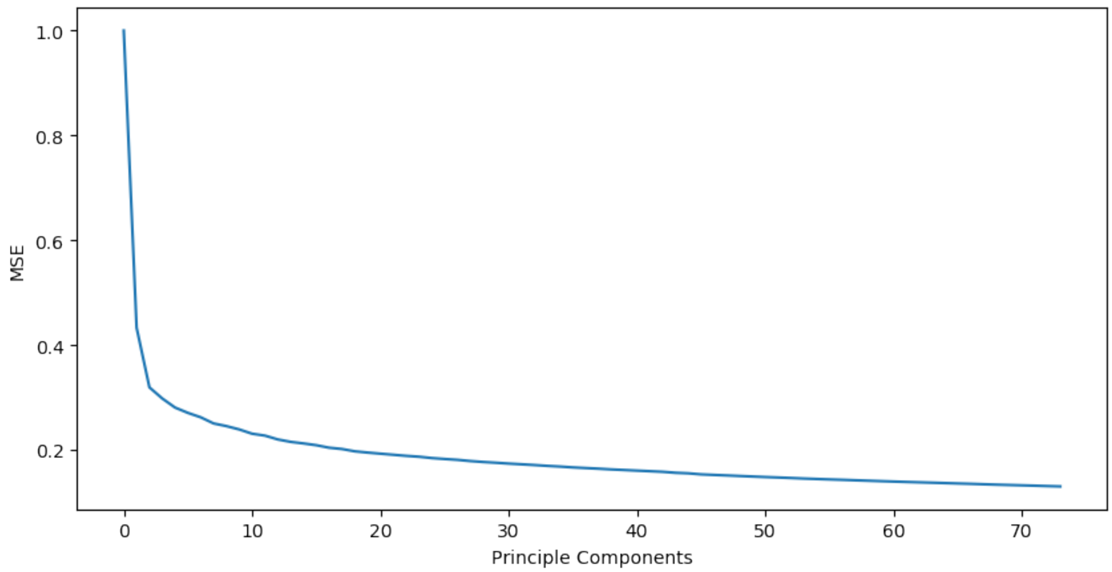





13 Plotting a graph showing the total mean square error over all train images vs the number of principal components used to reconstruct.

[119]:
```
mse=[]
comp=[]
for i in range(n_components):
    new_eigvec=eigenvectors[:,0:i]
    z=new_eigvec.T.dot(xtrain.T)
    xsvdtrain=new_eigvec.dot(z)
    error=0
    for j,k in enumerate(xtrain):
        norm_diff = np.linalg.norm(xtrain[j] - xsvdtrain.T[j])
        avg_norm_diff = norm_diff/np.linalg.norm(xtrain[j])
        error += avg_norm_diff
    error /= len(xtrain)
    comp.append(i)
    mse.append(error)
plt.rcParams['figure.figsize'] = [10, 5]
plt.rcParams['figure.dpi'] = 100
plt.plot(comp,mse)
plt.xlabel('Principle Components')
plt.ylabel('MSE')
```

[119]:
```
Text(0, 0.5, 'MSE')
```



14 Mean squared Error

[120]:
```
MSE = 0
for j,k in enumerate(xtrain):
    norm_diff = np.linalg.norm(xtrain[j] - re_xtrain.T[j])
    avg_norm_diff = norm_diff/np.linalg.norm(xtrain[j])
    MSE += avg_norm_diff
MSE /= len(xtrain)
MSE*100
```

[120]: 12.991960709721035

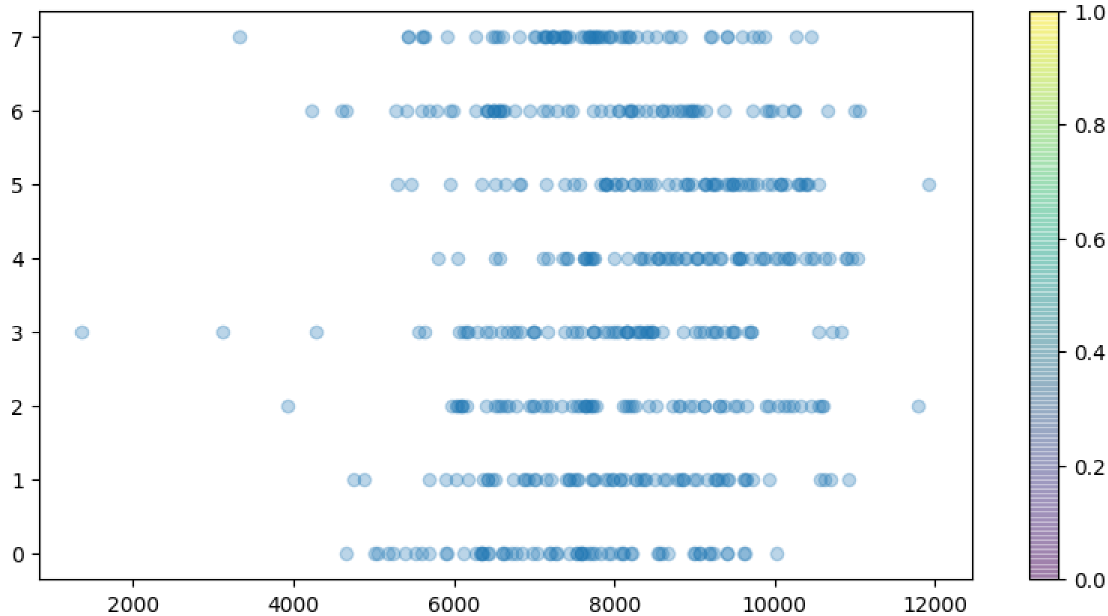
15 Checking if the accuracy is more than 80%

[121]:
```
ch_eigensum = np.sum(eigenvalues[:n_components])
tot_eigensum = np.sum(eigenvalues[:])
acc = ch_eigensum/tot_eigensum
if(acc>0.8):
    print("True")
else:
    print("False")
```

```
True
```

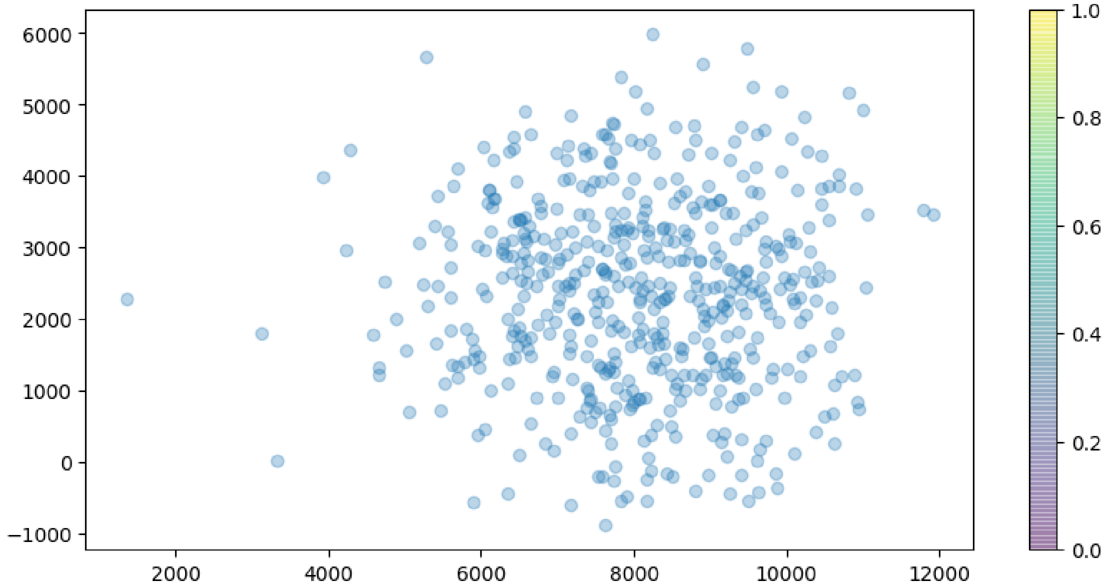
16 scatterplots to examine how the images are clustered in the 1D space using the number of principal components.

[124]:
```
oned_eigenvec = eigenvectors[:,:1]
onedx_pca = oned_eigenvec.T.dot(xtrain.T)
plt.scatter(onedx_pca.T[:,:1],classes, alpha=0.3,cmap='red')
plt.colorbar();
```



17 scatterplots to examine how the images are clustered in the 2D space using the number of principal components.

[125]:
```
twod_eigenvec = eigenvectors[:,:2]
twodx_pca = twod_eigenvec.T.dot(xtrain.T)
plt.scatter(twodx_pca.T[:,:1],twodx_pca.T[:,1:2], alpha=0.3,cmap='red')
plt.colorbar();
```

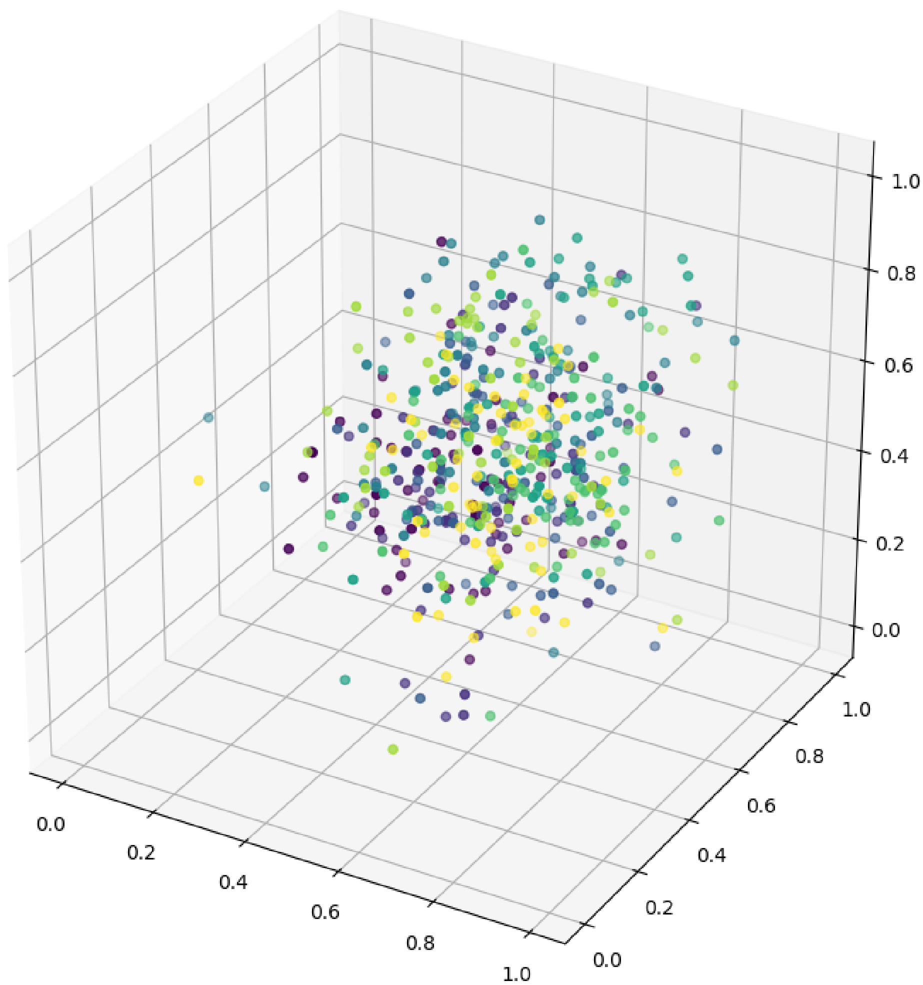


18 scatterplots to examine how the images are clustered in the 3D space using the number of principal components.

[126]:
```
thd_eigenvec = eigenvectors[:,:3]
thdx_pca = thd_eigenvec.T.dot(xtrain.T)
x = np.array(thdx_pca.T[:,:1]).astype(float)
x = (x - np.min(x))/np.ptp(x)
y = np.array(thdx_pca.T[:,1:2]).astype(float)
y = (y - np.min(y))/np.ptp(y)
z = np.array(thdx_pca.T[:,2:3]).astype(float)
z = (z - np.min(z))/np.ptp(z)
fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(projection='3d')
ax.scatter(x,y,z,c=classes,zdir='z',depthshade=True)
```

```
C:\Users\Indranil\Anaconda3\lib\site-packages\ipykernel_launcher.py:3:
ComplexWarning: Casting complex values to real discards the imaginary part
  This is separate from the ipykernel package so we can avoid doing imports
until
C:\Users\Indranil\Anaconda3\lib\site-packages\ipykernel_launcher.py:5:
ComplexWarning: Casting complex values to real discards the imaginary part
  """
C:\Users\Indranil\Anaconda3\lib\site-packages\ipykernel_launcher.py:7:
ComplexWarning: Casting complex values to real discards the imaginary part
  import sys
```

[126]: <mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x24e47a71648>



[ ]:

q2

April 3, 2020

1 Question 2

2 Implement logistic regression to classify the images provided in the dataset.

3 Importing Libraries

[1]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
from PIL import Image
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
```

4 Appying the principal component analysis to get the transformed trainning set where the number of principal component is 74. Returnning the transformed traning set and the resultant eigenvectors which will be used to tranformed the test dataset also.

[2]:
```
def apply_pca(train):
    n_components = 74
    x_mean = np.mean(train, axis =0)
    x_center = train - x_mean
    x_cov = np.cov(x_center.T)
    x_eigenvalues, x_eigenvectors = np.linalg.eig(x_cov)
    indexes = x_eigenvalues.argsort()[::-1]
    eigenvalues = x_eigenvalues[indexes]
    eigenvectors = x_eigenvectors[:,indexes]
    red_eigenvec = eigenvectors[:,:n_components]
    x_pca = red_eigenvec.T.dot(train.T)
    global_eigenvector = eigenvectors
```

```
    return x_pca.T,eigenvectors
```

5 Transforming the test data set by applying the principal component analysis.

[3]:
```
def transform_pca(eigenvectors, test):
    n_components = 74
    red_eigenvec = eigenvectors[:,:n_components]
    test_pca = red_eigenvec.T.dot(test.T)
    return test_pca.T
```

6 Code to calculate accuracy

[4]:
```
def accuracy(predictions, y):
    return ((predictions == y).mean()*100)
```

7 Sigmoid function

$$h_\theta(x) = \frac{1}{1 + e^{\theta^\top x}} \tag{1}$$

[5]:
```
def sigmoid_function(x):
    g = 1/(1 + np.exp(-x))
    return g
```

8 Calculation of the cost:

$$\begin{aligned} J(\theta) &= \frac{1}{m} \sum_{i=1}^{m} \mathrm{Cost}(h_\theta(x^{(i)}), y^{(i)}) && (2) \\ &= -\frac{1}{m} [\sum_{i=1}^{m} y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))] && (3) \\ & && (4) \end{aligned}$$

[6]:
```
def calculate_cost(x,y,w,h):
    total_cost = np.sum(-y * np.log(h) - (1 - y) * np.log(1 - h))
    cost = total_cost/len(y)
    return cost
```

9 Calulation of the gradient

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \tag{5}$$

[7]:
```
def claculate_gradient(x,y,h):
    gradient = (y-h).dot(x)/len(y)
    return gradient
```

10 Trainning the data :- One vs Rest classification technique is used. One-vs-Rest classification is a method which involves training N distinct binary classifiers, each designed for recognizing a particular class. Then those N classifiers are collectively used for multi-class classification. I take values of one class and turn them into one, and the rest of classes - into zeros. And everytime cosidering only one class, it converges by running through the number of iterations and calculate the optimized weight corrrsponding to each class.

[8]:
```
def train_data(xtrain,y,numofiter,learningrate):
    x = np.ones(shape=(xtrain.shape[0], xtrain.shape[1] + 1),dtype=complex)
    x[:, 1:] = xtrain
    classes = np.unique(y)
    weights =[]
    for cls in classes:
        weight = np.zeros(x.shape[1],dtype=complex) #dtype=complex
        y_map = []
        for i in y:
            if(i==cls):
                y_map.append(1)
            else:
                y_map.append(0)
        #print("ymap ",y_map)
        for i in range(numofiter):
            h = sigmoid_function(x.dot(weight))
            gradient = claculate_gradient(x,y_map,h)
            weight = weight + learningrate*gradient
        weights.append(weight)
    return weights
```

11 Predict class: Taking each sample and calculating

$$h_\theta(x) = [h_\theta^{(1)}(x), h_\theta^{(2)}(x), h_\theta^{(3)}(x), ...] \tag{6}$$

and taking the maximum value and its corresponding class as the predicted class.

[18]:
```
def predict_class(xval,y,weights):
    x = np.ones(shape=(xval.shape[0], xval.shape[1] + 1),dtype=complex)
    x[:, 1:] = xval
    temp_predictions = []
    for i in x:
        hypothesis = np.zeros(shape=(len(weights)),dtype=complex) #dtype=complex
        k =0
        for weight in weights:
            h = sigmoid_function(i.dot(weight))
            #print("h===== ",h)
            hypothesis[k] = h
            k = k+1
        temp_predictions.append(np.argmax(hypothesis))
    predictions = []
    #print("t_pred ",temp_predictions)
    for indx in temp_predictions:
        #print("y_indx",indx)
        predictions.append(y[indx])
    return predictions
```

12 Loading the tranning data and applying the own PCA on it taking 74 principal components and applying min-max scaler over it.

[260]:
```
path = r"G:\second_sem\SMAI\Assignment_3\q2\dataset"
images = []
labels = []
for f in os.listdir(path):
    label = f[1:f.find("_")]
    images.append(np.asarray(Image.open(path +'/'+f).convert('L').resize((64,␣
 ↪64))).flatten())
    labels.append(label)
train = np.array(images)
scalar = MinMaxScaler()
train = scalar.fit_transform(train)
train,eigenvects = apply_pca(train)
labels = np.array(labels)
```

[261]:
```
xtrain = train[:,:]
ytrain = labels[:]
```

13 Loading the test data and transforming the test data by applying the own PCA on it taking 74 principal components and applying min-max scaler over it.

[262]:
```
vpath = r"G:\second_sem\SMAI\Assignment_3\q2\A3\test"
vimages = []
vlabels = []
for f in os.listdir(vpath):
    label = f[1:f.find("_")]
    vimages.append(np.asarray(Image.open(vpath +'/'+f).convert('L').resize( (64,␣
↪64))).flatten())
    vlabels.append(label)
vtrain = np.array(vimages)
vtrain = scalar.transform(vtrain)
vtrain = transform_pca(eigenvects,vtrain)
vlabels = np.array(vlabels)
xvalidation = vtrain[:,:]
yvalidation = vlabels[:]
```

14 Experiment 1:- The number of iterations = 100000 and the learning rate = 0.0001

[78]:
```
numofiter = 100000
learningrate = 0.0001
weights = train_data(xtrain, ytrain, numofiter,learningrate)
classes = np.unique(ytrain)
predictions1 = predict_class(xvalidation,classes,weights)
```

15 Accuracy

[79]:
```
acc1 = accuracy(predictions1, yvalidation)
acc1
```

[79]:
```
68.75
```

16 Confusion matrix

[80]:
```
print(confusion_matrix(predictions1,yvalidation))
```

```
[[ 0  1  0  0  0  0]
 [ 0 13  0  0  0  0]
 [ 0  3  9  0  0  0]
 [ 0  1  2  0  0  0]
```

```
 [ 0  0  1  0  0  0]
 [ 0  2  0  0  0  0]]
```

17 Classification report

[81]:
```
print(classification_report(predictions1,yvalidation))
```

```
              precision    recall  f1-score   support

          01       0.00      0.00      0.00         1
          03       0.65      1.00      0.79        13
          04       0.75      0.75      0.75        12
          05       0.00      0.00      0.00         3
          06       0.00      0.00      0.00         1
          07       0.00      0.00      0.00         2

    accuracy                           0.69        32
   macro avg       0.23      0.29      0.26        32
weighted avg       0.55      0.69      0.60        32

C:\Users\Indranil\Anaconda3\lib\site-
packages\sklearn\metrics\classification.py:1437: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples.
  'precision', 'predicted', average, warn_for)
```

18 Experiment 2:- The number of iterations = 50000 and the learning rate = 0.0005

[82]:
```
numofiter = 50000
learningrate = 0.0005
weights = train_data(xtrain, ytrain, numofiter,learningrate)
classes = np.unique(ytrain)
predictions2 = predict_class(xvalidation,classes,weights)
```

19 Accuracy

[83]:
```
acc2 = accuracy(predictions2, yvalidation)
acc2
```

[83]: 78.125

20 Confusion Matrix

[84]:
```
print(confusion_matrix(predictions2,yvalidation))
```

```
[[ 0  0  1  0  0  0]
 [ 0  0  0  1  0  0]
 [ 0  0 15  0  0  0]
 [ 0  0  2 10  0  0]
 [ 0  0  0  1  0  0]
 [ 0  0  2  0  0  0]]
```

21 Classification report

[85]:
```
print(classification_report(predictions2,yvalidation))
```

```
              precision    recall  f1-score   support

          00       0.00      0.00      0.00         1
          02       0.00      0.00      0.00         1
          03       0.75      1.00      0.86        15
          04       0.83      0.83      0.83        12
          05       0.00      0.00      0.00         1
          07       0.00      0.00      0.00         2

    accuracy                           0.78        32
   macro avg       0.26      0.31      0.28        32
weighted avg       0.66      0.78      0.71        32
```

22 Experiment 3:- The number of iterations = 70000 and the learning rate = 0.005

[86]:
```
numofiter = 70000
learningrate = 0.005
weights = train_data(xtrain, ytrain, numofiter,learningrate)
classes = np.unique(ytrain)
predictions3 = predict_class(xvalidation,classes,weights)
```

23 Accuracy

[87]:
```
acc3 = accuracy(predictions3, yvalidation)
acc3
```

[87]: 84.375

24 Confusion Matrix

[88]:
```
print(confusion_matrix(predictions3,yvalidation))
```

```
[[16  0  0  0]
 [ 2 11  0  0]
 [ 0  1  0  0]
 [ 2  0  0  0]]
```

25 Classification report

[89]:
```
print(classification_report(predictions3,yvalidation))
```

```
              precision    recall  f1-score   support

          03       0.80      1.00      0.89        16
          04       0.92      0.85      0.88        13
          06       0.00      0.00      0.00         1
          07       0.00      0.00      0.00         2

    accuracy                           0.84        32
   macro avg       0.43      0.46      0.44        32
weighted avg       0.77      0.84      0.80        32
```

26 Experiment 4:- The number of iterations = 100000 and the learning rate = 0.01

[90]:
```
numofiter = 100000
learningrate = 0.01
weights = train_data(xtrain, ytrain, numofiter,learningrate)
classes = np.unique(ytrain)
predictions4 = predict_class(xvalidation,classes,weights)
```

27 Accuracy

[91]:
```
acc4 = accuracy(predictions4, yvalidation)
acc4
```

[91]: 87.5

28 Confusion matrix

[92]:
```
print(confusion_matrix(predictions4,yvalidation))
```

```
[[17  0  0  0]
 [ 1 11  0  0]
 [ 0  1  0  0]
 [ 2  0  0  0]]
```

29 Samle Test And Train

[10]:
```
train_path = r"G:\second_sem\SMAI\Assignment_3\q2\sample_train.txt"
train_images = []
train_labels = []
train_file = open(train_path,"r")
for train_line in train_file:
    path_label = train_line.split(" ")
    train_f = path_label[0]
    train_label = path_label[1].replace('\n', '')
    train_images.append(np.asarray(Image.open(train_f).convert('L').resize((64,␣
↪64))).flatten())
    train_labels.append(train_label)
train_train = np.array(train_images)
train_scalar = MinMaxScaler()
train_train = train_scalar.fit_transform(train_train)
train_train,train_eigenvects = apply_pca(train_train)
train_labels = np.array(train_labels)
train_file.close()
```

[11]:
```
train_xtrain = train_train[:,:]
```

[12]:
```
label_to_ordinal = {}
ordinal_to_label = {}
uniq_label = np.unique(train_labels)
for i in range(uniq_label.shape[0]):
    label_to_ordinal[uniq_label[i]] = i
for j in range(uniq_label.shape[0]):
    ordinal_to_label[j] = uniq_label[j]
train_ytrain = np.zeros(len(train_labels))
for i in range(len(train_labels)):
    train_ytrain[i] = label_to_ordinal[train_labels[i]]
```

[13]:
```
test_path = r"G:\second_sem\SMAI\Assignment_3\q2\sample_test.txt"
test_images = []
test_file = open(test_path,"r")
for test_line in test_file:
```

```
    test_line= test_line.replace('\n','')
    test_images.append(np.asarray(Image.open(test_line).convert('L').resize(␣
 ↪(64, 64))).flatten())
test_test = np.array(test_images)
test_test = train_scalar.transform(test_test)
test_test = transform_pca(train_eigenvects,test_test)
test_xtest = test_test[:,:]
test_file.close()
```

[19]:
```
numofiter = 100000
learningrate = 0.01
t_weights = train_data(train_xtrain, train_ytrain, numofiter,learningrate)
t_classes = np.unique(train_ytrain)
predictions5 = predict_class(test_xtest,t_classes,t_weights)
```

[20]:
```
predicted_labels = []
for i in predictions5:
    predicted_labels.append(ordinal_to_label[i])
predicted_labels
```

[20]: ['abc', 'abc', 'alice', 'bob', 'bob']

[ ]:

[ ]:

q3

April 3, 2020

1 Question 3

2 Implement Multilayer Perceptron(MLP), Convolutional Neural Network(CNN) as well as Support Vector Machines(SVM) to classify digits from the MNIST dataset.

3 Importing Libraries

[1]:
```
from mnist import MNIST
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
```

4 Support Vector Machine

5 Load dataset

[0]:
```
mndata = MNIST('/content/data/')
xtrain, ytrain = mndata.load_training()
xtest, ytest = mndata.load_testing()
xtrain = np.array(xtrain)
ytrain = np.array(ytrain)
xtest = np.array(xtest)
ytest = np.array(ytest)
```

[0]:
```
from sklearn import svm
from sklearn.preprocessing import StandardScaler
```

6 Normalizing the data between 0-254

[0]:
```
svmxtrain = xtrain/255.0
svmytrain = ytrain
svmxtest = xtest/255.0
svmytest = ytest
```

7 Applying standard scaler to transform the data such that its distribution will have a mean value 0 and standard deviation of 1.

[0]:
```
scaler = StandardScaler()
svmxtrain = scaler.fit_transform(svmxtrain)
svmxtest = scaler.transform(svmxtest)
```

8 Experiment 1:- Applying Support Vector Machine with Linear kernel and calculating the accuracy

[0]:
```
svc1 = svm.SVC(kernel='linear')
svc1.fit(svmxtrain, svmytrain)
y_predict1 = svc1.predict(svmxtest)
acc1 = accuracy_score(svmytest, y_predict1)
print('Accuracy = ',acc1)
```

```
Accuracy =  0.9293
```

9 Confusion Matrix

[0]:
```
print(confusion_matrix(svmytest, y_predict1))
```

```
[[ 951    0    5    2    2    8    8    2    1    1]
 [   0 1119    6    2    0    1    2    1    4    0]
 [  10   13  956   11    7    4    5    6   18    2]
 [   7    1   15  941    0   16    1    6   19    4]
 [   3    2   18    1  929    0    3    5    4   17]
 [   7    6    7   41    6  789   12    2   19    3]
 [  12    3   13    1    8   17  902    0    2    0]
 [   2    8   23   13   10    1    0  945    5   21]
 [  12    6   11   28    8   24    9    6  858   12]
 [   6    7    6   10   36    4    1   23   13  903]]
```

10 Classification Report

[0]:
```
print(classification_report(svmytest, y_predict1))
```

```
              precision    recall  f1-score   support

           0       0.94      0.97      0.96       980
           1       0.96      0.99      0.97      1135
           2       0.90      0.93      0.91      1032
           3       0.90      0.93      0.91      1010
           4       0.92      0.95      0.93       982
           5       0.91      0.88      0.90       892
           6       0.96      0.94      0.95       958
           7       0.95      0.92      0.93      1028
           8       0.91      0.88      0.90       974
           9       0.94      0.89      0.92      1009

    accuracy                           0.93     10000
   macro avg       0.93      0.93      0.93     10000
weighted avg       0.93      0.93      0.93     10000
```

11 Experiment 2:- Applying Support Vector Machine with polynomial kernel and calculating the accuracy.

[0]:
```
svc2 = svm.SVC(kernel='poly')
svc2.fit(svmxtrain, svmytrain)
y_predict2 = svc2.predict(svmxtest)
acc2 = accuracy_score(svmytest, y_predict2)
print('Accuracy = ',acc2)
```

```
Accuracy =  0.9611
```

12 Confusion Matrix

[0]:
```
print(confusion_matrix(svmytest, y_predict2))
```

```
[[ 962    0    0    1    0    3    4    0   10    0]
 [   0 1122    3    0    2    1    4    0    3    0]
 [   6    0  973    3    3    1    2    7   37    0]
 [   0    0    1  971    1    3    2    5   22    5]
 [   0    0    2    0  955    0    6    2    4   13]
 [   2    1    3    3    2  863    6    1    9    2]
 [   5    3    1    0    8    8  919    0   14    0]
 [   1    8   11    2   12    0    0  958   11   25]
```

```
 [   1    0    3    6    5    8    1    3  943    4]
 [   3    4    1   11   24    3    0    4   14  945]]
```

13 Classification Report

[0]:
```python
print(classification_report(svmytest, y_predict2))
```

```
              precision    recall  f1-score   support

           0       0.98      0.98      0.98       980
           1       0.99      0.99      0.99      1135
           2       0.97      0.94      0.96      1032
           3       0.97      0.96      0.97      1010
           4       0.94      0.97      0.96       982
           5       0.97      0.97      0.97       892
           6       0.97      0.96      0.97       958
           7       0.98      0.93      0.95      1028
           8       0.88      0.97      0.92       974
           9       0.95      0.94      0.94      1009

    accuracy                           0.96     10000
   macro avg       0.96      0.96      0.96     10000
weighted avg       0.96      0.96      0.96     10000
```

14 Experiment 3:- Applying Support Vector Machine with RBF kernel and calculating the accuracy

[0]:
```python
svc3 = svm.SVC(kernel='rbf')
svc3.fit(svmxtrain, svmytrain)
y_predict3 = svc3.predict(svmxtest)
acc3 = accuracy_score(svmytest, y_predict3)
print('Accuracy = ',acc3)
```

```
Accuracy =  0.9661
```

15 Confusion Matrix

[0]:
```python
print(confusion_matrix(svmytest, y_predict3))
```

```
[[ 968    0    1    1    0    3    3    2    2    0]
 [   0 1127    3    0    0    1    2    0    2    0]
 [   5    1  996    2    2    0    1   15    9    1]
 [   0    0    4  980    1    7    0   11    7    0]
 [   0    0   12    0  944    2    4    7    3   10]
```

```
[   2    0    1   10    2  854    6    8    7    2]
[   6    2    1    0    4    8  930    2    5    0]
[   1    6   13    2    3    0    0  990    0   13]
[   3    0    4    6    6    9    3   14  926    3]
[   4    6    5   11   12    2    0   20    3  946]]
```

16 Classification Report

[0]:
```
print(classification_report(svmytest, y_predict3))
```

```
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       980
           1       0.99      0.99      0.99      1135
           2       0.96      0.97      0.96      1032
           3       0.97      0.97      0.97      1010
           4       0.97      0.96      0.97       982
           5       0.96      0.96      0.96       892
           6       0.98      0.97      0.98       958
           7       0.93      0.96      0.94      1028
           8       0.96      0.95      0.96       974
           9       0.97      0.94      0.95      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000
```

17 Multi-Layer Perceptron

[0]:
```
from torchvision import datasets, transforms
import torch
import torchvision
from torch import nn
from torch import optim
from mnist import MNIST
import numpy as np
from torch.utils.data import TensorDataset
from torch import Tensor
```

18 Loading the training dataset and test dataset and reshaping it to the length of the dataset * 28 * 28 * 1 to feed into the multi layer perceptron and normalizing it between 0-254 and creating batches of size 64

[0]:
```
mndata = MNIST('/content/data/')
xtrain, ytrain = mndata.load_training()
xtest, ytest = mndata.load_testing()
original = np.array(ytest)
xtrain = np.array(xtrain)
ytrain = np.array(ytrain)
xtest = np.array(xtest)
ytest = np.array(ytest)
xtrain = xtrain.reshape(xtrain.shape[0], 28, 28, 1)
xtest = xtest.reshape(xtest.shape[0], 28, 28, 1)
xtrain = xtrain.astype('float')
xtest = xtest.astype('float')
xtrain = xtrain/255
xtest = xtest/255
```

[0]:
```
xtrain = Tensor(xtrain)
ytrain = Tensor(ytrain)
ytrain = ytrain.long()
xtest = Tensor(xtest)
ytest = Tensor(ytest)
trainset = TensorDataset(xtrain,ytrain)
testset = TensorDataset(xtest, ytest)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=True)
```

19 Experiment 4:- Multi layer perceptron of input layer consisting of 784(28 * 28) neurons, two hidden layers with 128 neurons and 64 neurons in each layer correspondingly. Output layers of 10 neurons for 10 classes. Applying ReLu(y = max(0, x)) activation function in both of the hidden layers. Linear activation function (A = cx) is used in output layer.

[0]:
```
inputlayer = 784
hiddenlayer = [128, 64]
outputlayer = 10

model = nn.Sequential(nn.Linear(inputlayer, hiddenlayer[0]),
                      nn.ReLU(),
```

```
                         nn.Linear(hiddenlayer[0], hiddenlayer[1]),
                         nn.ReLU(),
                         nn.Linear(hiddenlayer[1], outputlayer))
```

20 The loss function is Cross Entropy loss (H(P, Q) = – sum x in X P(x) * log(Q(x))) and Stochastic Gradient descent optimzer is used with learning rate 0.003 and momentum 0.9. Epochs is 100.

[0]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.003, momentum=0.9)
epochs = 100
for e in range(epochs):
    total_loss = 0
    for xtrain, ytrain in trainloader:
        xtrain = xtrain.view(xtrain.shape[0], -1)
        optimizer.zero_grad()
        result = model(xtrain)
        loss = criterion(result, ytrain)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
```

21 Prediction of the test data and calculating accuracy

[0]:
```
correct_prediction, total_count = 0, 0
original1 = []
predictions1 =[]
for xtest,ytest in testloader:
    for i in range(len(ytest)):
        img = xtest[i].view(1, 784)
        with torch.no_grad():
            ps = model(img)
        probab = list(ps.numpy()[0])
        prediction = probab.index(max(probab))
        real = ytest.numpy()[i]
        original1.append(real)
        predictions1.append(prediction)
        if(real == prediction):
            correct_prediction += 1
        total_count += 1
print("\nModel Accuracy =", (correct_prediction/total_count))
```

```
Model Accuracy = 0.9793
```

22 Confusion Matrix

[0]:
```
print(confusion_matrix(original1, predictions1))
```

```
[[ 969    1    1    1    1    0    3    1    2    1]
 [   0 1125    3    1    0    1    2    1    2    0]
 [   4    2 1007    6    3    0    1    4    5    0]
 [   0    0    3  992    0    3    0    4    4    4]
 [   1    0    2    1  960    0    4    3    1   10]
 [   3    0    0   10    2  866    4    1    3    3]
 [   3    2    2    1    4    3  941    0    2    0]
 [   0    3    7    1    0    0    0 1011    1    5]
 [   5    0    3    8    3    3    3    4  940    5]
 [   2    2    0    4    7    4    1    3    4  982]]
```

23 Classification Report

[0]:
```
print(classification_report(original1, predictions1))
```

```
              precision    recall  f1-score   support

         0.0       0.98      0.99      0.99       980
         1.0       0.99      0.99      0.99      1135
         2.0       0.98      0.98      0.98      1032
         3.0       0.97      0.98      0.97      1010
         4.0       0.98      0.98      0.98       982
         5.0       0.98      0.97      0.98       892
         6.0       0.98      0.98      0.98       958
         7.0       0.98      0.98      0.98      1028
         8.0       0.98      0.97      0.97       974
         9.0       0.97      0.97      0.97      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

24 Experiment 5:- Multi layer perceptron of input layer consisting of 784(28 * 28) neurons, three hidden layers with 128 neurons and 64 neurons and 32 in each layer correspondingly. Output layers of 10 neurons for 10 classes. Applying ReLu(y = max(0, x)) activation function in all of the hidden layers. Linear activation function (A = cx) is used in output layer.

[0]:
```
inputlayer = 784
hiddenlayer = [128, 64, 32]
outputlayer = 10

model2 = nn.Sequential(nn.Linear(inputlayer, hiddenlayer[0]),
                      nn.ReLU(),
                      nn.Linear(hiddenlayer[0], hiddenlayer[1]),
                      nn.ReLU(),
                      nn.Linear(hiddenlayer[1], hiddenlayer[2]),
                      nn.ReLU(),
                      nn.Linear(hiddenlayer[2], outputlayer))
```

25 The loss function is Cross Entropy loss (H(P, Q) = – sum x in X P(x) * log(Q(x))) and Stochastic Gradient descent optimzer is used with learning rate 0.003 and momentum 0.9. Epochs is 100.

[0]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model2.parameters(), lr=0.003, momentum=0.9)
epochs = 100
for e in range(epochs):
    total_loss = 0
    for xtrain, ytrain in trainloader:
        xtrain = xtrain.view(xtrain.shape[0], -1)
        optimizer.zero_grad()
        result = model2(xtrain)
        loss = criterion(result, ytrain)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
```

26 Prediction of the test data and calculating accuracy

[0]:
```
correct_prediction, total_count = 0, 0
original2 = []
predictions2 = []
for xtest,ytest in testloader:
    for i in range(len(ytest)):
        img = xtest[i].view(1, 784)
        with torch.no_grad():
            ps = model2(img)
        probab = list(ps.numpy()[0])
        prediction = probab.index(max(probab))
        real = ytest.numpy()[i]
        original2.append(real)
        predictions2.append(prediction)
        if(real == prediction):
            correct_prediction += 1
        total_count += 1
print("\nModel Accuracy =", (correct_prediction/total_count))
```

```
Model Accuracy = 0.9775
```

27 Consusion Matrix

[0]:
```
print(confusion_matrix(original2, predictions2))
```

```
[[ 971    0    1    0    0    1    3    1    2    1]
 [    0 1128    2    0    0    0    2    1    2    0]
 [    6    3 1002    5    2    0    3    5    6    0]
 [    0    0    3  987    0    4    0    7    5    4]
 [    1    0    2    1  960    0    4    3    0   11]
 [    2    0    0   11    1  865    4    2    4    3]
 [    4    3    5    1    5    6  933    0    1    0]
 [    2    7    4    5    1    0    1  999    3    6]
 [    5    1    3    5    3    3    1    2  947    4]
 [    3    3    0    3    9    4    0    2    2  983]]
```

28 Classification Report

[0]:
```
print(classification_report(original2, predictions2))
```

```
              precision    recall  f1-score   support

         0.0       0.98      0.99      0.98       980
         1.0       0.99      0.99      0.99      1135
```

```
         2.0       0.98      0.97      0.98      1032
         3.0       0.97      0.98      0.97      1010
         4.0       0.98      0.98      0.98       982
         5.0       0.98      0.97      0.97       892
         6.0       0.98      0.97      0.98       958
         7.0       0.98      0.97      0.97      1028
         8.0       0.97      0.97      0.97       974
         9.0       0.97      0.97      0.97      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

29 Experiment 6:- Multi layer perceptron of input layer consisting of 784(28 * 28) neurons, three hidden layers with 128 neurons and 64 neurons and 32 in each layer correspondingly. Output layers of 10 neurons for 10 classes. Applying sigmoid(y = 1/(1+e^(-x)) activation function in all of the hidden layers. Linear activation function (A = cx) is used in output layer.

[0]:
```
inputlayer = 784
hiddenlayer = [128, 64, 32]
outputlayer = 10

model3 = nn.Sequential(nn.Linear(inputlayer, hiddenlayer[0]),
                      nn.Sigmoid(),
                      nn.Linear(hiddenlayer[0], hiddenlayer[1]),
                      nn.Sigmoid(),
                      nn.Linear(hiddenlayer[1], hiddenlayer[2]),
                      nn.Sigmoid(),
                      nn.Linear(hiddenlayer[2], outputlayer))
```

30 The loss function is Cross Entropy loss (H(P, Q) = – sum x in X P(x) * log(Q(x))) and Stochastic Gradient descent optimzer is used with learning rate 0.003 and momentum 0.9. Epochs is 100.

[0]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model3.parameters(), lr=0.003, momentum=0.9)
epochs = 100
for e in range(epochs):
    total_loss = 0
    for xtrain, ytrain in trainloader:
```

```
        xtrain = xtrain.view(xtrain.shape[0], -1)
        optimizer.zero_grad()
        result = model3(xtrain)
        loss = criterion(result, ytrain)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
```

31 Prediction of the test data and calculating accuracy

[0]:
```
correct_prediction, total_count = 0, 0
original3 = []
predictions3 = []
for xtest,ytest in testloader:
    for i in range(len(ytest)):
        img = xtest[i].view(1, 784)
        with torch.no_grad():
            ps = model3(img)
        probab = list(ps.numpy()[0])
        prediction = probab.index(max(probab))
        real = ytest.numpy()[i]
        original3.append(real)
        predictions3.append(prediction)
        if(real == prediction):
            correct_prediction += 1
        total_count += 1
print("\nModel Accuracy =", (correct_prediction/total_count))
```

```
Model Accuracy = 0.9673
```

32 Confusion Matrix

[0]:
```
print(confusion_matrix(original3, predictions3))
```

```
[[ 966    0    4    0    1    5    1    1    2    0]
 [    0 1121    2    5    0    1    0    2    4    0]
 [    4    5  999    8    1    0    2    9    4    0]
 [    0    1    3  986    0    7    0    7    5    1]
 [    4    0    2    0  943    0    6    6    1   20]
 [    5    0    1   25    1  846    1    0   10    3]
 [   11    2    6    0    8    5  922    0    4    0]
 [    1   10    6    6    1    0    0  996    0    8]
 [    1    3    4    8    4   11    3    4  932    4]
 [    3    3    0    6    8    9    1   16    1  962]]
```

33 Classification Report

[0]:
```
print(classification_report(original3, predictions3))
```

```
              precision    recall  f1-score   support

         0.0       0.97      0.99      0.98       980
         1.0       0.98      0.99      0.98      1135
         2.0       0.97      0.97      0.97      1032
         3.0       0.94      0.98      0.96      1010
         4.0       0.98      0.96      0.97       982
         5.0       0.96      0.95      0.95       892
         6.0       0.99      0.96      0.97       958
         7.0       0.96      0.97      0.96      1028
         8.0       0.97      0.96      0.96       974
         9.0       0.96      0.95      0.96      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000
```

34 Experiment 7:- Multi layer perceptron of input layer consisting of 784(28 * 28) neurons, three hidden layers with 128 neurons and 64 neurons and 32 in each layer correspondingly. Output layers of 10 neurons for 10 classes. Applying ReLu(y = max(a,0)) in the first two hidden layers. Applying sigmoid(y = 1/(1+e^(-x)) activation function in the third hidden layer. Linear activation function (A = cx) is used in output layer.

[0]:
```
inputlayer = 784
hiddenlayer = [128, 64,32]
outputlayer = 10

model5 = nn.Sequential(nn.Linear(inputlayer, hiddenlayer[0]),
                      nn.ReLU(),
                      nn.Linear(hiddenlayer[0], hiddenlayer[1]),
                      nn.ReLU(),
                      nn.Linear(hiddenlayer[1], hiddenlayer[2]),
                      nn.Sigmoid(),
                      nn.Linear(hiddenlayer[2], outputlayer))
```

35 The loss function is Cross Entropy loss (H(P, Q) = – sum x in X P(x) * log(Q(x))) and Stochastic Gradient descent optimzer is used with learning rate 0.003 and momentum 0.9. Epoch is 100.

[0]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model5.parameters(), lr=0.003, momentum=0.9)
epochs = 100
for e in range(epochs):
    total_loss = 0
    for xtrain, ytrain in trainloader:
        xtrain = xtrain.view(xtrain.shape[0], -1)
        optimizer.zero_grad()
        result = model5(xtrain)
        loss = criterion(result, ytrain)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
```

36 Prediction of the test data and calculating accuracy

[0]:
```
correct_prediction, total_count = 0, 0
predictions5 = []
original5 = []
for xtest,ytest in testloader:
    for i in range(len(ytest)):
        img = xtest[i].view(1, 784)
        with torch.no_grad():
            ps = model5(img)
        probab = list(ps.numpy()[0])
        prediction = probab.index(max(probab))
        real = ytest.numpy()[i]
        original5.append(real)
        predictions5.append(prediction)
        if(real == prediction):
            correct_prediction += 1
        total_count += 1
print("\nModel Accuracy =", (correct_prediction/total_count))
```

```
Model Accuracy = 0.977
```

37 Confusion Matrix.

```
[0]: print(confusion_matrix(original5,predictions5))
```

```
[[ 969    0    2    1    1    2    2    1    1    1]
 [   0 1125    3    0    0    1    3    1    2    0]
 [   2    1 1010    7    1    0    3    4    4    0]
 [   0    0    4  986    0   10    0    4    4    2]
 [   0    1    4    0  955    0    6    2    2   12]
 [   4    0    0   10    1  861    6    1    4    5]
 [   6    3    0    1    5    4  937    0    2    0]
 [   1    6    5    3    2    0    0 1000    2    9]
 [   2    0    3    6    6    6    3    1  942    5]
 [   4    2    0    2    7    3    0    3    3  985]]
```

38 Classification Report

```
[0]: print(classification_report(original5, predictions5))
```

```
              precision    recall  f1-score   support

         0.0       0.98      0.99      0.98       980
         1.0       0.99      0.99      0.99      1135
         2.0       0.98      0.98      0.98      1032
         3.0       0.97      0.98      0.97      1010
         4.0       0.98      0.97      0.97       982
         5.0       0.97      0.97      0.97       892
         6.0       0.98      0.98      0.98       958
         7.0       0.98      0.97      0.98      1028
         8.0       0.98      0.97      0.97       974
         9.0       0.97      0.98      0.97      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

39 Experiment 7:- Multi layer perceptron of input layer consisting of 784(28 * 28) neurons, three hidden layers with 128 neurons and 64 neurons and 32 in each layer correspondingly. Output layers of 10 neurons for 10 classes. Applying Tanh(f(x) = 1 — exp(-2x) / 1 + exp(-2x)) in the first two hidden layers. Applying sigmoid(y = 1/(1+e^(-x)) activation function in the third hidden layer. Linear activation function (A = cx) is used in output layer.

[0]:
```
inputlayer = 784
hiddenlayer = [128, 64,32]
outputlayer = 10

model6 = nn.Sequential(nn.Linear(inputlayer, hiddenlayer[0]),
                      nn.Tanh(),
                      nn.Linear(hiddenlayer[0], hiddenlayer[1]),
                      nn.Tanh(),
                      nn.Linear(hiddenlayer[1], hiddenlayer[2]),
                      nn.Sigmoid(),
                      nn.Linear(hiddenlayer[2], outputlayer))
```

40 The loss function is Cross Entropy loss (H(P, Q) = – sum x in X P(x) * log(Q(x))) and Stochastic Gradient descent optimzer is used with learning rate 0.003 and momentum 0.9. Epoch is 100.

[0]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model6.parameters(), lr=0.003, momentum=0.9)
epochs = 100
for e in range(epochs):
    total_loss = 0
    for xtrain, ytrain in trainloader:
        xtrain = xtrain.view(xtrain.shape[0], -1)
        optimizer.zero_grad()
        result = model6(xtrain)
        loss = criterion(result, ytrain)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
```

41 Prediction of the test data and calculating accuracy

[0]:

```
correct_prediction, total_count = 0, 0
predictions6 = []
original6 = []
for xtest,ytest in testloader:
    for i in range(len(ytest)):
        img = xtest[i].view(1, 784)
        with torch.no_grad():
            ps = model6(img)
        probab = list(ps.numpy()[0])
        prediction = probab.index(max(probab))
        real = ytest.numpy()[i]
        original6.append(real)
        predictions6.append(prediction)
        if(real == prediction):
            correct_prediction += 1
        total_count += 1
print("\nModel Accuracy =", (correct_prediction/total_count))
```

```
Model Accuracy = 0.9772
```

42 Confusion Matrix

[0]:

```
print(confusion_matrix(original6,predictions6))
```

```
[[ 966    0    2    0    1    3    2    1    3    2]
 [   0 1127    2    1    0    0    1    1    3    0]
 [   3    3 1010    5    1    0    1    3    6    0]
 [   0    0    4  986    0    5    0    3   10    2]
 [   1    1    1    0  960    1    6    1    1   10]
 [   4    0    0    9    1  863    2    3    6    4]
 [   5    2    3    0    3    4  939    0    2    0]
 [   0    4    9    3    1    1    0  998    1   11]
 [   4    1    4    4    2    6    2    2  946    3]
 [   0    2    0    4   10    2    1    9    4  977]]
```

43 Classification Report

[0]:

```
print(classification_report(original6, predictions6))
```

```
              precision    recall  f1-score   support

         0.0       0.98      0.99      0.98       980
         1.0       0.99      0.99      0.99      1135
```

```
         2.0       0.98      0.98      0.98      1032
         3.0       0.97      0.98      0.98      1010
         4.0       0.98      0.98      0.98       982
         5.0       0.98      0.97      0.97       892
         6.0       0.98      0.98      0.98       958
         7.0       0.98      0.97      0.97      1028
         8.0       0.96      0.97      0.97       974
         9.0       0.97      0.97      0.97      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

44 Convolution Neural Network

[3]:
```
# from torchvision import datasets, transforms
# import torch
# import torchvision
# from torch import nn
# from torch import optim
from mnist import MNIST
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D
from keras.optimizers import SGD
import keras
from keras.layers.advanced_activations import LeakyReLU
```

```
Using TensorFlow backend.
```

45 Loading the training dataset and test dataset and reshaping it to the length of the dataset * 28 * 28 * 1 to feed into the multi layer perceptron and normalizing it between 0-254.

[4]:
```
mndata = MNIST('G:\second_sem\SMAI\Assignment_3\q3\dataset')
xtrain, ytrain = mndata.load_training()
xtest, ytest = mndata.load_testing()
xtrain = np.array(xtrain)
ytrain = keras.utils.to_categorical(ytrain, 10)
xtest = np.array(xtest)
original = np.array(ytest)
ytest = keras.utils.to_categorical(ytest, 10)
xtrain = xtrain.reshape(xtrain.shape[0], 28, 28, 1)
xtest = xtest.reshape(xtest.shape[0], 28, 28, 1)
```

```
xtrain = xtrain.astype('float')
xtest = xtest.astype('float')
xtrain = xtrain/255
xtest = xtest/255
```

46 Experiment 8:- The model is comprised of one output layer and one 2d convolution layer and one hidden layer. On the hidden layer and 2d convolution layer ReLu activation function has been used. Softmax activation function is used in output layer. 2D convolution layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. The number of output filters in the convolution is 32. Kernel of size 3x3 is used. After that max pooling of 2x2 matrix is used. The dropout rate is set to 30% from convolution layer to the hidden layer, meaning one in almost 3 inputs will be randomly excluded from each update cycle. The convoltuion layer to hidden layer is densed and 128 neurons is used in the hidden layer. From the hidden layer to output layer dropout rate is set 25% meaning one in 4 inputs will be discarded randomly and it is densed. 10 neuronse for 10 classes is used in output layer. Stochastic gradient descent optimizer is used with learning rate 0.003 and momentum 0.9. The loss funtion is used cross entropy loss. And the metrics for the model is "accuracy". Batch size of 64 is created with epochs is set 10.

[14]:
```
model1 = Sequential()
model1.add(Conv2D(filters=32, kernel_size=(3, 3),␣
 ↪activation='relu',input_shape=(28,28,1)))
model1.add(MaxPooling2D(pool_size=(2, 2)))
model1.add(Dropout(0.30))
model1.add(Flatten())
model1.add(Dense(units=128, activation='relu'))
model1.add(Dropout(0.25))
model1.add(Dense(units=10, activation='softmax'))
optimizer = SGD(lr=0.003, momentum=0.9)
model1.compile(loss=keras.losses.categorical_crossentropy,optimizer=optimizer,␣
 ↪metrics=['accuracy'])
model1.fit(x=xtrain,y=ytrain,batch_size=64,epochs=10)
acc1 = model1.evaluate(xtest, ytest)
acc1
```

```
Epoch 1/10
60000/60000 [==============================] - 33s 558us/step - loss: 0.5601 -
acc: 0.8288
Epoch 2/10
60000/60000 [==============================] - 33s 551us/step - loss: 0.2524 -
acc: 0.9228
Epoch 3/10
60000/60000 [==============================] - 33s 551us/step - loss: 0.1992 -
acc: 0.9394
Epoch 4/10
60000/60000 [==============================] - 33s 553us/step - loss: 0.1717 -
acc: 0.9481
Epoch 5/10
60000/60000 [==============================] - 33s 547us/step - loss: 0.1505 -
acc: 0.9541
Epoch 6/10
60000/60000 [==============================] - 33s 548us/step - loss: 0.1393 -
acc: 0.9569
Epoch 7/10
60000/60000 [==============================] - 34s 567us/step - loss: 0.1261 -
acc: 0.9617
Epoch 8/10
60000/60000 [==============================] - 33s 551us/step - loss: 0.1156 -
acc: 0.9644
Epoch 9/10
60000/60000 [==============================] - 33s 549us/step - loss: 0.1059 -
acc: 0.9673
Epoch 10/10
60000/60000 [==============================] - 33s 548us/step - loss: 0.0999 -
acc: 0.9696
10000/10000 [==============================] - 2s 228us/step
```

[14]:
```
[0.0648645735614933, 0.9797]
```

47 Prediction of the digit.

[0]:
```
temp1 = model1.predict(xtest)
pred1 = np.argmax(np.round(temp1),axis=1)
```

48 Confusion Matrix

[0]:
```
print(confusion_matrix(original,pred1))
```

```
[[ 976    0    0    0    0    0    0    1    3    0]
 [   2 1124    2    1    0    0    2    0    4    0]
```

```
[  13    0 1007    1    1    0    0    4    6    0]
[   3    0    1  994    0    1    0    4    7    0]
[   7    0    6    0  941    0    1    1    2   24]
[  10    0    0   10    0  863    2    1    5    1]
[  11    3    1    1    4    4  931    0    3    0]
[  14    7   10    1    0    0    0  984    3    9]
[   9    0    2    5    0    2    0    2  952    2]
[  11    5    0    7    6    2    0    4    2  972]]
```

49 Classification Report

[0]:
```
print(classification_report(original,pred1))
```

```
              precision    recall  f1-score   support

           0       0.92      1.00      0.96       980
           1       0.99      0.99      0.99      1135
           2       0.98      0.98      0.98      1032
           3       0.97      0.98      0.98      1010
           4       0.99      0.96      0.97       982
           5       0.99      0.97      0.98       892
           6       0.99      0.97      0.98       958
           7       0.98      0.96      0.97      1028
           8       0.96      0.98      0.97       974
           9       0.96      0.96      0.96      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000
```

50 Experiment 9:- The model is comprised of one output layer and one 2d convolution layer and one hidden layer. On the hidden layer and 2d convolution layer ReLu activation function has been used. Softmax activation function is used in output layer. 2D convolution layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. The number of output filters in the convolution is 64. Kernel of size 4x4 is used. After that max pooling of 2x2 matrix is used. The dropout rate is set to 50% from convolution layer to the hidden layer, meaning one in 2 inputs will be randomly excluded from each update cycle. The convoltuion layer to hidden layer is densed and 128 neurons is used in the hidden layer. From the hidden layer to output layer dropout rate is set 20% meaning one in 5 inputs will be discarded randoml and it is densed. 10 neuronse for 10 classes is used in output layer. Stochastic gradient descent optimizer is used with learning rate 0.003 and momentum 0.9. The loss funtion is used cross entropy loss. And the metrics for the model is "accuracy". Batch size of 64 is created with epochs is set 10.

[0]:
```
model2 = Sequential()
model2.add(Conv2D(filters=64, kernel_size=(4, 4),␣
 ↪activation='relu',input_shape=(28,28,1)))
model2.add(MaxPooling2D(pool_size=(2, 2)))
model2.add(Dropout(0.5))
model2.add(Flatten())
model2.add(Dense(units=128, activation='relu'))
model2.add(Dropout(0.2))
model2.add(Dense(units=10, activation='softmax'))
optimizer = SGD(lr=0.003, momentum=0.9)
model2.compile(loss=keras.losses.categorical_crossentropy,optimizer=optimizer,␣
 ↪metrics=['accuracy'])
model2.fit(x=xtrain,y=ytrain,batch_size=64,epochs=10)
acc2 = model2.evaluate(xtest, ytest)
acc2
```

```
Epoch 1/10
60000/60000 [==============================] - 50s 840us/step - loss: 0.5143 -
acc: 0.8440
Epoch 2/10
60000/60000 [==============================] - 51s 849us/step - loss: 0.2270 -
acc: 0.9303
Epoch 3/10
```

```
60000/60000 [==============================] - 50s 831us/step - loss: 0.1683 -
acc: 0.9492
Epoch 4/10
60000/60000 [==============================] - 50s 827us/step - loss: 0.1356 -
acc: 0.9594
Epoch 5/10
60000/60000 [==============================] - 50s 828us/step - loss: 0.1162 -
acc: 0.9651
Epoch 6/10
60000/60000 [==============================] - 50s 833us/step - loss: 0.1025 -
acc: 0.9696
Epoch 7/10
60000/60000 [==============================] - 50s 828us/step - loss: 0.0926 -
acc: 0.9719
Epoch 8/10
60000/60000 [==============================] - 50s 827us/step - loss: 0.0838 -
acc: 0.9740
Epoch 9/10
60000/60000 [==============================] - 51s 846us/step - loss: 0.0760 -
acc: 0.9769
Epoch 10/10
60000/60000 [==============================] - 51s 846us/step - loss: 0.0718 -
acc: 0.9784
10000/10000 [==============================] - 2s 241us/step
```

[0]: [0.04387621255386621, 0.9851]

51 Prediction of the digits

[0]:
```
temp2 = model2.predict(xtest)
pred2 = np.argmax(np.round(temp2),axis=1)
```

52 Confusion Matrix

[0]:
```
print(confusion_matrix(original,pred2))
```

```
[[ 977    0    0    0    0    0    0    1    2    0]
 [   3 1126    2    1    0    0    1    0    2    0]
 [   7    1 1014    0    2    0    0    6    2    0]
 [   3    0    3  997    0    2    0    2    3    0]
 [   6    0    1    0  972    0    0    0    1    2]
 [   2    0    0    4    0  880    3    1    1    1]
 [  11    3    0    1    3    3  937    0    0    0]
 [   5    1    7    1    0    0    0 1010    1    3]
 [   7    1    2    3    1    2    1    3  950    4]
```

```
[  13    5    0    1    9    0    0    4    1  976]]
```

53 Classification Report

[0]:
```
print(classification_report(original,pred2))
```

```
              precision    recall  f1-score   support

           0       0.94      1.00      0.97       980
           1       0.99      0.99      0.99      1135
           2       0.99      0.98      0.98      1032
           3       0.99      0.99      0.99      1010
           4       0.98      0.99      0.99       982
           5       0.99      0.99      0.99       892
           6       0.99      0.98      0.99       958
           7       0.98      0.98      0.98      1028
           8       0.99      0.98      0.98       974
           9       0.99      0.97      0.98      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

54 Experiment 10:- The model is comprised of one output layer and one 2d convolution layer and one hidden layer. On the hidden layer and 2d convolution layer ReLu activation function has been used. Softmax activation function is used in output layer. 2D convolution layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. The number of output filters in the convolution is 32. Kernel of size 3x3 is used. After that max pooling of 2x2 matrix is used. The dropout rate is set to 20% from convolution layer to the hidden layer, meaning one in 5 inputs will be randomly excluded from each update cycle. The convoltuion layer to hidden layer is densed and 64 neurons is used in the hidden layer. From the hidden layer to output layer dropout rate is set 10% meaning one in 10 inputs will be discarded randoml and it is densed. 10 neuronse for 10 classes is used in output layer. Stochastic gradient descent optimizer is used with learning rate 0.003 and momentum 0.9. The loss funtion is used cross entropy loss. And the metrics for the model is "accuracy". Batch size of 64 is created with epochs is set 10.

[0]:
```
model3 = Sequential()
model3.add(Conv2D(filters=32, kernel_size=(3, 3),␣
 ↪activation='relu',input_shape=(28,28,1)))
model3.add(MaxPooling2D(pool_size=(2, 2)))
model3.add(LeakyReLU(alpha=0.2))
model3.add(Flatten())
model3.add(Dense(units=64, activation='relu'))
model3.add(LeakyReLU(alpha=0.1))
model3.add(Dense(units=10, activation='softmax'))
optimizer = SGD(lr=0.003, momentum=0.9)
model3.compile(loss=keras.losses.categorical_crossentropy,optimizer=optimizer,␣
 ↪metrics=['accuracy'])
model3.fit(x=xtrain,y=ytrain,batch_size=64,epochs=10)
acc3 = model3.evaluate(xtest, ytest)
acc3
```

```
Epoch 1/10
60000/60000 [==============================] - 24s 404us/step - loss: 0.4256 -
acc: 0.8799
Epoch 2/10
60000/60000 [==============================] - 24s 397us/step - loss: 0.1935 -
acc: 0.9422
Epoch 3/10
```

```
60000/60000 [==============================] - 24s 397us/step - loss: 0.1447 -
acc: 0.9569
Epoch 4/10
60000/60000 [==============================] - 24s 397us/step - loss: 0.1170 -
acc: 0.9649
Epoch 5/10
60000/60000 [==============================] - 24s 393us/step - loss: 0.0977 -
acc: 0.9703
Epoch 6/10
60000/60000 [==============================] - 23s 391us/step - loss: 0.0842 -
acc: 0.9743
Epoch 7/10
60000/60000 [==============================] - 23s 390us/step - loss: 0.0745 -
acc: 0.9776
Epoch 8/10
60000/60000 [==============================] - 24s 398us/step - loss: 0.0654 -
acc: 0.9804
Epoch 9/10
60000/60000 [==============================] - 24s 399us/step - loss: 0.0593 -
acc: 0.9822
Epoch 10/10
60000/60000 [==============================] - 24s 402us/step - loss: 0.0540 -
acc: 0.9836
10000/10000 [==============================] - 2s 208us/step
```

[0]: [0.07566864336489235, 0.9758]

55 Prediction of the digits

[0]:
```
temp3 = model3.predict(xtest)
pred3 = np.argmax(np.round(temp3),axis=1)
```

56 Consfusion Matrix

[0]:
```
print(confusion_matrix(original,pred3))
```

```
[[ 974    0    1    0    0    0    1    1    3    0]
 [   2 1123    4    0    0    1    4    0    1    0]
 [   7    0 1009    1    2    0    1    3    9    0]
 [   7    0    9  980    0    0    0    3    7    4]
 [   2    0    4    0  973    0    1    0    1    1]
 [  12    0    0    9    1  855    4    1    9    1]
 [  14    2    0    1    3    2  933    0    3    0]
 [   5    6   14    1    0    0    0  996    3    3]
 [  10    0    1    2    3    0    1    1  956    0]
```

```
 [  21    6    1    7   16    2    0    9    7  940]]
```

57 Classification Report

[0]:
```
print(classification_report(original,pred3))
```

```
              precision    recall  f1-score   support

           0       0.92      0.99      0.96       980
           1       0.99      0.99      0.99      1135
           2       0.97      0.98      0.97      1032
           3       0.98      0.97      0.97      1010
           4       0.97      0.99      0.98       982
           5       0.99      0.96      0.98       892
           6       0.99      0.97      0.98       958
           7       0.98      0.97      0.98      1028
           8       0.96      0.98      0.97       974
           9       0.99      0.93      0.96      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000
```

58 Experiment 11:- The model is comprised of one output layer and one 2d convolution layer and one hidden layer. On the hidden layer and 2d convolution layer Tanh activation function has been used. Softmax activation function is used in output layer. 2D convolution layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. The number of output filters in the convolution is 64. Kernel of size 2x2 is used. After that max pooling of 2x2 matrix is used. The dropout rate is set to 30% from convolution layer to the hidden layer, meaning one in almost 3 inputs will be randomly excluded from each update cycle. The convoltuion layer to hidden layer is densed and 128 neurons is used in the hidden layer. From the hidden layer to output layer dropout rate is set 20% meaning one in 5 inputs will be discarded randoml and it is densed. 10 neuronse for 10 classes is used in output layer. Stochastic gradient descent optimizer is used with learning rate 0.003 and momentum 0.9. The loss funtion is used cross entropy loss. And the metrics for the model is "accuracy". Batch size of 64 is created with epochs is set 10.

[10]:
```
model4 = Sequential()
model4.add(Conv2D(filters=64, kernel_size=(2, 2), strides=(2, 2),␣
 ↪activation='tanh',input_shape=(28,28,1)))
model4.add(MaxPooling2D(pool_size=(2, 2)))
model4.add(Dropout(0.3))
model4.add(Flatten())
model4.add(Dense(units=128, activation='tanh'))
model4.add(Dropout(0.2))
model4.add(Dense(units=10, activation='softmax'))
optimizer = SGD(lr=0.003, momentum=0.9)
model4.compile(loss=keras.losses.categorical_crossentropy,optimizer=optimizer,␣
 ↪metrics=['accuracy'])
model4.fit(x=xtrain,y=ytrain,batch_size=64,epochs=10)
acc4 = model4.evaluate(xtest, ytest)
acc4
```

```
Epoch 1/10
60000/60000 [==============================] - 18s 301us/step - loss: 0.7211 -
acc: 0.7884
Epoch 2/10
60000/60000 [==============================] - 18s 295us/step - loss: 0.4089 -
acc: 0.8733
Epoch 3/10
```

```
60000/60000 [==============================] - 18s 296us/step - loss: 0.3310 -
acc: 0.8982
Epoch 4/10
60000/60000 [==============================] - 18s 295us/step - loss: 0.2722 -
acc: 0.9171
Epoch 5/10
60000/60000 [==============================] - 18s 299us/step - loss: 0.2326 -
acc: 0.9295
Epoch 6/10
60000/60000 [==============================] - 18s 298us/step - loss: 0.2073 -
acc: 0.9378
Epoch 7/10
60000/60000 [==============================] - 18s 294us/step - loss: 0.1897 -
acc: 0.9422
Epoch 8/10
60000/60000 [==============================] - 18s 299us/step - loss: 0.1763 -
acc: 0.9458
Epoch 9/10
60000/60000 [==============================] - 18s 298us/step - loss: 0.1664 -
acc: 0.9489
Epoch 10/10
60000/60000 [==============================] - 18s 297us/step - loss: 0.1591 -
acc: 0.9516
10000/10000 [==============================] - 1s 106us/step
```

[10]:
```
[0.115869736199826, 0.964]
```

59 Prediction of the digits

[0]:
```
temp4 = model4.predict(xtest)
pred4 = np.argmax(np.round(temp4),axis=1)
```

60 Confusion Matrix

[12]:
```
print(confusion_matrix(original,pred4))
```

```
[[ 972    0    0    0    0    1    1    3    3    0]
 [   4 1122    3    0    0    1    3    1    1    0]
 [  24    2  981    5    4    0    3    5    7    1]
 [  10    0    3  980    0    5    1    5    4    2]
 [  11    1    2    0  935    0    8    4    3   18]
 [  26    1    1   15    0  836    8    2    3    0]
 [  13    2    1    0    2    8  931    0    1    0]
 [  25    4   15    2    1    0    0  965    1   15]
 [  23    0    6   10    4    2    2    3  920    4]
```

```
 [  28    2    1    7   10    4    0   13    1  943]]
```

61 Classification Report

[13]:
```
print(classification_report(original,pred4))
```

```
              precision    recall  f1-score   support

           0       0.86      0.99      0.92       980
           1       0.99      0.99      0.99      1135
           2       0.97      0.95      0.96      1032
           3       0.96      0.97      0.97      1010
           4       0.98      0.95      0.96       982
           5       0.98      0.94      0.96       892
           6       0.97      0.97      0.97       958
           7       0.96      0.94      0.95      1028
           8       0.97      0.94      0.96       974
           9       0.96      0.93      0.95      1009

    accuracy                           0.96     10000
   macro avg       0.96      0.96      0.96     10000
weighted avg       0.96      0.96      0.96     10000
```

62 Summary : The best performing model is as The model is comprised of one output layer and one 2d convolution layer and one hidden layer. On the hidden layer and 2d convolution layer ReLu activation function has been used. Softmax activation function is used in output layer. 2D convolution layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. The number of output filters in the convolution is 64. Kernel of size 4x4 is used. After that max pooling of 2x2 matrix is used. The dropout rate is set to 50% from convolution layer to the hidden layer, meaning one in 2 inputs will be randomly excluded from each update cycle. The convoltuion layer to hidden layer is densed and 128 neurons is used in the hidden layer. From the hidden layer to output layer dropout rate is set 20% meaning one in 5 inputs will be discarded randoml and it is densed. 10 neuronse for 10 classes is used in output layer. Stochastic gradient descent optimizer is used with learning rate 0.003 and momentum 0.9. The loss funtion is used cross entropy loss. And the metrics for the model is "accuracy". Batch size of 64 is created with epochs is set 10.

[ ]:

Question 4

you are expected to perform regression over the dataset of global_active_power values. You are supposed to take the active power values in the past one hour and predict the next active power value.

In [1]:

```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from math import sqrt
from keras.models import Sequential
from keras.layers import Dense

from keras.optimizers import SGD
from sklearn.preprocessing import MinMaxScaler
from tqdm import tqdm
```

```
Using TensorFlow backend.
```

Loading the dataset

In [2]:

```
whole_df = pd.read_csv('G:\second_sem\SMAI\Assignment_3\q4\dataset\household_power_consumpt
ion.txt', sep=';',parse_dates={'datetime' : ['Date', 'Time']}, infer_datetime_format=True,
na_values=['nan','?'])
```

In [3]:

```
t_df = whole_df.filter(["datetime","Global_active_power"],axis =1)
copydf = t_df.filter(["Global_active_power"],axis=1)
```

Generating correlation matrix to find out the correlation between features.

In [0]:

```
corr = whole_df.corr()
corr.style.background_gradient(cmap='coolwarm')
```

Out[0]:

	Global_active_power	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	Sub_metering_2
Global_active_power	1	0.247017	-0.399762	0.998889	0.484401	0.434569
Global_reactive_power	0.247017	1	-0.112246	0.26612	0.123111	0.139231
Voltage	-0.399762	-0.112246	1	-0.411363	-0.195976	-0.167405
Global_intensity	0.998889	0.26612	-0.411363	1	0.489298	0.440347

	Global_active_power	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	Sub_metering_2
Sub_metering_1	0.484401	0.123111	-0.195976	0.489298	1	0.0547209
Sub_metering_2	0.434569	0.139231	-0.167405	0.440347	0.0547209	1
Sub_metering_3	0.638555	0.0896165	-0.268172	0.626543	0.102571	0.080872

Seperating the test data which is to be predicted by index

In [4]:

```
test_index = list(copydf['Global_active_power'].index[copydf['Global_active_power'].apply(
np.isnan)])
test_index.sort()
```

A given univariate sequence into multiple samples where each sample has a specified number of time steps and the output is a single time step

In [5]:

```
def form_dataset(dataset,window_size):
    new_dataset = []
    new_label = []
    length = len(dataset)
    for i in range(length):
        last_index = i + window_size
        if(last_index > length-1):
            break
        temp_x = dataset[i:last_index]
        temp_y = dataset[last_index]
        new_dataset.append(temp_x)
        new_label.append(temp_y)
    return new_dataset, new_label
```

Filling the first window if there is any missing value with mean of the previous values of the missing value.

In [6]:

```
def fill_firstwindow(dataset,test_index,window_size):
    pred_first_window = []
    if(test_index[0] == 0):
        dataset = dataset.remove(0)
    cp_index = test_index.copy()
    for i in cp_index:
        if(i >= window_size):
            break
        test_index.remove(i)
        mean = np.mean(np.array(dataset[:i]))
        pred_first_window.append(mean)
        dataset[i] = mean
    return dataset, pred_first_window, test_index
```

Prediction of missing values.

In [0]:

```
pred = []
for i in test_index[:10]:
```

```
    t_dataset = np.array(copyset60[i-window_size:i])
    temp1 = (t_dataset.reshape(1,window_size))
    temp2 = (lr60.predict(temp1)).reshape(1)[0]
    copyset60[i] = temp2
```

Calculating mean absolute percentage error

In [7]:

```
def mean_absolute_percentage_error(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
```

Window size = 60

In [8]:

```
window_size = 60
df60 = copydf.copy()
```

Preparing the test dataset of each of window size of 60.

In [9]:

```
dataset60 = list(df60["Global_active_power"])
# dataset60,pred1,test_index = fill_firstwindow(dataset60,test_index,window_size)
# copyset60 = dataset60.copy()
pre_xtrain, pre_ytrain = form_dataset(dataset60,window_size)
xtest60 = []
for i in test_index:
    xtest60.append(pre_xtrain[i-window_size])
```

In [0]:

```
df60 = df60.dropna(subset=["Global_active_power"])
```

Splitting the data into training data and validation data.

In [0]:

```
dataset60 = list(df60["Global_active_power"])
pre_xtrain60, pre_ytrain60 = form_dataset(dataset60,window_size)
fraction60 = int(0.8 * (len(pre_xtrain60)))
xtrain60 = pre_xtrain60[:fraction60]
xvalid60 = pre_xtrain60[fraction60:len(pre_xtrain60)]
ytrain60 = pre_ytrain60[:fraction60]
yvalid60 = pre_ytrain60[fraction60:len(pre_xtrain60)]
```

In [0]:

```
xtrain60 = np.array(xtrain60)
xvalid60 = np.array(xvalid60)
ytrain60 = np.array(ytrain60)
yvalid60 = np.array(yvalid60)
```

Experiment 1:- The model using Linear regression of window size 60.

In [0]:

```
lr60 = LinearRegression(fit_intercept=False)
lr60 = lr60.fit(xtrain60, ytrain60)
ypred60 = lr60.predict(xvalid60)
```

R2Score on the validation set

In [250]:

```
r2score60 = r2_score(yvalid60, ypred60)
r2score60
```

Out[250]:

```
0.939163062012902
```

Root mean squared error on validation set

In [251]:

```
rmse60 = sqrt(mean_squared_error(yvalid60, ypred60))
rmse60
```

Out[251]:

```
0.2210054279867361
```

Mean absolute percentage error on validation set

In [252]:

```
yv60 = yvalid60.reshape(-1,1)
yp60 = ypred60.reshape(-1,1)
mape60 = mean_absolute_percentage_error(yv60,yp60)
mape60
```

Out[252]:

```
9.097724829563784
```

Window Size = 120

In [0]:

```
window_size = 120
df120 = copydf
```

Preparing the test dataset of each of window size of 120.

In [0]:

```
dataset120 = list(df120["Global_active_power"])
pre_xtrain, pre_ytrain = form_dataset(dataset120,window_size)
xtest120 = []
for i in test_index:
    xtest120.append(pre_xtrain[i-window_size])
```

In [0]:

```
df120 = df120.dropna(subset=["Global_active_power"])
```

Splitting the data into training data and validation data.

In [0]:

```
dataset120 = list(df120["Global_active_power"])
```

```
pre_xtrain120, pre_ytrain120 = form_dataset(dataset120,window_size)
fraction120 = int(0.8 * (len(pre_xtrain120)))
xtrain120 = pre_xtrain120[:fraction120]
xvalid120 = pre_xtrain120[fraction120:len(pre_xtrain120)]
ytrain120 = pre_ytrain120[:fraction120]
yvalid120 = pre_ytrain120[fraction120:len(pre_xtrain120)]
```

In [0]:

```
xtrain120 = np.array(xtrain120)
xvalid120 = np.array(xvalid120)
ytrain120 = np.array(ytrain120)
yvalid120 = np.array(yvalid120)
```

Experiment 2:- The model using Linear regression of window size 120.

In [0]:

```
lr120 = LinearRegression(fit_intercept=False)
lr120 = lr120.fit(xtrain120, ytrain120)
ypred120 = lr120.predict(xvalid120)
```

R2Score on the validation set

In [0]:

```
r2score120 = r2_score(yvalid120, ypred120)
r2score120
```

Out[0]:

```
0.9393241694688419
```

Root mean squared error on validation set

In [0]:

```
rmse120 = sqrt(mean_squared_error(yvalid120, ypred120))
rmse120
```

Out[0]:

```
0.220714164945173
```

Mean absolute percentage error on validation set

In [0]:

```
yv120 = yvalid120.reshape(-1,1)
yp120 = ypred120.reshape(-1,1)
mape120 = mean_absolute_percentage_error(yv120,yp120)
mape120
```

Out[0]:

```
9.285595430162703
```

Multi layer perceptron of window size 60

In [0]:

```
window_size = 60
```

```
df60_mlp = copydf
dataset60_mlp = list(df60_mlp["Global_active_power"])
pre_xtrain, pre_ytrain = form_dataset(dataset60_mlp,window_size)
xtest60_mlp = []
for i in test_index:
    xtest60_mlp.append(pre_xtrain[i-window_size])
df60_mlp = df60_mlp.dropna(subset=["Global_active_power"])
dataset60_mlp = list(df60_mlp["Global_active_power"])
pre_xtrain60_mlp, pre_ytrain60_mlp = form_dataset(dataset60_mlp,window_size)
fraction60_mlp = int(0.8 * (len(pre_xtrain60_mlp)))
xtrain60_mlp = pre_xtrain60_mlp[:fraction60_mlp]
xvalid60_mlp = pre_xtrain60_mlp[fraction60_mlp:len(pre_xtrain60_mlp)]
ytrain60_mlp = pre_ytrain60_mlp[:fraction60_mlp]
yvalid60_mlp = pre_ytrain60_mlp[fraction60_mlp:len(pre_xtrain60_mlp)]
xtrain60_mlp = np.array(xtrain60_mlp)
xvalid60_mlp = np.array(xvalid60_mlp)
ytrain60_mlp = np.array(ytrain60_mlp)
yvalid60_mlp = np.array(yvalid60_mlp)
```

Experiment 3:- One hidden layer and one outlayer is used. ReLu activation function has been used in the hidden layer with input dimension of 60. The hidden layer and the output layer is connected densly. Stochastic gradient descent optimizer is used with learning rate 0.003 and momentum 0.9 and epoch is 10. Loss function mean squred error is used. Hidden layer is of 100 neurons and output layer has single neuron. Window size is 60.

In [0]:

```
model60_mlp_1 = Sequential()
model60_mlp_1.add(Dense(100, activation='relu',input_dim=window_size))
model60_mlp_1.add(Dense(1))
optimizer = SGD(lr=0.003, momentum=0.9)
model60_mlp_1.compile(optimizer=optimizer, loss='mse')
model60_mlp_1.fit(xtrain60_mlp, ytrain60_mlp, epochs=10)
```

```
Epoch 1/10
1639376/1639376 [==============================] - 56s 34us/step - loss: 0.0824
Epoch 2/10
1639376/1639376 [==============================] - 56s 34us/step - loss: 0.0794
Epoch 3/10
1639376/1639376 [==============================] - 57s 35us/step - loss: 0.0782
Epoch 4/10
1639376/1639376 [==============================] - 56s 34us/step - loss: 0.0767
Epoch 5/10
1639376/1639376 [==============================] - 56s 34us/step - loss: 0.0760
Epoch 6/10
1639376/1639376 [==============================] - 56s 34us/step - loss: 0.0756
Epoch 7/10
1639376/1639376 [==============================] - 56s 34us/step - loss: 0.0747
Epoch 8/10
1639376/1639376 [==============================] - 57s 35us/step - loss: 0.0741
Epoch 9/10
1639376/1639376 [==============================] - 56s 34us/step - loss: 0.0736
Epoch 10/10
1639376/1639376 [==============================] - 56s 34us/step - loss: 0.0734
```

Out[0]:

```
<keras.callbacks.History at 0x7fb07ecdda90>
```

Prediction of validation set

In [0]:

```
ypred60_mlp_1 = []
for i in tqdm(range(xvalid60_mlp.shape[0])):
    temp1 = (xvalid60_mlp[i].reshape(1,window_size))
    temp2 = (model60_mlp_1.predict(temp1)).reshape(1)[0]
    ypred60_mlp_1.append(temp2)
```

```
100%|██████████| 409844/409844 [04:12<00:00, 1622.06it/s]
```

Root mean squared error on validation set

In [0]:

```
rmse60_mlp_1 = sqrt(mean_squared_error(yvalid60_mlp, ypred60_mlp_1))
rmse60_mlp_1
```

Out[0]:

```
0.21956389321865233
```

R2score on validation set

In [0]:

```
r2score60_mlp_1 = r2_score(yvalid60_mlp, ypred60_mlp_1)
r2score60_mlp_1
```

Out[0]:

```
0.9399541063761594
```

Mean absolute percentage error on validation

In [0]:

```
arr_ypred60_mlp_1 = np.array(ypred60_mlp_1)
yv60_mlp_1 = yvalid60_mlp.reshape(-1,1)
yp60_mlp_1 = arr_ypred60_mlp_1.reshape(-1,1)
mape60_mlp_1 = mean_absolute_percentage_error(yv60_mlp_1,yp60_mlp_1)
mape60_mlp_1
```

Out[0]:

```
10.29135787493216
```

Experiment 4:- Three hidden layers and one output layer is used. ReLu activation has been used in first two hidden layers. Sigmoid activation function has been used in the third hidden layer. There are 1000 neurons in the first hidden layer, 500 neurons in second hidden layer, 200 neurons in the third hidden layer and single neuron in the output layer. All the layers are densly connected. Stochastic gradient descent optimizer is used with learning rate 0.005 and momentum 0.7 and epoch is 10. Loss function mean squred error is used. Window size is 60.

In [0]:

```
model60_mlp_2 = Sequential()
model60_mlp_2.add(Dense(1000, activation='relu', input_dim = window_size))
model60_mlp_2.add(Dense(500, activation='relu'))
model60_mlp_2.add(Dense(200, activation ='sigmoid'))
model60_mlp_2.add(Dense(1))
optimizer = SGD(lr=0.005, momentum=0.7)
```

```
model60_mlp_2.compile(optimizer=optimizer, loss='mse')
model60_mlp_2.fit(xtrain60_mlp, ytrain60_mlp, epochs=10, verbose =0)
```

Out[0]:

```
<keras.callbacks.History at 0x7f0d237306d8>
```

Prediction on validation set

In [0]:

```
ypred60_mlp_2 = []
for i in tqdm(range(xvalid60_mlp.shape[0])):
    temp1 = (xvalid60_mlp[i].reshape(1,window_size))
    temp2 = (model60_mlp_2.predict(temp1)).reshape(1)[0]
    ypred60_mlp_2.append(temp2)
```

```
100%|██████████| 409844/409844 [05:12<00:00, 1313.55it/s]
```

Root mean squared error on validation set

In [0]:

```
rmse60_mlp_2 = sqrt(mean_squared_error(yvalid60_mlp, ypred60_mlp_2))
rmse60_mlp_2
```

Out[0]:

```
0.23210497908565997
```

R2score on validation set

In [0]:

```
r2score60_mlp_2 = r2_score(yvalid60_mlp, ypred60_mlp_2)
r2score60_mlp_2
```

Out[0]:

```
0.9328987855139678
```

Mean absolute error on validation set

In [0]:

```
arr_ypred60_mlp_2 = np.array(ypred60_mlp_2)
yv60_mlp_2 = yvalid60_mlp.reshape(-1,1)
yp60_mlp_2 = arr_ypred60_mlp_2.reshape(-1,1)
mape60_mlp_2 = mean_absolute_percentage_error(yv60_mlp_2,yp60_mlp_2)
mape60_mlp_2
```

Out[0]:

```
20.467525521011463
```

Experiment 5:- Three hidden layers and one output layer is used. Tanh activation has been used in first two hidden layers. Sigmoid activation function has been used in the third hidden layer. There are 500 neurons in the first hidden layer, 200 neurons in second hidden layer, 100 neurons in the third hidden layer and single neuron in the output layer. All the layers are densely connected. Stochastic gradient descent optimizer is used with learning rate

0.01 and momentum 0.9 and epoch is 10. Loss function mean squred error is used. Window size is 60.

In [0]:

```
model60_mlp_3 = Sequential()
model60_mlp_3.add(Dense(500, activation='tanh', input_dim = window_size))
model60_mlp_3.add(Dense(200, activation='tanh'))
model60_mlp_3.add(Dense(100, activation ='sigmoid'))
model60_mlp_3.add(Dense(1))
optimizer = SGD(lr=0.01, momentum=0.9)
model60_mlp_3.compile(optimizer=optimizer, loss='mse')
model60_mlp_3.fit(xtrain60_mlp, ytrain60_mlp, epochs=10, verbose=1)
```

```
Epoch 1/10
1639376/1639376 [==============================] - 147s 90us/step - loss: 0.0836
Epoch 2/10
1639376/1639376 [==============================] - 149s 91us/step - loss: 0.0765
Epoch 3/10
1639376/1639376 [==============================] - 150s 92us/step - loss: 0.0747
Epoch 4/10
1639376/1639376 [==============================] - 148s 90us/step - loss: 0.0734
Epoch 5/10
1639376/1639376 [==============================] - 145s 88us/step - loss: 0.0727
Epoch 6/10
1639376/1639376 [==============================] - 145s 89us/step - loss: 0.0717
Epoch 7/10
1639376/1639376 [==============================] - 144s 88us/step - loss: 0.0711
Epoch 8/10
1639376/1639376 [==============================] - 147s 90us/step - loss: 0.0710
Epoch 9/10
1639376/1639376 [==============================] - 144s 88us/step - loss: 0.0708
Epoch 10/10
1639376/1639376 [==============================] - 143s 87us/step - loss: 0.0709
```

Out[0]:

```
<keras.callbacks.History at 0x7f977e0702e8>
```

Prediction on validation set

In [0]:

```
ypred60_mlp_3 = []
for i in tqdm(range(xvalid60_mlp.shape[0])):
    temp1 = (xvalid60_mlp[i].reshape(1,window_size))
    temp2 = (model60_mlp_3.predict(temp1)).reshape(1)[0]
    ypred60_mlp_3.append(temp2)
```

```
100%|██████████| 409844/409844 [04:08<00:00, 1650.73it/s]
```

Root mean squred error on validation set

In [0]:

```
rmse60_mlp_3 = sqrt(mean_squared_error(yvalid60_mlp, ypred60_mlp_3))
rmse60_mlp_3
```

Out[0]:

```
0.2184594115901879
```

R2score on validation set

In [0]:

```
r2score60_mlp_3 = r2_score(yvalid60_mlp, ypred60_mlp_3)
```

```
r2score60_mlp_3
```

Out[0]:

```
0.9405566897998614
```

Mean absolute percentage error on validation set

In [0]:

```
arr_ypred60_mlp_3 = np.array(ypred60_mlp_3)
yv60_mlp_3 = yvalid60_mlp.reshape(-1,1)
yp60_mlp_3 = arr_ypred60_mlp_3.reshape(-1,1)
mape60_mlp_3 = mean_absolute_percentage_error(yv60_mlp_3,yp60_mlp_3)
mape60_mlp_3
```

Out[0]:

```
10.723973980677123
```

Multi layer perceptron with window size 120

In [0]:

```
window_size = 120
df120_mlp = copydf
dataset120_mlp = list(df120_mlp["Global_active_power"])
pre_xtrain, pre_ytrain = form_dataset(dataset120_mlp,window_size)
xtest120_mlp = []
for i in test_index:
    xtest120_mlp.append(pre_xtrain[i-window_size])
df120_mlp = df120_mlp.dropna(subset=["Global_active_power"])
dataset120_mlp = list(df120_mlp["Global_active_power"])
pre_xtrain120_mlp, pre_ytrain120_mlp = form_dataset(dataset120_mlp,window_size)
fraction120_mlp = int(0.8 * (len(pre_xtrain120_mlp)))
xtrain120_mlp = pre_xtrain120_mlp[:fraction120_mlp]
xvalid120_mlp = pre_xtrain120_mlp[fraction120_mlp:len(pre_xtrain120_mlp)]
ytrain120_mlp = pre_ytrain120_mlp[:fraction120_mlp]
yvalid120_mlp = pre_ytrain120_mlp[fraction120_mlp:len(pre_xtrain120_mlp)]
xtrain120_mlp = np.array(xtrain120_mlp)
xvalid120_mlp = np.array(xvalid120_mlp)
ytrain120_mlp = np.array(ytrain120_mlp)
yvalid120_mlp = np.array(yvalid120_mlp)
```

Experiment 6:- One hidden layer and one outlayer is used. ReLu activation function has been used in the hidden layer with input dimension of 60. The hidden layer and the output layer is connected densly. Stochastic gradient descent optimizer is used with learning rate 0.003 and momentum 0.9 and epoch is 10. Loss function mean squred error is used. Hidden layer is of 100 neurons and output layer has single neuron. Window size is 120.

In [0]:

```
model120_mlp_1 = Sequential()
model120_mlp_1.add(Dense(100, activation='relu',input_dim=window_size))
model120_mlp_1.add(Dense(1))
optimizer = SGD(lr=0.003, momentum=0.9)
model120_mlp_1.compile(optimizer=optimizer, loss='mse')
model120_mlp_1.fit(xtrain120_mlp, ytrain120_mlp, epochs=10, verbose=1)
```

```
Epoch 1/10
1639328/1639328 [==============================] - 58s 36us/step - loss: 0.0872
Epoch 2/10
```

```
Epoch 2/10
1639328/1639328 [==============================] - 57s 35us/step - loss: 0.0839
Epoch 3/10
1639328/1639328 [==============================] - 58s 35us/step - loss: 0.0826
Epoch 4/10
1639328/1639328 [==============================] - 58s 35us/step - loss: 0.0808
Epoch 5/10
1639328/1639328 [==============================] - 58s 35us/step - loss: 0.0792
Epoch 6/10
1639328/1639328 [==============================] - 58s 36us/step - loss: 0.0782
Epoch 7/10
1639328/1639328 [==============================] - 58s 35us/step - loss: 0.0774
Epoch 8/10
1639328/1639328 [==============================] - 58s 35us/step - loss: 0.0770
Epoch 9/10
1639328/1639328 [==============================] - 58s 35us/step - loss: 0.0765
Epoch 10/10
1639328/1639328 [==============================] - 58s 35us/step - loss: 0.0763
```

Out[0]:

```
<keras.callbacks.History at 0x7f0db99d2940>
```

Prediction on validation set

In [0]:

```
ypred120_mlp_1 = []
for i in tqdm(range(xvalid120_mlp.shape[0])):
    temp1 = (xvalid120_mlp[i].reshape(1,window_size))
    temp2 = (model120_mlp_1.predict(temp1)).reshape(1)[0]
    ypred120_mlp_1.append(temp2)
```

```
100%|██████████| 409832/409832 [04:07<00:00, 1654.24it/s]
```

Root mean squared error on validation set

In [0]:

```
rmse120_mlp_1 = sqrt(mean_squared_error(yvalid120_mlp, ypred120_mlp_1))
rmse120_mlp_1
```

Out[0]:

```
0.22120759768793463
```

R2score on validation set

In [0]:

```
r2score120_mlp_1 = r2_score(yvalid120_mlp, ypred120_mlp_1)
r2score120_mlp_1
```

Out[0]:

```
0.9390525701532718
```

Mean Absolute error on validation set

In [0]:

```
arr_ypred120_mlp_1 = np.array(ypred120_mlp_1)
yv120_mlp_1 = yvalid120_mlp.reshape(-1,1)
yp120_mlp_1 = arr_ypred120_mlp_1.reshape(-1,1)
mape120_mlp_1 = mean_absolute_percentage_error(yv120_mlp_1,yp120_mlp_1)
mape120_mlp_1
```

Out[0]:

```
10.142149633248108
```

Experiment 7:- Three hidden layers and one output layer is used. ReLu activation has been used in first two hidden layers. Sigmoid activation function has been used in the third hidden layer. There are 1000 neurons in the first hidden layer, 500 neurons in second hidden layer, 200 neurons in the third hidden layer and single neuron in the output layer. All the layers are densly connected. Stochastic gradient descent optimizer is used with learning rate 0.005 and momentum 0.7 and epoch is 10. Loss function mean squred error is used. Window size is 120.

In [0]:

```
model120_mlp_2 = Sequential()
model120_mlp_2.add(Dense(1000, activation='relu', input_dim = window_size))
model120_mlp_2.add(Dense(500, activation='relu'))
model120_mlp_2.add(Dense(200, activation ='sigmoid'))
model120_mlp_2.add(Dense(1))
optimizer = SGD(lr=0.005, momentum=0.7)
model120_mlp_2.compile(optimizer=optimizer, loss='mse')
model120_mlp_2.fit(xtrain120_mlp, ytrain120_mlp, epochs=10, verbose=1)
```

```
Epoch 1/10
1639328/1639328 [==============================] - 563s 343us/step - loss: 0.0821
Epoch 2/10
1639328/1639328 [==============================] - 567s 346us/step - loss: 0.0763
Epoch 3/10
1639328/1639328 [==============================] - 572s 349us/step - loss: 0.0748
Epoch 4/10
1639328/1639328 [==============================] - 572s 349us/step - loss: 0.0737
Epoch 5/10
1639328/1639328 [==============================] - 572s 349us/step - loss: 0.0727
Epoch 6/10
1639328/1639328 [==============================] - 575s 351us/step - loss: 0.0721
Epoch 7/10
1639328/1639328 [==============================] - 578s 352us/step - loss: 0.0714
Epoch 8/10
1639328/1639328 [==============================] - 601s 367us/step - loss: 0.0707
Epoch 9/10
1639328/1639328 [==============================] - 591s 360us/step - loss: 0.0701
Epoch 10/10
1639328/1639328 [==============================] - 580s 354us/step - loss: 0.0697
```

Out[0]:

```
<keras.callbacks.History at 0x7f0db8010da0>
```

Prediction on validation set

In [0]:

```
ypred120_mlp_2 = []
for i in tqdm(range(xvalid120_mlp.shape[0])):
    temp1 = (xvalid120_mlp[i].reshape(1,window_size))
    temp2 = (model120_mlp_2.predict(temp1)).reshape(1)[0]
    ypred120_mlp_2.append(temp2)
```

```
100%|██████████| 409832/409832 [05:31<00:00, 1234.89it/s]
```

Root mean squared error on validation set

In [0]:

```
rmse120_mlp_2 = sqrt(mean_squared_error(yvalid120_mlp, ypred120_mlp_2))
rmse120_mlp_2
```

Out[0]:

```
0.21438090078325006
```

R2score on validation set

In [0]:

```
r2score120_mlp_2 = r2_score(yvalid120_mlp, ypred120_mlp_2)
r2score120_mlp_2
```

Out[0]:

```
0.9427563259360379
```

Mean abslute percentage error on validation set

In [0]:

```
arr_ypred120_mlp_2 = np.array(ypred120_mlp_2)
yv120_mlp_2 = yvalid120_mlp.reshape(-1,1)
yp120_mlp_2 = arr_ypred120_mlp_2.reshape(-1,1)
mape120_mlp_2 = mean_absolute_percentage_error(yv120_mlp_2,yp120_mlp_2)
mape120_mlp_2
```

Out[0]:

```
11.88265276965425
```

Experiment 8:- Three hidden layers and one output layer is used. Tanh activation has been used in first two hidden layers. Sigmoid activation function has been used in the third hidden layer. There are 500 neurons in the first hidden layer, 200 neurons in second hidden layer, 100 neurons in the third hidden layer and single neuron in the output layer. All the layers are densly connected. Stochastic gradient descent optimizer is used with learning rate 0.01 and momentum 0.9 and epoch is 10. Loss function mean squred error is used. Window size is 120.

In [0]:

```
model120_mlp_3 = Sequential()
model120_mlp_3.add(Dense(500, activation='tanh', input_dim = window_size))
model120_mlp_3.add(Dense(200, activation='tanh'))
model120_mlp_3.add(Dense(100, activation ='sigmoid'))
model120_mlp_3.add(Dense(1))
optimizer = SGD(lr=0.01, momentum=0.9)
model120_mlp_3.compile(optimizer=optimizer, loss='mse')
model120_mlp_3.fit(xtrain120_mlp, ytrain120_mlp, epochs=10, verbose=1)
```

```
Epoch 1/10
1639328/1639328 [==============================] - 189s 116us/step - loss: 0.0907
Epoch 2/10
1639328/1639328 [==============================] - 178s 108us/step - loss: 0.1062
Epoch 3/10
1639328/1639328 [==============================] - 175s 107us/step - loss: 0.1435
Epoch 4/10
1639328/1639328 [==============================] - 170s 104us/step - loss: 0.1359
Epoch 5/10
```

```
1639328/1639328 [==============================] - 170s 104us/step - loss: 0.1183
Epoch 6/10
1639328/1639328 [==============================] - 170s 104us/step - loss: 0.1128
Epoch 7/10
1639328/1639328 [==============================] - 169s 103us/step - loss: 0.1170
Epoch 8/10
1639328/1639328 [==============================] - 170s 104us/step - loss: 0.1780
Epoch 9/10
1639328/1639328 [==============================] - 172s 105us/step - loss: 0.1877
Epoch 10/10
1639328/1639328 [==============================] - 175s 107us/step - loss: 0.1638
```

Out[0]:

```
<keras.callbacks.History at 0x7f0db7574550>
```

Prediction on validation set

In [0]:

```
ypred120_mlp_3 = []
for i in tqdm(range(xvalid120_mlp.shape[0])):
    temp1 = (xvalid120_mlp[i].reshape(1,window_size))
    temp2 = (model120_mlp_3.predict(temp1)).reshape(1)[0]
    ypred120_mlp_3.append(temp2)
```

```
100%|██████████| 409832/409832 [04:51<00:00, 1404.81it/s]
```

Root mean squared error on validation set

In [0]:

```
rmse120_mlp_3 = sqrt(mean_squared_error(yvalid120_mlp, ypred120_mlp_3))
rmse120_mlp_3
```

Out[0]:

```
0.33799011048392225
```

R2score on validation set

In [0]:

```
r2score120_mlp_3 = r2_score(yvalid120_mlp, ypred120_mlp_3)
r2score120_mlp_3
```

Out[0]:

```
0.8577136287451945
```

Mean absolute percentage error on validation set

In [0]:

```
arr_ypred120_mlp_3 = np.array(ypred120_mlp_3)
yv120_mlp_3 = yvalid120_mlp.reshape(-1,1)
yp120_mlp_3 = arr_ypred120_mlp_3.reshape(-1,1)
mape120_mlp_3 = mean_absolute_percentage_error(yv120_mlp_3,yp120_mlp_3)
mape120_mlp_3
```

Out[0]:

```
31.974084487858207
```

Summary : The best performing model is as three hidden layers

and one output layer is used. ReLu activation has been used in first two hidden layers. Sigmoid activation function has been used in the third hidden layer. There are 1000 neurons in the first hidden layer, 500 neurons in second hidden layer, 200 neurons in the third hidden layer and single neuron in the output layer. All the layers are densly connected. Stochastic gradient descent optimizer is used with learning rate 0.005 and momentum 0.7 and epoch is 10. Loss function mean squred error is used. Window size is 120.

In [0]:

```

```