1.0 PART 1

1.1 Description of Data Set

A Vicon motion capture camera system was used to record users performing 3 hand postures with markers attached to a left-handed glove. Markers with a rigid pattern on the glove was used to establish a local co-ordinate system. The goal of the classification task is to correctly predict the hand movements.

As shown in **Table 1**, the size of our original data set was 78,095 cases and 34 features. This was later reduced to 10, 002 cases and 15 features as some of the features have incomplete data and were thus eliminated. **Table 2** show the original classes and their sizes. We later selected only 3 classes of size 3,334 each as shown in **Table 3**. The 3,334 cases were randomly selected to ensure that there is no hidden order in the dataset. We later divided the reduced dataset into 80% training sets and 20% test sets (**Table 4**). **Table 4** shows the sizes of the training and test sets as well as the size and proportions of their respective classes. As it can be seen the proportion of each class in the training and test set is practically the same. This is an essential part towards the classification problem.

In addition, the three basic co-ordinates that re presents the features are:
'Xi' - Real. The x-coordinate of the i-th unlabeled marker position. 'i' ranges from 0 to 4.
'Yi' - Real. The y-coordinate of the i-th unlabeled marker position. 'i' ranges from 0 to 4.
'Zi' - Real. The z-coordinate of the i-th unlabeled marker position. 'i' ranges from 0 to 4.

Table 1: Size of data

ID	Data	Size
1	Size of original data set	78,095 cases and 34 Features
2	Size of reduced data set	10,002 cases and 15 features

Table 2: Size of original classes

Classes	Class Meaning	Size
1	Fist (with thumb out)	16,265
2	Stop (hand flat)	14, 978
3	Point 1 (point with pointer finger)	16, 344

4	Point 2 (point with pointer and middle fingers)	14, 775
5	Grab (fingers curled as if to grab)	15, 733

Table 3: Size of kept classes

Classes	Class Meaning	Sizes
1	Fist (with thumb out)	3,334 (Randomly Selected)
2	Stop (hand flat)	3,334 (Randomly Selected)
3	Point1 (point with pointer finger)	3,334 (Randomly Selected)

Table 4: Proportion of cases

Classes	Total Size	Data	Size	Proportions
Training Set	8001	Class 1	2671	33.38 %
		Class 2	2676	33.45 %
		Class 3	2654	33.17 %
Test Set	2001	Class 1	663	33.13 %
		Class 2	658	32.89 %
		Class 3	680	33.98 %

```
#Import and shuffle the data to achieve randomness
data1 = shuffle(pd.read_csv("C:/Users/jamiu/OneDrive/Documents/Summarized_Data.csv").iloc[:,1:17]).reset_index(dro
data1.head(5)
```

	X0	Y0	Z0	X1	Y1	Z1	X2	Y2	Z2	X3	Y3	Z3
0	87.016700	129.618498	-24.270710	76.748724	83.786963	-26.998948	86.095194	10.914820	-87.851044	24.918868	101.571559	-20.856917
1	30.697188	151.065603	34.220707	43.720312	137.198268	25.032245	14.890409	143.246123	38.634689	1.596005	118.486356	35.050281
2	89.944859	76.902952	-24.410426	101.321836	133.235552	9.625583	61.003285	141.479748	33.619975	10.356824	81.486461	25.228471
3	113.246715	40.010163	-58.375726	75.525373	128.192859	-18.520465	-6.657308	74.721169	-45.417755	72.858145	84.277519	-22.487738
4	89.741977	74.558727	-40.883478	19.285592	140.050484	42.328346	53.944025	147.298576	24.894506	-1.287505	122.126818	32.984610

Figure 1: Kept data.

```
#Get the classes for each input variables
D1= data1[data1['Class']==1].iloc[:,0:15]
D2= data1[data1['Class']==2].iloc[:,0:15]
D3= data1[data1['Class']==3].iloc[:,0:15]
print ("Class 1 : "+ str(D1.shape[0]),"Class 2: "+ str(D2.shape[0]), "Class 3: "+str(D3.shape[0]) )
```

```
Class 1 : 3334 Class 2: 3334 Class 3: 3334
```

Figure 2: Size of kept classes.

```
print ("Class 1 (training set): "+ str(D1tr.shape[0]),"Class 2 (training set): "+ str(D2tr.shape[0]), "Class 3 (tr
print ("Class 1 (test set): "+ str(D1te.shape[0]),"Class 2 (test set): "+ str(D2te.shape[0]), "Class 3 (test set):

Class 1 (training set): 2671 Class 2 (training set): 2676 Class 3 (training set): 2654
Class 1 (test set): 663 Class 2 (test set): 658 Class 3 (test set): 680
```

Figure 3: Size of classes in both training and test sets.

2.0 PART 2

2.1 Description of the MLP Architecture

We selected an MLP architecture defined by three layers. The first is the input layer with number of neurons equals 15 (p=15). This represent the number of features of our cases (input data). The second layer is the hidden later with RELU activation function and an unknown dimension (for now) and lastly the output layer (dimension=3 which represent the number of classes) extended by the softmax function.

The following function takes input a vector X following form:

```
def MLP(x):
    layer_1=tf.add(tf.matmul(x,weights['h']), biases['b'])
    layer_1=tf.nn.relu(layer_1)
    out_layer=tf.matmul(layer_1, weights['out'])+biases['out']

    return out_layer

#construct model
logits=MLP(X)
#define loss and optimizer
loss=tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=Y,logits=logits))
```

Figure 4: MLP architecture

The softmax function takes as an input, the output result from the MLP consisting of real numbers and then transforms them into a probability distribution proportional to the exponential of the input numbers to a vector of size 3. Once the vector of probabilities is obtained, the cell corresponding to the highest probability in the vector is said to be the predicted class. This can be compared with the binary code for the true output to measure the discrepancy in the form of average entropy.

If $\zeta_1, \zeta_2, ..., \zeta_j$ represents the output from the MLP, the softmax function is given by:

$$q_j = \frac{\exp(\zeta_j)}{\exp(\zeta_1)+\exp(\zeta_2)+...+\exp(\zeta_j)}, \text{ where } q_1+q_2+q_3+...+q_j = 1 \text{ and } 1 \geq q_j \geq 0 \quad (1)$$

If ζ_j is large, then q_j tend to 1 and if ζ_j is small then q_j tend to 0. It is important to note that this function has no weights and the maximum of the probabilities gives the true class of the case considered.

True output = max ($q_1, q_2, q_3, ..., q_j$) (2)

2.2 Estimation of plausible value of h = h95 < p

For the hidden layer, there is a need to have a guideline to compute the unknown neurons in the hidden layer. This is because if h is taken to be very large, the automatic learning will be long and will perform extremely well for the training set and poorly for the test set. This is called overfitting and thus the capacity of the generalization might be very weak. Besides, if we select h to be very small, there is a possibility for the architecture to be too weak to analyse the data. The performance will be poor. Hence, we need to have a value for h such that the result obtained will be stable for both training and test sets. This is the reason why we performed PCA analysis to determine the value of h to choose from, that will result in best performance.

For the first scenario which has to do with estimation of the lowest value of h, we performed a PCA analysis on the input data.

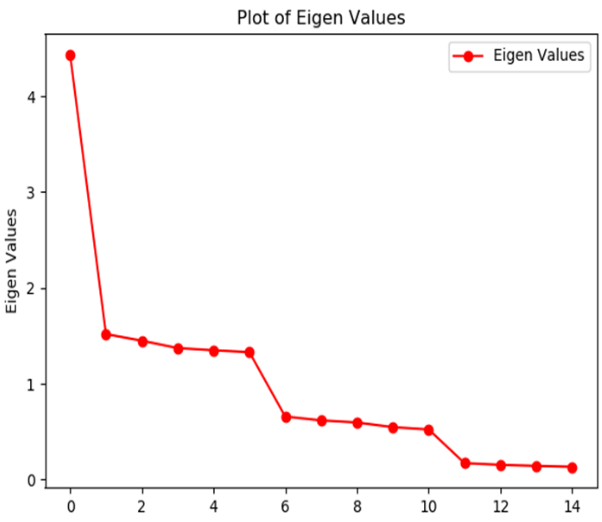


Figure 5: Eigen values for input data

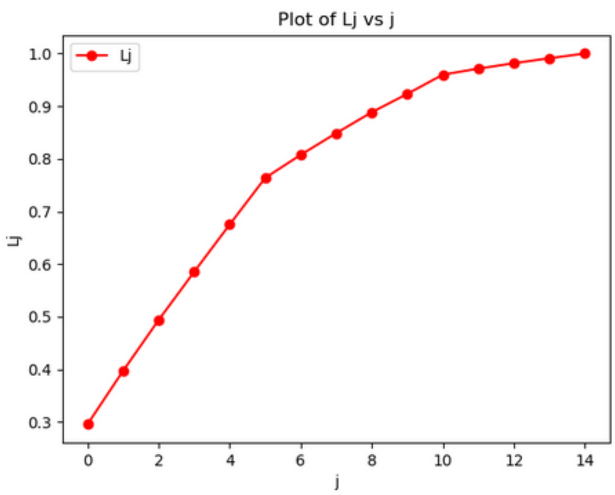


Figure 6: Li (Ri) for input data

```
#Smallest Number h95
compare1=sum(L1)*0.95
s1=0
count1=0
for i in L1:
    count1=count1+1
    s1=s1+i
   # print(i)
    if s1>compare1:
       # print(count1)
        break

h95=count1
print("The smallest number h95 is: " + str(h95))
```

```
The smallest number h95 is: 11
```

Figure 7: R95 computed for h95.

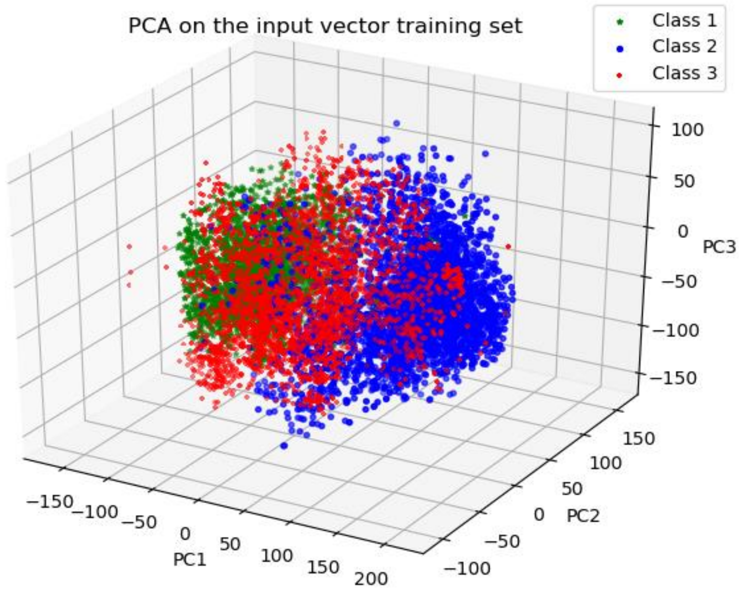


Figure 8: 3D Projections of eigen vectors on input data

Figure 4 shows the decreasing trend of the positive eigen values. **Figure 5** represents the cumulative sum of the eigen vectors which gives the proportion of the explained variance. The eigen vectors generally explained certain percentage of the total dispersion of the data. Hence, the explained dispersion Li tends to 1 when R gets to p. Generally, there is a need to find a good truncation value of R such that the ratio is close to 95%. This corresponds to 11 as we estimated from the graph (**Figure 6)** and calculated using the code in **Figure 7**. The value R=11 thus represents the minimum number of hidden layers to use that will capture about 95% variance in our data. Consequently, this value of R95 gives the smallest dimension h95 for the hidden layer.

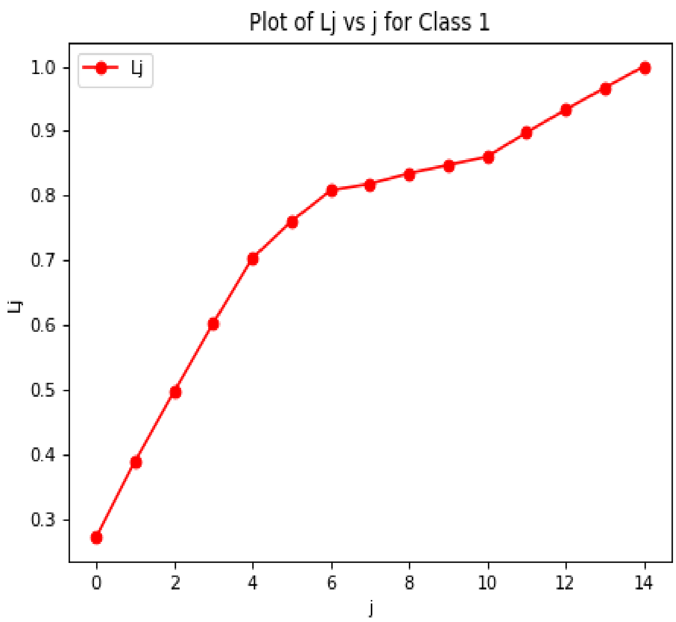
We also projected the input data configuration into 3 dimensional space generated by the three eigen orthogonal vectors of length 1 in as shown in **Figure 8**. We look at the projection of the input data when we are in class 1, class 2 and class 3. As it is shown, this projection gives a weak view and thus taking h=3 will not react well to input with different classes as they are not completely separated in the Figure. As we can see in the figure there is no visible separation which shows that higher dimensions are need to completely separate the classes in this case.

2.3 <u>Estimate one larger plausible value hL for the size h ,</u>

To get another higher plausible value for hL, we apply PCA analysis to the data and compute the smallest number U_j of eigenvalues corresponding to the classes and also preserves 95% of the variance. **Figures 9, 10, 11** demonstrate the analysis for this estimation corresponding to each of the classes similar what was done previous for h=11. The result shows that:

$$hL = U_1 + U_2 + U_3 = 12 + 11 + 11 = 34 \quad (3)$$

Hence, we will need 34 neurons for automatic learning.




```
#Smallest Number h95
compare1=sum(L1)*0.95
s1=0
count1=0
for i in L1:
    count1=count1+1
    s1=s1+i
   # print(i)
    if s1>compare1:
       # print(count1)
        break

h95=count1
print("The smallest number h95 is: " + str(h95))
```

```
The smallest number h95 is: 11
```

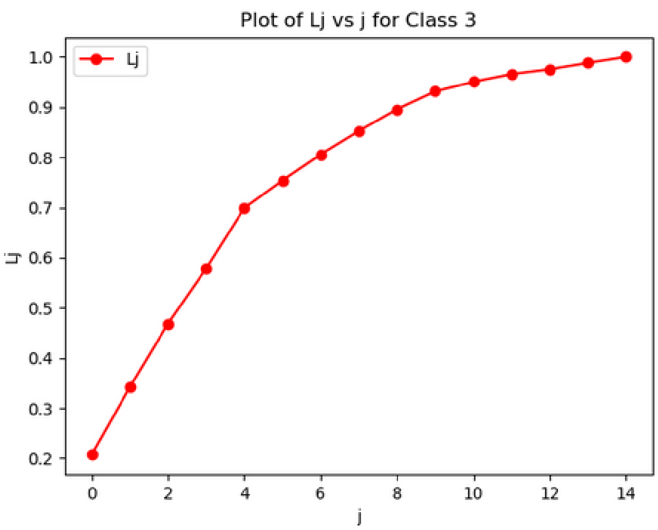
Figure 9: Li plots and computed R95 for Class 1


```
#Smallest Value
compareD1=sum(LD1)*0.95
sD1=0
countD1=0
L1=sorted(eig_valsD1,reverse=True, )
for i in LD1:
    countD1=countD1+1
    sD1=sD1+i
    #print(i)
    if sD1>compareD1:
        #print(countD1)
        break

U1=countD1
print("The smallest number is: " + str(U1))
```

```
The smallest number is: 12
```

Figure 10 : Li plots and computed R95 for Class 2




```
compareD3=sum(LD3)*0.95
sD3=0
countD3=0
for i in LD3:
    countD3=countD3+1
    sD3=sD3+i
   # print(i)
    if sD3>compareD3:
       # print(countD3)
        break

U3=countD3
U3
print("The smallest number is: " + str(U3))
```

```
The smallest number is: 11
```

Figure 11: Li plots and computed R95 for Class 3

3.0 PART 3 AND 4

3.1 <u>Automatic Learning and Performance Analysis</u>

TensorFlow was used to implement the automatic learning in python. A batch size of 1000 with 1000 epochs were selected (**Table 5**). To implement average cross entropy (avCRE) we use ***softmax turn logits*** (numeric output of the last linear layer) which turn the MLP output into probabilities by taking the exponents of each output and then normalize each number by the sum of those exponents so the entire output vector adds up to one. The maximum of this probabilities in the layer gives the true class of the case.

Cross entropy loss is the loss function. The function takes in two probability distributions p(x) and q(x), where p(x) is the true probability distribution and q(x) is the estimated probability distribution. It measures how far is the predicted distribution from the true distribution.

```
tf.nn.softmax_cross_entropy_with_logits(labels=Y,logits=logits)
```

The function computes cross entropy after applying softmax function, where ‘Y’ is the vector containing true labels and logits are the MLP output result before softmax is applied.

We used gradient descent optimizer for minimizing the loss function:

optimizer=

tf.train.GradientDescentOptimizer(learning_rate=learning_rate).minimize(loss)

For the learning we chose Batch size =1000, epochs=1000, gradient descent step size **ε(n)=0.9**, **the** learning stops once we pass through all the batches and epochs, we print the loss values to the screen for monitoring the learning quality and we do random normal initialization of weights.

```
weights={
         'h': tf.Variable(tf.random_normal([n_input,n_hidden])),
         'out':tf.Variable(tf.random_normal([n_hidden,n_classes]))
         }

biases={
         'b':tf.Variable(tf.random_normal([n_hidden])),
         'out':tf.Variable(tf.random_normal([n_classes]))
         }
```

For this MLP, we use the gradient descent algorithm with an initial learning rate of 0.01. To lower the learning rate as the learning proceeds the following function applies an exponential decay function. It requires an initial global_step value=0 to compute the decayed learning rate for each batch.

Table 5: Proportion of cases

ID	Parameters	Value
1	Batch size	1000
2	Epoch	1000
3	Number of hidden neurons	11 and 34

3.2 Learning for hidden layer size $h_L = 11$

Figure 12 shows the batch size average cross-entropy error. We can observe that the BACRE has an extreme negative slope for the first 200 batches with less oscillations then for batches after that, the oscillations increase rapidly along with a simultaneous decrease in the slope. The slope starts to stabilize after roughly 2000 batches with noise. **Figure 13** shows the batch size average accuracy estimated as the percentage of correct prediction. The observation is similar to **Figure 12** but in opposite direction and with a heavy noise and oscillations.

For **Figure 14, t**he norm of the weights decreases sharply for the first 100 batches with very little noise. Then the norm decreases slowly with some noise until the first 1000 batches with peaks. Oscillations are observed until 3500 batches but after that the change in norm of weights starts to stabilise, also the noise smoothens.

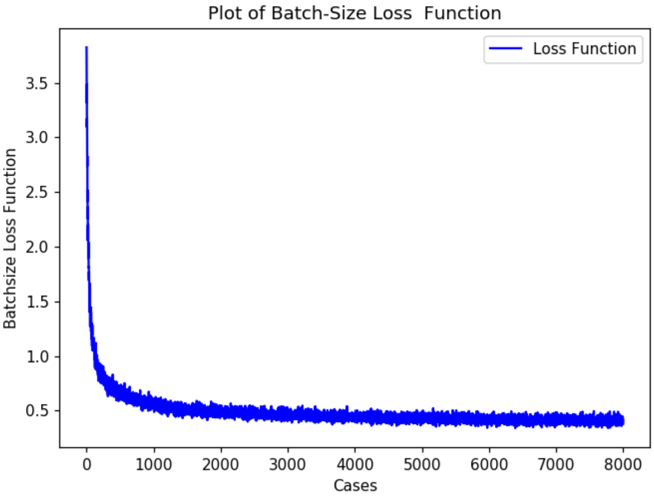


Figure 12: Plots of baCREn

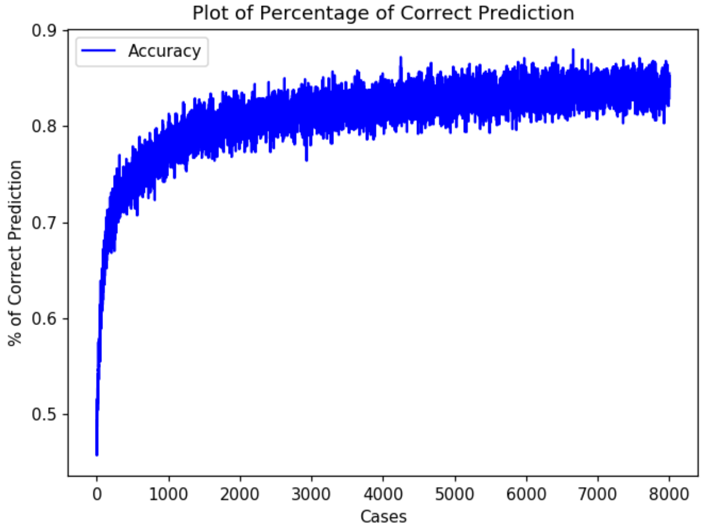


Figure 13: Plots of batch size accuracy

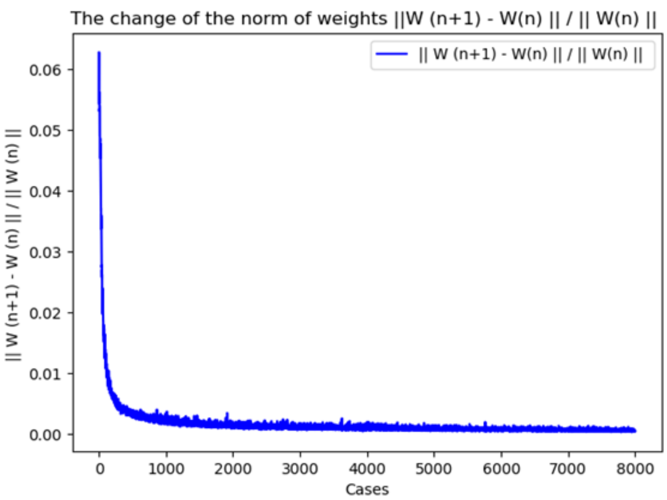


Figure 14: Plots of || W(n+1)- W(n)|| / ||Wn||

Figure 15, shows the plot of the norm of the gradient vector over the square root of D. |(|G_n |)|/d has a sharp decrease in the slope for the first 100 batches with little noise. After the 100 batches the noise increases substantially starting after the first 1000 batches. The noise is apparent but a constant value is achieved for |(|G_n |)|/d.

Figure 16 shows the plot of performance for both the training set and test sets. The performance of the training set was a bit higher than the test set which is expected. Also, the results tend to show the stability of our MLP has the both the training and test set gives almost similar result. This is verified by **Figure 17** which compare the loss function for both. We can see that the error for the training set after it stabilizes was lower than the test set. Here the performance starts to stabilise after roughly 30th iteration or 4th epoch, we can consider the the best epoch $m^* = 4$.

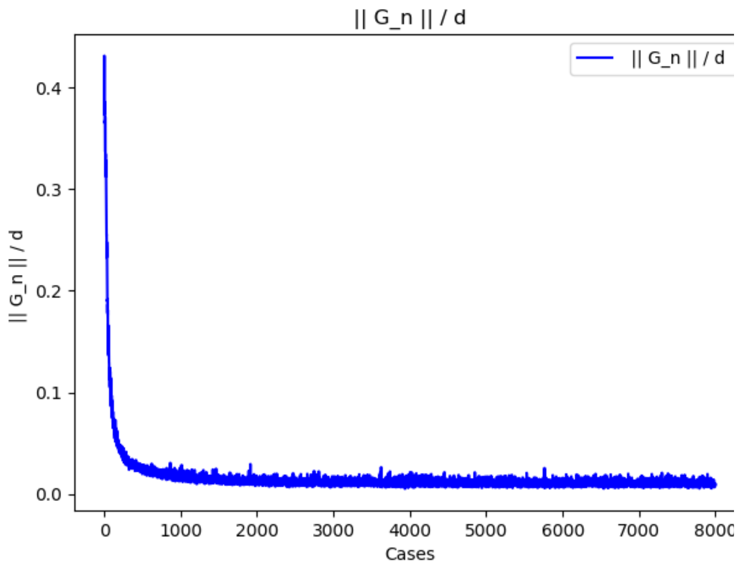
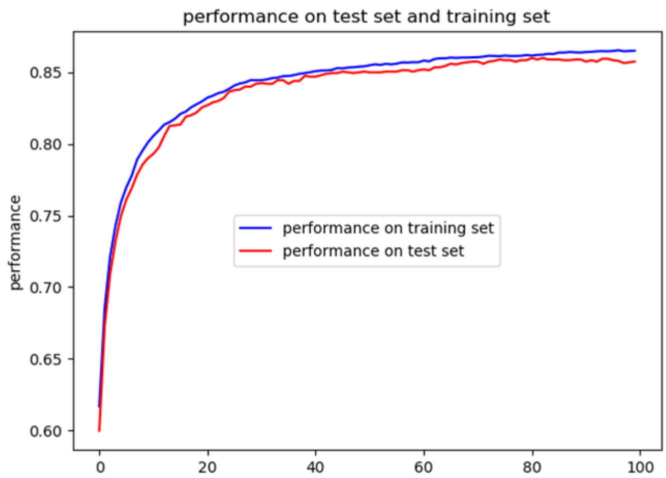


Figure 15: Plots of ||Gn|| / d



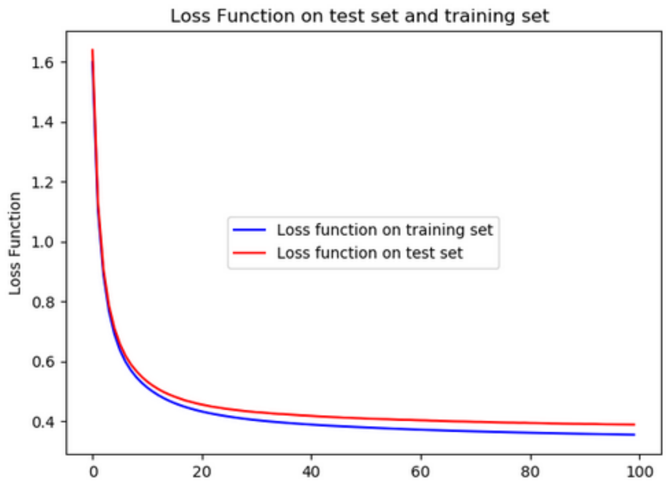


Figure 16: Plots of performance for trsining and test set

Figure 17: Plots of loss function for training and test sets

3.2.1 ***Confusion Matrix***

```
#Confusion Matrix for the Training Set
vv = np.array([sum(aycom[0,:]),sum(aycom[1,:]),sum(aycom[2,:])])
np.around((aycom.T/vv).T, decimals=3)
```

```
array([[0.825, 0.104, 0.071],
       [0.095, 0.864, 0.041],
       [0.084, 0.01 , 0.907]])
```

```
#Confusion Matrix for the Test Set
vvt = np.array([sum(axcom[0,:]),sum(axcom[1,:]),sum(axcom[2,:])])
np.around((axcom.T/vvt).T, decimals=3)
```

```
array([[0.808, 0.12 , 0.072],
       [0.115, 0.85 , 0.034],
       [0.07 , 0.011, 0.919]])
```

Table 6: Summary of performance

Data	Data	Correct Classification	Incorrect Classification
Training Set	Class 1	82.5%	17.5%
	Class 2	86.4%	13.6 %
	Class 3	90.7%	9.3 %
Test Set	Class 1	80.8%	19.2 %
	Class 2	85%	15 %
	Class 3	91.9%	8.1 %

Table 6 shows the percentage of correct classification for the classes in both the training and test sets. Apparently, the MLP does a good job in correctly classifying all the three classes. Training set performed better for (which is expected) class 1 and class 2. However, it appears the MLP does a better job in classifying class 3 for the test set than the training set.

3.3 Learning for hidden layer size $h_L = 34$

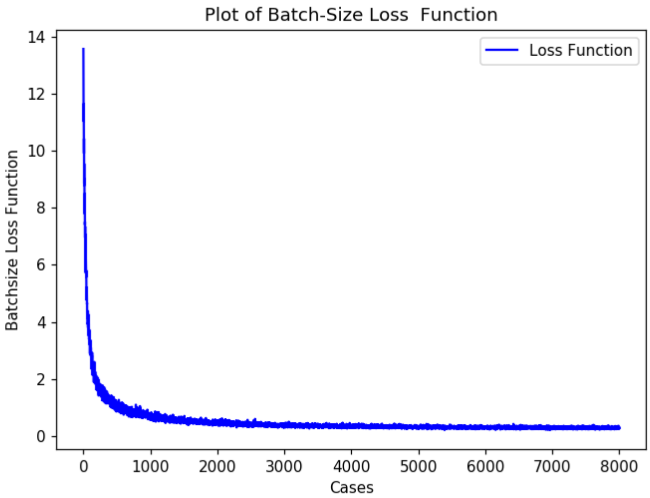


Figure 18: Plots of baCREn.

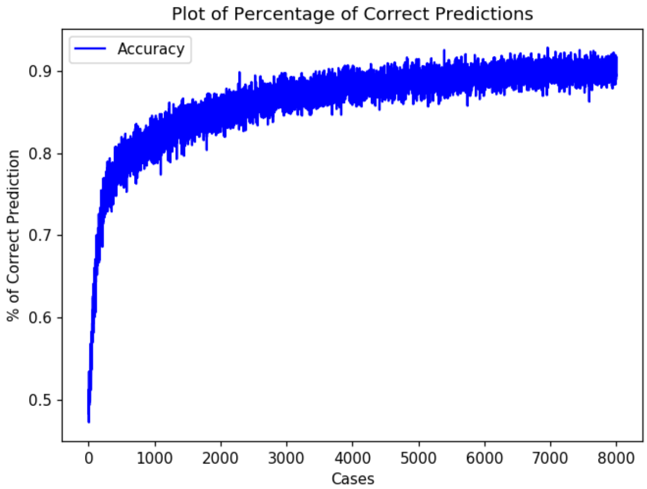


Figure 19: Plots of batch size accuracy.

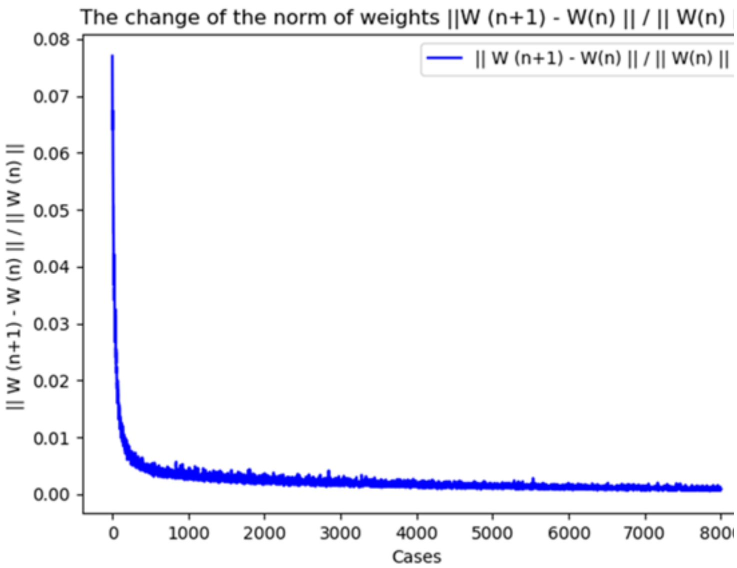


Figure 20: Plots of || W(n+1)- W(n)|| / ||Wn||

For **Figure 18**, it can be observed that the BACRE has an extreme negative slope for the first 500 batches but has less oscillations then for batches after that, the slope and oscillations start to decrease. The slope starts to stabilize after roughly 2000 batches with noise. **Figure 19** shows the batch size average accuracy estimated as the percentage of correct prediction. The observation is similar to **Figure 18** but in opposite direction and with a heavy noise and oscillations.

For **Figure 20, t**he norm of the weights decreases sharply for the first 100 batches with very little noise. Then the norm decreases slowly with some noise until the first 1000 batches. Oscillations are observed until 3500 batches but after that the change in norm of weights starts to stabilise, also the noise smoothens.

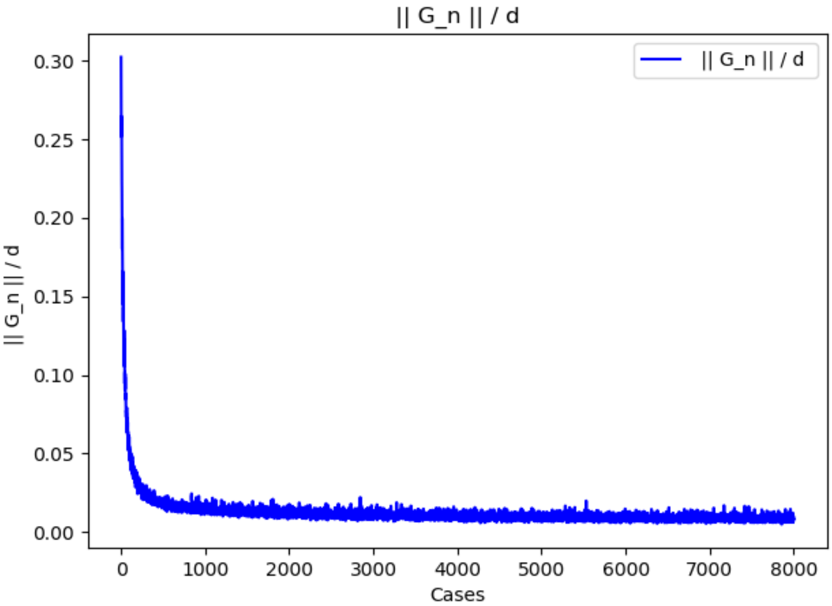


Figure 21: Plots of ||Gn|| / d

Figure 21, $||G_n||/d$ has a sharp decrease in the slope for the first 100 batches with little noise. After the 100 batches the noise increases substantially starting after the first 1000 batches. Further $||G_n||/d$stabilises after that but continues to have noise.

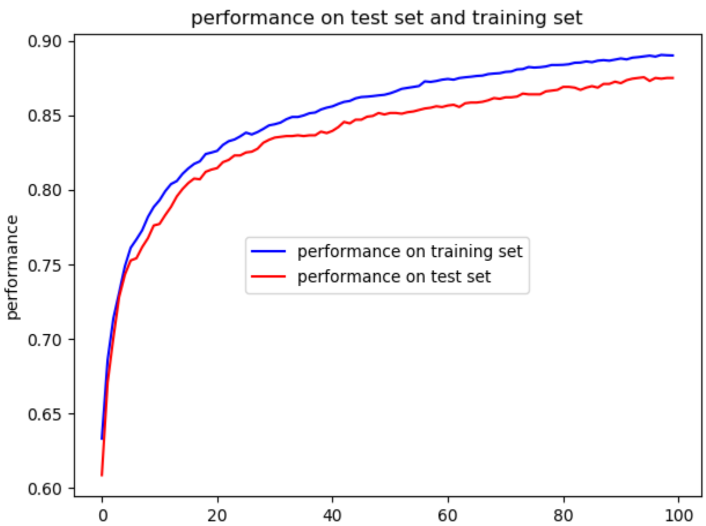


Figure 22: Plots of performance for trsining and test set

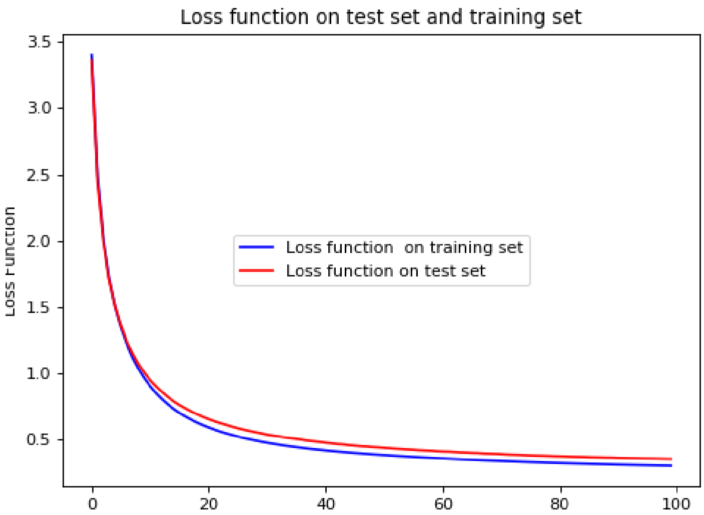


Figure 23: Plots of loss function for training and test sets

Figure 22 shows the plot of performance for both the training set and test sets as done for h=11 while **Figure 23** compare the loss function. There is a significant difference between the performance on the training and test set. When compared with the performance of h=11, h=34 appear to perform better for both the training and test sets.

3.4 *Confusion Matrix*

```
#Confusion Matrix for the Training Set
vvtr = np.array([sum(ay2com[0,:]),sum(ay2com[1,:]),sum(ay2com[2,:])])
np.around((ay2com.T/vvtr).T, decimals=3)
```

```
array([[0.845, 0.097, 0.059],
       [0.079, 0.901, 0.02 ],
       [0.059, 0.016, 0.926]])
```

```
#Confusion Matrix for the Test Set
vvte1 = np.array([sum(ax2com[0,:]),sum(ax2com[1,:]),sum(ax2com[2,:])])
np.around((ax2com.T/vvte1).T, decimals=3)
```

```
array([[0.829, 0.108, 0.064],
       [0.097, 0.874, 0.029],
       [0.05 , 0.023, 0.927]])
```

Table 7: Summary of performance

Data	Data	Correct Classification	Incorrect Classification
Training Set	Class 1	84.5 %	15.5%
	Class 2	90.1 %	9.9 %
	Class 3	92.6 %	7.4 %
Test Set	Class 1	82.9 %	17.1 %
	Class 2	87.4 %	12.6 %
	Class 3	92.7 %	7.3 %

Table 7 shows the percentage of correct classification for the classes in both the training and test sets. The MLP performed better on the training set compared to the test set. In general, the performance of MLP for classification of the input appears to be a lot better when the number of neurons in the hidden layer is 34 compared to when it is 11. Consequently, we fixed our hidden layer neuron to be 34.

5.0 PART 5

5.1 Impact of various learning options

5.1.1 *Batch Size*

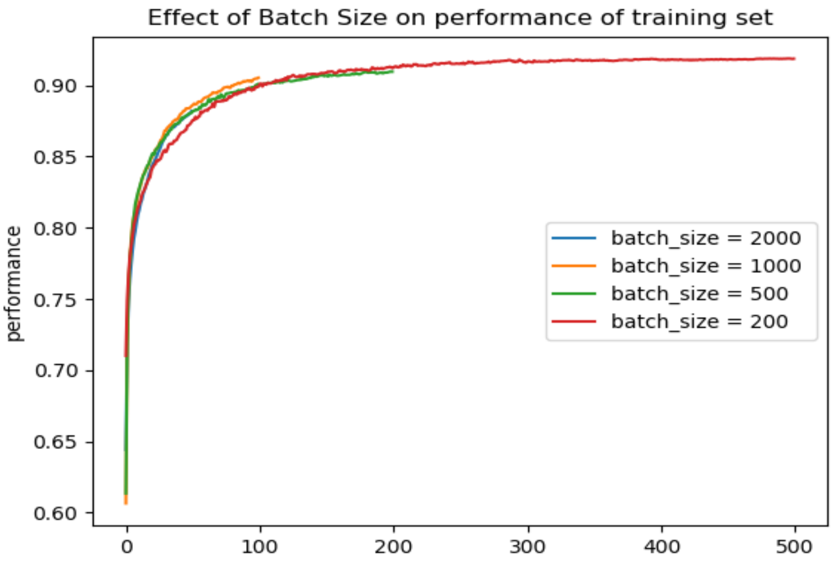


Figure 24: Effect of Batch Size on performance of training set

We tested the network with four batch sizes (2000, 1000, 500, 200) with 1000 epochs for each. We can easily see that the performance is almost the same for all of them. Hence, it is better to use a larger batch size like 1000 or 2000 for saving computing resources.

5.1.2 *Weight Initialization*

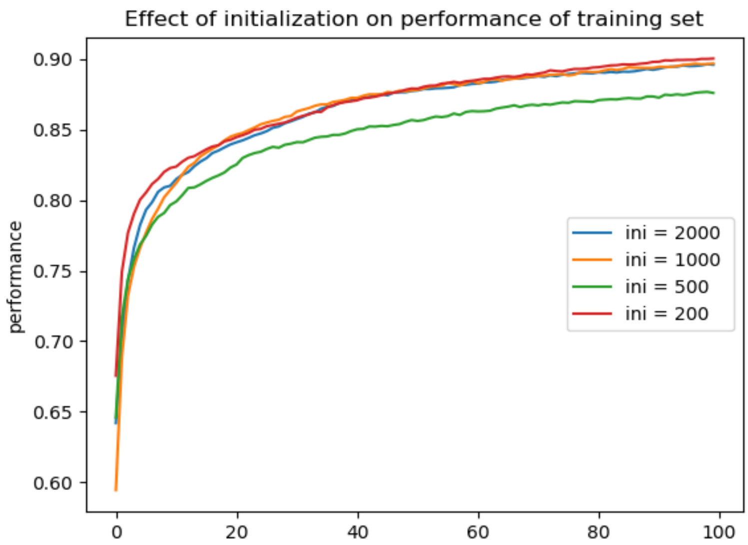


Figure 25: Effect of initialization on performance of training set

We initialised the weights and biases randomly by random seeding of 2000, 1000, 500 and 200 for a random normal distribution. All of the seedlings had close results but seeding

=200 had a different result. This show that there is a possibility of having a different performance if the initialization of weights and biases are done differently.

5.1.3 *Size of the hidden layer*

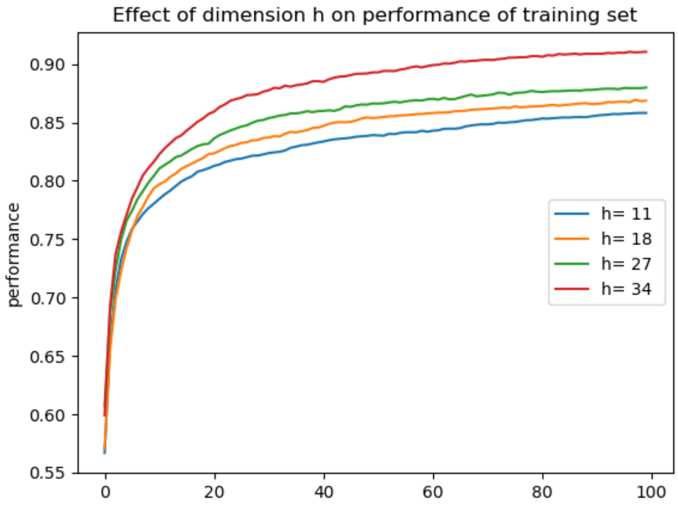


Figure 26: Effect of dimension h on performance of training set

We tested the network for four sizes of the hidden layer (11, 18, 27, 34). As expected the lesser number of neurons gave less accurate results. Also, the best performance came from the highest number of neurons which is 34.

5.1.4. *Initial Learning rate (Gradient Descent)*

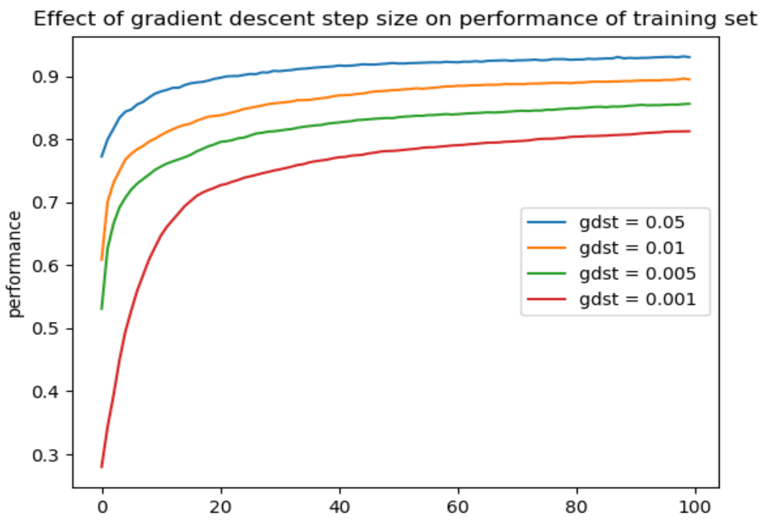


Figure 27: Effect of gradient descent step size on performance of training set

We tested the network for four initial learning rates (0.05, 0.01, 0.005, 0.001). It shows that as the gradient descent step size increases, the performance on the training set increases. The performance was best for the higher learning rates (0.05, 0.01) while the lowest to highest learning rate difference was of 20%.

6.0 PART 6

6.1 <u>Analysis of hidden layer behaviour</u>

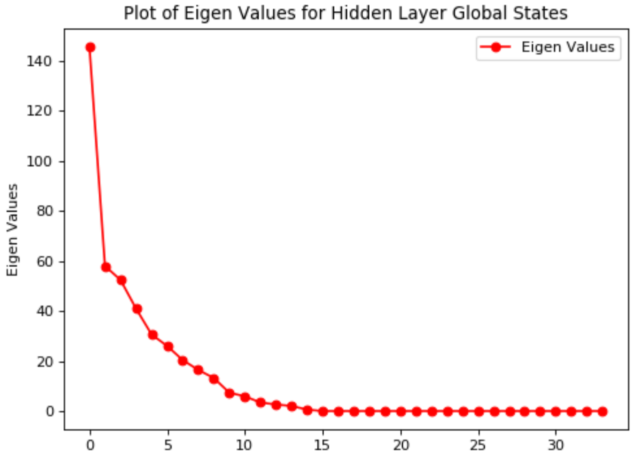


Figure 28: Eigen values for global states

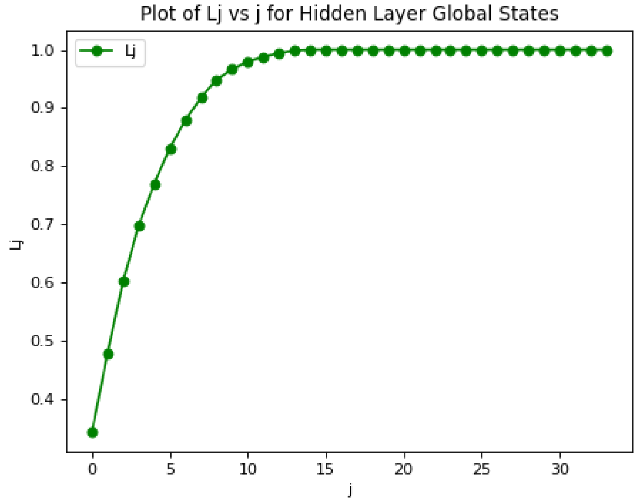


Figure 29: Li (Ri) for global states

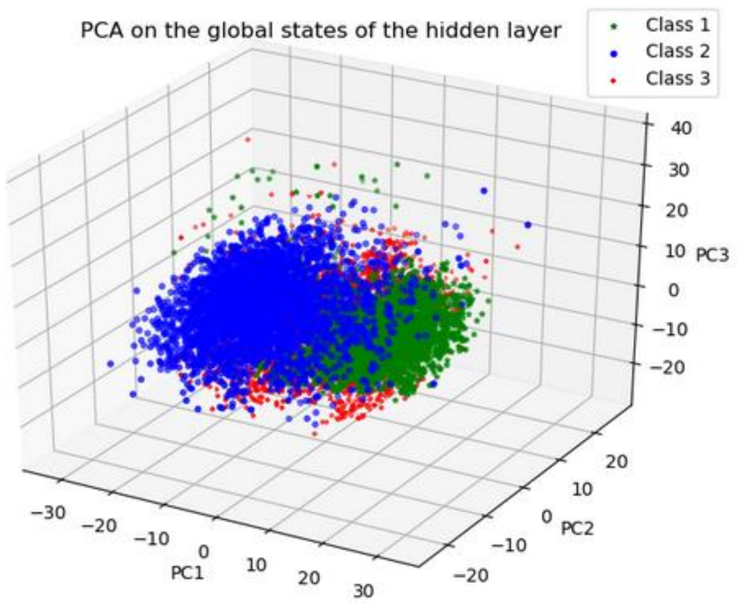


Figure 30: 3D Projections of the global states.

By retrieving the weights and biases during automatic learning, the global states of the hidden neurons was computed and PCA analysis was later performed on it.

Figure 28 shows the decreasing curves for the eigen values while **Figure 29** represents the cumulative sum of the eigen vectors which gives the proportion of the explained variance for the global states of the hidden neurons. It appears with just about 15 neurons, close to 95 % of the variance can be explained. **Figure 30** shows a 3D representation of the projection. The separation between the classes are not that great. Hence, higher dimension is needed to achieve separation. An option could be considering automatic clustering.

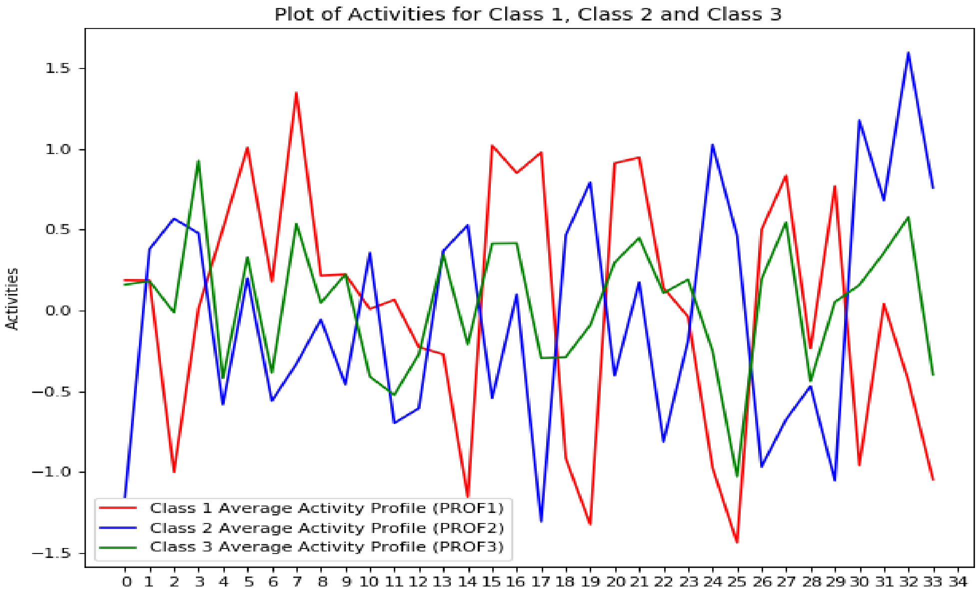


Figure 31: Profiles for Class 1 (PROF1), Class 2 (PROF2) and Class 3 (PROF3).

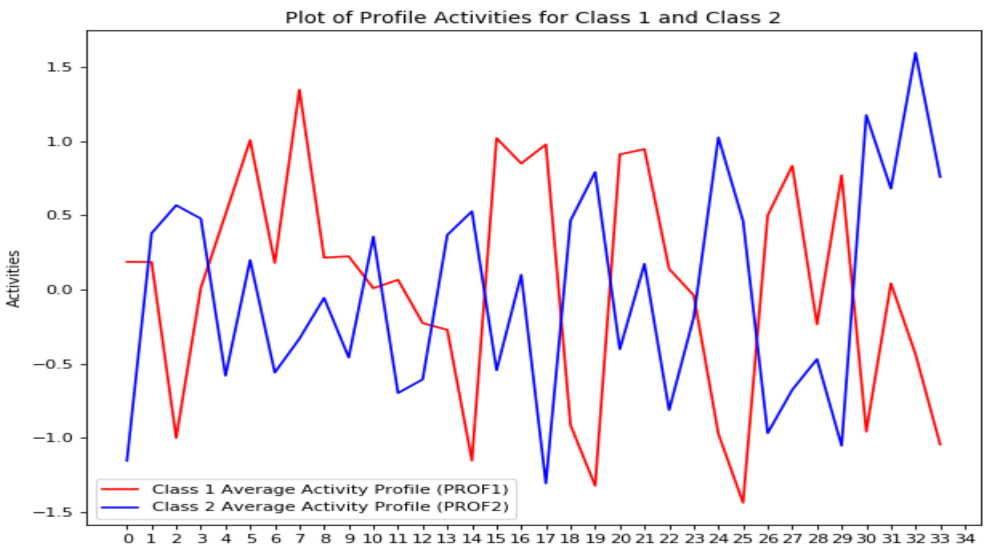


Figure 32: Profiles for Class 1 (PROF1) and Class 2 (PROF2).

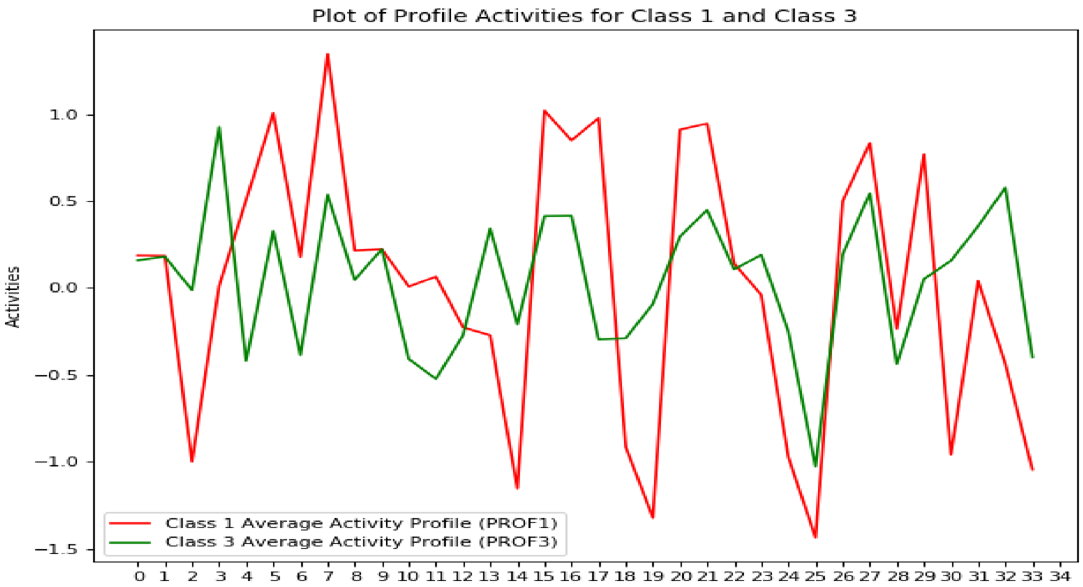


Figure 33: Profiles for Class 1 (PROF1) and Class 3 (PROF3).

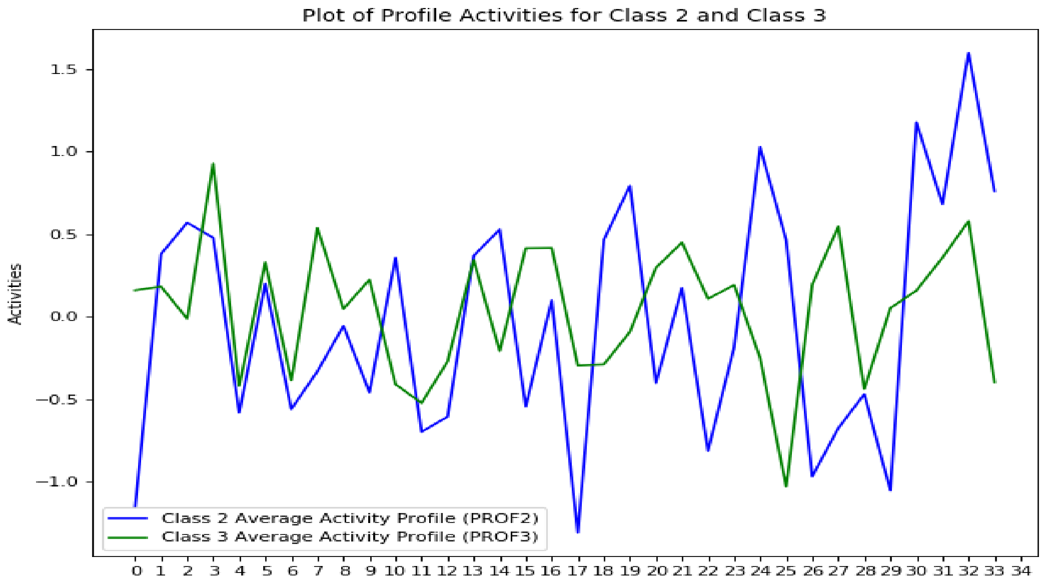


Figure 34: Profiles for Class 2 (PROF2) and Class 3 (PROF3)

To find the lazy and active neurons which are very active in the hidden layer, one way of doing this is to construct the profile activities. **Figure 31** shows the average activities for all the three classes. It appears most of the neurons do a great job in differentiating between the classes.

Figure 32 shows the profiles for class 1 and 2. The best differentiation is achieved for neurons 0, 2, 14, 15, 17, 19, 25, 26, 30, 33. However, for **Figure 33,** the best differentiation for class 1 and 3 achieved for 2, 7, 14, 19, 30. For Class 2 and 3, **Figure 34** shows that the best differentiation is achieved for 0, 17, 19, 22, 24, 25, 26, 27.

In [3]:
```
import numpy as np
import tensorflow as tf
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
from numpy import linalg as LA
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from mpl_toolkits import mplot3d
from sklearn.utils import shuffle
import random
```

In [4]:
```
#Import and shuffle the data to achieve randomness
data1 = shuffle(pd.read_csv("C:/Users/jamiu/OneDrive/Documents/Summarized_Dat
a.csv").iloc[:,1:17]).reset_index(drop=True)
data1.head(5)
```

Out[4]:

	X0	Y0	Z0	X1	Y1	Z1	X2	Y2	
0	100.953911	55.961128	-64.827305	55.895537	162.282295	-11.235967	57.637481	96.730735	-
1	113.566147	10.119615	-80.509512	90.442881	5.216347	-78.614536	65.451932	-11.751467	-
2	82.991675	37.717783	-48.641430	71.087686	61.890004	-53.845487	60.857715	10.064146	-
3	74.306212	56.891575	-65.768832	60.768323	93.783658	-24.968925	80.659233	82.367063	-
4	13.762436	77.700770	-47.506657	72.524665	64.100462	-87.839070	35.675513	72.891603	-

In [5]:
```
#Get the input cases and output (target class)
y_output = data1['Class']
x_input = data1.iloc[:,0:15]
```

In [6]:
```
#Get the classes for each input variables
D1= data1[data1['Class']==1].iloc[:,0:15]
D2= data1[data1['Class']==2].iloc[:,0:15]
D3= data1[data1['Class']==3].iloc[:,0:15]
print ("Class 1 : "+ str(D1.shape[0]),"Class 2: "+ str(D2.shape[0]), "Class 3:
"+str(D3.shape[0]) )
```

```
Class 1 : 3334 Class 2: 3334 Class 3: 3334
```

In [7]:
```
#convert the output to binary code
yy={}
for i in range(10002):

    if y_output[i]==1:
      yy[i]=[0,0,1]

    elif y_output[i]==2:
        yy[i]=[0,1,0]
    else:
        yy[i]=[1,0,0]
yyy=pd.DataFrame.from_dict(data=yy,orient='index')
```

PART 2 : Selecting 2 tentative sizes (h) for the hidden layer by PCA Analysis

In [8]:
```
#Convert the data to training and test set
scaler=preprocessing.StandardScaler()
scaled_data=pd.DataFrame(scaler.fit_transform(x_input))
data_train0,data_test0,y_train0,y_test0= train_test_split(scaled_data,y_output
,test_size=0.2, random_state=100)
data_train,data_test,y_train,y_test= train_test_split(scaled_data,yyy,test_siz
e=0.2, random_state=100)
```

In [9]:
```
D1tr=y_train0[y_train0[:,]==1]
D2tr=y_train0[y_train0[:,]==2]
D3tr=y_train0[y_train0[:,]==3]
D1te=y_test0[y_test0[:,]==1]
D2te=y_test0[y_test0[:,]==2]
D3te=y_test0[y_test0[:,]==3]
```

In [10]:
```
print ("Class 1 (training set): "+ str(D1tr.shape[0]),"Class 2 (training set):
"+ str(D2tr.shape[0]), "Class 3 (training set): "+str(D3tr.shape[0]) )
print ("Class 1 (test set): "+ str(D1te.shape[0]),"Class 2 (test set): "+ str(
D2te.shape[0]), "Class 3 (test set): "+str(D3te.shape[0]) )
```

```
Class 1 (training set): 2663 Class 2 (training set): 2668 Class 3 (training s
et): 2670
Class 1 (test set): 671 Class 2 (test set): 666 Class 3 (test set): 664
```

In [11]:
```
data_train.head(5)
```

Out[11]:

	0	1	2	3	4	5	6	7	
4052	-1.336146	-0.257159	-0.643712	0.583427	-1.197125	-1.588397	-0.021294	-0.318572	-1.1
8692	-2.010570	-0.274503	-0.325620	0.506873	-1.271424	-1.632140	0.348953	-0.525397	-1.5
2924	-0.432939	0.300219	1.015603	-2.070615	0.446521	1.117696	0.546144	-0.885026	-0.2
6898	-0.625425	0.023007	1.217695	0.748327	-0.130835	0.685058	0.100970	0.122157	0.9
6956	1.040640	1.217523	1.038581	0.775127	0.133373	0.641581	1.460384	-1.090782	-1.7

In [12]:
```
#Standardized the data
data_train_std=StandardScaler().fit_transform(data_train)
#Covariance matrix
cov_mat1=np.corrcoef(np.transpose(data_train_std))
pd.DataFrame(cov_mat1).head(4)
```

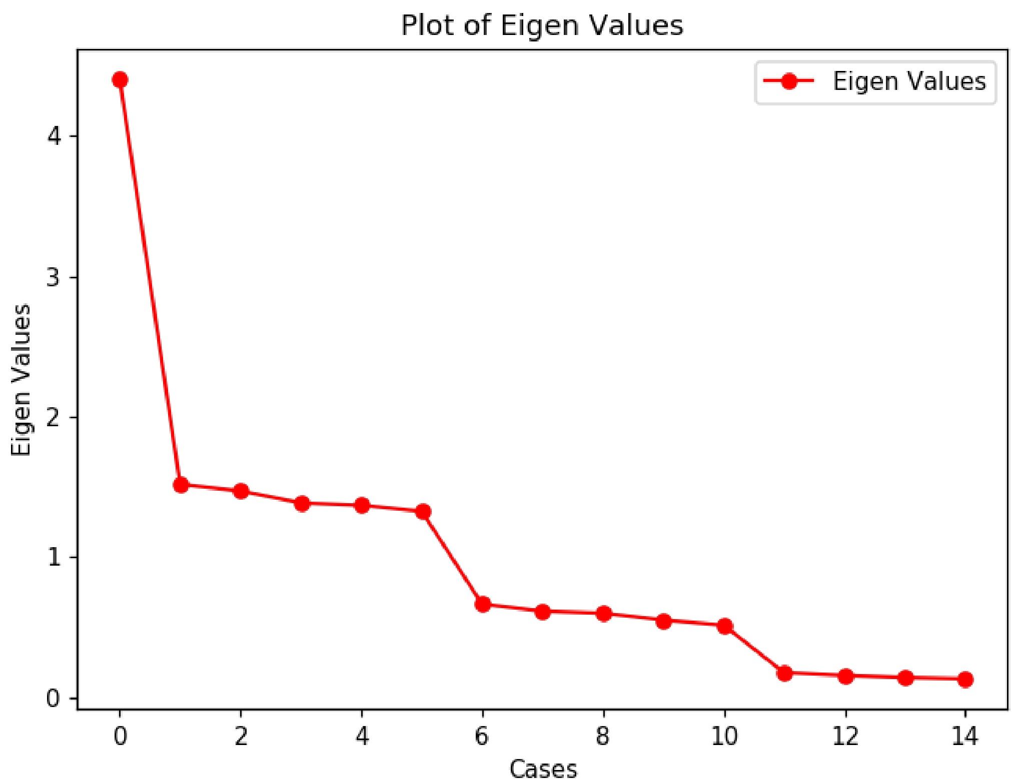
Out[12]:

	0	1	2	3	4	5	6	7	8
0	1.000000	-0.224676	-0.258118	0.127463	0.009750	0.040116	0.099500	0.028067	0.03727
1	-0.224676	1.000000	0.652873	0.003080	0.172088	0.272436	-0.017443	0.141050	0.24816
2	-0.258118	0.652873	1.000000	0.004790	0.273581	0.562019	-0.015688	0.238826	0.53274
3	0.127463	0.003080	0.004790	1.000000	-0.255218	-0.320866	0.126369	-0.011332	-0.02783

In [13]:
```
eig_vals1,eig_vecs1=np.linalg.eig(cov_mat1) #Get the Eigen values and Eigen ve
ctors
print((eig_vals1)) #print the eigen values
```

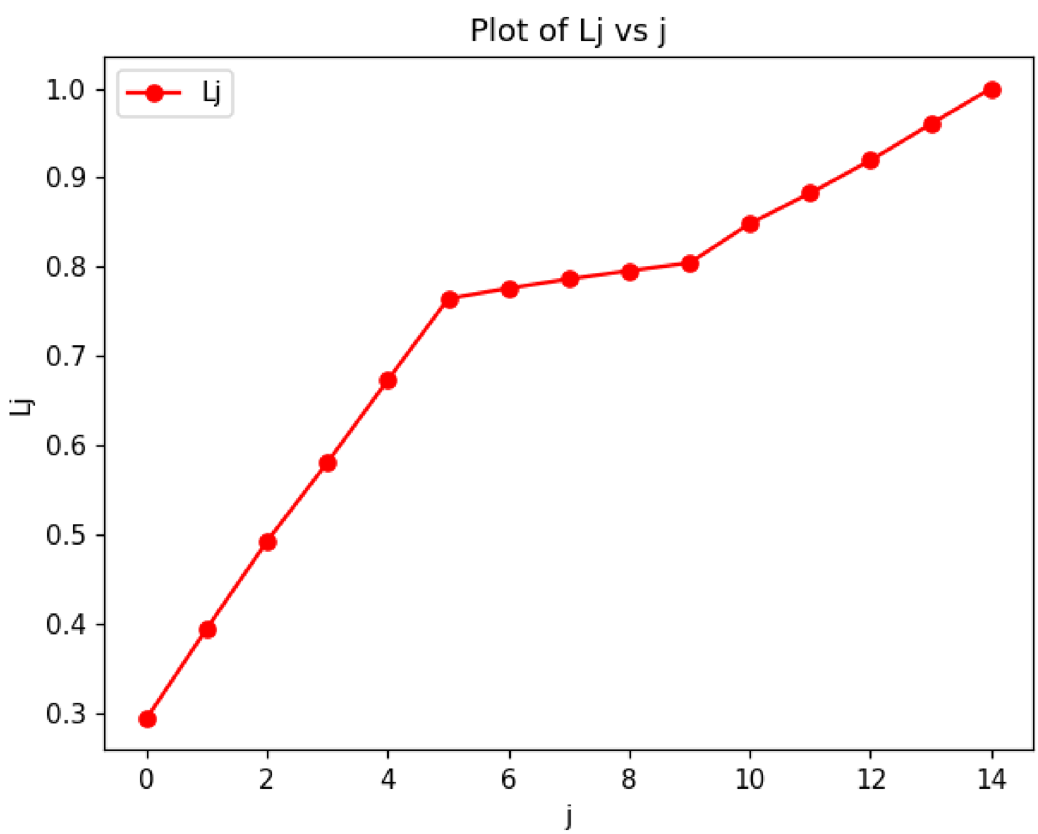
```
[4.40388059 1.51622632 1.46907783 1.3241208  1.38421915 1.36709768
 0.17477514 0.15475699 0.13959492 0.1308936  0.66249449 0.51272559
 0.54945309 0.61281647 0.59786736]
```

In [14]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.figure(1)
L1=sorted(eig_vals1,reverse=True, )
plt.plot(L1,  marker='o', label='Eigen Values', color='r')
plt.ylabel('Eigen Values')
plt.xlabel('Cases')
plt.title('Plot of Eigen Values')
plt.legend()
```



Out[14]: <matplotlib.legend.Legend at 0x295b8f382e8>

In [15]:
```
%matplotlib notebook
dgvv1=eig_vals1
da1vv1=[]
for i in range(15):
    if i ==0:
        Ri = dgvv1[i]
    else:
        Ri+=dgvv1[i]
    da1vv1.append(Ri)
%matplotlib notebook
plt.plot(da1vv1/(sum(dgvv1)),  marker='o', label='Lj', color='r')
plt.ylabel('Lj')
plt.xlabel('j')
plt.title('Plot of Lj vs j')
plt.legend()
```



Out[15]: <matplotlib.legend.Legend at 0x295b9339c18>

In [16]:
```
L1=sorted(eig_vals1,reverse=True)
```

In [17]:
```
#Smallest Number h95
compare1=sum(L1)*0.95
s1=0
count1=0
for i in L1:
    count1=count1+1
    s1=s1+i
   # print(i)
    if s1>compare1:
       # print(count1)
        break

h95=count1
print("The smallest number h95 is: " + str(h95))
```

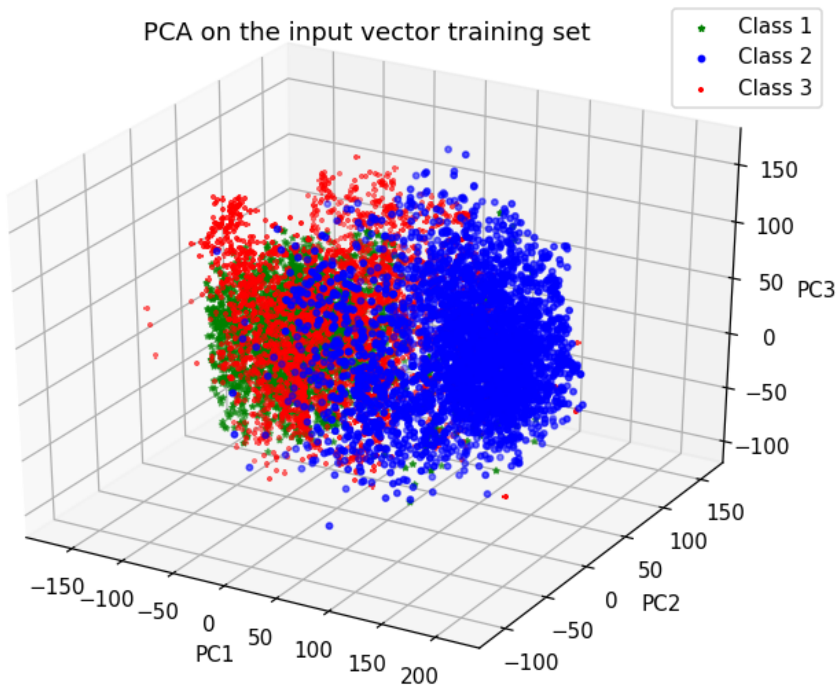
```
The smallest number h95 is: 11
```

In [18]:
```
#PCA Analysis
pca0 = PCA(n_components=3)
pca0.fit(data_train)
print(pca0.explained_variance_)
# Store results of PCA in a data frame
result1=pd.DataFrame(pca0.transform(D1), columns=['PCA%i' % i for i in range(3
)], index=D1.index)
result2=pd.DataFrame(pca0.transform(D2), columns=['PCA%i' % i for i in range(3
)], index=D2.index)
result3=pd.DataFrame(pca0.transform(D3), columns=['PCA%i' % i for i in range(3
)], index=D3.index)
print(result2)
```

```
[4.38747335 1.51888369 1.46467196]
            PCA0       PCA1       PCA2
0      83.833221  86.120268 -17.309299
6      -2.554499  44.543609  72.935766
10    137.983682 -52.237173  -2.054982
16    110.304488 -70.875316  70.047700
18     33.639885  93.931832 118.729867
...          ...        ...        ...
9985   86.049608  61.869883 -19.546658
9986  152.022960  24.740445  56.724937
9990  108.102031  67.800031   6.416309
9994  141.928073  -8.305663  31.743545
9995  151.732443  30.838453  43.504822

[3334 rows x 3 columns]
```

In [19]:
```
# Plot of Principal Components
%matplotlib notebook
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1,1,1, projection='3d')
ax.scatter(result1['PCA0'], result1['PCA1'], result1['PCA2'],s=8,marker='*', c
olor='g', label='Class 1')
ax.scatter(result2['PCA0'], result2['PCA1'], result2['PCA2'],s=8,marker='o', c
olor = 'b', label='Class 2')
ax.scatter(result3['PCA0'], result3['PCA1'], result3['PCA2'],s=8, marker='+',
color='r', label='Class 3')
ax.set_xlabel("PC1")
ax.set_ylabel("PC2")
ax.set_zlabel("PC3")
ax.legend (loc='best')
ax.set_title("PCA on the input vector training set")
```



Out[19]: `Text(0.5, 0.92, 'PCA on the input vector training set')`

Class 1

In [20]:
```
#To estimate one larger plausible value hL for the size h
#Class 1
D1_std=StandardScaler().fit_transform(D1)
pd.DataFrame(D1_std).head(4) #standardized data for class 1
```

Out[20]:

	0	1	2	3	4	5	6	7	
0	1.504661	-1.232811	0.098815	1.034310	-0.192492	-0.052171	0.651067	-2.045807	-0.75267
1	-0.814471	0.391377	0.147340	1.085777	-0.105632	-1.475132	-0.256490	0.301184	-0.12708
2	-1.583494	0.639555	1.123890	1.097321	-0.644815	-0.874277	1.532909	0.387816	-0.74888
3	0.480367	1.128298	1.819266	-1.008071	1.481760	3.001304	-0.214018	1.334375	2.45850

In [21]:
```
cov_D1=np.corrcoef(np.transpose(D1_std))
pd.DataFrame(cov_D1)#Covariance matrix for class 1
```

Out[21]:

	0	1	2	3	4	5	6	7	
0	1.000000	-0.455564	-0.453574	-0.017178	0.011144	0.023761	0.022920	0.011972	-0.011
1	-0.455564	1.000000	0.357741	-0.012848	0.139953	0.136815	-0.026042	0.108315	0.138
2	-0.453574	0.357741	1.000000	-0.052547	0.141651	0.561305	-0.101851	0.137300	0.572
3	-0.017178	-0.012848	-0.052547	1.000000	-0.442355	-0.511940	0.017063	0.028694	-0.066
4	0.011144	0.139953	0.141651	-0.442355	1.000000	0.453021	-0.016105	0.108041	0.156
5	0.023761	0.136815	0.561305	-0.511940	0.453021	1.000000	-0.075984	0.138631	0.594
6	0.022920	-0.026042	-0.101851	0.017063	-0.016105	-0.075984	1.000000	-0.442319	-0.513
7	0.011972	0.108315	0.137300	0.028694	0.108041	0.138631	-0.442319	1.000000	0.403
8	-0.011811	0.138714	0.572514	-0.066861	0.156252	0.594104	-0.513774	0.403294	1.000
9	-0.007318	-0.019898	-0.171599	0.022461	-0.007328	-0.177376	0.058685	-0.024843	-0.203
10	0.038528	0.085134	0.171948	0.005278	0.077578	0.204427	0.015116	0.066038	0.189
11	0.003827	0.121885	0.564156	-0.068645	0.161437	0.604691	-0.121925	0.164685	0.600
12	-0.048142	0.066551	-0.136837	0.030147	-0.013037	-0.151424	-0.019517	0.082763	-0.132
13	0.064318	0.043641	0.132977	0.045679	0.085708	0.144456	0.071793	-0.009619	0.115
14	-0.011670	0.092368	0.537997	-0.078728	0.169353	0.566218	-0.084713	0.107638	0.541

In [22]:
```
#EigenValues and EigenVectors
eig_valsD1,eig_vecsD1=np.linalg.eig(cov_D1)
print(eig_valsD1)
```

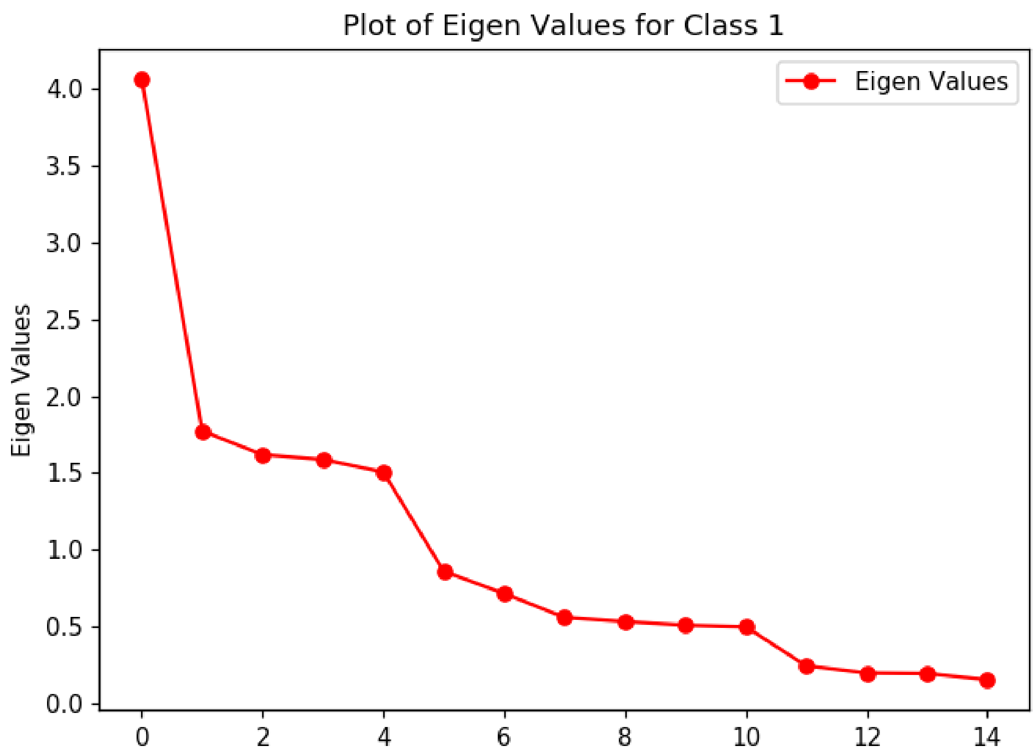
```
[4.06337566 1.77257635 1.61860994 1.58636602 1.50559822 0.85852878
 0.71390154 0.15283266 0.24326227 0.19317562 0.19666644 0.55835813
 0.53149571 0.5078201  0.49743256]
```

In [23]:
```
pd.DataFrame(eig_vecsD1)
```

Out[23]:

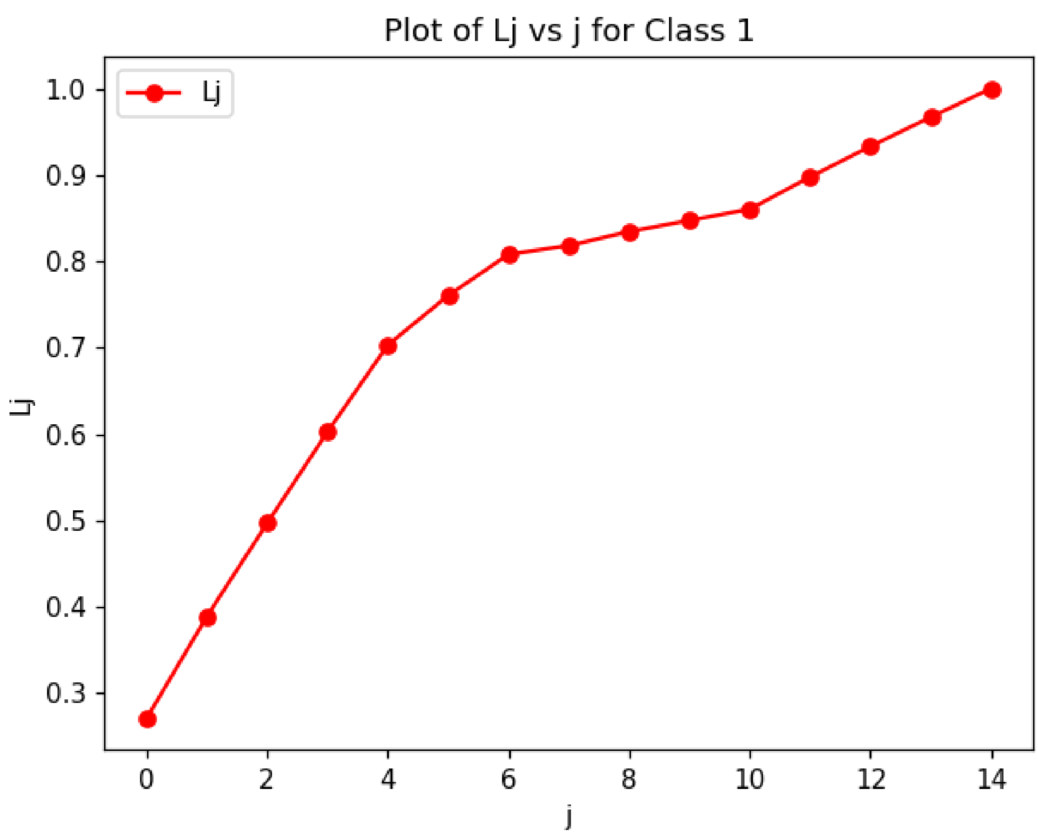
	0	1	2	3	4	5	6	7	
0	0.065825	-0.347653	0.403771	0.398800	-0.079570	0.126367	-0.307577	-0.096907	0.126
1	-0.137104	0.320966	-0.371387	-0.272185	0.085675	0.390500	0.077418	-0.007530	0.015
2	-0.371656	0.149091	-0.199813	-0.259981	0.016539	-0.237632	-0.236992	-0.140132	0.262
3	0.126661	0.105361	0.356701	-0.509136	-0.135052	0.147172	-0.359013	0.368059	0.064
4	-0.178918	-0.059867	-0.345872	0.420012	0.076889	0.376212	0.037771	-0.076418	0.042
5	-0.398099	-0.096625	-0.182457	0.261905	0.062560	-0.155044	-0.229540	0.768919	0.082
6	0.139558	-0.351309	-0.228988	-0.171543	0.429283	0.066888	-0.434734	-0.183800	0.134
7	-0.153075	0.347370	0.182489	0.173112	-0.361264	0.382140	-0.044913	0.019066	0.020
8	-0.393721	0.161188	0.146810	0.063510	-0.249540	-0.148924	-0.161439	-0.333072	0.244
9	0.207871	-0.015681	-0.361014	0.055316	-0.426637	-0.006637	-0.446040	-0.128690	-0.043
10	-0.192811	0.030986	0.293709	-0.027758	0.440403	0.388841	0.016714	0.021707	-0.042
11	-0.409531	-0.016301	0.220988	-0.046518	0.218835	-0.065102	-0.118314	-0.259806	0.067
12	0.154229	0.474299	0.013605	0.227507	0.257924	0.033244	-0.476145	-0.054569	-0.444
13	-0.128836	-0.400254	-0.077221	-0.222771	-0.264446	0.495966	-0.052679	-0.003679	-0.011
14	-0.381827	-0.270496	0.009521	-0.155729	-0.143546	-0.112197	-0.020959	-0.075931	-0.785

In [24]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.figure(1)
L1=sorted(eig_valsD1,reverse=True, )
plt.plot(L1,  marker='o', label='Eigen Values', color='r')
plt.ylabel('Eigen Values')
plt.xlabel('')
plt.title('Plot of Eigen Values for Class 1')
plt.legend()
```



Out[24]: <matplotlib.legend.Legend at 0x295b9bceb38>

In [25]:
```
%matplotlib notebook
dgvv=eig_valsD1
da1vv=[]
for i in range(15):
    if i ==0:
        Ri = dgvv[i]
    else:
        Ri+=dgvv[i]
    da1vv.append(Ri)
%matplotlib notebook
plt.plot(da1vv/(sum(dgvv)),  marker='o', label='Lj', color='r')
plt.ylabel('Lj')
plt.xlabel('j')
plt.title('Plot of Lj vs j for Class 1')
plt.legend()
```



Out[25]: <matplotlib.legend.Legend at 0x295b9fc7cc0>

In [26]:
```
LD1=sorted(eig_valsD1,reverse=True)
```

In [27]:
```
#Smallest Value
compareD1=sum(LD1)*0.95
sD1=0
countD1=0
L1=sorted(eig_valsD1,reverse=True, )
for i in LD1:
    countD1=countD1+1
    sD1=sD1+i
    #print(i)
    if sD1>compareD1:
        #print(countD1)
        break

U1=countD1
print("The smallest number is: " + str(U1))
```

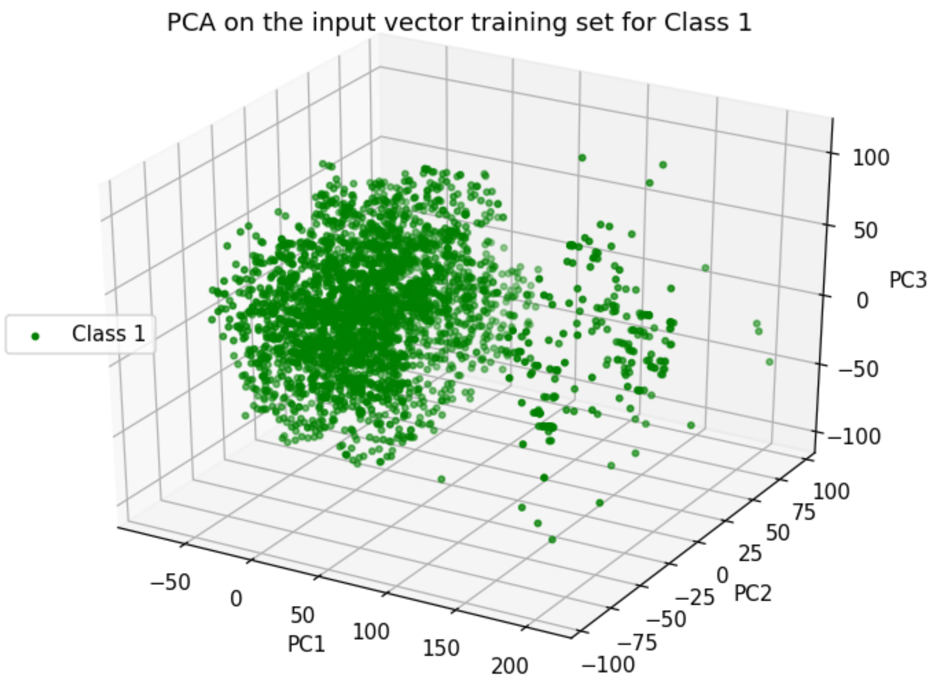
```
The smallest number is: 12
```

In [28]:
```
#PCA Analysis
pcac1 = PCA(n_components=3)
pcac1.fit(D1)
# Store results of PCA in a data frame
resultD1=pd.DataFrame(pcac1.transform(D1), columns=['PCA%i' % i for i in range
(3)], index=D1.index)
print(resultD1)
```

```
             PCA0       PCA1       PCA2
2       18.789331  79.891750 -40.874546
4      -65.934417 -46.259117  35.197098
7        1.138674  -8.412521  38.768932
17     143.491940  39.589372  23.600128
22     -26.634232  11.521728  26.692040
...           ...        ...        ...
9989   -14.663742  24.675748  50.385601
9997     3.734942 -52.193157  15.043205
9998    13.822864  -7.304827 -44.134068
9999    33.436587 -15.307020  49.457841
10000   32.836089 -40.435823 -22.075187

[3334 rows x 3 columns]
```

In [29]:
```
# Plot of Principal Components
%matplotlib notebook
fig2 = plt.figure(figsize=(8, 6))
ax2 = fig2.add_subplot(1,1,1, projection='3d')
ax2.scatter(resultD1['PCA0'], resultD1['PCA1'], resultD1['PCA2'],s=8,marker=
'o', color='g', label='Class 1')
ax2.set_xlabel("PC1")
ax2.set_ylabel("PC2")
ax2.set_zlabel("PC3")
ax2.legend (loc='center left')
ax2.set_title("PCA on the input vector training set for Class 1")
```



Out[29]: Text(0.5, 0.92, 'PCA on the input vector training set for Class 1')

Class 2

In [30]:
```python
D2_std=StandardScaler().fit_transform(D2)
pd.DataFrame(D2_std).head(4)  #standardized data for class 2
```

Out[30]:

	0	1	2	3	4	5	6	7	
0	1.547031	-1.335023	-1.985179	0.119540	1.587402	-0.276369	0.205653	-0.204236	-0.69772
1	-1.265657	-0.032247	0.140151	0.234832	-2.481356	-2.317018	-0.418473	0.098789	-0.27835
2	-0.387641	-0.060208	0.457523	0.450231	0.968825	0.401492	0.073763	1.187924	0.83408
3	-0.598171	0.384612	1.275666	0.452349	0.892797	1.308154	-0.465679	-0.414415	0.52027

In [31]:
```python
#Covariant Matrix for Clas 2
cov_D2=np.corrcoef(np.transpose(D2_std))
pd.DataFrame(cov_D2).head(4)#Covariance matrix for class 2
```

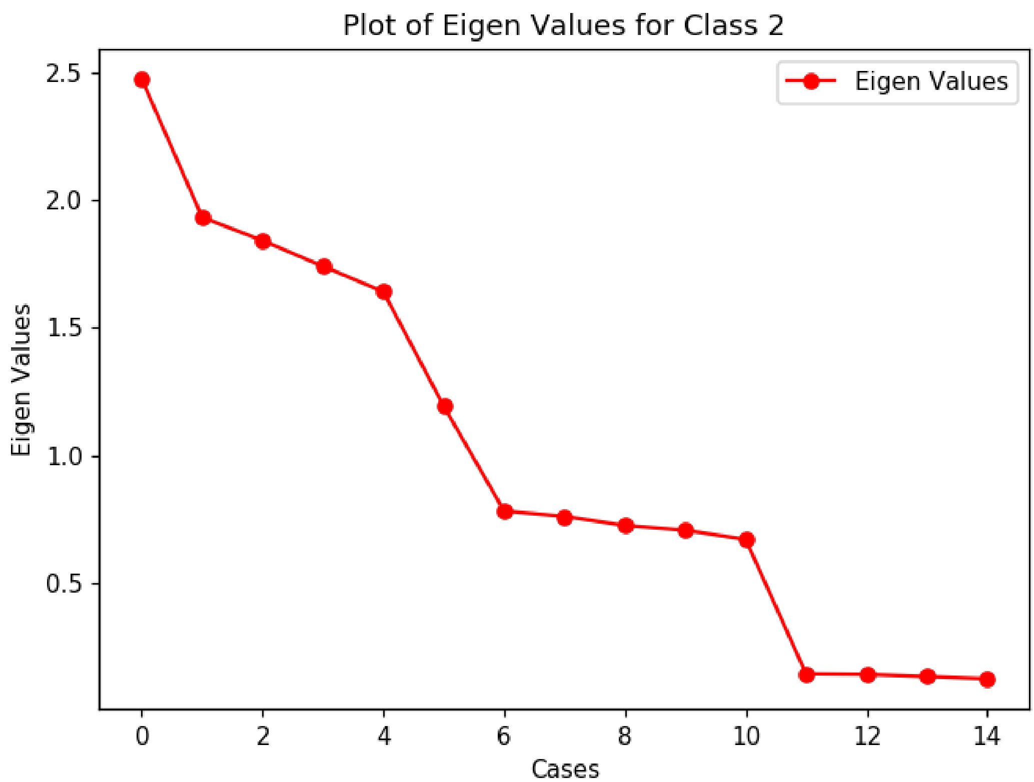
Out[31]:

	0	1	2	3	4	5	6	7	
0	1.000000	-0.188126	-0.494252	0.160735	-0.003887	0.017436	0.129569	0.005477	0.0205
1	-0.188126	1.000000	0.621127	-0.030694	0.079175	-0.009472	-0.029369	0.059603	-0.0324
2	-0.494252	0.621127	1.000000	-0.022273	0.003063	0.185475	-0.003458	-0.007210	0.1572
3	0.160735	-0.030694	-0.022273	1.000000	-0.179735	-0.463646	0.192800	-0.032609	-0.0279

In [32]:
```python
#EigenValues and EigenVectors for Class 2
eig_valsD2,eig_vecsD2=np.linalg.eig(cov_D2)
print(eig_valsD2)
```

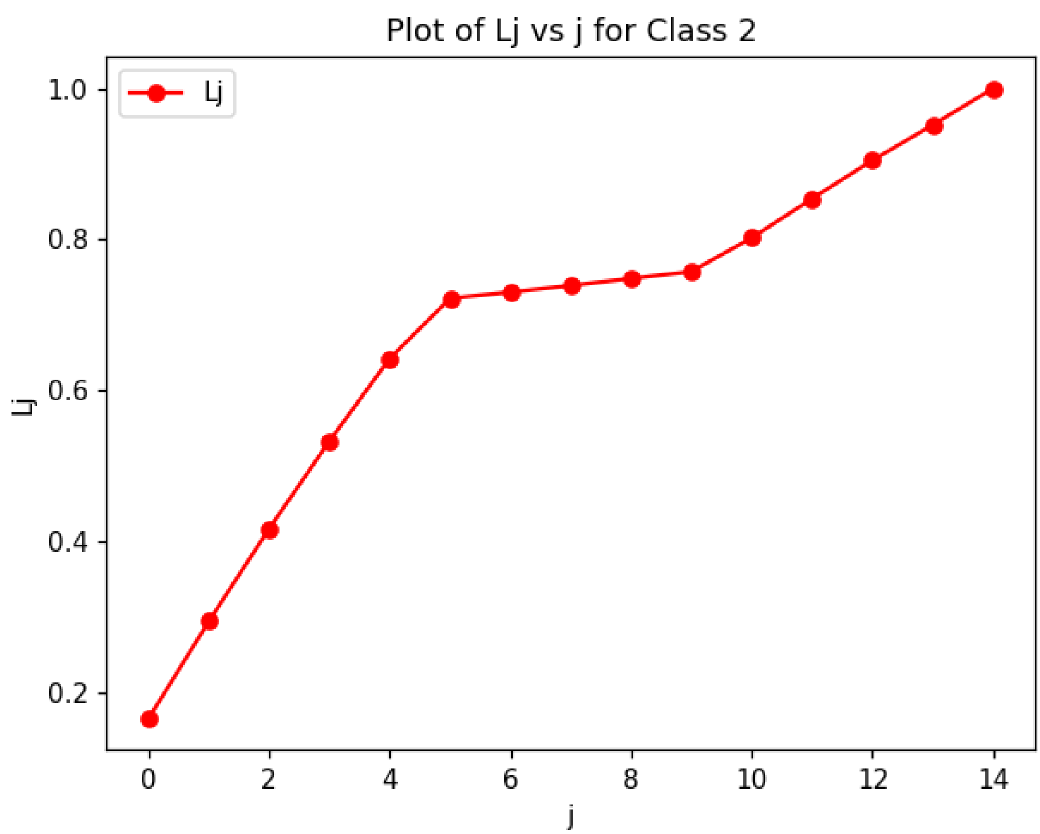
```
[2.47646514 1.93380782 1.84244204 1.74146772 1.6417704  1.19259234
 0.12197182 0.13080307 0.14092376 0.13914761 0.66929308 0.78058318
 0.75955087 0.70509272 0.72408843]
```

In [33]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.figure(1)
L1=sorted(eig_valsD2,reverse=True, )
plt.plot(L1,  marker='o', label='Eigen Values', color='r')
plt.ylabel('Eigen Values')
plt.xlabel('Cases')
plt.title('Plot of Eigen Values for Class 2')
plt.legend()
```



Out[33]: <matplotlib.legend.Legend at 0x295bb7ef1d0>

In [34]:
```
%matplotlib notebook
dgvv2=eig_valsD2
da1vv2=[]
for i in range(15):
    if i ==0:
        Ri = dgvv2[i]
    else:
        Ri+=dgvv2[i]
    da1vv2.append(Ri)
%matplotlib notebook
plt.plot(da1vv2/(sum(dgvv2)),  marker='o', label='Lj', color='r')
plt.ylabel('Lj')
plt.xlabel('j')
plt.title('Plot of Lj vs j for Class 2')
plt.legend()
```



Out[34]: <matplotlib.legend.Legend at 0x295bbba6cc0>

In [35]:
```
LD2=sorted(eig_valsD2,reverse=True)
```

In [36]:
```
compareD2=sum(LD2)*0.95
sD2=0
countD2=0
for i in LD2:
    countD2=countD2+1
    sD2=sD2+i
    #print(i)
    if sD2>compareD2:
     #   print(countD2)
        break

U2=countD2
U2
print("The smallest number is: " + str(U2))
```

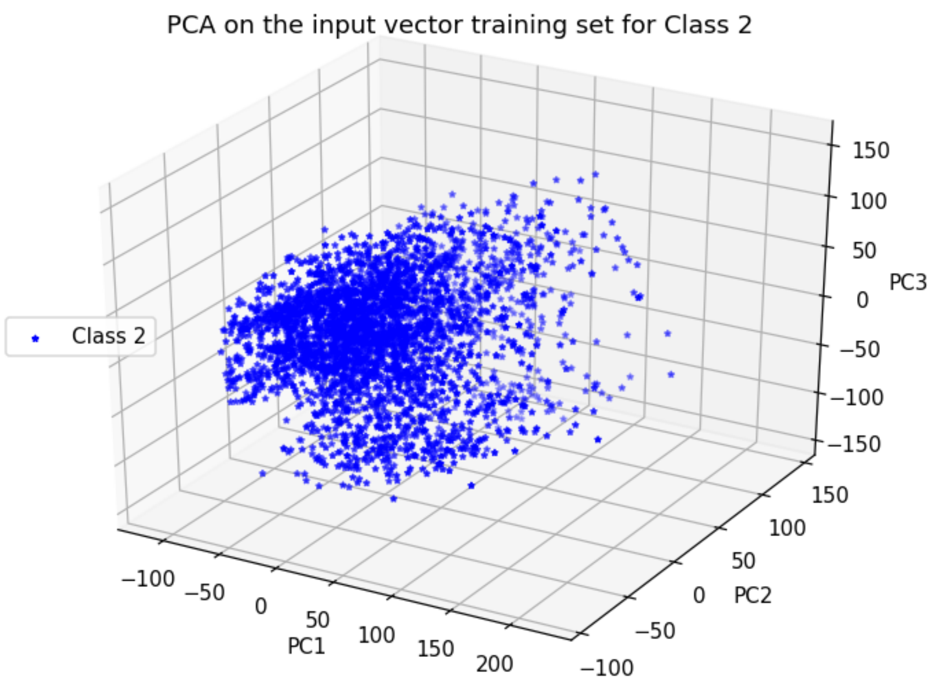
```
The smallest number is: 11
```

In [37]:
```
#PCA Analysis
pcac3 = PCA(n_components=3)
pcac3.fit(D2)
# Store results of PCA in a data frame
resultD2=pd.DataFrame(pcac3.transform(D2), columns=['PCA%i' % i for i in range
(3)], index=D2.index)
print(resultD2)
```

```
            PCA0        PCA1       PCA2
0      19.882945  -42.137204  56.613356
6     103.194411   -7.667214 -56.536947
10    -63.513354   84.398086  16.716125
16    -23.656855  109.724422 -30.871048
18     79.689567  -88.339771 -62.186070
...          ...         ...        ...
9985   28.697932    1.724991  80.274300
9986  -35.574612   -8.269584 -39.921107
9990   19.337726    9.392943  31.779461
9994  -29.283802   46.562206  12.894110
9995  -36.171361  -14.441834  11.210825

[3334 rows x 3 columns]
```

In [38]:
```
# Plot of Principal Components
fig3 = plt.figure(figsize=(8, 6))
ax3 = fig3.add_subplot(1,1,1, projection='3d')
ax3.scatter(resultD2['PCA0'], resultD2['PCA1'], resultD2['PCA2'],s=8,marker=
'*', color='b', label='Class 2')
ax3.set_xlabel("PC1")
ax3.set_ylabel("PC2")
ax3.set_zlabel("PC3")
ax3.legend (loc='center left')
ax3.set_title("PCA on the input vector training set for Class 2")
```



Out[38]:
```
Text(0.5, 0.92, 'PCA on the input vector training set for Class 2')
```

Class 3

In [39]:

```
D3_std=StandardScaler().fit_transform(D3)
pd.DataFrame(D3_std).head(4)  #standardized data for class 3
```

Out[39]:

	0	1	2	3	4	5	6	7	
0	1.600146	-1.662380	-1.605408	0.927640	-1.576025	-1.472627	0.172335	-1.777729	-1.1824
1	0.400539	-0.565212	-1.118359	-0.008332	0.405190	0.297154	0.622695	0.315968	-0.8027
2	-2.665469	0.090633	1.084350	-0.310466	0.662577	0.666607	-0.326063	1.840615	0.7401
3	-0.267809	1.291773	0.597533	0.204279	-1.691111	-1.133181	1.458491	-0.528331	-0.7506

In [40]:

```
#Covariant Matrix for Clas 3
cov_D3=np.corrcoef(np.transpose(D3_std))
pd.DataFrame(cov_D3).head(4)#Covariance matrix for class 3
```

Out[40]:

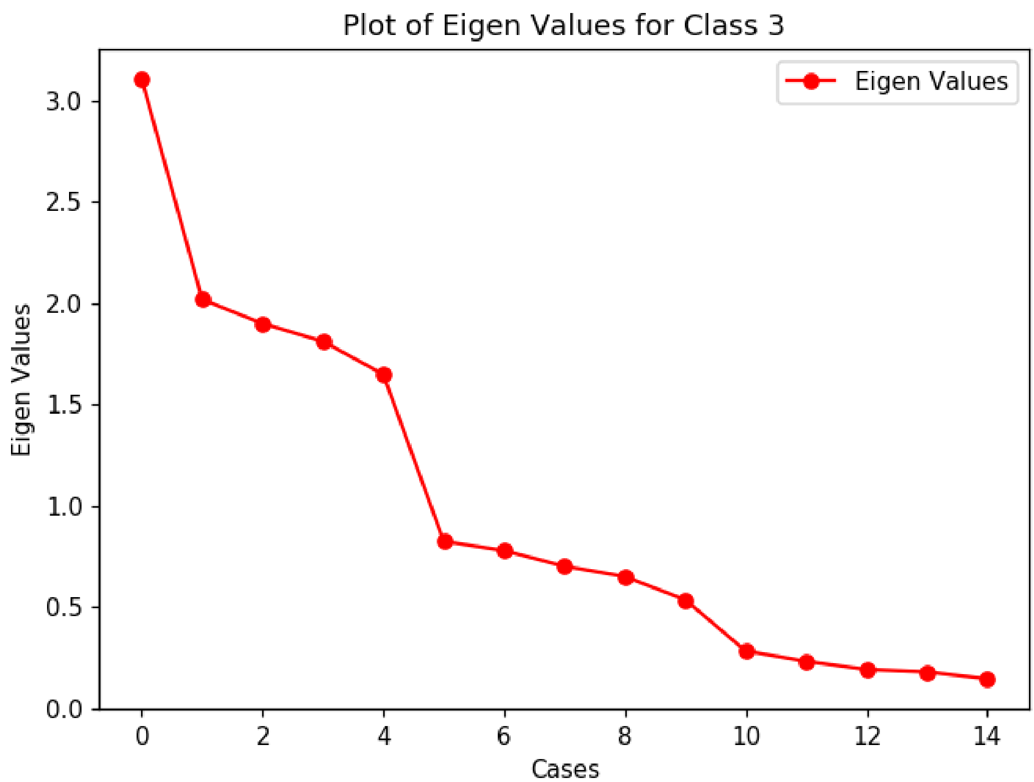
	0	1	2	3	4	5	6	7	
0	1.000000	-0.368009	-0.384430	0.020087	-0.068541	-0.188989	0.004910	-0.028518	-0.1505
1	-0.368009	1.000000	0.618927	-0.054968	-0.075118	0.082797	-0.084154	-0.126560	0.0342
2	-0.384430	0.618927	1.000000	-0.135704	0.056986	0.392538	-0.144237	-0.016704	0.3231
3	0.020087	-0.054968	-0.135704	1.000000	-0.329794	-0.439380	0.021648	-0.056917	-0.1706

In [41]:

```
#EigenValues and EigenVectors for Class 3
eig_valsD3,eig_vecsD3=np.linalg.eig(cov_D3)
print(eig_valsD3)
```

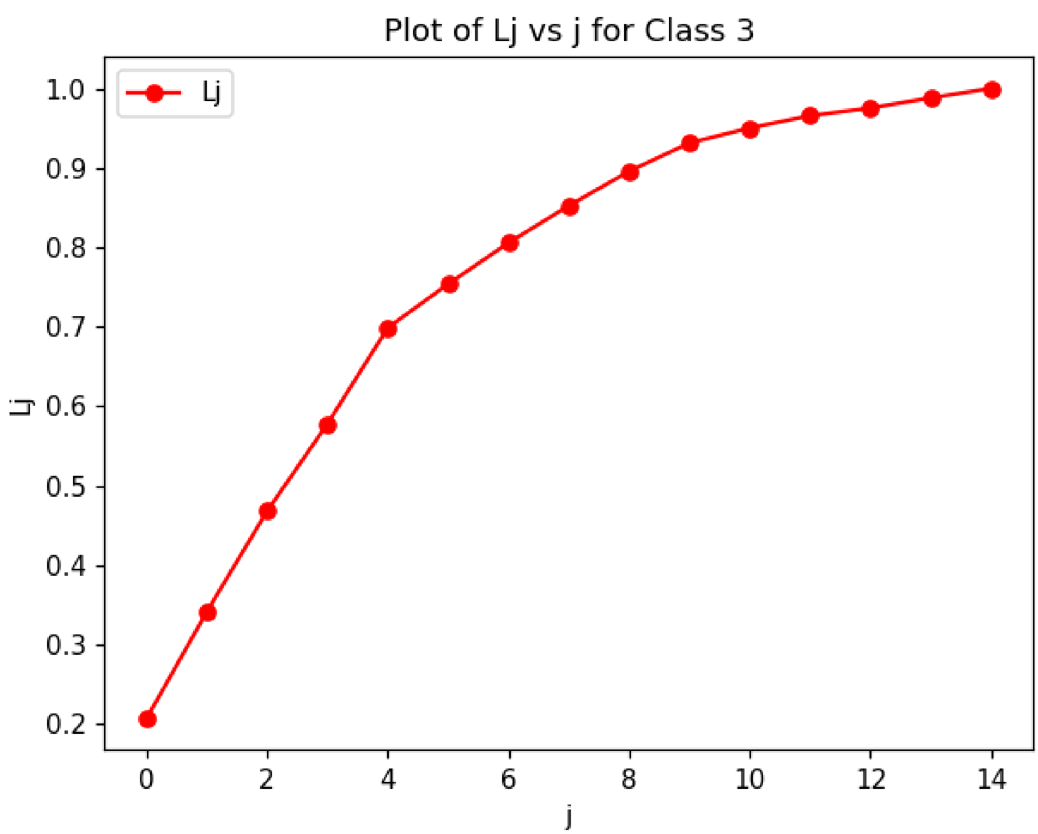
```
[3.10655699 2.01812158 1.89936038 1.64837126 1.81105203 0.82402068
 0.77797873 0.70066727 0.65007362 0.53496336 0.28233858 0.23130757
 0.14504887 0.19105152 0.17908755]
```

In [42]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.figure(1)
L1=sorted(eig_valsD3,reverse=True, )
plt.plot(L1,  marker='o', label='Eigen Values', color='r')
plt.ylabel('Eigen Values')
plt.xlabel('Cases')
plt.title('Plot of Eigen Values for Class 3')
plt.legend()
```



Out[42]: <matplotlib.legend.Legend at 0x295bbf32048>

In [43]:
```
%matplotlib notebook
dgvv3=eig_valsD3
da1vv3=[]
for i in range(15):
    if i ==0:
        Ri = dgvv3[i]
    else:
        Ri+=dgvv3[i]
    da1vv3.append(Ri)
%matplotlib notebook
plt.plot(da1vv3/(sum(dgvv3)),  marker='o', label='Lj', color='r')
plt.ylabel('Lj')
plt.xlabel('j')
plt.title('Plot of Lj vs j for Class 3')
plt.legend()
```



Out[43]: <matplotlib.legend.Legend at 0x295bc7d1a90>

In [44]:
```
LD3=sorted(eig_valsD3,reverse=True)
```

In [45]:
```
compareD3=sum(LD3)*0.95
sD3=0
countD3=0
for i in LD3:
    countD3=countD3+1
    sD3=sD3+i
   # print(i)
    if sD3>compareD3:
       # print(countD3)
        break

U3=countD3
U3
print("The smallest number is: " + str(U3))
```

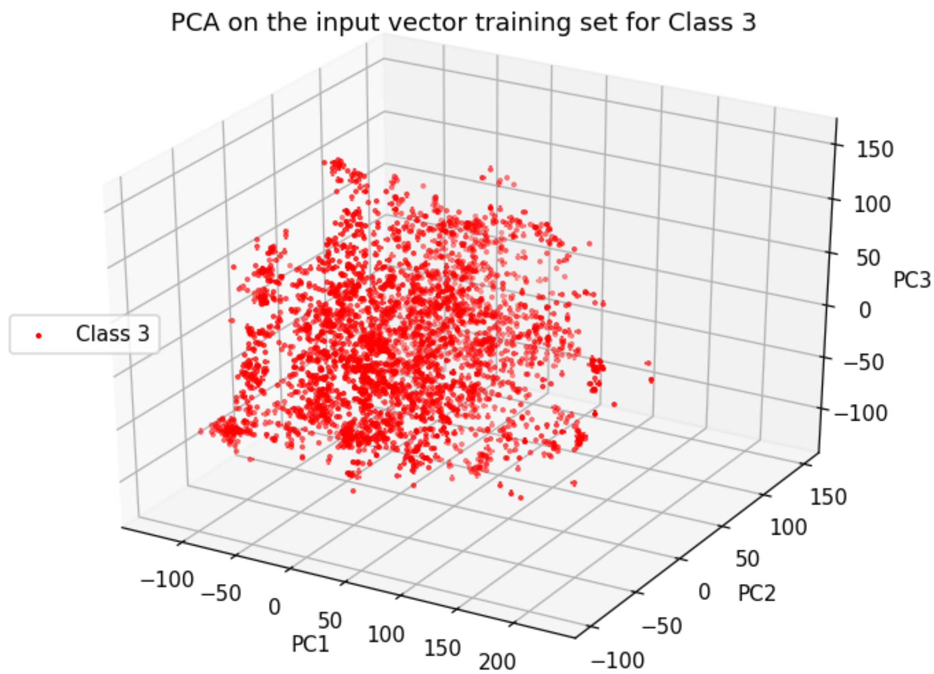
```
The smallest number is: 11
```

In [46]:
```
#PCA Analysis
pcac4 = PCA(n_components=3)
pcac4.fit(D3)
# Store results of PCA in a data frame
resultD3=pd.DataFrame(pcac4.transform(D3), columns=['PCA%i' % i for i in range
(3)], index=D3.index)
print(resultD3)
```

```
            PCA0        PCA1        PCA2
1     -47.080085  113.573884   41.091023
3     -19.777299   64.341962  -27.666581
5      32.188087  -75.233057   18.300147
8       9.888209   53.371407   -4.346209
9     -76.712459  102.095774  -40.463964
...          ...         ...         ...
9991   40.812441  -62.668756   40.578467
9992  -21.295435   96.418971   10.154413
9993  -65.313322  104.110450   79.149839
9996   64.145621  -30.583498   46.527712
10001  19.254683   56.730044  -93.928886

[3334 rows x 3 columns]
```

In [47]:
```
# Plot of Principal Components
%matplotlib notebook
fig4 = plt.figure(figsize=(8, 6))
ax4 = fig4.add_subplot(1,1,1, projection='3d')
ax4.scatter(resultD3['PCA0'], resultD3['PCA1'], resultD3['PCA2'],s=8,marker=
'+', color='r', label='Class 3')
ax4.set_xlabel("PC1")
ax4.set_ylabel("PC2")
ax4.set_zlabel("PC3")
ax4.legend (loc='center left')
ax4.set_title("PCA on the input vector training set for Class 3")
```



Out[47]: Text(0.5, 0.92, 'PCA on the input vector training set for Class 3')

In [48]:
```
#Size of hL
hL= U1+U2+U3
print("Plausible value of hL is: " + str(hL))
```

```
Plausible value of hL is: 34
```

PART 3 & 4 : Automatic Training and Performance Analysis

For number of hidden neuron h95 = 11

In [60]:
```
def fun (b,w,l, h ):
    import math
    training_epochs = 1000 #training epoch
    batch_size= b #batch size
    display_step=1 #step_size

    n_hidden = h #number of hidden layer =h95 first
    n_input = 15  #number of inputs equal number of features
    n_classes = 3  #number of classes number of output
    d=math.sqrt(h*15 + h + h *3 + 3) #square root of dimension

    #learning
    global1_step=tf.Variable(0,trainable=False)
    initial_learning_rate=l
    learning_rate=tf.compat.v1.train.exponential_decay(initial_learning_rate,\
    global_step=global1_step, decay_steps=training_epochs, decay_rate=0.9)

    add_global=global1_step.assign_add(1)

    X=tf.compat.v1.placeholder("float",[None,n_input])
    Y=tf.compat.v1.placeholder("float",[None,n_classes])

    random.seed(w)
    weights={
            'h': tf.Variable(tf.random_normal([n_input,n_hidden])),
            'out':tf.Variable(tf.random_normal([n_hidden,n_classes]))
            }

    biases={
            'b':tf.Variable(tf.random_normal([n_hidden])),
            'out':tf.Variable(tf.random_normal([n_classes]))
            }

    def MLP(x):
        layer_1=tf.add(tf.matmul(x,weights['h']), biases['b'])
        layer_1=tf.nn.relu(layer_1)
        out_layer=tf.matmul(layer_1, weights['out'])+biases['out']

        return out_layer

    #construct model
    logits=MLP(X)

    #define loss and optimizer


    loss=tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=Y,logit
s=logits))
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
.minimize(loss)
    correct_prediction=tf.equal(tf.argmax(logits,1),tf.argmax(Y,1))
    accuracy=tf.reduce_mean(tf.cast(correct_prediction,tf.float32))
    confusion_matrix=tf.math.confusion_matrix(tf.argmax(logits,1),tf.argmax(Y,
1))
    init=tf.global_variables_initializer()
```

```
    #Initializing the variables
    with tf.Session() as sess:
        sess.run(init)
        ini_acu=sess.run(accuracy,feed_dict={X:data_train,Y:y_train})
        train_accu1=[]
        test_accu1=[]
        train_loss=[]
        test_loss=[]
        L_R1=[]
        LOSS1=[]
        W_n1=[]
        relW=[]
        G_n1=[]
        G_ave1=[]
        BACRE1=[]
        LOSSBA = []
        ACRE1=[]
        #Training cycle
        for epoch in range(training_epochs):

            avg_cost=0
            total_batch=int(data_train.shape[0]/batch_size)
            store=np.append(np.reshape(sess.run(weights['h']),(1,n_hidden*n_in
put)),np.reshape(sess.run(weights['out']),(1,n_classes*n_hidden)))
            store=np.append(store,np.reshape(sess.run(biases['b']),(1,n_hidden
)))
            store=np.append(store,np.reshape(sess.run(biases['out']),(1,n_clas
ses)))
            for i in range(total_batch):

                step,rate=sess.run([add_global,learning_rate])
                L_R1.append(rate)
                # print(rate)
                random.seed(1000)
                randidx=np.random.randint(8001,size=batch_size)
                batch_xs=data_train.iloc[randidx,:]
                batch_ys=y_train.iloc[randidx,:]

                sess.run(optimizer,feed_dict={X:batch_xs,Y:batch_ys})
                c = sess.run(loss,feed_dict={X:batch_xs,Y:batch_ys})
                BACRE1.append(sess.run(accuracy,feed_dict={X:batch_xs,Y:batch_
ys}))
                LOSSBA.append(sess.run(loss,feed_dict={X:batch_xs,Y:batch_ys
}))
                #print(c)
                LOSS1.append(c)
                W1=np.reshape(sess.run(weights['h']),(1,n_hidden*n_input))
                W2=np.reshape(sess.run(weights['out']),(1,n_classes*n_hidden))
                W3=np.reshape(sess.run(biases['b']),(1,n_hidden))
                W4=np.reshape(sess.run(biases['out']),(1,n_classes))
                W=np.concatenate((W1,W2,W3,W4),axis=1)
                #W=np.append(W1,W2)
                #W=np.append(W,W3)
                #W=np.append(W,W4)
                WW=LA.norm(W-store)
                relW.append(WW/(LA.norm(store)))
                W_n1.append(WW)
```

```
                # print(WW)
                G_n1.append(WW/rate)
                G_ave1.append(WW/(rate*d))
                store=W
                avg_cost+=c/total_batch
                if step%80==0:
                    train=sess.run(accuracy,feed_dict={X:data_train,Y:y_train
})
                    train_accu1.append(train)
                    test=sess.run(accuracy,feed_dict={X:data_test,Y:y_test})
                    test_accu1.append(test)
                    train_loss.append(sess.run(loss,feed_dict={X:data_train,Y:
y_train}))
                    test_loss.append(sess.run(loss,feed_dict={X:data_test,Y:y_
test}))
        ax=sess.run(confusion_matrix,feed_dict={X:data_test,Y:y_test})
        ay=sess.run(confusion_matrix,feed_dict={X:data_train,Y:y_train})

    return train_accu1, test_accu1,train_loss, test_loss,ax,ay,LOSS1,W_n1,G_av
e1,W1,W3,BACRE1, LOSSBA
```

In [76]:
```
train_accu1, test_accu1,train_loss, test_loss,axcom,aycom,LOSS1,W_n1,G_ave1,W1
1,W31,BACRE1,LOSSBA11 = fun (1000, 1000, 0.01, 11)
```

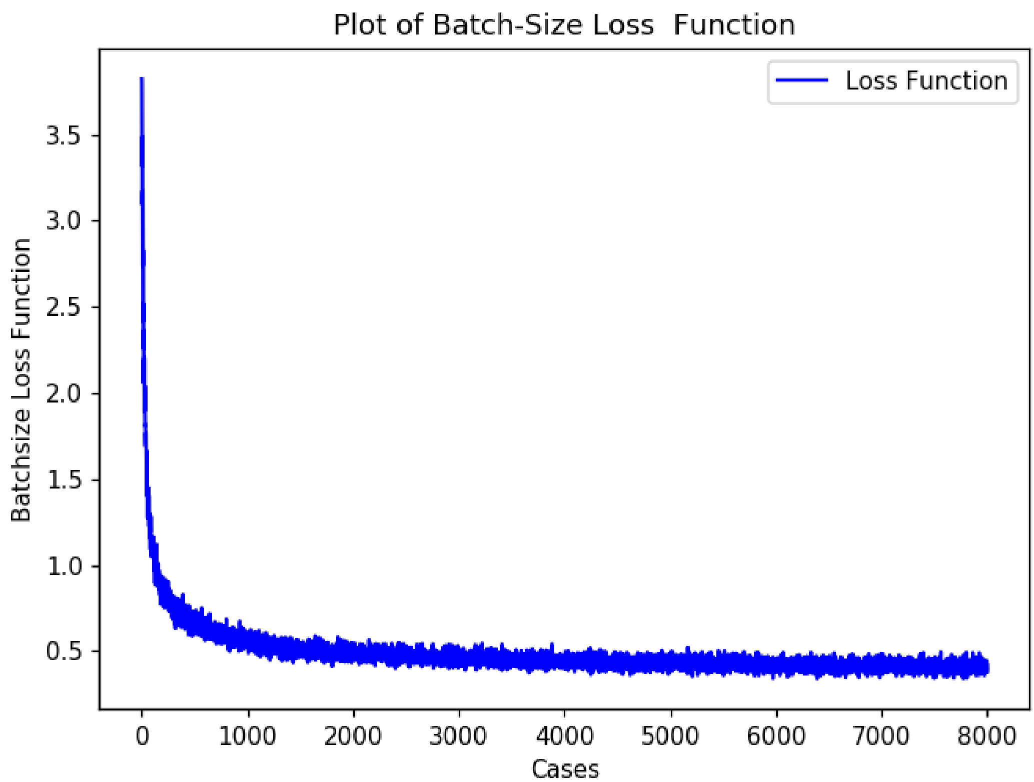
In [80]:
```
#Confusion Matrix for the Training Set
vv = np.array([sum(aycom[0,:]),sum(aycom[1,:]),sum(aycom[2,:])])
np.around((aycom.T/vv).T, decimals=3)
```

Out[80]:
```
array([[0.802, 0.113, 0.085],
       [0.125, 0.837, 0.038],
       [0.101, 0.018, 0.881]])
```

In [81]:
```
#Confusion Matrix for the Test Set
vvt = np.array([sum(axcom[0,:]),sum(axcom[1,:]),sum(axcom[2,:])])
np.around((axcom.T/vvt).T, decimals=3)
```

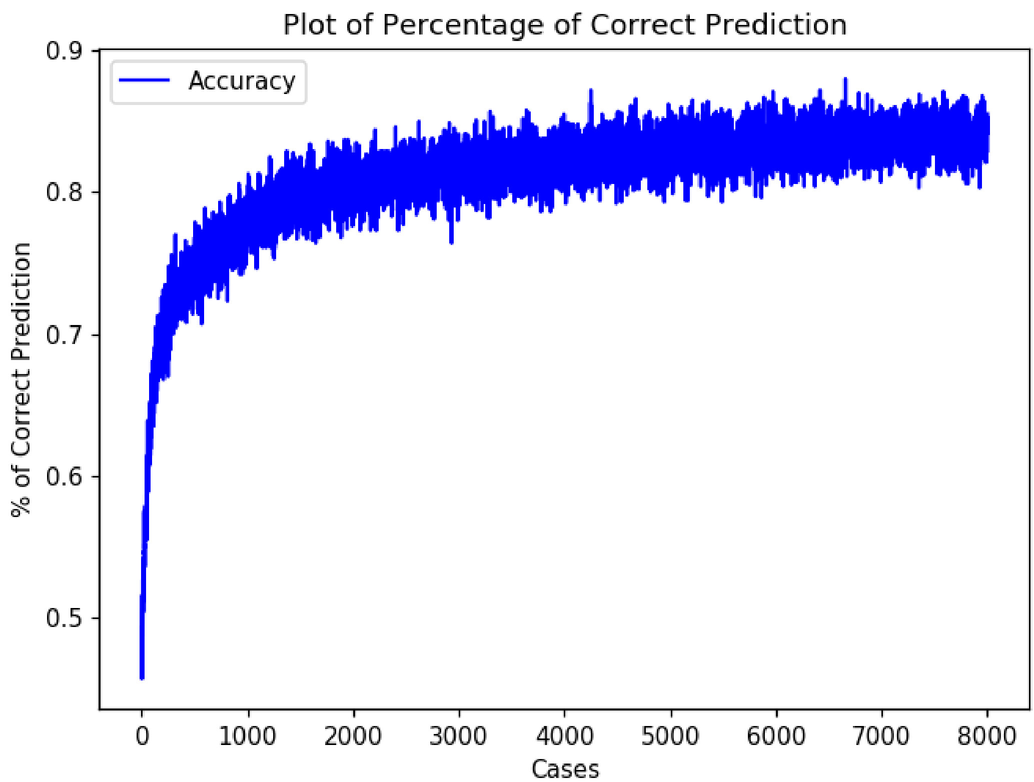
Out[81]:
```
array([[0.791, 0.115, 0.094],
       [0.131, 0.837, 0.032],
       [0.115, 0.019, 0.866]])
```

In [84]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.plot(range(8000),LOSSBA11,  color='b', label='Loss Function')
plt.ylabel('Batchsize Loss Function')
plt.xlabel('Cases')
plt.title('Plot of Batch-Size Loss  Function')
plt.legend()
```



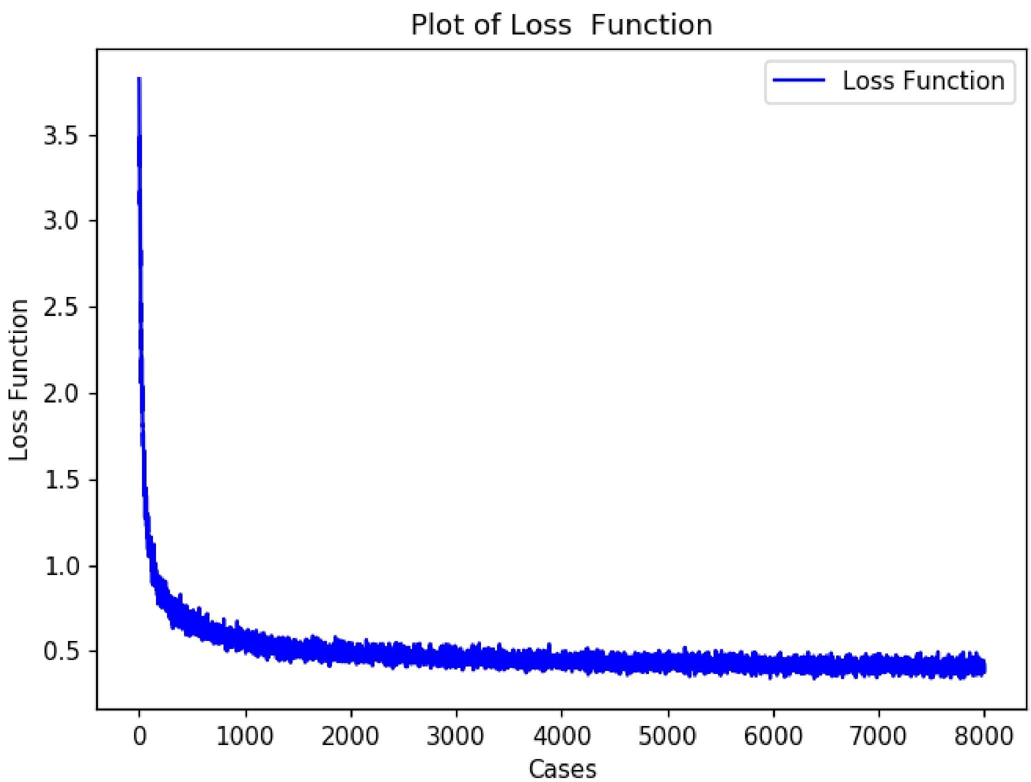
Out[84]: <matplotlib.legend.Legend at 0x295c477bbe0>

In [85]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.plot(range(8000),BACRE1,  color='b', label='Accuracy')
plt.ylabel('% of Correct Prediction')
plt.xlabel('Cases')
plt.title('Plot of Percentage of Correct Prediction')
plt.legend()
```



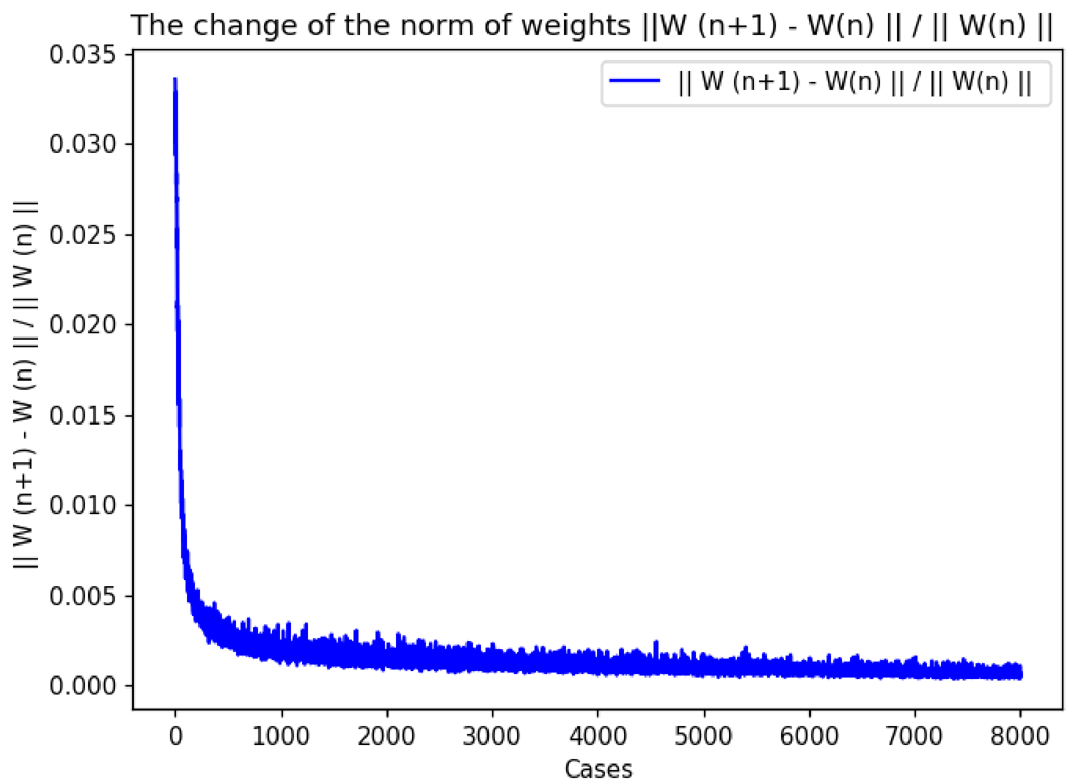
Out[85]: <matplotlib.legend.Legend at 0x295c4b36f60>

In [86]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.plot(range(8000),LOSS1,  color='b', label='Loss Function')
plt.ylabel('Loss Function')
plt.xlabel('Cases')
plt.title('Plot of Loss  Function')
plt.legend()
```



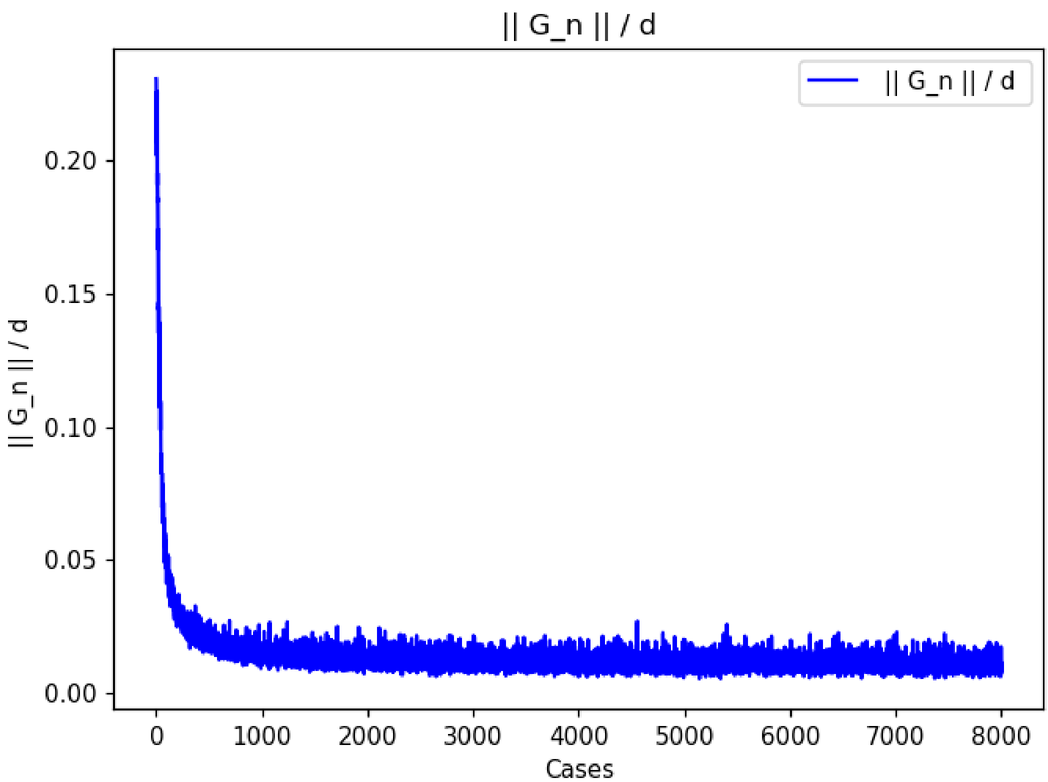
Out[86]: <matplotlib.legend.Legend at 0x295c4ee17f0>

In [87]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.plot(range(8000),W_n1,  color='b',  label='|| W (n+1) - W(n) || / || W(n)
 || ')
plt.ylabel('|| W (n+1) - W (n) || / || W (n) || ')
plt.xlabel('Cases')
plt.title('The change of the norm of weights ||W (n+1) - W(n) || / || W(n) ||
 ')
plt.legend()
```



Out[87]: <matplotlib.legend.Legend at 0x295c642ea58>

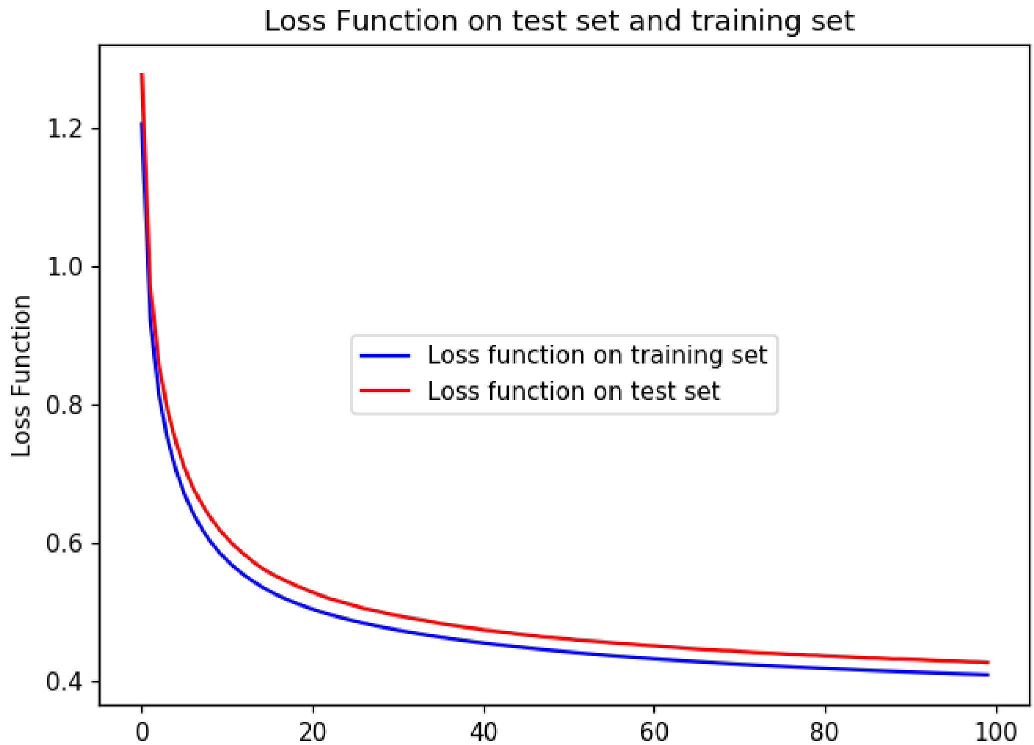
In [88]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.plot(range(8000),G_ave1,  color='b',  label=' || G_n || / d ')
plt.ylabel('|| G_n || / d ')
plt.xlabel('Cases')
plt.title(' || G_n || / d ')
plt.legend()
```



Out[88]: <matplotlib.legend.Legend at 0x295c68022e8>

In [89]:
```
%matplotlib notebook
fig = plt.figure(figsize=(7,5))
ax = plt.subplot(111)
ax.plot( train_accu1, color='b',label='performance on training set')
ax.plot( test_accu1, color='r',label='performance on test set')
plt.ylabel('performance')
plt.title('performance on test set and training set')
ax.legend(loc='center')
plt.show()
```

In [90]:
```
%matplotlib notebook
fig = plt.figure(figsize=(7,5))
ax = plt.subplot(111)
ax.plot( train_loss, color='b',label='Loss function on training set')
ax.plot( test_loss, color='r',label='Loss function on test set')
plt.ylabel('Loss Function')
plt.title('Loss Function on test set and training set ')
ax.legend(loc='center')
plt.show()
```



For number of hidden neuron hL = 34

In [94]:
```
train_accu12, test_accu12,train_loss2, test_loss2,ax2com,ay2com,LOSS12,W_n12,G
_ave12,W34,W35,BACRE2,LOSSBA12  = fun (1000, 1000, 0.01, 34)
```

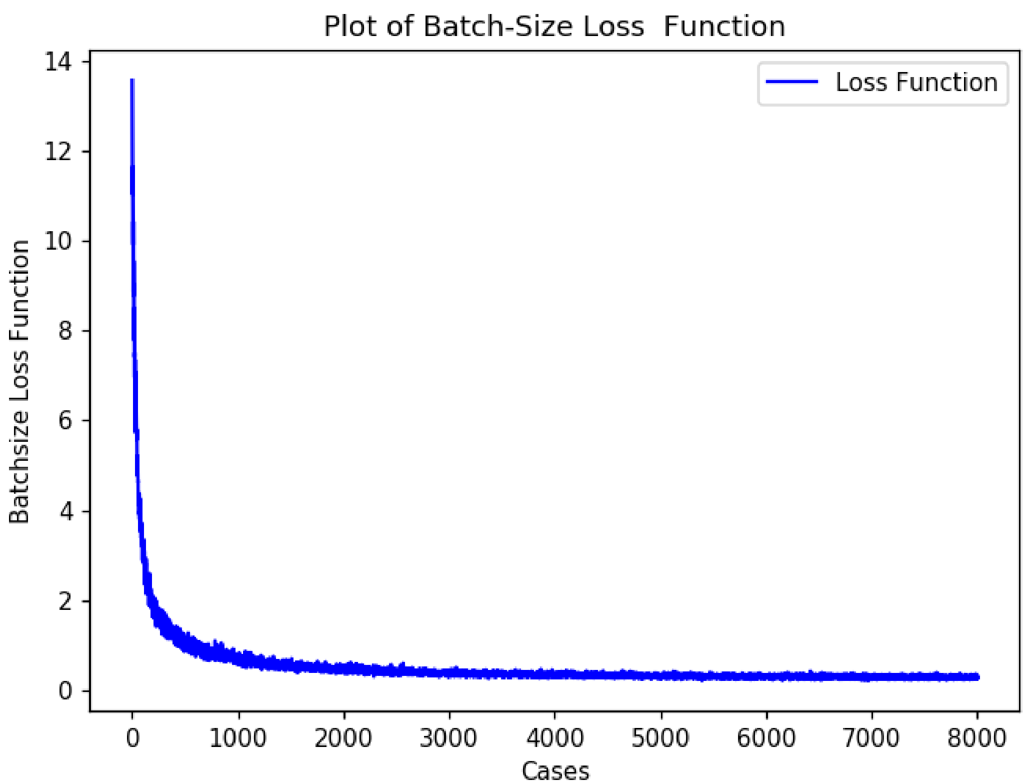
In [95]:
```
#Confusion Matrix for the Training Set
vvtr = np.array([sum(ay2com[0,:]),sum(ay2com[1,:]),sum(ay2com[2,:])])
np.around((ay2com.T/vvtr).T, decimals=3)
```

Out[95]:
```
array([[0.873, 0.091, 0.035],
       [0.101, 0.884, 0.015],
       [0.046, 0.012, 0.942]])
```

In [96]:
```
#Confusion Matrix for the Test Set
vvte1 = np.array([sum(ax2com[0,:]),sum(ax2com[1,:]),sum(ax2com[2,:])])
np.around((ax2com.T/vvte1).T, decimals=3)
```

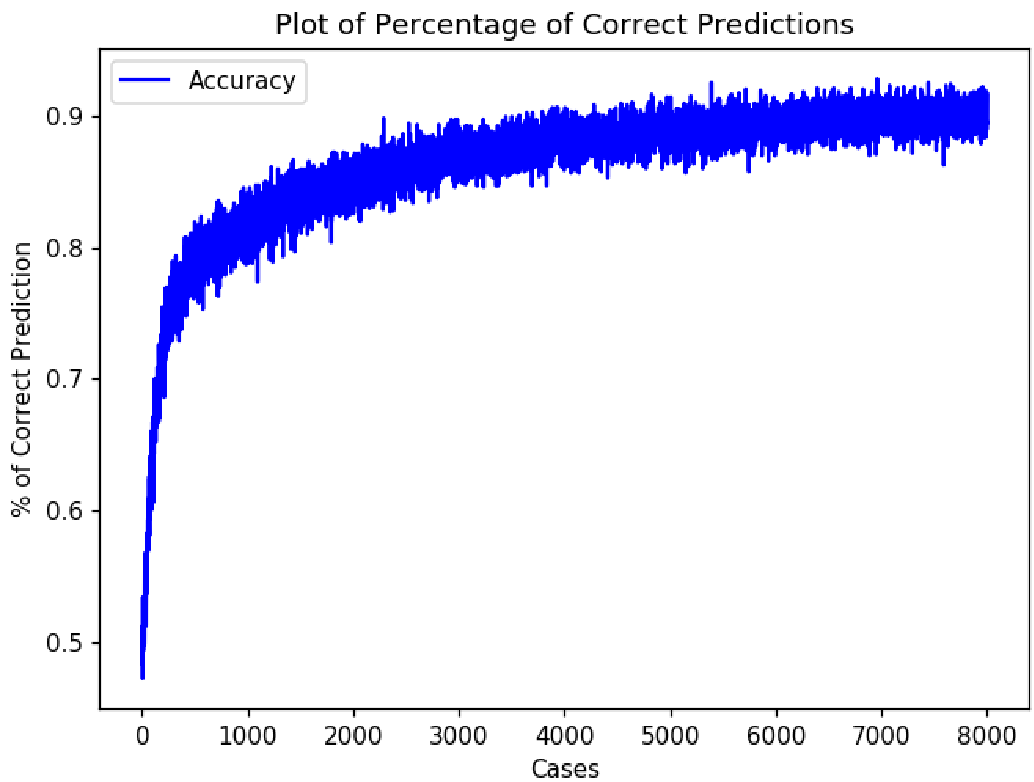
Out[96]:
```
array([[0.854, 0.107, 0.039],
       [0.088, 0.899, 0.012],
       [0.057, 0.007, 0.935]])
```

In [101]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.plot(range(8000),LOSSBA12,  color='b', label='Loss Function')
plt.ylabel('Batchsize Loss Function')
plt.xlabel('Cases')
plt.title('Plot of Batch-Size Loss  Function')
plt.legend()
```



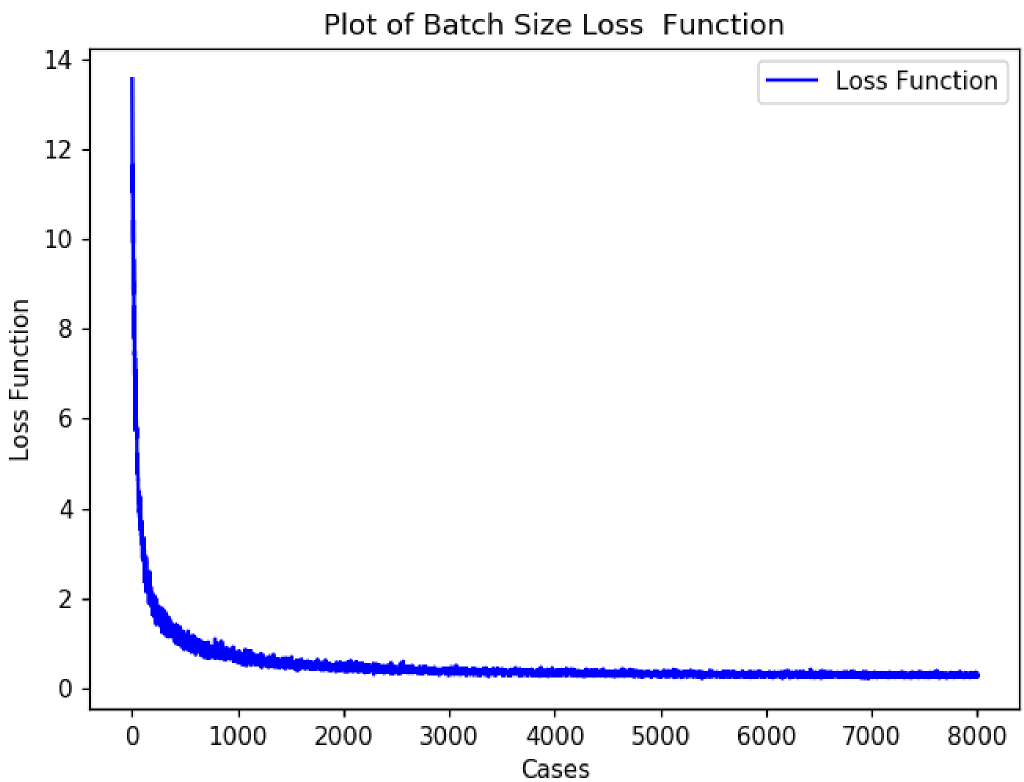
Out[101]:
```
<matplotlib.legend.Legend at 0x295c980ee10>
```

In [107]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.plot(range(8000),BACRE2,  color='b', label='Accuracy')
plt.ylabel('% of Correct Prediction')
plt.xlabel('Cases')
plt.title('Plot of Percentage of Correct Predictions')
plt.legend()
```



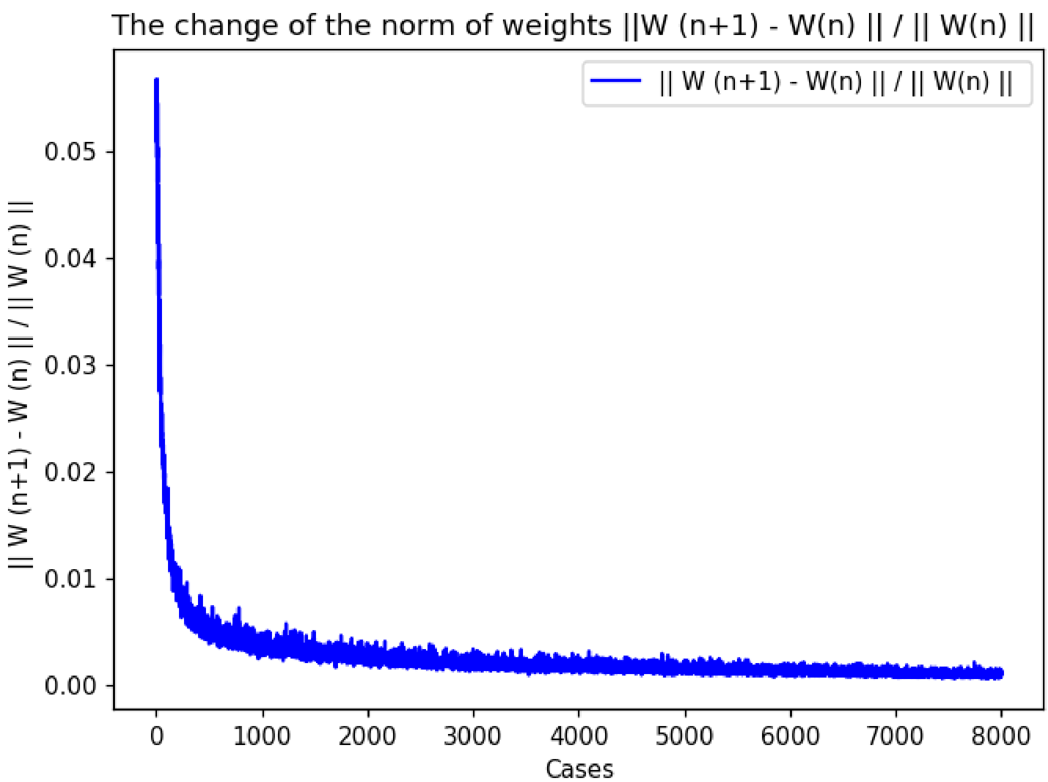
Out[107]: `<matplotlib.legend.Legend at 0x295c1faedd8>`

In [108]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.plot(range(8000),LOSS12,  color='b',  label='Loss Function')
plt.ylabel('Loss Function')
plt.xlabel('Cases')
plt.title('Plot of Batch Size Loss  Function')
plt.legend()
```



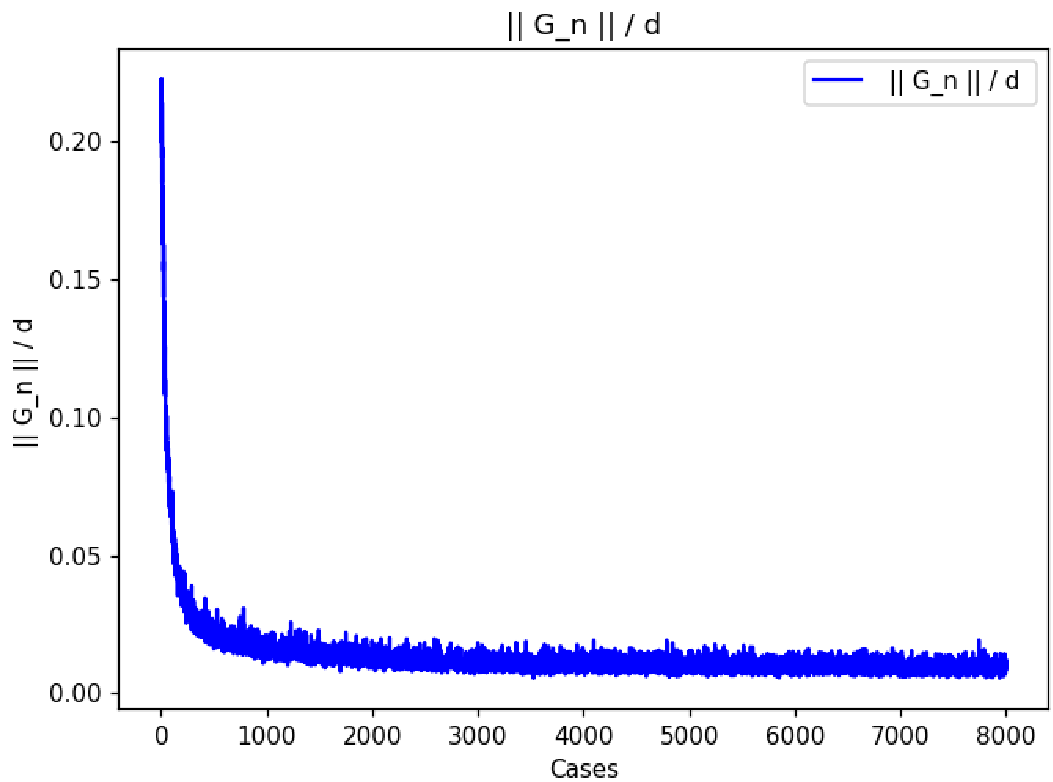
Out[108]: <matplotlib.legend.Legend at 0x295c0e78dd8>

In [109]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.plot(range(8000),W_n12,  color='b', label='|| W (n+1) - W(n) || / || W(n)
 || ')
plt.ylabel('|| W (n+1) - W (n) || / || W (n) || ')
plt.xlabel('Cases')
plt.title('The change of the norm of weights ||W (n+1) - W(n) || / || W(n) ||
 ')
plt.legend()
```



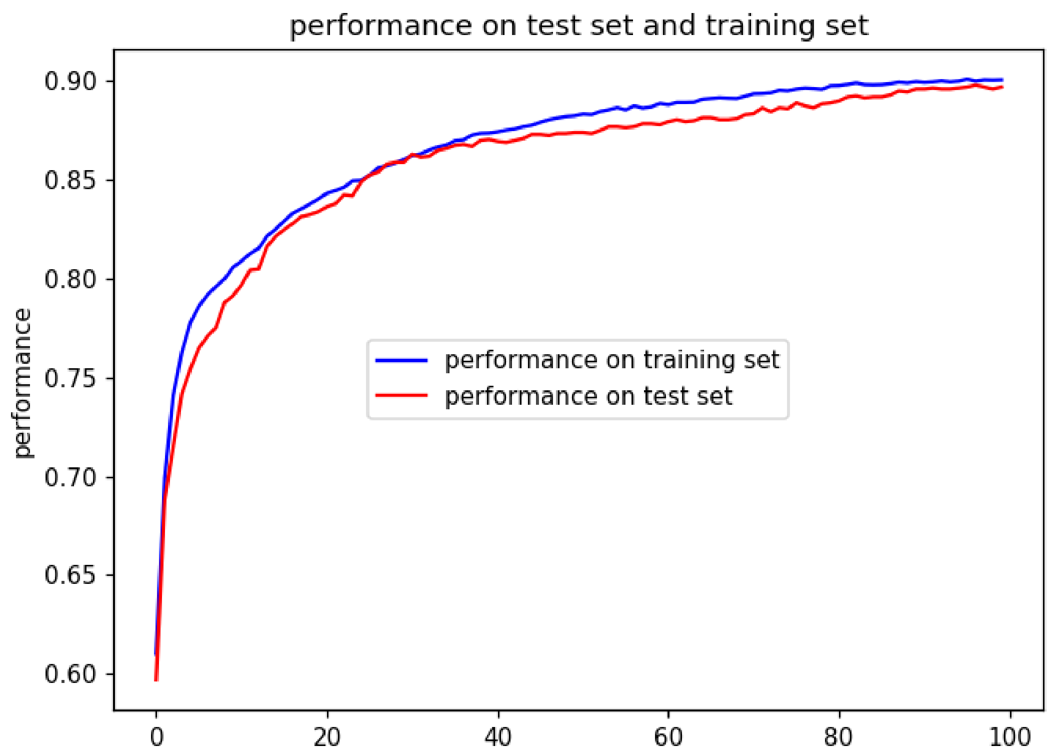
Out[109]: <matplotlib.legend.Legend at 0x295c0edad30>

In [110]:
```
%matplotlib notebook
plt.figure(figsize=(7, 5))
plt.plot(range(8000),G_ave12,  color='b', label=' || G_n || / d ')
plt.ylabel('|| G_n || / d ')
plt.xlabel('Cases')
plt.title(' || G_n || / d ')
plt.legend()
```

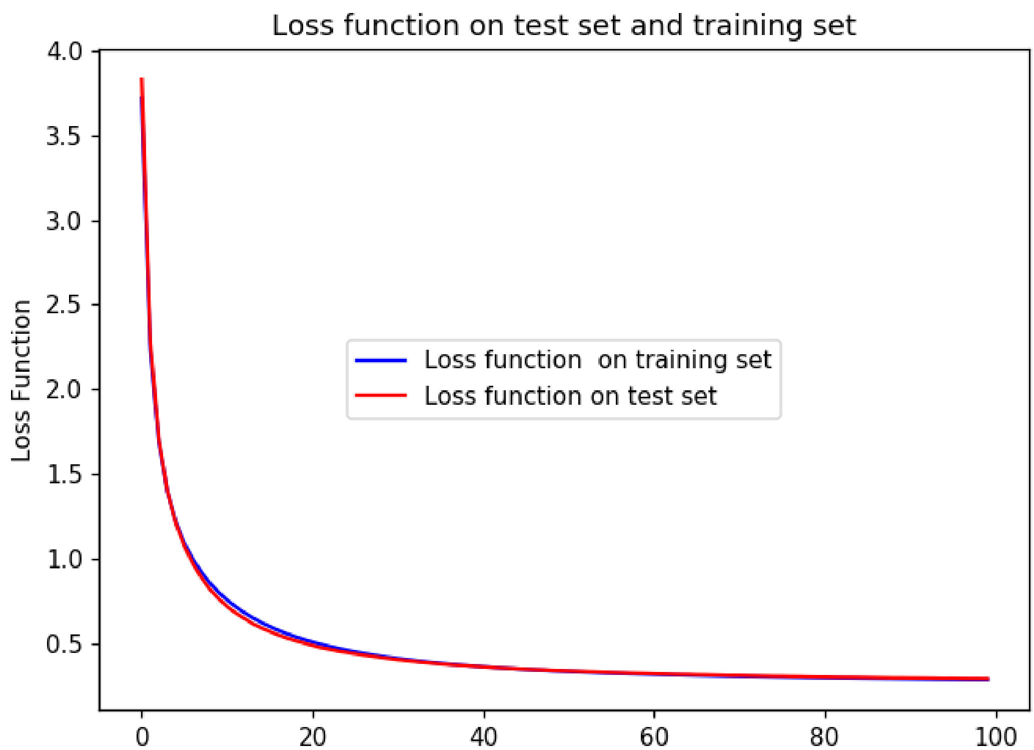


Out[110]: `<matplotlib.legend.Legend at 0x295c0f48b00>`

In [111]:
```
%matplotlib notebook
fig = plt.figure(figsize=(7,5))
ax = plt.subplot(111)
ax.plot( train_accu12, color='b',label='performance on training set')
ax.plot( test_accu12, color='r',label='performance on test set')
plt.ylabel('performance')
plt.title('performance on test set and training set')
ax.legend(loc='center')
plt.show()
```



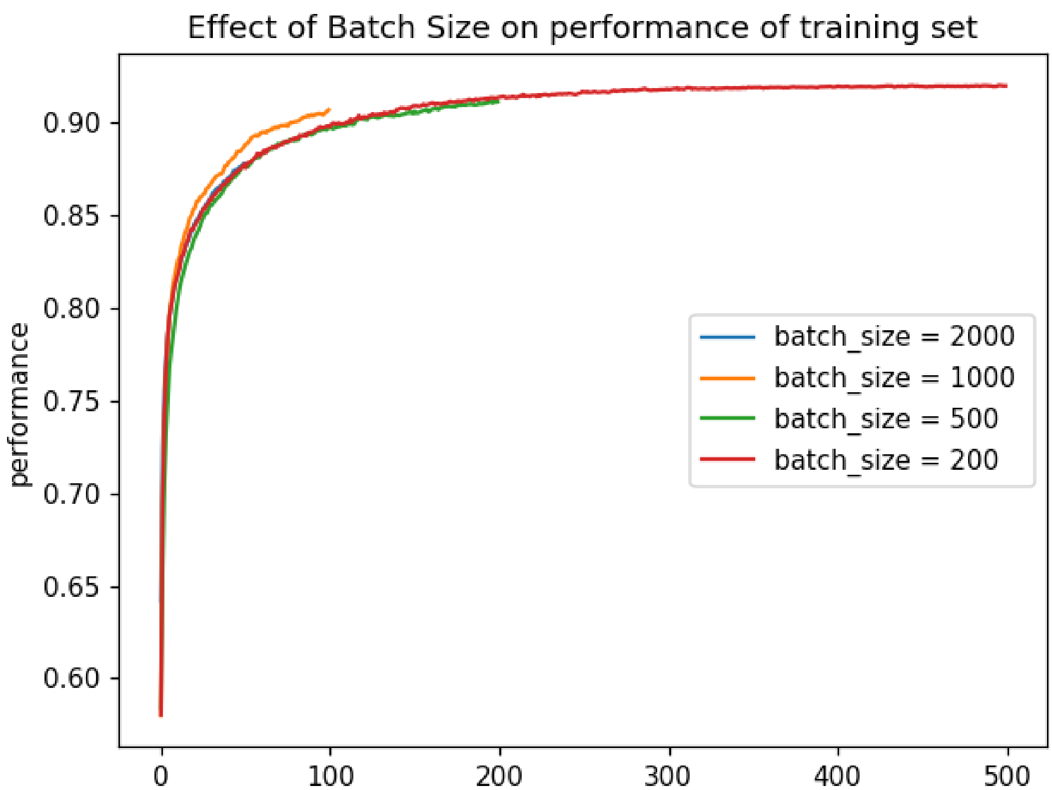
In [112]:
```
%matplotlib notebook
fig = plt.figure(figsize=(7,5))
ax = plt.subplot(111)
ax.plot( train_loss2, color='b',label='Loss function  on training set')
ax.plot( test_loss2, color='r',label='Loss function on test set')
plt.ylabel('Loss Function')
plt.title('Loss function on test set and training set')
ax.legend(loc='center')
plt.show()
```



PART 5: Impact of Various Learning Options

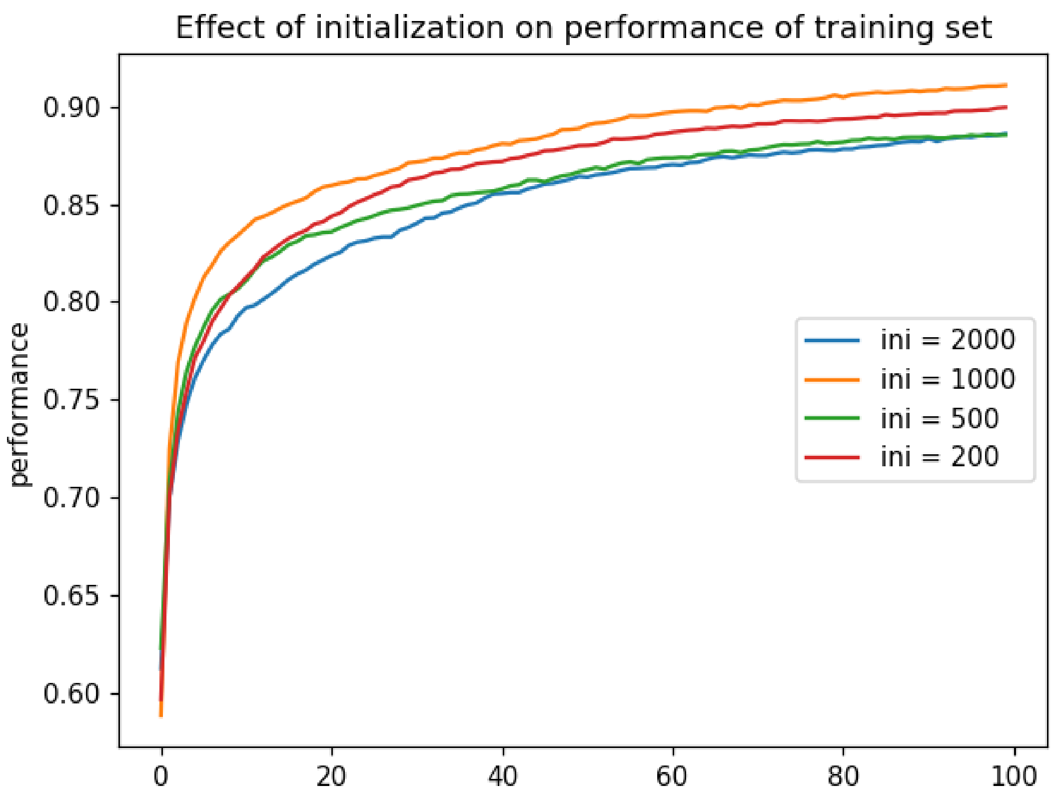
Effect of Batch Size on Performance

In [113]:
```
%matplotlib notebook
for b in [2000,1000, 500,  200]:
    train_accu1b, test_accu1b,train_lossb, test_lossb,axb,ayb,LOSS1b,W_n1b,G_a
ve1b,Wb1,Wb2,BACREb1,LOSSB31 = fun (b, 1000, 0.01, 34)
    plt.plot(train_accu1b, label='batch_size = %s '%b)
plt.ylabel('performance')
plt.title('Effect of Batch Size on performance of training set')
plt.legend(loc='center right')
plt.show()
```



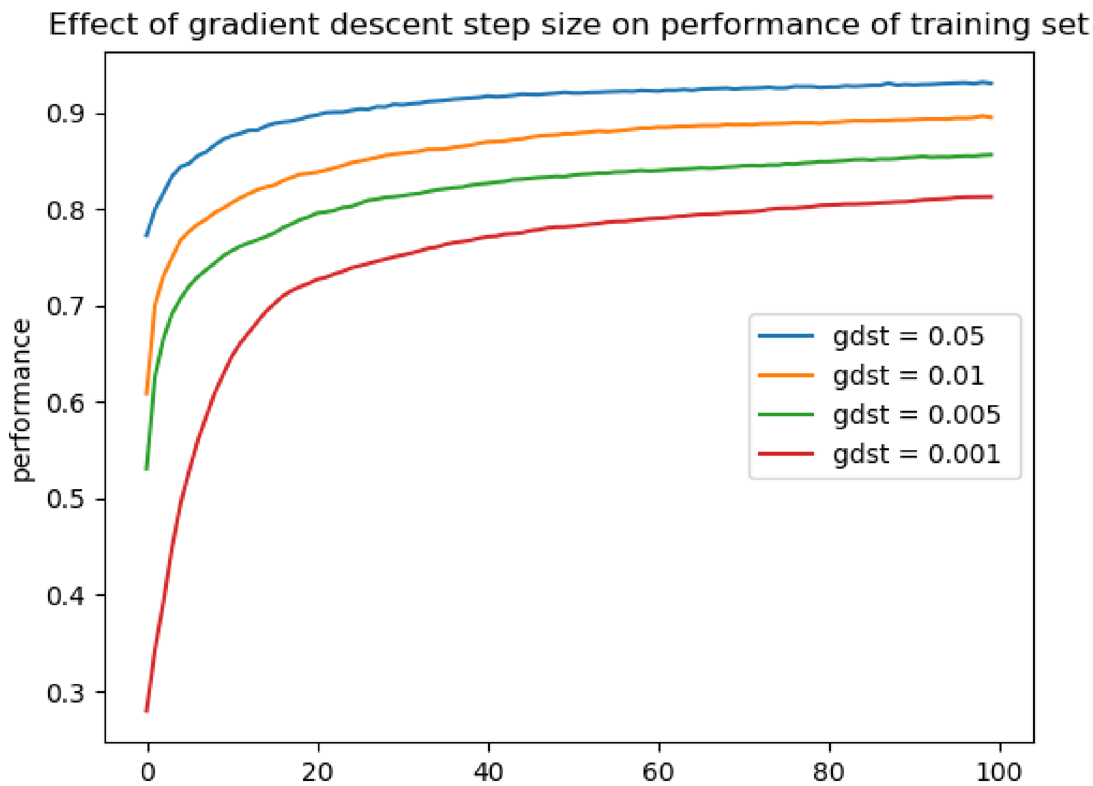
Effect of Initialization on Performance

In [115]:
```
%matplotlib notebook
for b in [2000,1000, 500,  200]:
    train_accu1b, test_accu1b,train_lossb, test_lossb,axb,ayb,LOSS1b,W_n1b,G_a
ve1b, Wi1, Wi2,BACREi, LOSSBA3 = fun (1000, b, 0.01, 34)
    plt.plot(train_accu1b, label='ini = %s '%b)
plt.ylabel('performance')
plt.title('Effect of initialization on performance of training set')
plt.legend(loc='center right')
plt.show()
```



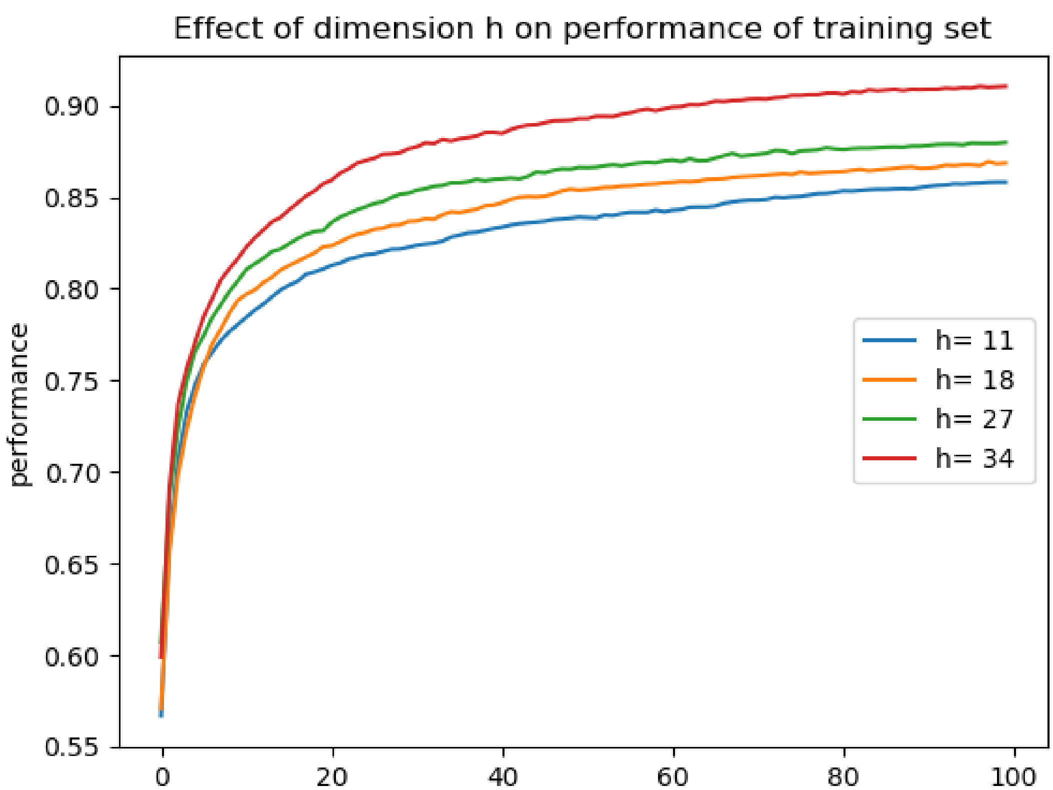
Effect of gradient descent step size

In [282]:
```
%matplotlib notebook
for b in [0.05,0.01, 0.005, 0.001]:
    train_accu1b, test_accu1b,train_lossb, test_lossb,axb,ayb,LOSS1b,W_n1b,G_a
ve1b, Wi3, Wi4, BACREh, LOSSBA4 = fun (1000, 1000, b, 34)
    plt.plot(train_accu1b, label='gdst = %s '%b)
plt.ylabel('performance')
plt.title('Effect of gradient descent step size on performance of training se
t')
plt.legend(loc='center right')
plt.show()
```



Effect of dimension h of H

In [283]:
```
%matplotlib notebook
for b in [11,18,27,34]:
    train_accu1b, test_accu1b,train_lossb, test_lossb,axb,ayb,LOSS1b,W_n1b,G_a
ve1b, Wh1, Wh2, BACREbh,LOSSBA5 = fun (1000, 2000, 0.01, b)
    plt.plot(train_accu1b, label='h= %s '%b)
    plt.ylabel('performance')
    plt.title('Effect of dimension h on performance of training set')
plt.legend(loc='center right')
plt.show()
```



PART 6: Analysis of hidden layer behaviour

In [284]:
```
#Global States of the Hidden Layer
hidden_global_states = []
for a in range(10002):
    cc=np.reshape(W34, (15,34))
    dd= np.array([scaled_data.iloc[a,:]])
    ry = pd.DataFrame(np.matmul(dd,cc)+np.matrix(W35))
    hidden_global_states.append(ry)
global_states=pd.concat(hidden_global_states)
```

In [285]:
```
global_states.head(5)#First 5 cases
```

Out[285]:

	0	1	2	3	4	5	6	7	8
0	-2.002860	0.961967	4.181001	1.416856	-2.814030	3.746032	0.531388	-0.185582	0.98378
0	-4.074219	5.536793	1.443264	2.285714	2.732781	-2.453837	-2.766361	-4.437343	1.97895
0	-6.634599	2.529500	4.381410	1.510009	-3.409478	-1.466038	-1.343636	-4.204399	-2.43952
0	-1.622281	-3.454943	0.268150	9.713295	-2.949385	-1.692263	-6.831481	2.866246	0.97267
0	-6.067314	2.702971	5.070668	3.287963	-2.283503	-2.103664	-1.052264	-3.194274	-3.78275

5 rows × 34 columns

In [286]:
```
#redundancy pruning check
global_states_std=StandardScaler().fit_transform(global_states)
corr_global_states=np.corrcoef(np.transpose(global_states_std))
a=abs(pd.DataFrame(corr_global_states))
a.head(10)
```

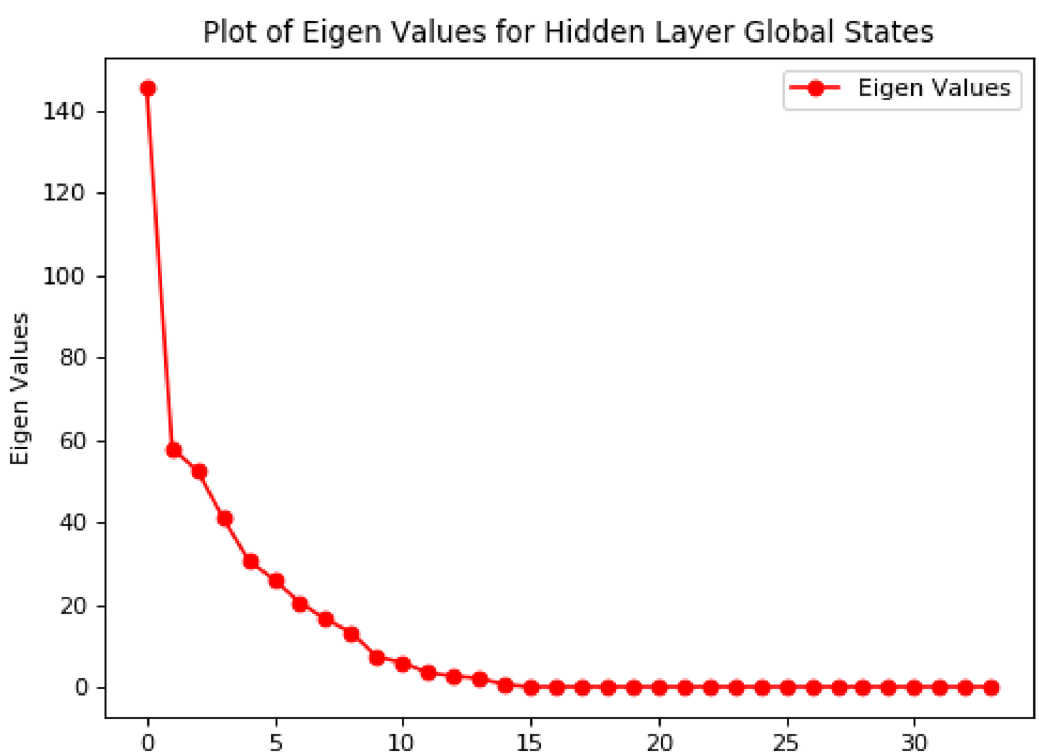
Out[286]:

	0	1	2	3	4	5	6	7	8	
0	1.000000	0.569764	0.288284	0.316506	0.239918	0.187949	0.089524	0.717138	0.383318	0.
1	0.569764	1.000000	0.162831	0.171160	0.011525	0.161641	0.017456	0.621285	0.235194	0.
2	0.288284	0.162831	1.000000	0.285679	0.615649	0.444783	0.079531	0.639377	0.386054	0.
3	0.316506	0.171160	0.285679	1.000000	0.487033	0.577500	0.694568	0.088485	0.214565	0.
4	0.239918	0.011525	0.615649	0.487033	1.000000	0.231659	0.365210	0.343841	0.137727	0.
5	0.187949	0.161641	0.444783	0.577500	0.231659	1.000000	0.276402	0.299962	0.161028	0.
6	0.089524	0.017456	0.079531	0.694568	0.365210	0.276402	1.000000	0.013197	0.369196	0.
7	0.717138	0.621285	0.639377	0.088485	0.343841	0.299962	0.013197	1.000000	0.513608	0.
8	0.383318	0.235194	0.386054	0.214565	0.137727	0.161028	0.369196	0.513608	1.000000	0.
9	0.163567	0.133474	0.114822	0.540818	0.245703	0.039016	0.089463	0.261728	0.029007	1.

10 rows × 34 columns

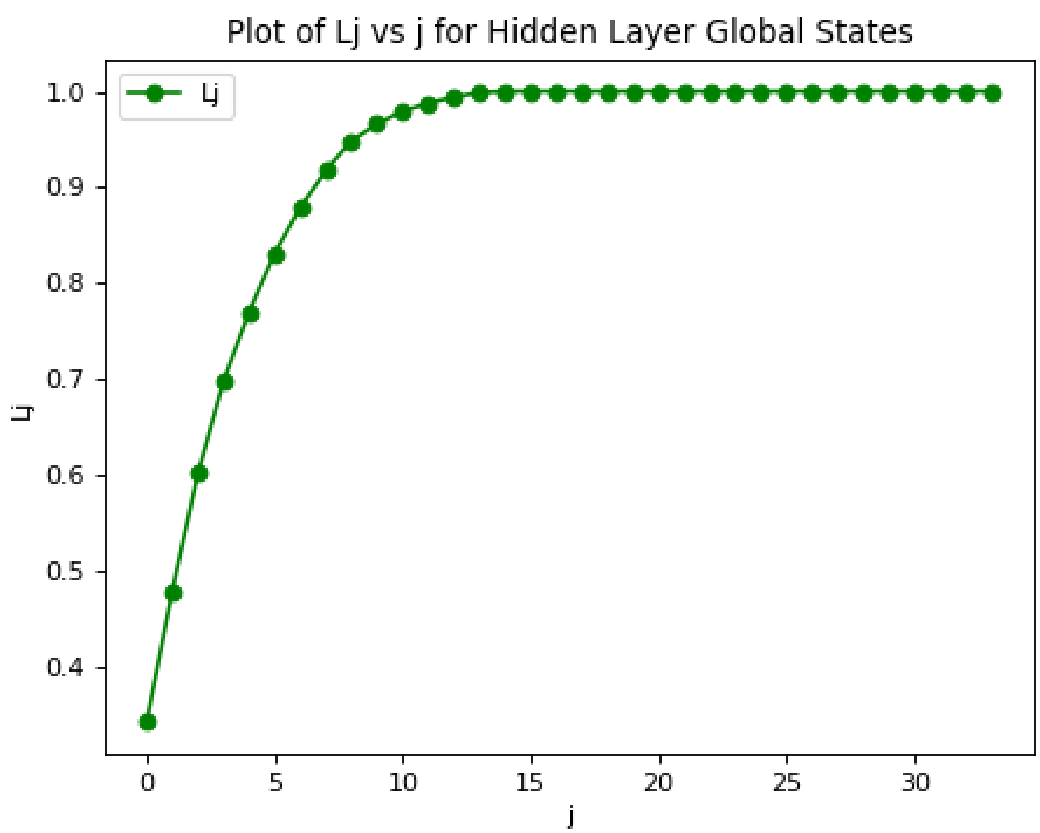
In [377]:
```
%matplotlib notebook
pca2l = PCA(n_components=34)
pca2l.fit(global_states)
print((pca2l.explained_variance_ratio_))
plt.figure(figsize=(7, 5))
plt.figure(1)
Le=sorted(pca2l.explained_variance_,reverse=True, )
plt.plot(Le,  marker='o', label='Eigen Values', color='r')
plt.ylabel('Eigen Values')
plt.xlabel('')
plt.title('Plot of Eigen Values for Hidden Layer Global States')
plt.legend()
```

```
[3.42329015e-01 1.36272545e-01 1.23144288e-01 9.62042151e-02
 7.19280859e-02 6.11173004e-02 4.79189191e-02 3.86379613e-02
 3.10298298e-02 1.72454967e-02 1.38519450e-02 7.98067824e-03
 6.18849645e-03 4.78018773e-03 1.37103596e-03 7.63398165e-33
 3.45048295e-33 2.54155901e-33 2.25505339e-33 2.10991356e-33
 2.07107152e-33 2.03439844e-33 1.77123489e-33 1.66044393e-33
 1.49268133e-33 1.48846617e-33 1.48846617e-33 1.48846617e-33
 1.48846617e-33 1.48846617e-33 1.48846617e-33 1.48846617e-33
 1.48846617e-33 1.48846617e-33]
```



Out[377]: `<matplotlib.legend.Legend at 0x16529521588>`

In [376]:
```
dg=pca2l.explained_variance_ratio_
da1=[]
for i in range(34):
    if i ==0:
        Ri = dg[i]
    else:
        Ri+=dg[i]
    da1.append(Ri)
%matplotlib notebook
plt.plot(da1,  marker='o', label='Lj', color='g')
plt.ylabel('Lj')
plt.xlabel('j')
plt.title('Plot of Lj vs j for Hidden Layer Global States')
plt.legend()
```



Out[376]: <matplotlib.legend.Legend at 0x1652949be10>

In [289]:
```
#Smallest Number uB
bb1=pca2l.explained_variance_
compare1bb=sum(bb1)*0.95
s1bb=0
count1bb=0
for i in bb1:
    count1bb=count1bb+1
    s1bb=s1bb+i
   # print(i)
    if s1bb>compare1bb:
       # print(count1)
        break

Ub=count1bb
print("The smallest number Ub is: " + str(Ub))
```

```
The smallest number Ub is: 10
```

In [290]:
```
#Get the classes for the global states
ff=pd.concat([(global_states.reset_index(drop=True)),y_output], axis=1)
D111= ff[ff['Class']==1].iloc[:,0:34]
D211= ff[ff['Class']==2].iloc[:,0:34]
D311= ff[ff['Class']==3].iloc[:,0:34]
print ("Class 1: "+ str(D111.shape[0]),"Class 2: "+ str(D211.shape[0]), "Class 3: "+str(D311.shape[0]) )
```

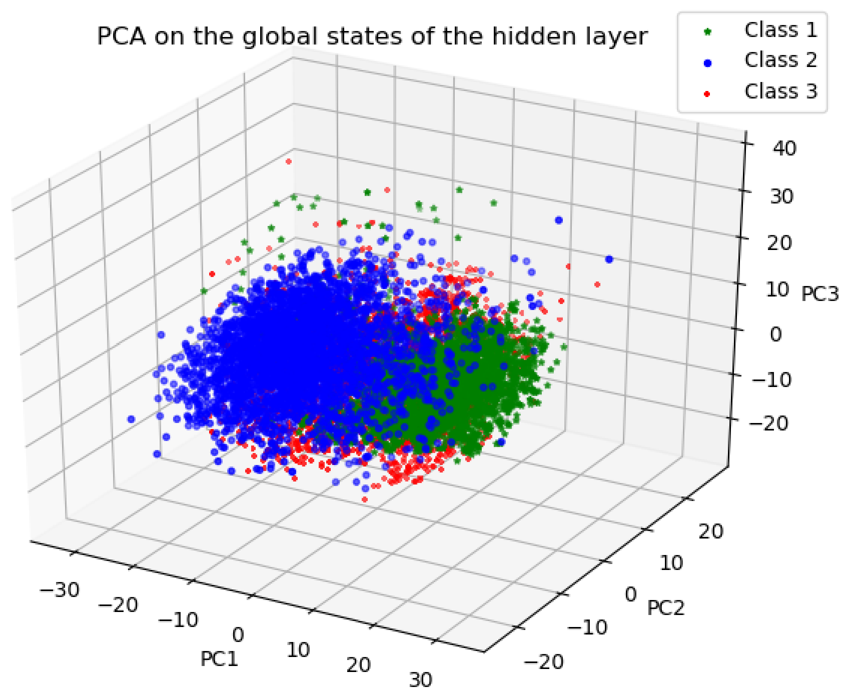
```
Class 1: 3334 Class 2: 3334 Class 3: 3334
```

In [291]:
```
# PCA Analysis for Global states of the hidden layer
pca2l1 = PCA(n_components=3)
pca2l1.fit(global_states)
result11=pd.DataFrame(pca2l1.transform(D111), columns=['PCA%i' % i for i in range(3)], index=D111.index)
result12=pd.DataFrame(pca2l1.transform(D211), columns=['PCA%i' % i for i in range(3)], index=D211.index)
result13=pd.DataFrame(pca2l1.transform(D311), columns=['PCA%i' % i for i in range(3)], index=D311.index)
print(result12)
```

```
           PCA0       PCA1       PCA2
1    -17.220172   5.818170  15.391655
2    -28.208133  -7.349290  -8.395854
4    -25.491351  -3.047852   0.157710
6    -14.543473   1.276715  11.309739
7    -14.329854 -13.068049  -6.630295
...         ...        ...        ...
9992 -10.751792  -4.809884  10.353612
9993  -2.956948   1.528698  -2.603284
9996  -8.485723 -10.305815  -2.364113
9997 -17.981210   3.761675   0.841383
9998 -26.612454  -4.768697 -12.311448

[3334 rows x 3 columns]
```

In [292]:
```
# Plot of Principal Components
%matplotlib notebook
fig = plt.figure(figsize=(8, 6))
axv = fig.add_subplot(1,1,1, projection='3d')
axv.scatter(result11['PCA0'], result11['PCA1'], result11['PCA2'],s=8,marker=
'*', color='g', label='Class 1')
axv.scatter(result12['PCA0'], result12['PCA1'], result12['PCA2'],s=8,marker=
'o', color = 'b', label='Class 2')
axv.scatter(result13['PCA0'], result13['PCA1'], result13['PCA2'],s=8, marker=
'+', color='r', label='Class 3')
axv.set_xlabel("PC1")
axv.set_ylabel("PC2")
axv.set_zlabel("PC3")
axv.legend (loc='best')
axv.set_title("PCA on the global states of the hidden layer")
```

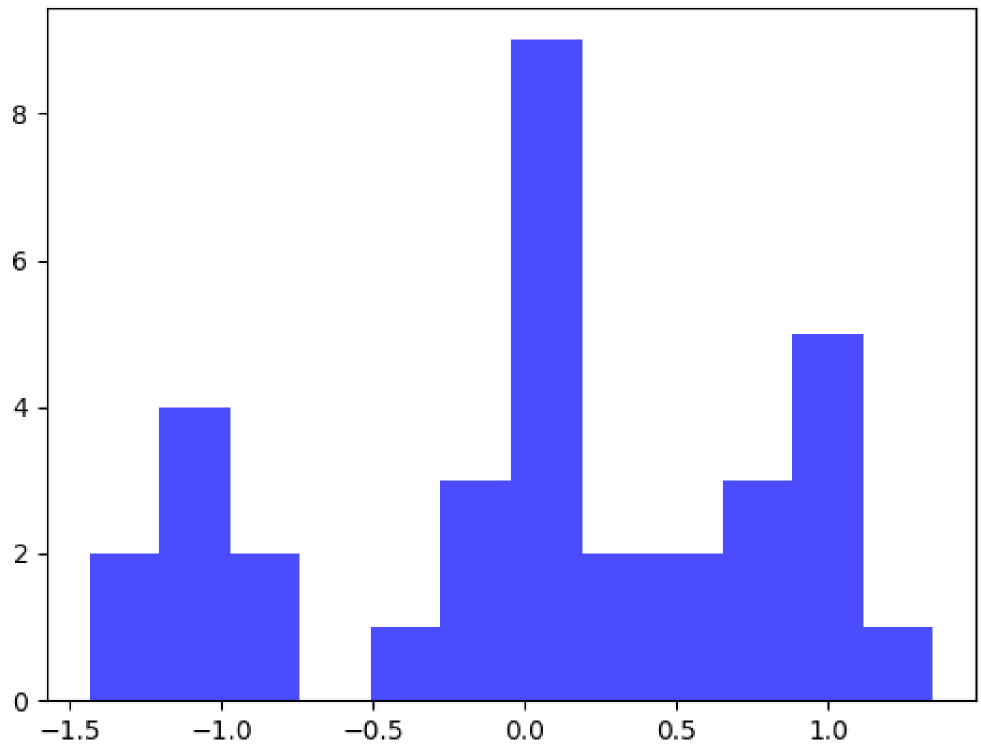


Out[292]: `Text(0.5, 0.92, 'PCA on the global states of the hidden layer')`

In [293]:
```
average_D111= (D111.sum(axis=0))/10002 #Class 1
average_D211= (D211.sum(axis=0))/10002 #Class 2
average_D311=(D311.sum(axis=0))/10002 #Class 3
average_All = (global_states.sum(axis=0))/10002 #Class 3
```

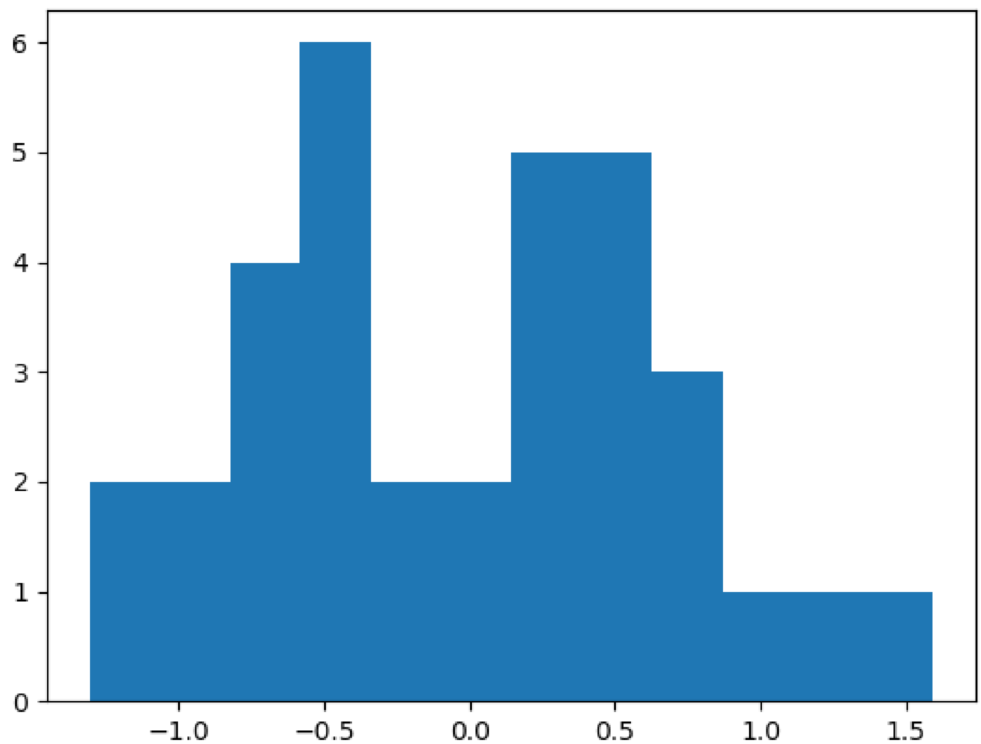
Histogram of Class 1 Average Activity

In [303]:
```
%matplotlib notebook
n, bins, patches = plt.hist(average_D111, bins=12, color='b',alpha=0.7)
plt.show()
```



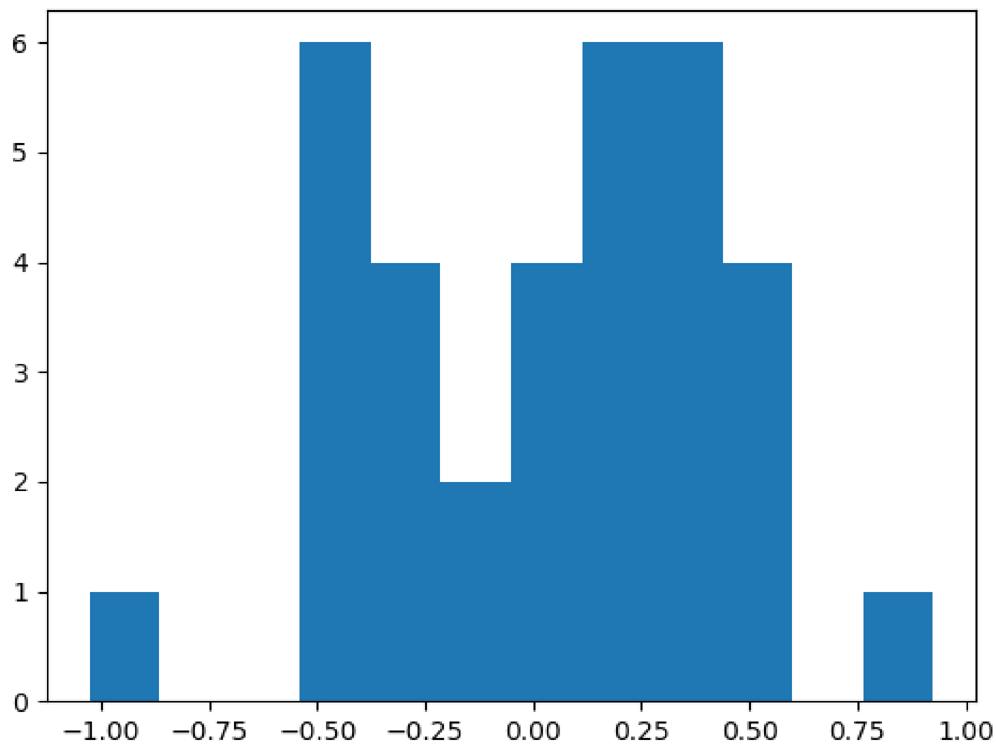
Histogram of Class 2 Average Activity

In [304]:
```
%matplotlib notebook
n, bins, patches = plt.hist(average_D211, bins=12)
plt.show()
```



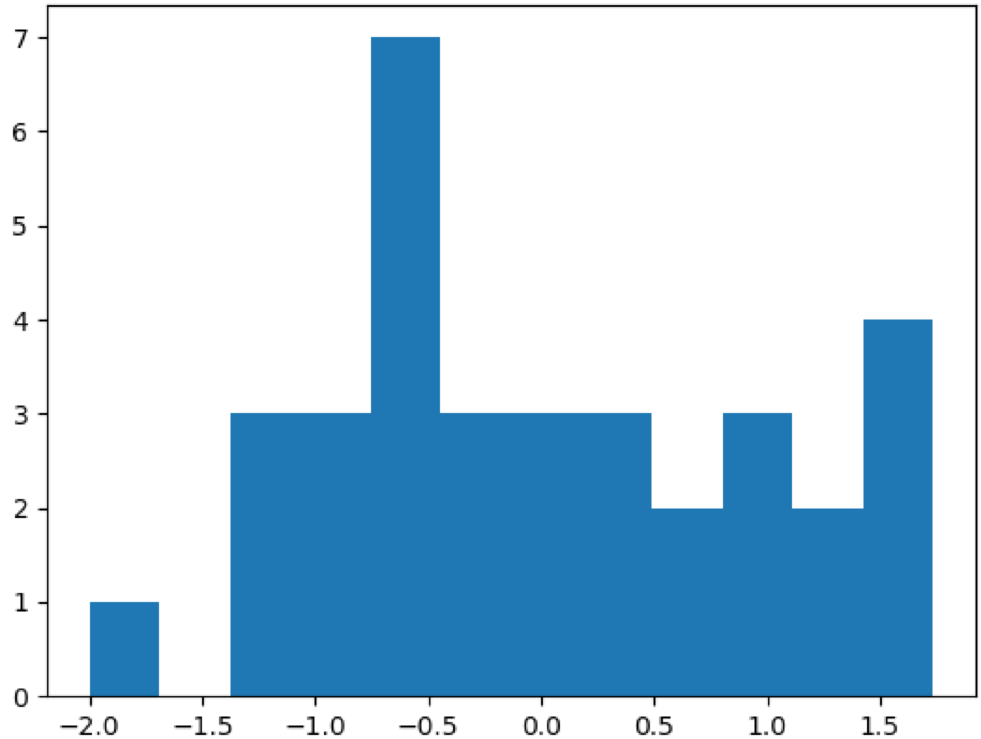
Histogram of Class 3 Average Activity

In [305]:
```
%matplotlib notebook
n, bins, patches = plt.hist(average_D311, bins=12)
plt.show()
```



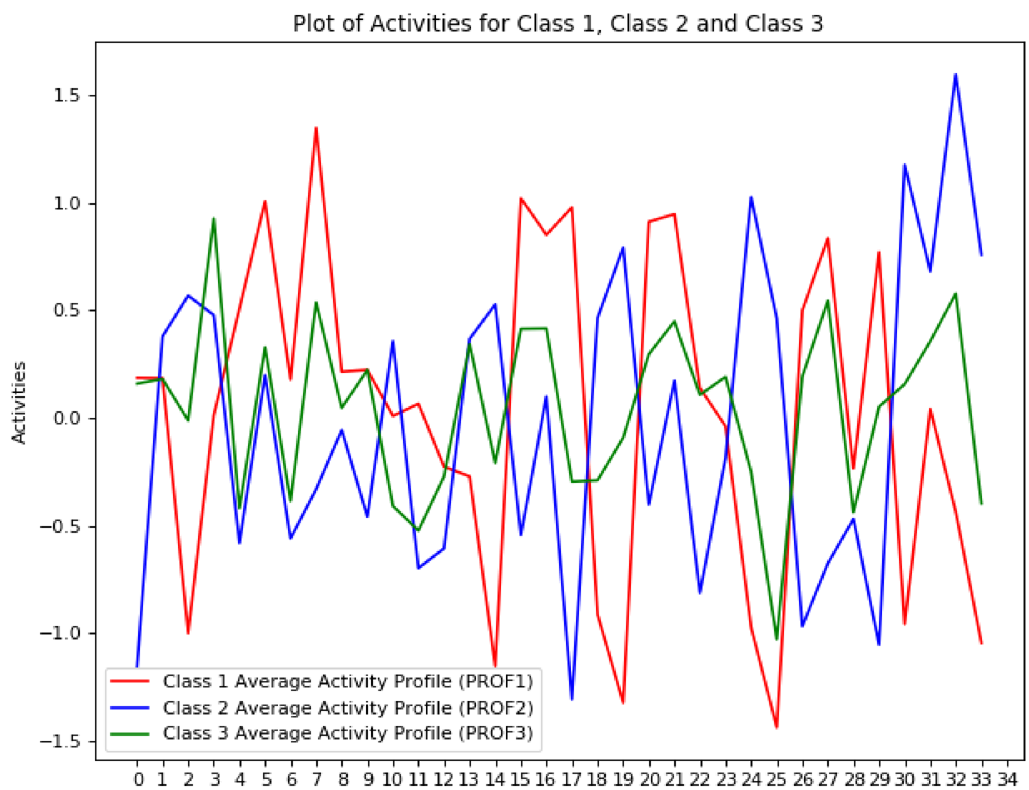
Histogram of Global Average Activity

In [306]:
```
%matplotlib notebook
n, bins, patches = plt.hist(average_All, bins=12)
plt.show()
```



Plot of hidden neuron profile activity for each class (DIFFERENTIATION)

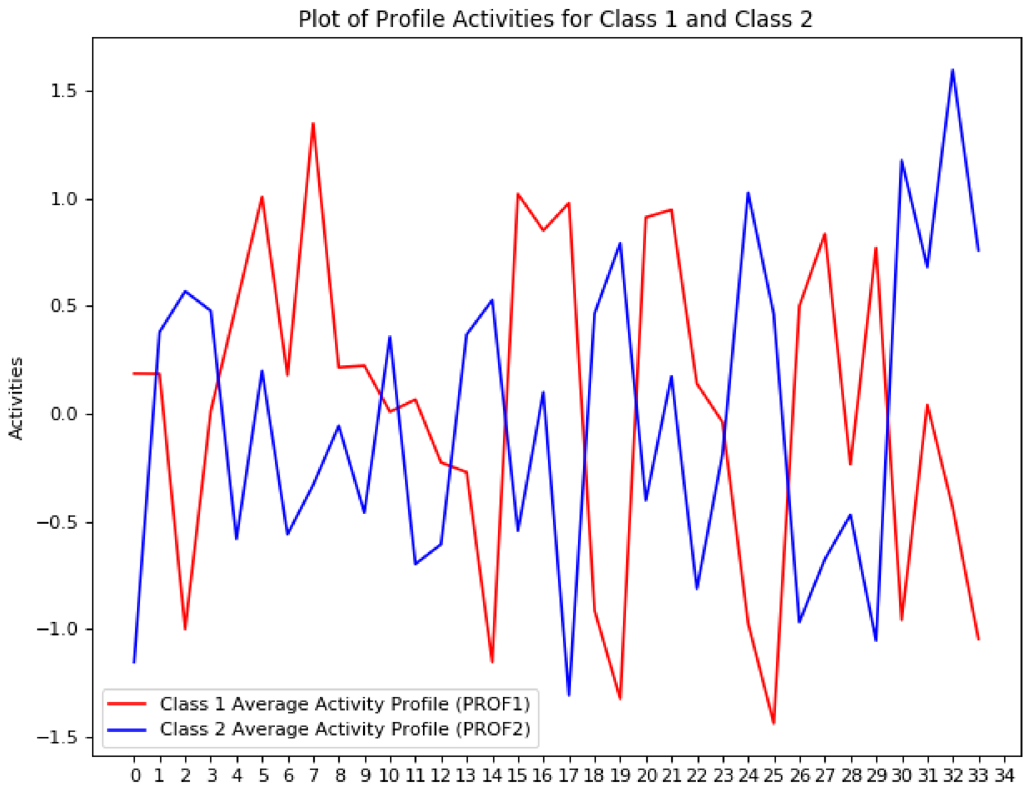
In [380]:
```
%matplotlib notebook
plt.figure(figsize=(9, 7))
plt.plot(average_D111,color='r',  label='Class 1 Average Activity Profile (PRO
F1)')
plt.plot(average_D211,color='b',  label='Class 2 Average Activity Profile (PRO
F2)')
plt.plot(average_D311,color='g',  label='Class 3 Average Activity Profile (PRO
F3)')
plt.ylabel('Activities')
plt.xlabel('')
plt.xticks(np.arange(0, 34+1, 1.0))
plt.title('Plot of Activities for Class 1, Class 2 and Class 3')
plt.legend(loc='lower left')
```



Out[380]: <matplotlib.legend.Legend at 0x16529d15ba8>

Plot of hidden neuron profile activity for class 1 and class 2

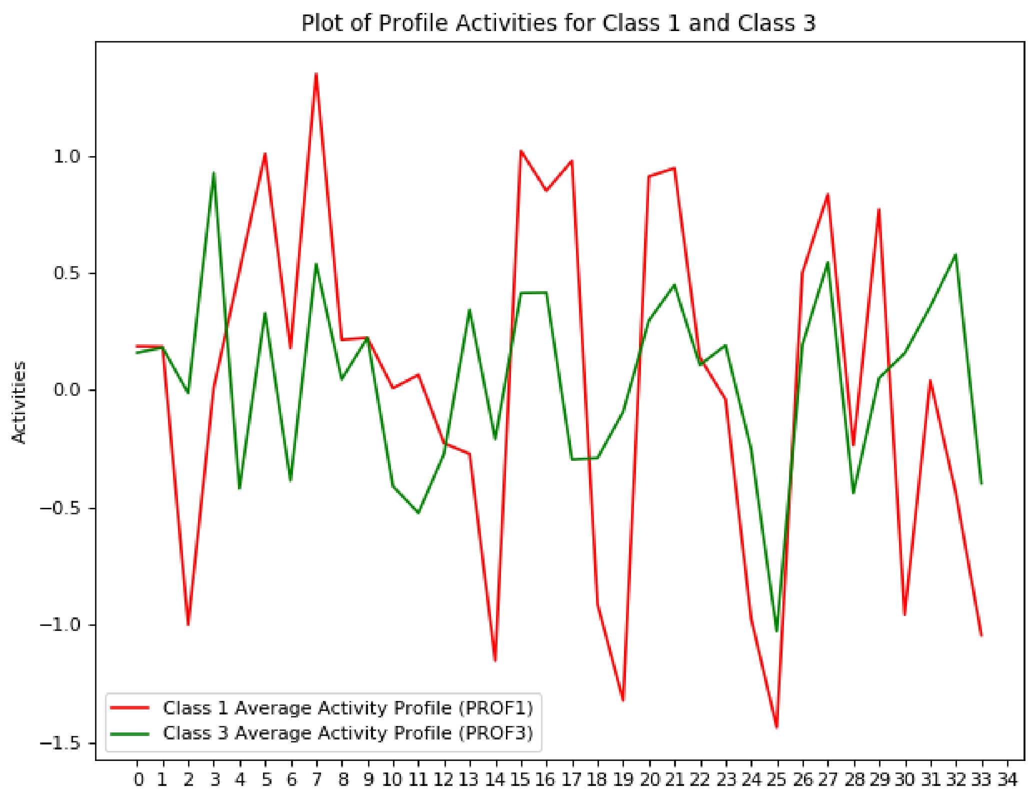
In [381]:
```
%matplotlib notebook
plt.figure(figsize=(9, 7))
plt.plot(average_D111,color='r',  label='Class 1 Average Activity Profile (PRO
F1)')
plt.plot(average_D211,color='b',  label='Class 2 Average Activity Profile (PRO
F2)')
plt.ylabel('Activities')
plt.xlabel('')
plt.xticks(np.arange(0, 34+1, 1.0))
plt.title('Plot of Profile Activities for Class 1 and Class 2')
plt.legend(loc='lower left')
```



Out[381]: <matplotlib.legend.Legend at 0x1652a164da0>

Plot of hidden neuron profile activity for class 1 and class 3

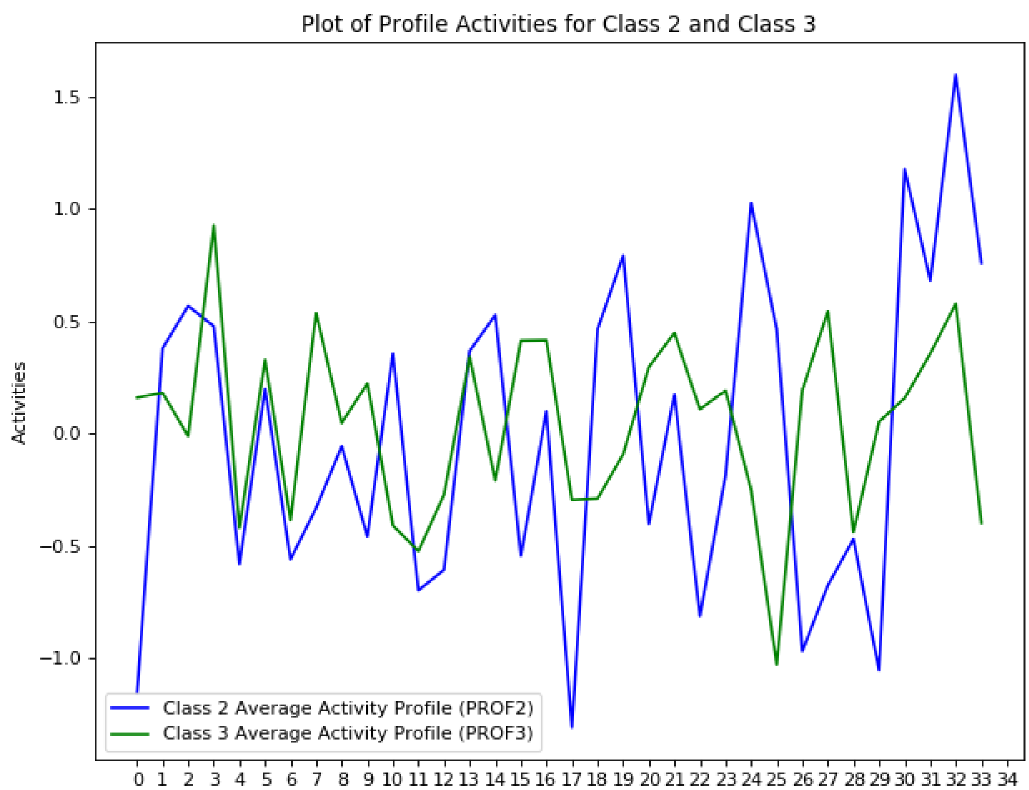
In [382]:
```
%matplotlib notebook
plt.figure(figsize=(9, 7))
plt.plot(average_D111,color='r',  label='Class 1 Average Activity Profile (PRO
F1)')
plt.plot(average_D311,color='g',  label='Class 3 Average Activity Profile (PRO
F3)')
plt.ylabel('Activities')
plt.xlabel('')
plt.xticks(np.arange(0, 34+1, 1.0))
plt.title('Plot of Profile Activities for Class 1 and Class 3')
plt.legend(loc='lower left')
```



Out[382]: <matplotlib.legend.Legend at 0x1652a4de668>

Plot of hidden neuron profile activity for class 2 and class 3

In [383]:
```
%matplotlib notebook
plt.figure(figsize=(9, 7))
plt.plot(average_D211,color='b',  label='Class 2 Average Activity Profile (PRO
F2)')
plt.plot(average_D311,color='g',  label='Class 3 Average Activity Profile (PRO
F3)')
plt.ylabel('Activities')
plt.xlabel('')
plt.xticks(np.arange(0, 34+1, 1.0))
plt.title('Plot of Profile Activities for Class 2 and Class 3')
plt.legend(loc='lower left')
```



Out[383]: <matplotlib.legend.Legend at 0x1652a931d68>

In [ ]: