**HW2 for deep learning and data mining**

Team Member: Mei Ni; Hsien-Hao Xu; Yingyu Huang

Contribution:

Mei Ni ([plumnini.xj@gmail.com](mailto:plumnini.xj@gmail.com)):

Hsien-Hao Hsu ([sam10030179@gmail.com](mailto:sam10030179@gmail.com)):

Yingyu Huang ([yingyu010365@gmail.com](mailto:yingyu010365@gmail.com)):

1. Data set and the associated classification task

1.1 Background of the data and the prediction task

This Avila data set has been extracted from 800 images of the "Avila Bible’. Every case in the data contains 10 features and corresponds to a group of 4 consecutive rows. The class labels are given as the patterns named A,B,C,D,E,F,G,H,I,W,X, and Y, corresponding to the 12 copyists.

In this project, the prediction task is to find an MLP model which can classify the sub dataset into 3 patterns and associate each pattern to a copyist.

1.2 Data set

Choose the first three largest class with totally 14684 samples. Randomly select 80% of the samples as training set and the rest as test set.

Expand the size of class 2 by cloning one time, and class 3 by cloning two times. The the three classes has almost same size.

Total number of cases in the first three largest class is



2. explain precisely what is the part played by the softmax function, and how the final

classification of Xk is computed from OUTk

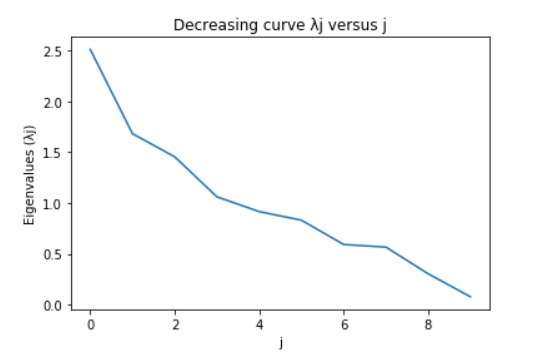
The MLP classifier transforms the input X(n) into an output vector Z(n) readable on the last layer of the MLP and the softmax function transforms Z(n) into a probability vector q = q1…qk.

For each input X(n) theideal terminal output r = tr(n) of the automatic classifier will be encoded by the binary vector p(n) = [p1…pk] where pr = 1,and pj = 0 for all jr. Note that p(n) is a probability on the alphabet [1,2,... k]. Each probability p(n) is quite special because it is actually deterministic, and hence its entropy Ent(p(n)) is zero.

3.

After apply PCA analysis to the training set of input vectors to generate p eigenvalues and compute the smallest number "h95" of eigenvalues preserving 95% of the total sum of eigenvalues ordered in decreasing order, we get h95=7.

The decreasing sequence of eigenvalues:



After apply PCA analysis to the set of Mj input vectors corresponding to the class Cj, with j=1,2 3 to generate Mj eigenvalues in decreasing order, and compute the smallest number "Uj" of eigenvalues preserving 95% of the total sum of these Mj eigenvalues, we get hL = U1 + U2 +U3 = 7+7+7=21

**4. Implement automatic training**

**4.1 What is average CRE?**

Let X = [ (X(1), X(2)…,X(n)] is the input data set, with p features in each cases. Classes number is k.

* To any X(n) in X, we input it to the MLP, then we will get output Q(n) = [q1, q2,….,qk], qi is the probability that X(n) belong to class i, which is computed by MLP. q1+q2+…+qk=1 and 0<=qi<=1
* To any X(n), the true class has been given as a vector Y= [p1, p2, … pk]=[0,0…1,…0] in Rk, a special probability vector that X(n) belongs to class i, i= 1…k.
* Cross Entropy (X(n)) = - p1\*log q1 – p2\*logq2 - …. - pk\*logqk = - log qi, i is the true class of X(n).

If let v= Cross Entropy (X(n)), e-v is the probability that X(n) is correctly classified.

* If we input m samples in data set to MLP, we will get m CREs of all the inputs

Average CRE = 1/m ( CRE(X(1)) + CRE(X(2)) + … + CRE(X(m)))

* Average CRE is the loss function of MLP, which is the function of the unknown parameters. The goal of the training is to get the minimum of average CRE, or to say, to get the best model with smallest loss value.
* In the training procedure, at the end of each epoch, we can get an average CRE based on the current parameter. Then the parameter will be optimized by the learning algorithm to get a smaller avCRE in the next iteration.

**4.2 What is the software environment**

In this project, we use tensor flow 2.1.0 to construct the MLP automatic learning model in Python.

**4.3 The functions used to implement MLP**

* Construct the architecture of model:

Input layer: neurons = 10

Hidden layer: neurons = 7, active function is RELU

Output Layer: neurons = 3, active function is RELU and SOFTMAX

model = Sequential()

model.add(Dense(7, activation='relu', input\_dim=10, kernel\_initializer=keras.initializers.RandomNormal(mean=0.0,

stddev=0.05, seed=1), bias\_initializer='zeros'))

model.add(Dense(NumClass, activation='softmax', kernel\_initializer=keras.initializers.RandomNormal(mean=0.0, stddev=0.05,

seed=2), bias\_initializer='zeros'))

* Model compiling

Loss function is Cross Entropy.

Learning algorithm is Gradient Descent.

loss\_fn = losses.CategoricalCrossentropy() # set the loss function

sgd = optimizers.SGD(learning\_rate=lr\_schedule) # set the learning algorithm -- gradient descent

model.compile(optimizer=sgd,  loss=loss\_fn, metrics=['accuracy'])

* Model fitting

model.fit(trainX, trainY, epochs=100, batch\_size=MiniBatchSize, callbacks=[checkpointer],validation\_data=(testX, testY))

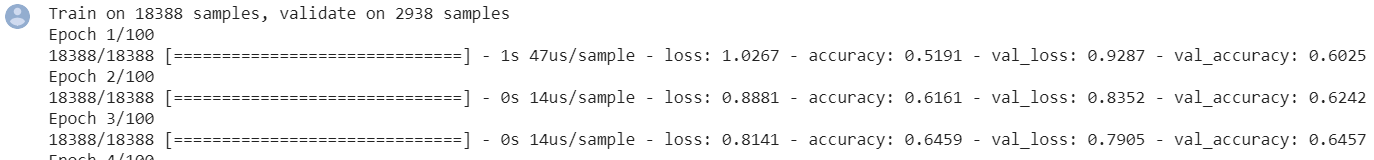
**4.4 what are the options offered for this task**

* Learning algorithm: “Batch Size” Gradient Descent
  + Gradient descent is an iterative algorithm, that starts from a random point on a function and travels down its slope in steps until it reaches the lowest point of that function.
  + Let W be the vector of all the unknown parameters of MLP, which contains weights and thresholds and need to be trained. The dimension of W is D = (input size \* hidden layer size + hidden layer size + hidden layer size \* output size + output size). In order to minimize the loss value (acCRE), W should be optimized in each of the iteration by the algorithm

W(s+1) = W(s) – ε(s+1) \* Gradient (W(s))

where s is the current learning step, Gradient(W(s)) is a long vector in RD with the coordinates equal to , ε(n) is the learning rate which control the speed at which the model learns. The step size of learning is learning rate\* gradient.

* + We can use all the training data to each of the learning iterations or we can divide the data set into many batches and use one batch in one iteration. In this project, the number of the training cases is 18388. We choose the ‘Batch Size’ gradient descent algorithm to train the model. Batch size is 128.
* initialization of the weights, thresholds, batch size and learning rate in this implementation:
  + initialization of the weights is random value of a normal distribution with mean=0 and standard deviation = 0.05.
  + initialization of threshold is zero.
  + batch size = 128
  + initial learning rate = 0.1
* The successive gradient descent steps sizes ε(n)
  + Learning rate (ε(n)) decay schedule: Exponential Decay
  + learning rate = initial learning rate \* decay rate s/decay steps, s is the current learning steps
  + initial learning rate = 0.1
  + decay rate =
  + decay steps = size of the training set / batch size = 18388/128 ≅ 144
* Monitor learning quality and stopping the learning
  + Firstly, set a large number for epoch (eg. epoch = 300 )
  + Then set the loss and accuracy of each epoch of training set and test set to be visible to monitor the performance and find the stop point.

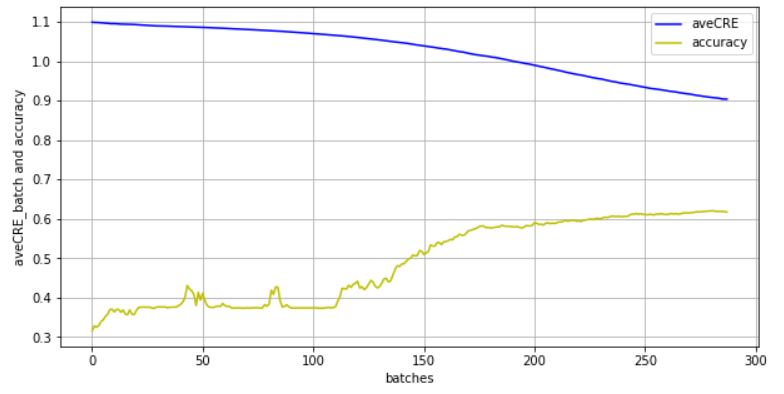


**5. Automatic learning typically implements successive steps**

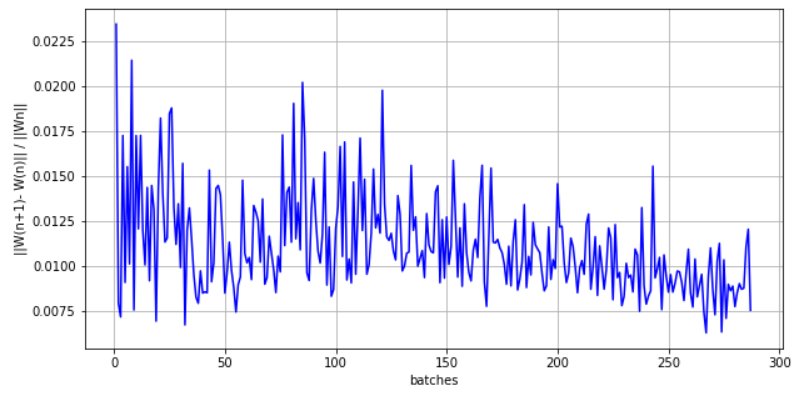
**5.1 implement MLP on training set with hidden layer = 21 and epoch = 2**

**5.2 Extract loss value and accuracy of training data by batches, plot the curve n vsbaCREn**

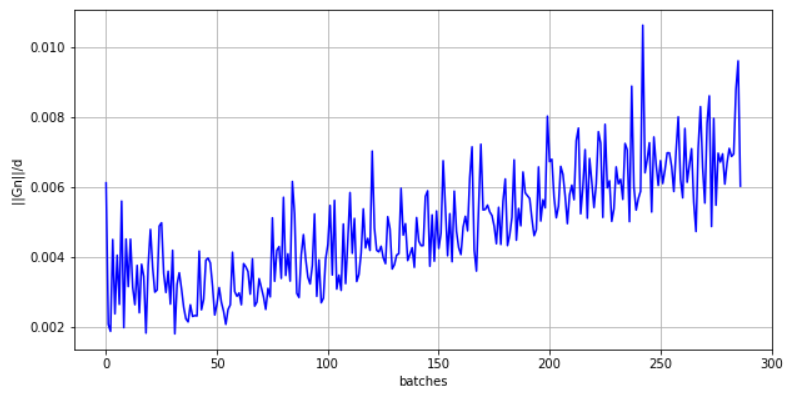
* In the first 2 epoch, the learning algorithm iterates 2\*144=288 times or batches.
* At the end of each batch, extract the model performance (avCRE, acc), parameter’s current values (weight and threshold) and learning step size (ε ) by callback function in Keras.
* Plot the performance vs batches. As the curves show, the cost decreases and the accuracy increases.



* Plot the curve n vs|| W(n+1)- W(n)|| / ||Wn||

As the n increases, || W(n+1)- W(n)|| / ||Wn|| decreases and the amplitude of vibration reduces.   


* plot the curve n vs ||Gn|| / d, where d = 10\*21+21+21\*3 + 3 = 297, ||Gn|| = ( 1/ ε(n) ) || W(n+1)- W(n)||



**6 Performance analysis**

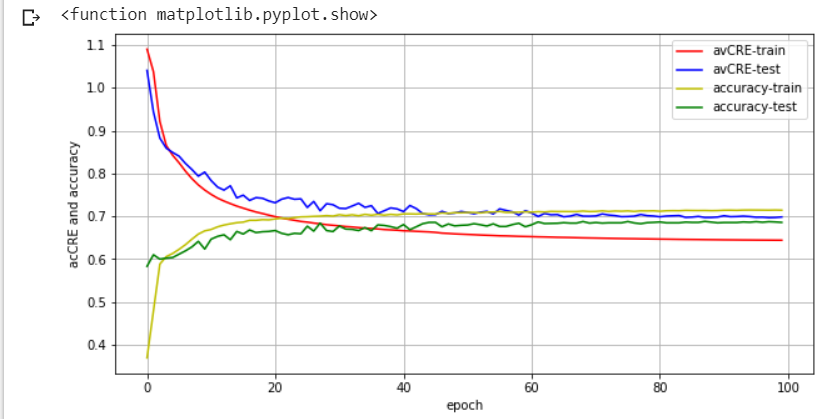
* h = 7, implement MLP with 7 neurons in hidden layer.
* avCRE of training set is 0.64; of test set is 0.71.
* accuracy (% of the correct answer) of training set is 71%±0.3%, of test set is 69% ±0.3%.
* Confusion matrix:

Class E has the best performance, Class A and F have 26% cases are mixed up and wrongly classified.



* Plot the loss and accuracy:

Because the hidden layer has a small size, the optimization almost stops at 50th epoch, and the gap between training set and test set is small. The model doesn’t over fit but the performance is not good enough. Maybe a larger size of hidden layer will improve the accuracy.



**7. Impact of various learning options**

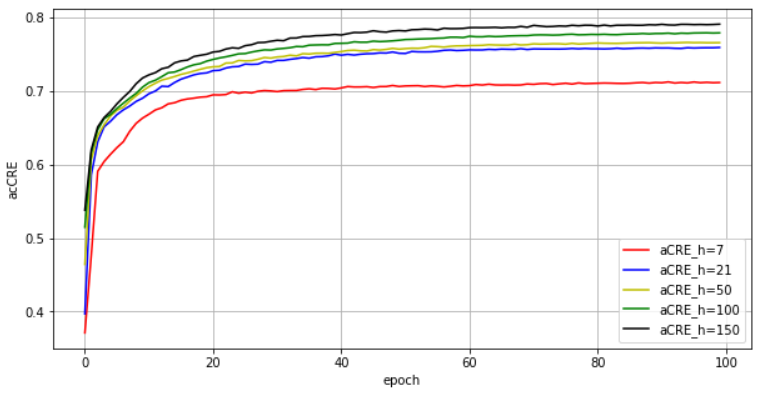
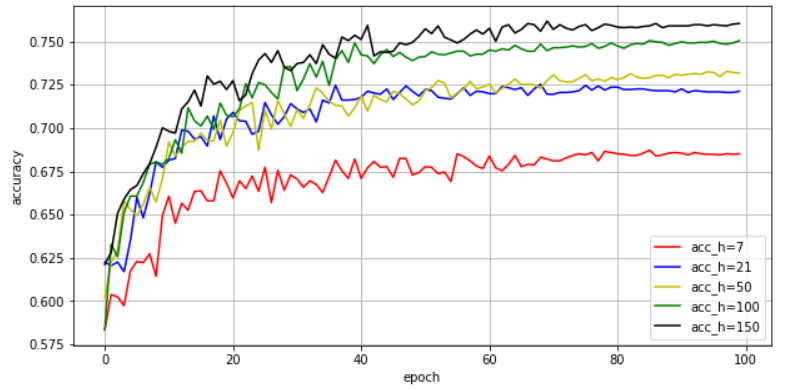
**7.1 Change the size of hidden layer**

Implement MLP with the size of hidden layer as [7,21, 50, 100, 150]. As the number of neuron increase, the performance is improved. Especially, when h=21, the improvement is significant, while h = [50,100,150], the improvement is not significant.

**Performance of the last epoch = 100**



**Plot the performance of test set**



**7.2 Change the batch size from 128 to 256**

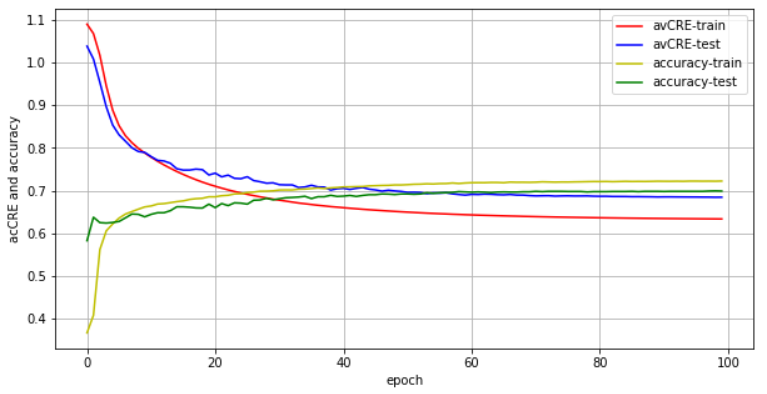
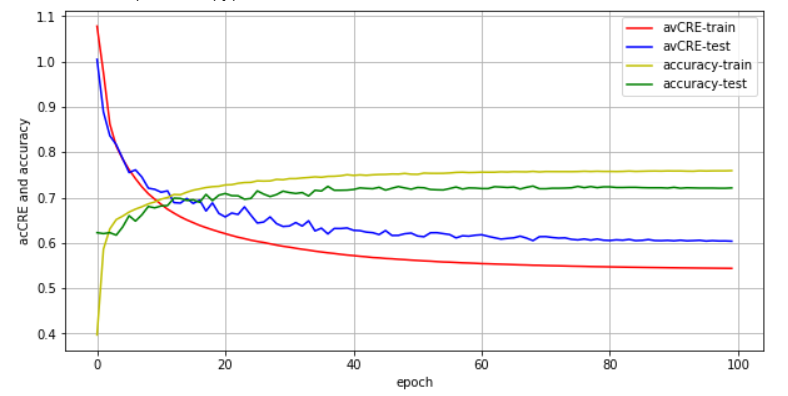
When the batch size doubles, the performance at 100th epoch is worse. Because the iterations of learning decreases, the learning speed becomes slower. This is proved by the plot curve of the performance, where the loss value drops down with less sharpness when batch size = 256. So, small batch size will lead to a faster convergence than the large one.

**Performance of the last epoch = 100**



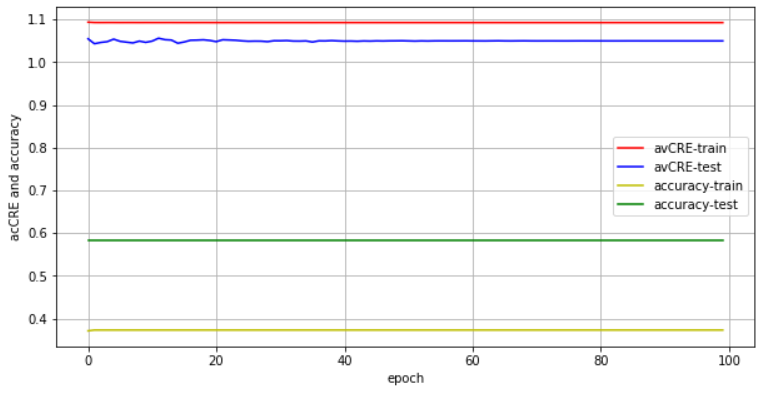
**Plot the performance**

**Batch size = 128 Batch size = 256**



**7.3 Change the initialization of weight**

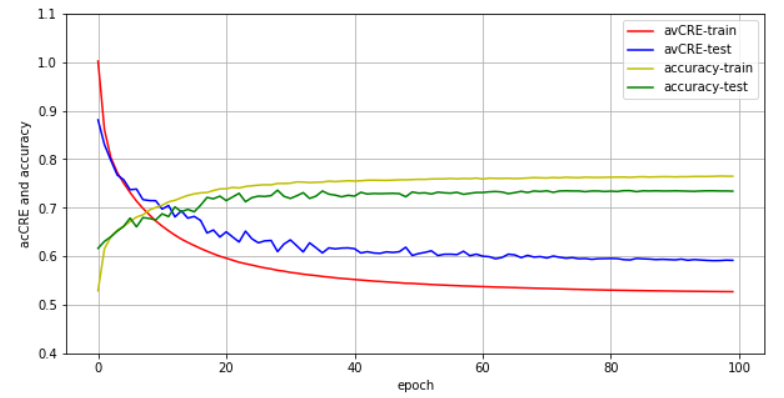
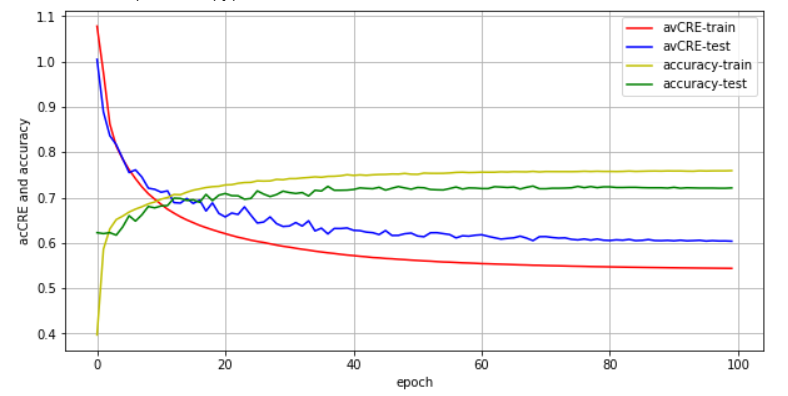
1. Implementing MLP with zero as the initial value of weight and threshold, the learning will fail.



1. Initialize the weight with random sample from uniform distribution,



Random normal Random uniform



**7.4 change the learning step size**

**8. Analysis of the hidden layer behaviour after completion of automatic learning**

**8.1 Call the best h**

h = 21

**8.2 Do PCA Analysis to hidden layer generated by training set**

First we do PCA to whole training set

Code:

Houtput = model.layers[0](trainX).numpy()

df = pd.DataFrame(Houtput)

from sklearn.decomposition import PCA

pca = PCA(n\_components=3)

principalComponents = pca.fit\_transform(df)

principalDf = pd.DataFrame(data = principalComponents

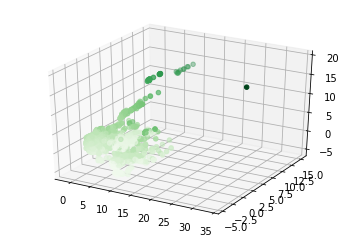
, columns = ['principal component 1', 'principal component 2','principal component 3'])

from mpl\_toolkits import mplot3d

fig = plt.figure()

ax = plt.axes(projection='3d')

ax.scatter3D(principalDf.iloc[:,0],principalDf.iloc[:,1],principalDf.iloc[:,2], c=principalDf.iloc[:,2], cmap='Greens')



Secondly, we do PCA analysis to 3 different groups of training set each generated by training set cases which belong to Class 1, Class 2 and Class 3

Code:

df\_trainx = pd.DataFrame(trainX.numpy())

df\_trainy = pd.DataFrame(np.argmax(trainY, axis=1))

CL1 = pd.DataFrame()

CL2 = pd.DataFrame()

CL3 = pd.DataFrame()

for i in range(18388):

if df\_trainy.iloc[i,0] == 0:

CL1 = CL1.append(pd.DataFrame(df\_trainx.iloc[i,:]).T)

elif df\_trainy.iloc[i,0] == 1:

CL2 = CL2.append(pd.DataFrame(df\_trainx.iloc[i,:]).T)

else:

CL3 = CL3.append(pd.DataFrame(df\_trainx.iloc[i,:]).T)

CL1=np.random.permutation(CL1)

CL2=np.random.permutation(CL2)

CL3=np.random.permutation(CL3)

CL1=tf.convert\_to\_tensor(CL1, dtype=tf.float32)

CL2=tf.convert\_to\_tensor(CL2, dtype=tf.float32)

CL3=tf.convert\_to\_tensor(CL3, dtype=tf.float32)

Houtput\_1 = model.layers[0](CL1).numpy()

Houtput\_2 = model.layers[0](CL2).numpy()

Houtput\_3 = model.layers[0](CL3).numpy()

df\_1 = pd.DataFrame(Houtput\_1)

df\_2 = pd.DataFrame(Houtput\_2)

df\_3 = pd.DataFrame(Houtput\_3)

from sklearn.decomposition import PCA

pca\_1 = PCA(n\_components=3)

pca\_2 = PCA(n\_components=3)

pca\_3 = PCA(n\_components=3)

principalComponents\_1 = pca\_1.fit\_transform(df\_1)

principalDf\_1 = pd.DataFrame(data = principalComponents\_1,columns = ['1','2','3'])

principalComponents\_2 = pca\_2.fit\_transform(df\_2)

principalDf\_2 = pd.DataFrame(data = principalComponents\_2,columns = ['1','2','3'])

principalComponents\_3 = pca\_3.fit\_transform(df\_3)

principalDf\_3 = pd.DataFrame(data = principalComponents\_3,columns = ['1','2','3'])

from mpl\_toolkits.mplot3d import Axes3D

import matplotlib.pyplot as plt

fig = plt.figure()

fig, ax = plt.subplots(figsize=(15,15))

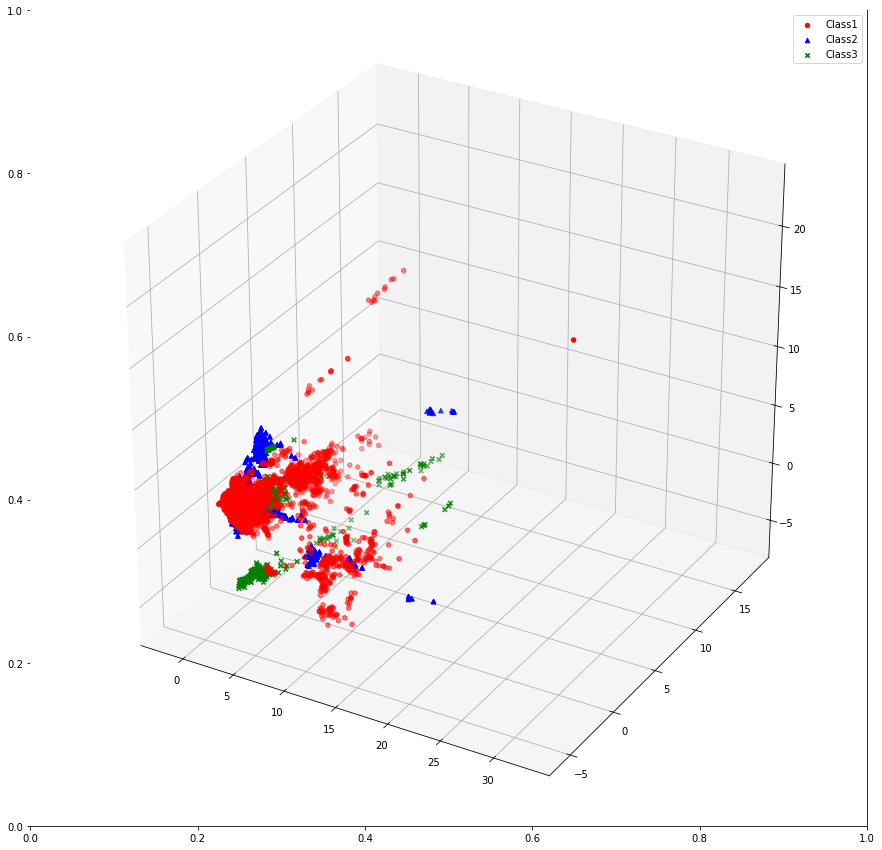
ax = fig.add\_subplot(111, projection='3d')

ax.scatter(principalDf\_1.iloc[:,0],principalDf\_1.iloc[:,1],principalDf\_1.iloc[:,2],c='r',marker='o',label='Class1')

ax.scatter(principalDf\_2.iloc[:,0],principalDf\_2.iloc[:,1],principalDf\_2.iloc[:,2],c='b',marker='^',label='Class2')

ax.scatter(principalDf\_3.iloc[:,0],principalDf\_3.iloc[:,1],principalDf\_3.iloc[:,2],c='g',marker='x',label='Class3')

ax.legend()



We can see that the hidden layer of three classes are not well separated. But still can distinguish.

**8.3 Display and compare the average hidden neurons activity profiles PROF1 , PROF2, PROF3**

Code:

fig, ax = plt.subplots(figsize=(15,5))

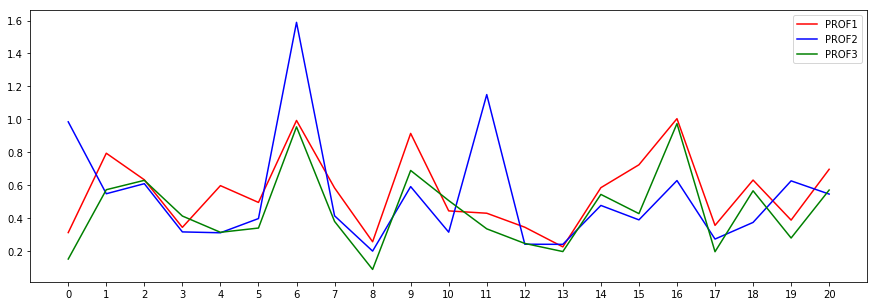
plt.plot(pd.DataFrame(Houtput\_1).mean(),color="red",label="PROF1")

plt.plot(pd.DataFrame(Houtput\_2).mean(),color="blue",label="PROF2")

plt.plot(pd.DataFrame(Houtput\_3).mean(),color="green",label="PROF3")

plt.xticks(range(0, 21))

plt.legend()

****

**8.4 List the hidden neurons which achieve best DIFFERENTIATION between class C1 vs C2, C1 vs C3, C2 vs C3**

Code:

for i in range(20):

if abs(pd.DataFrame(Houtput\_1).mean()[i] - pd.DataFrame(Houtput\_2).mean()[i])> 0.7:

print(i)

* 11

for i in range(20):

if abs(pd.DataFrame(Houtput\_1).mean()[i] - pd.DataFrame(Houtput\_3).mean()[i])> 0.29:

print(i)

* 15

for i in range(20):

if abs(pd.DataFrame(Houtput\_2).mean()[i] - pd.DataFrame(Houtput\_3).mean()[i])> 0.83:

print(i)

* 0

Interpret:

the 11th hidden neuron achieve best DIFFERENTIATION between class C1 versus C2

which means the 11th hidden neuron do lots of work on Classify C2 but do relevantly nothing on C1

the 15th hidden neuron achieve best DIFFERENTIATION between class C1 versus C3

which means the 15th hidden neuron do lots of work on Classify C1 but do relevantly nothing on C3

the 1st hidden neuron achieve best DIFFERENTIATION between class C2 versus C3

which means the 1st hidden neuron do lots of work on Classify C2 but do relevantly nothing on C3