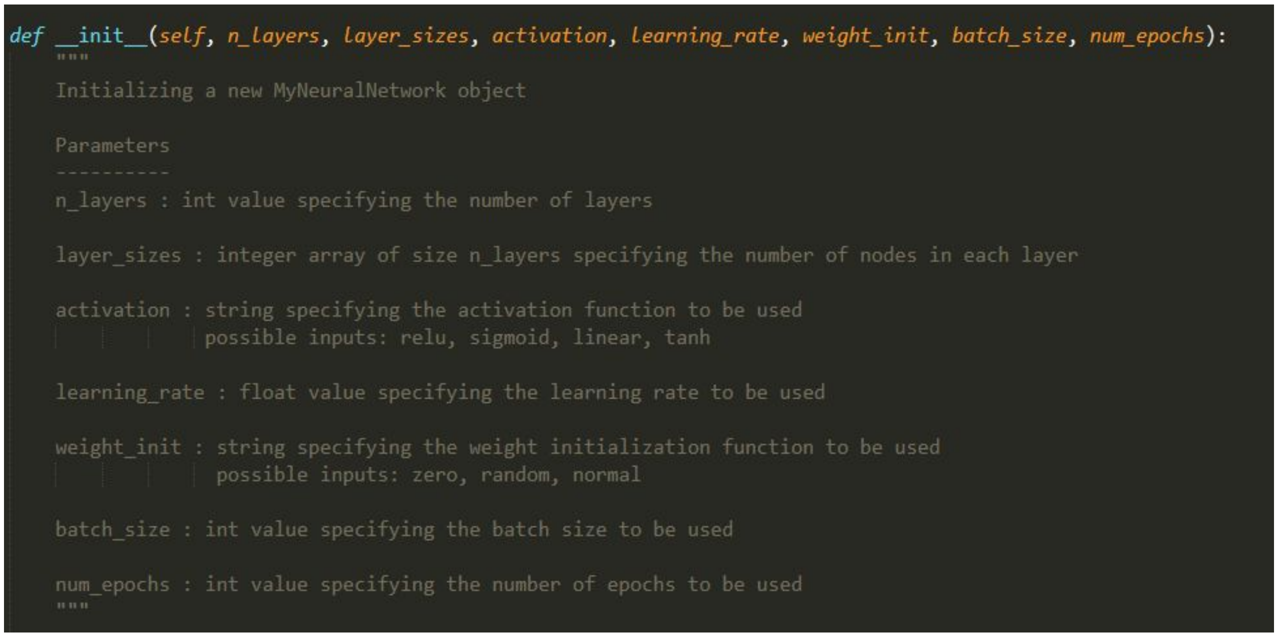
ML ASSIGNMENT-3 REPORT

Q1. You have to implement a general algorithm for Neural Networks. You can only use the NumPy library. Use the attached Q1.py for implementing the algorithm.
1. The network should have the following parameters
n layers: Number of Layers (int)
layer sizes: an array of size n layers which contains the number of nodes in each layer(array of int)
activation: Activation function to be used (string)
learning rate: the learning rate to be used (float)
weight init: initialisation function to be used
batch size: batch size to be used (int)
num epochs: number of epochs to be used (int)



```
def __init__(self, n_layers, layer_sizes, activation, learning_rate, weight_init, batch_size, num_epochs):
    """
    Initializing a new MyNeuralNetwork object

    Parameters
    ----------
    n_layers : int value specifying the number of layers

    layer_sizes : integer array of size n_layers specifying the number of nodes in each layer

    activation : string specifying the activation function to be used
                 possible inputs: relu, sigmoid, linear, tanh

    learning_rate : float value specifying the learning rate to be used

    weight_init : string specifying the weight initialization function to be used
                  possible inputs: zero, random, normal

    batch_size : int value specifying the batch size to be used

    num_epochs : int value specifying the number of epochs to be used
    """
```

DONE

2. Implement the following activation functions with their gradient calculation too:
ReLU,
sigmoid,
linear,
Tanh,
Softmax.
DONE

Ishan Kapur 2018042

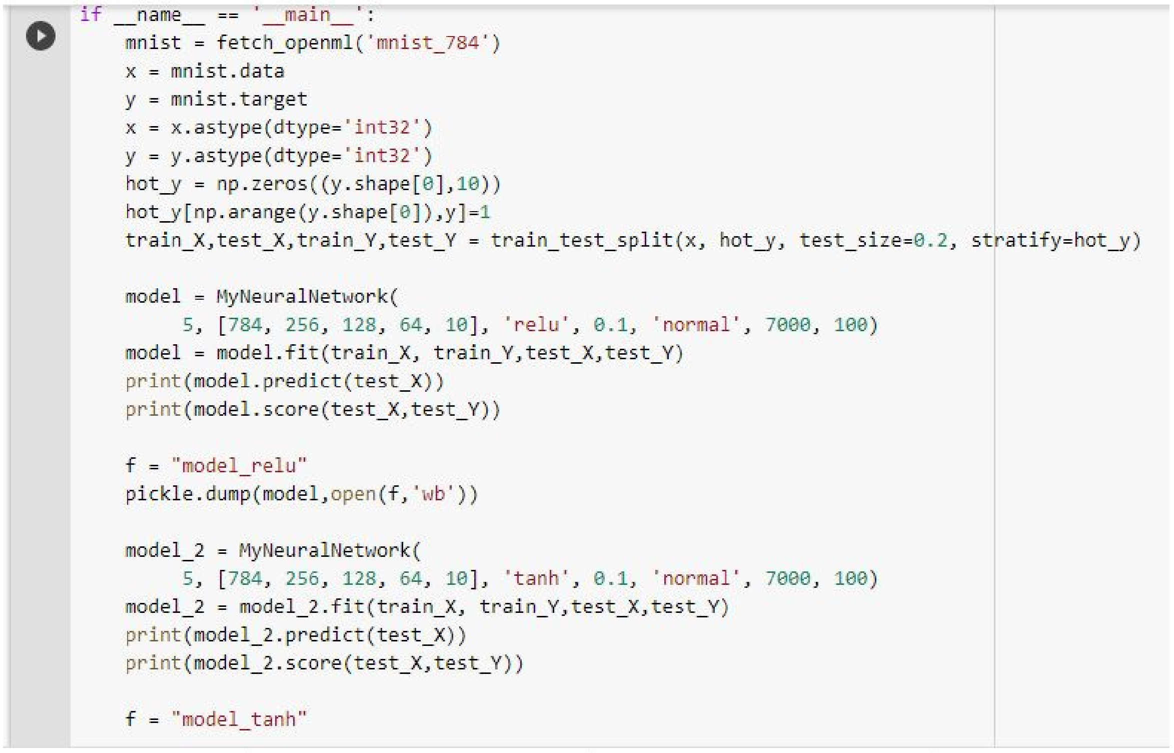
3. Implement the following weight initialisation techniques for the hidden layers:
zero: Zero initialisation
random: Random initialisation with a scaling factor of 0.01
normal: Normal(0,1) initialization with a scaling factor of 0.01

DONE

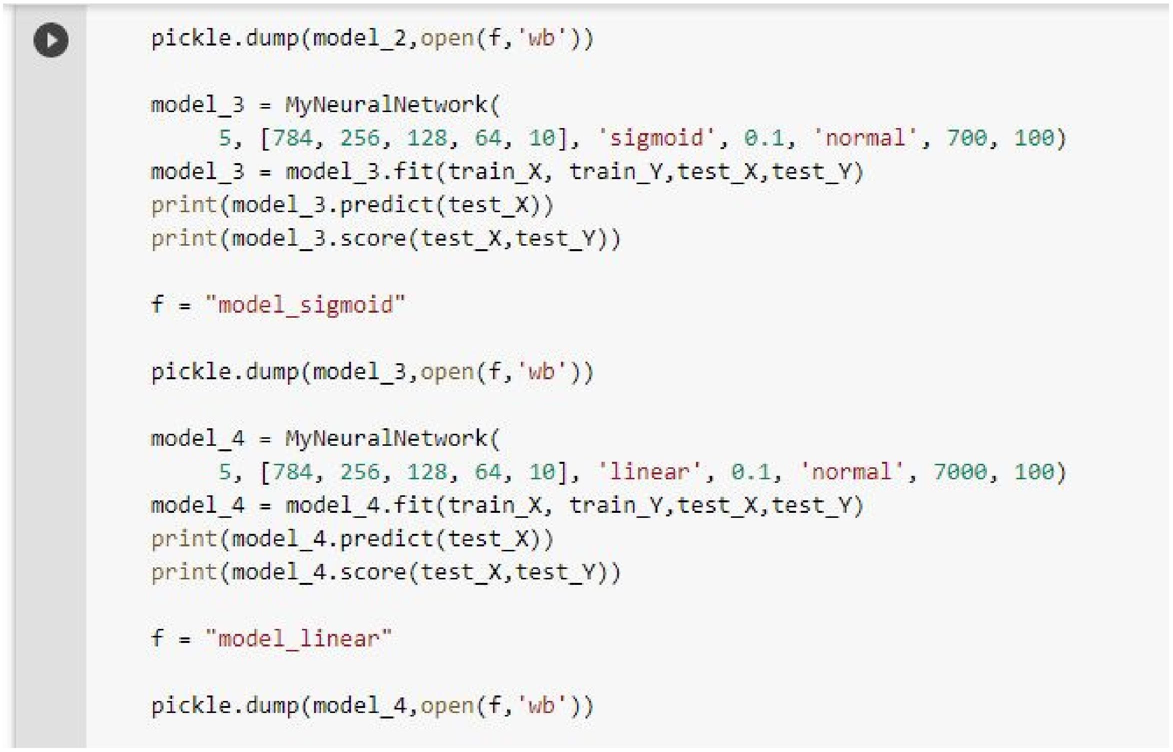
4. Implement the following functions with bias=0 and cross-entropy loss as the loss function. You can create other helper functions too.
fit(): accepts input data & input labels and trains a new model
predict proba(): takes input data and returns the class-wise probability
predict(): takes input data and returns the prediction using the trained model
score(): takes input data and their labels and returns the accuracy of the model

DONE

Q2. Use the MNIST dataset to train and test the neural network model created in Question 1 ONLY. Use the training dataset for calculating the training error and test dataset for calculating the testing error.
1. Use the following architecture [#input, 256, 128, 64, #output], learning rate=0.1,and number of epochs=100. Use normal weight initialisation as defined in the first question. Save the trained model weights separately for each activation function defined above and report the test accuracy.

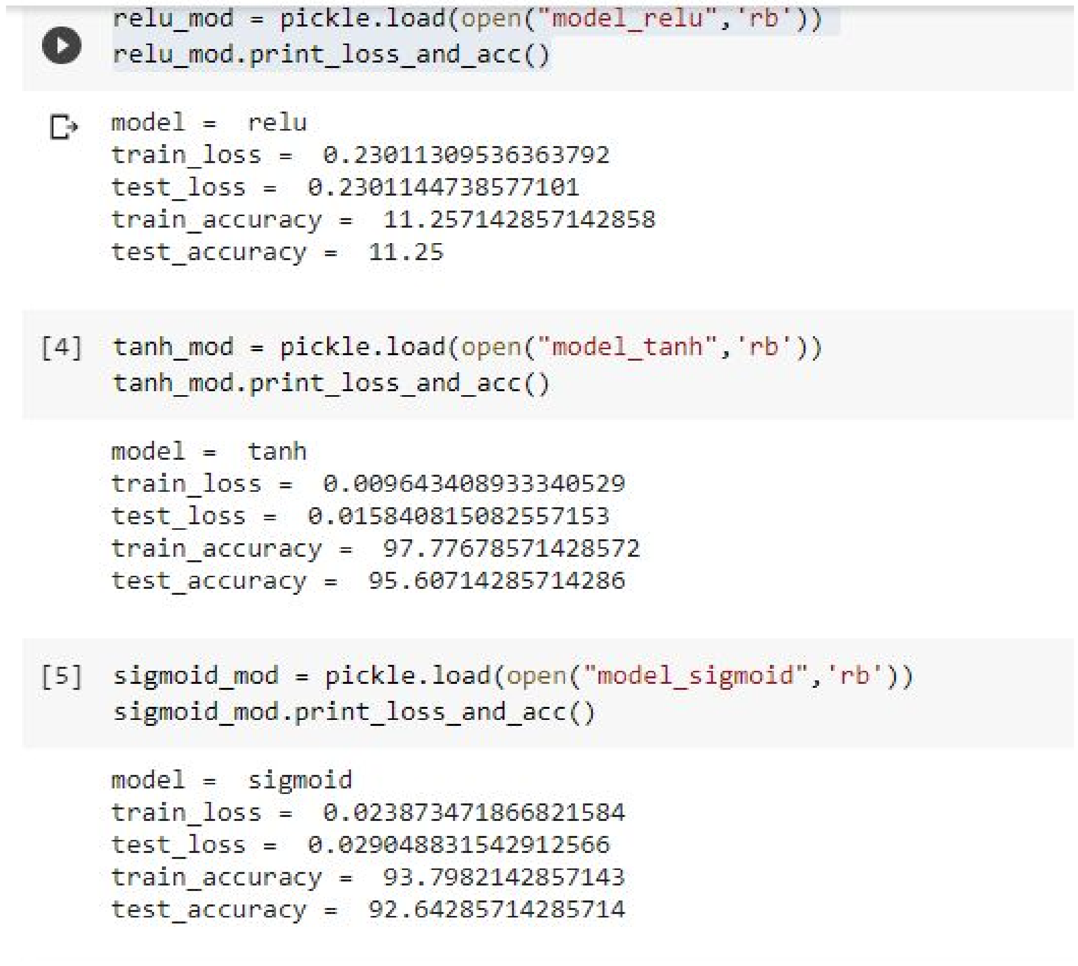


```
if __name__ == '__main__':
    mnist = fetch_openml('mnist_784')
    x = mnist.data
    y = mnist.target
    x = x.astype(dtype='int32')
    y = y.astype(dtype='int32')
    hot_y = np.zeros((y.shape[0],10))
    hot_y[np.arange(y.shape[0]),y]=1
    train_X,test_X,train_Y,test_Y = train_test_split(x, hot_y, test_size=0.2, stratify=hot_y)

    model = MyNeuralNetwork(
        5, [784, 256, 128, 64, 10], 'relu', 0.1, 'normal', 7000, 100)
    model = model.fit(train_X, train_Y,test_X,test_Y)
    print(model.predict(test_X))
    print(model.score(test_X,test_Y))

    f = "model_relu"
    pickle.dump(model,open(f,'wb'))

    model_2 = MyNeuralNetwork(
        5, [784, 256, 128, 64, 10], 'tanh', 0.1, 'normal', 7000, 100)
    model_2 = model_2.fit(train_X, train_Y,test_X,test_Y)
    print(model_2.predict(test_X))
    print(model_2.score(test_X,test_Y))

    f = "model_tanh"
```



```
pickle.dump(model_2,open(f,'wb'))

model_3 = MyNeuralNetwork(
     5, [784, 256, 128, 64, 10], 'sigmoid', 0.1, 'normal', 700, 100)
model_3 = model_3.fit(train_X, train_Y,test_X,test_Y)
print(model_3.predict(test_X))
print(model_3.score(test_X,test_Y))

f = "model_sigmoid"

pickle.dump(model_3,open(f,'wb'))

model_4 = MyNeuralNetwork(
     5, [784, 256, 128, 64, 10], 'linear', 0.1, 'normal', 7000, 100)
model_4 = model_4.fit(train_X, train_Y,test_X,test_Y)
print(model_4.predict(test_X))
print(model_4.score(test_X,test_Y))

f = "model_linear"

pickle.dump(model_4,open(f,'wb'))
```

ACCURACIES USING LOSS



```
relu_mod = pickle.load(open("model_relu",'rb'))
relu_mod.print_loss_and_acc()
```

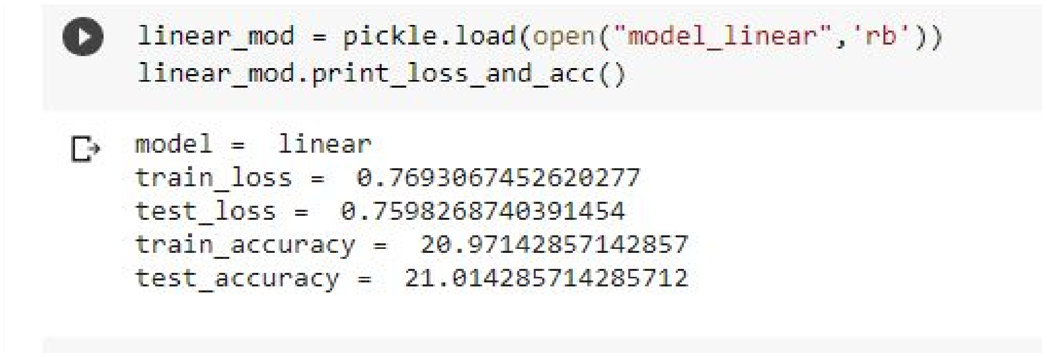
```
model =  relu
train_loss =  0.23011309536363792
test_loss =  0.2301144738577101
train_accuracy =  11.257142857142858
test_accuracy =  11.25
```

```
[4]  tanh_mod = pickle.load(open("model_tanh",'rb'))
     tanh_mod.print_loss_and_acc()
```

```
model =  tanh
train_loss =  0.009643408933340529
test_loss =  0.015840815082557153
train_accuracy =  97.77678571428572
test_accuracy =  95.60714285714286
```

```
[5]  sigmoid_mod = pickle.load(open("model_sigmoid",'rb'))
     sigmoid_mod.print_loss_and_acc()
```

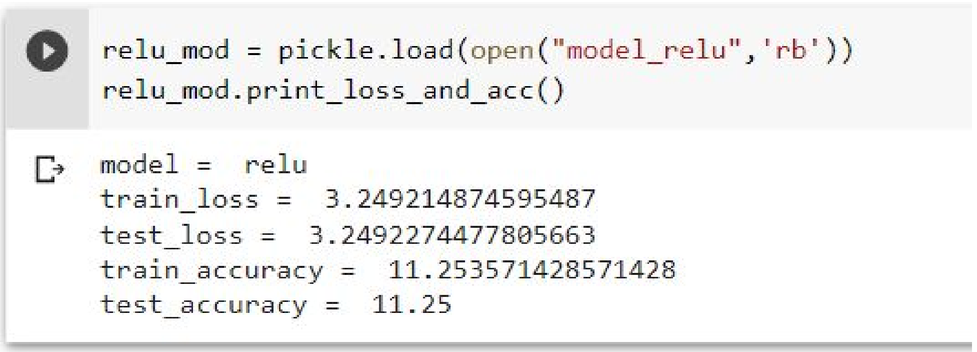
```
model =  sigmoid
train_loss =  0.023873471866821584
test_loss =  0.029048831542912566
train_accuracy =  93.7982142857143
test_accuracy =  92.64285714285714
```



```
linear_mod = pickle.load(open("model_linear",'rb'))
linear_mod.print_loss_and_acc()
```

```
model =  linear
train_loss =  0.7693067452620277
test_loss =  0.7598268740391454
train_accuracy =  20.97142857142857
test_accuracy =  21.014285714285712
```

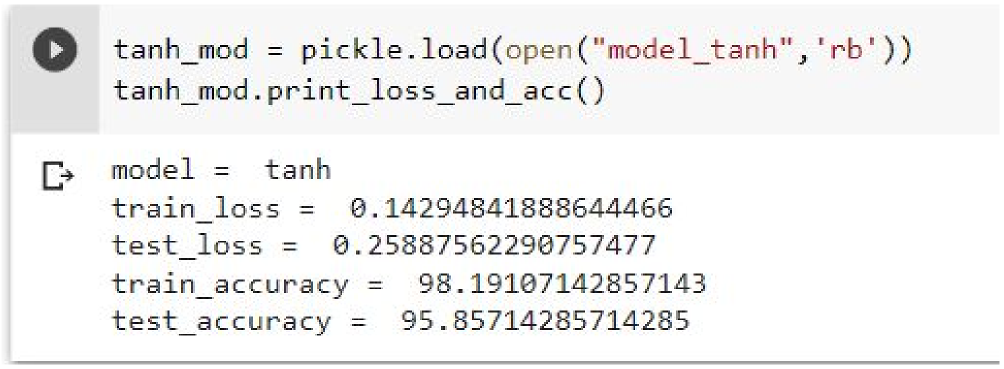
ACCURACIES OF MODELS using LOSS_2:
RELU



```
relu_mod = pickle.load(open("model_relu",'rb'))
relu_mod.print_loss_and_acc()
```

```
model =  relu
train_loss =  3.249214874595487
test_loss =  3.2492274477805663
train_accuracy =  11.253571428571428
test_accuracy =  11.25
```

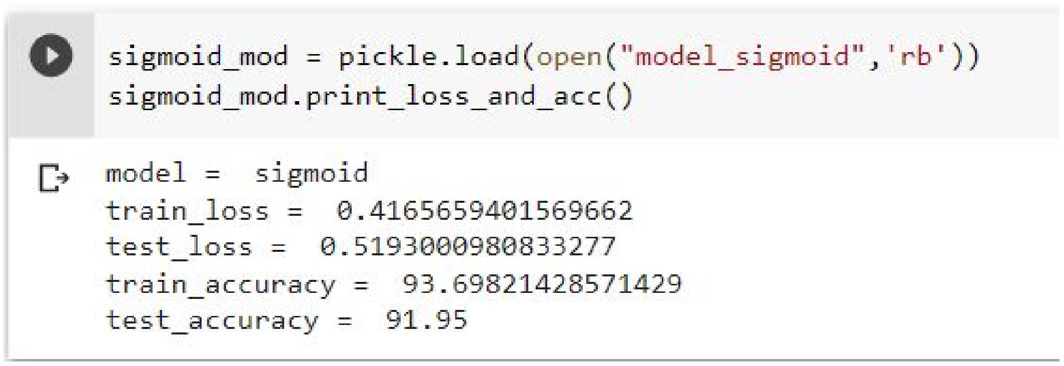
TANH



```
tanh_mod = pickle.load(open("model_tanh",'rb'))
tanh_mod.print_loss_and_acc()
```

```
model =  tanh
train_loss =  0.14294841888644466
test_loss =  0.25887562290757477
train_accuracy =  98.19107142857143
test_accuracy =  95.85714285714285
```

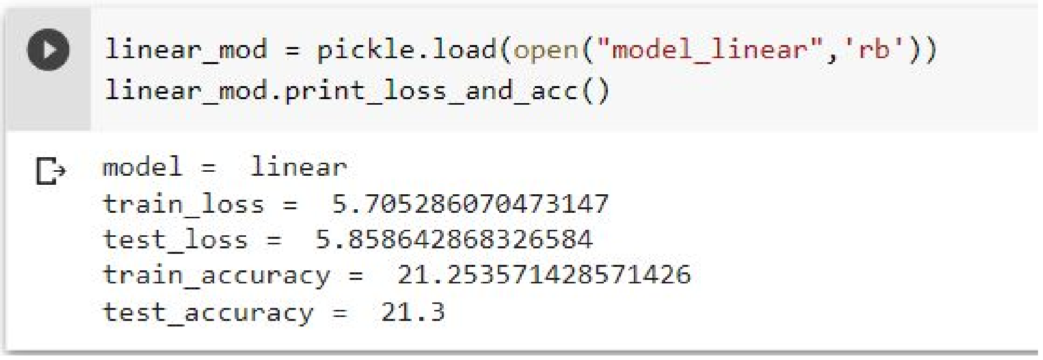
SIGMOID



```
sigmoid_mod = pickle.load(open("model_sigmoid",'rb'))
sigmoid_mod.print_loss_and_acc()
```

```
model =  sigmoid
train_loss =  0.4165659401569662
test_loss =  0.5193000980833277
train_accuracy =  93.69821428571429
test_accuracy =  91.95
```

LINEAR



```
linear_mod = pickle.load(open("model_linear",'rb'))
linear_mod.print_loss_and_acc()
```

```
model =  linear
train_loss =  5.705286070473147
test_loss =  5.858642868326584
train_accuracy =  21.253571428571426
test_accuracy =  21.3
```

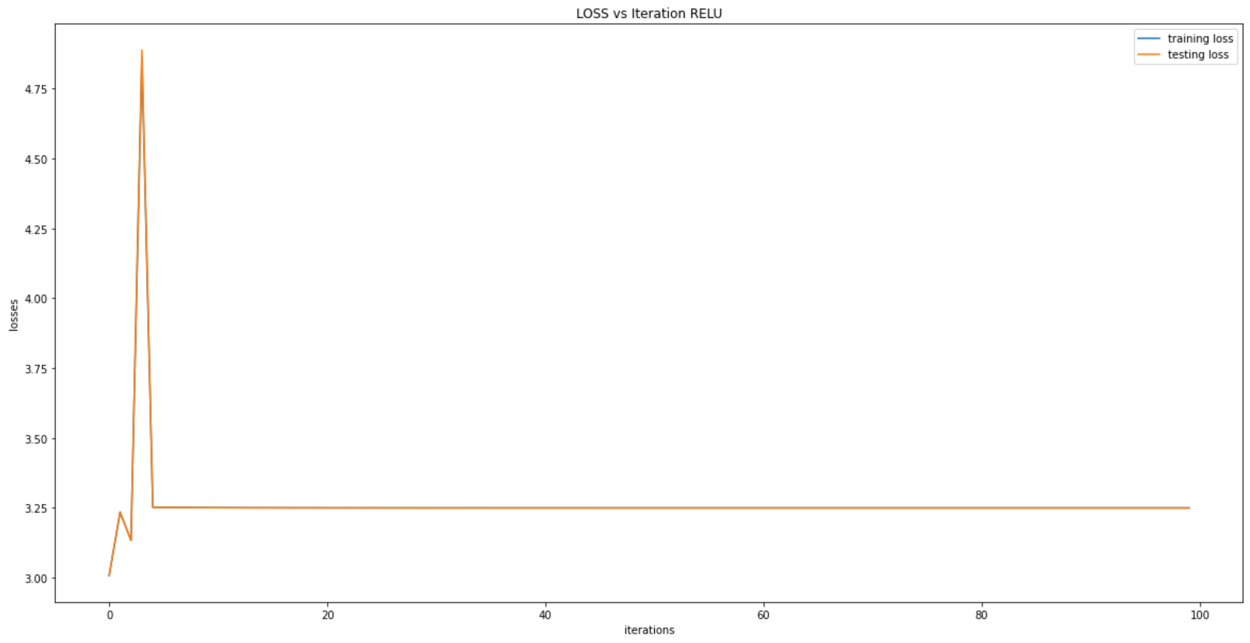
Ishan Kapur 2018042

Scratch code	reLU	tanh	sigmoid	linear
training	11.2578	97.76	93.79	20.97
testing	11.25	95.60	92.64	21.014

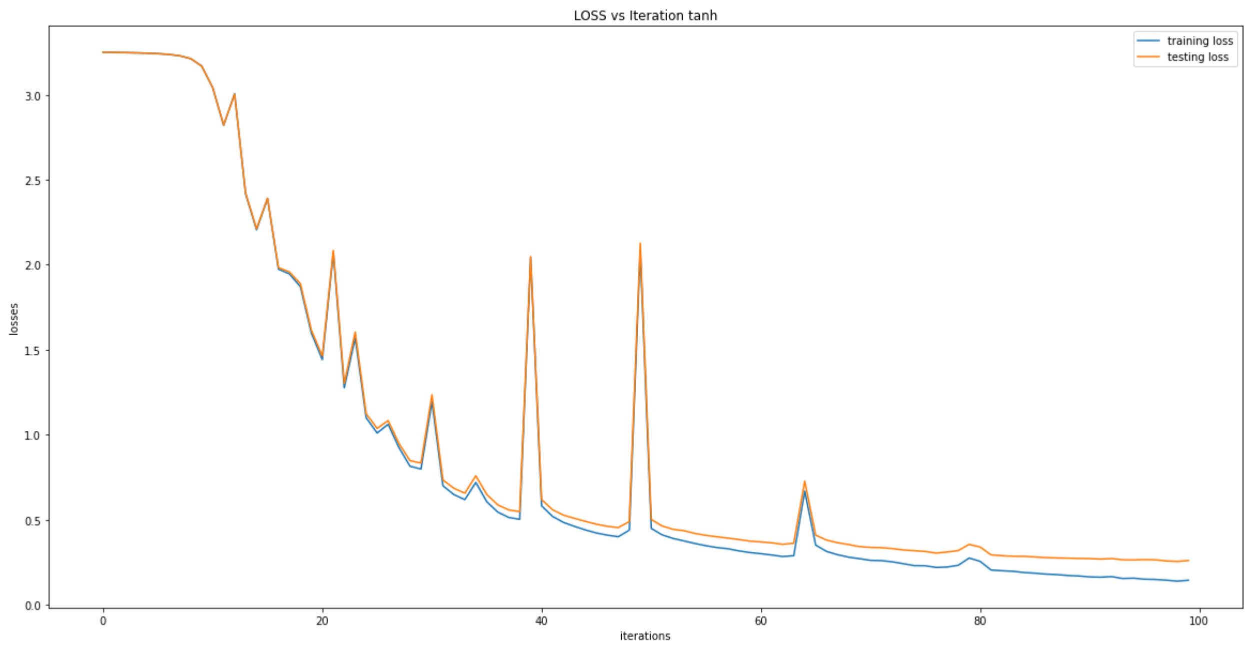
2. Plot training error vs epoch curve and testing error vs epoch curve for ReLU, sigmoid, linear and tanh activation function. Finally, you should have 4 graphs for the 4 activation functions.

Using LOSS_2

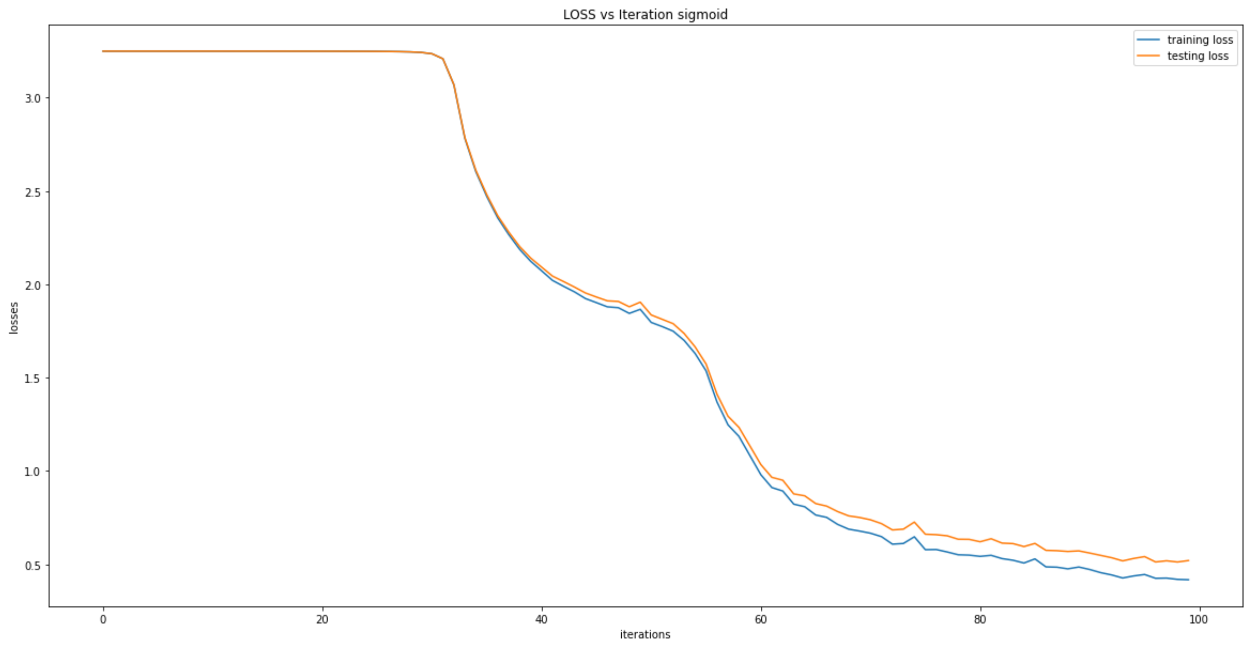
ReLU:



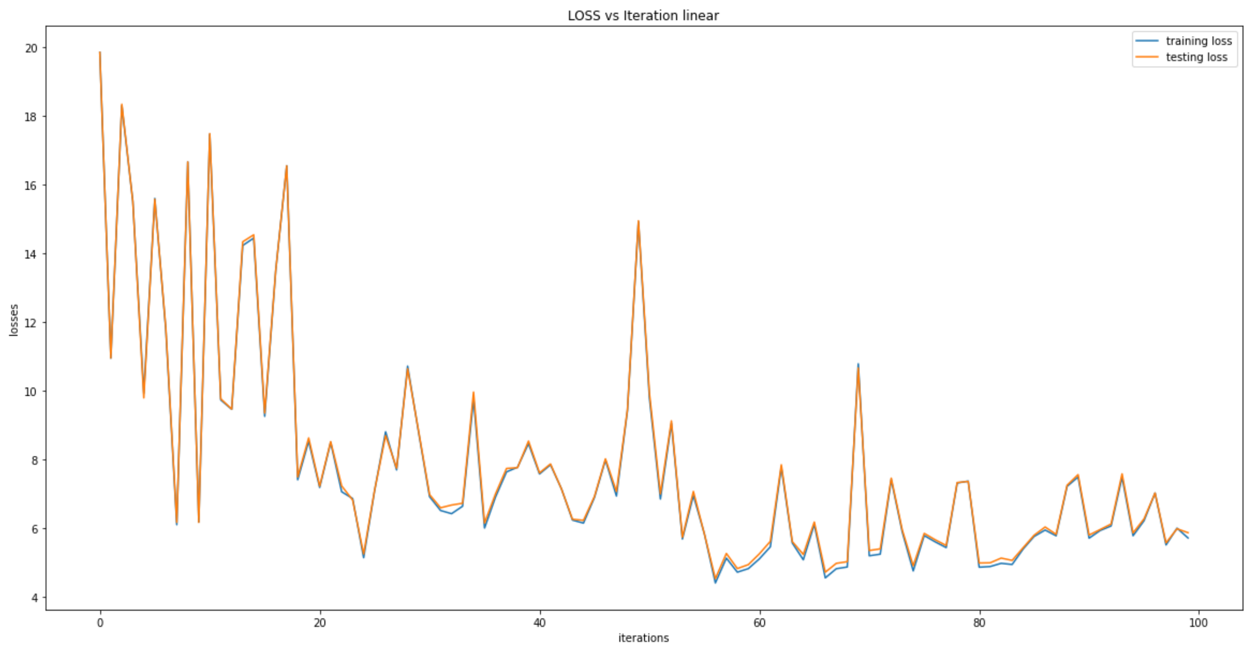
Tanh



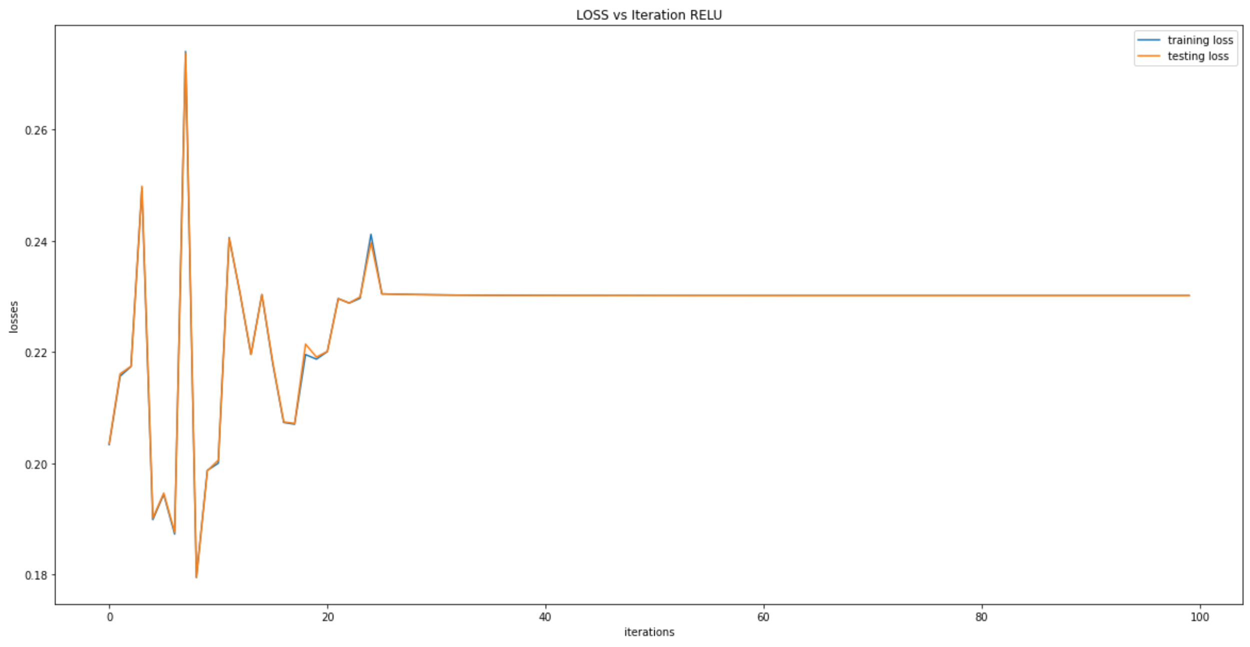
Sigmoid:



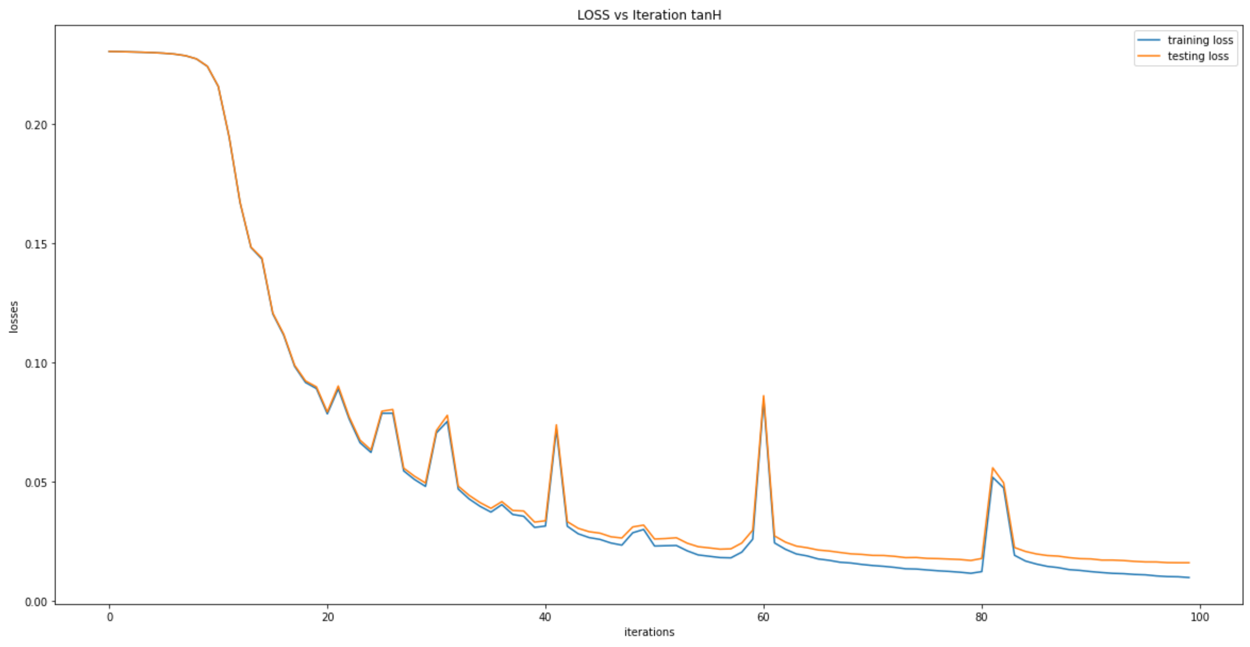
Linear:



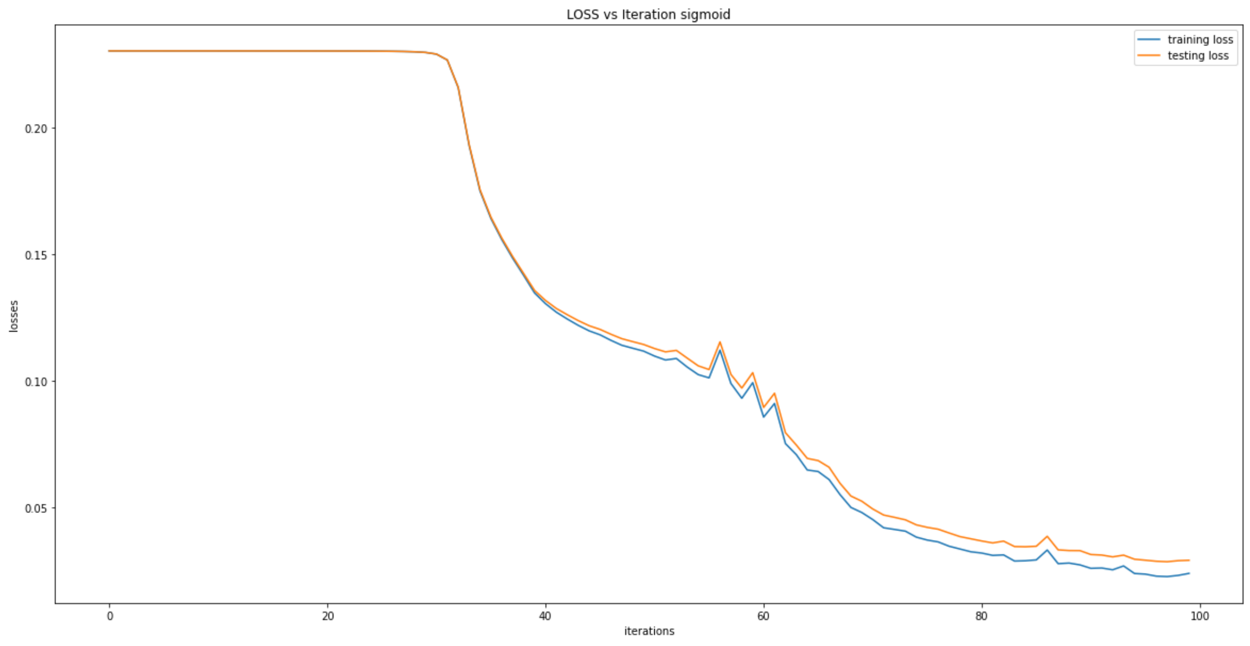
USING LOSS-RELU



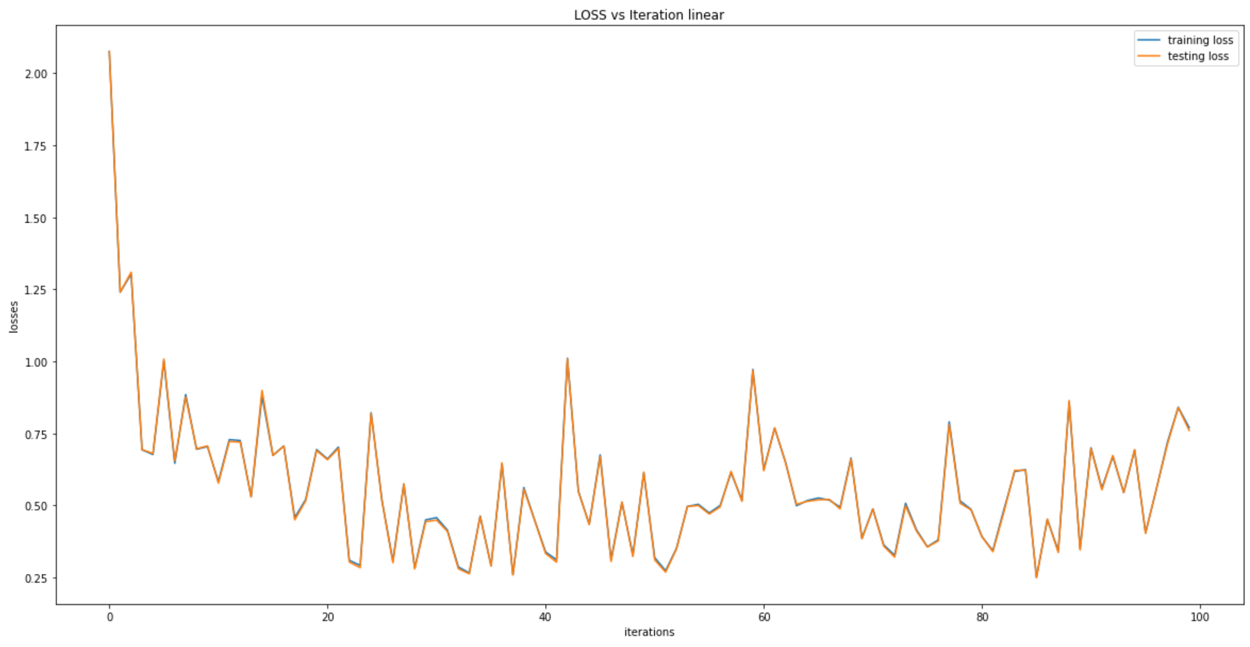
TANH



SIGMOID



Linear



3. In every case, what should be the activation function for the output layer? Give reasons to support your answer.

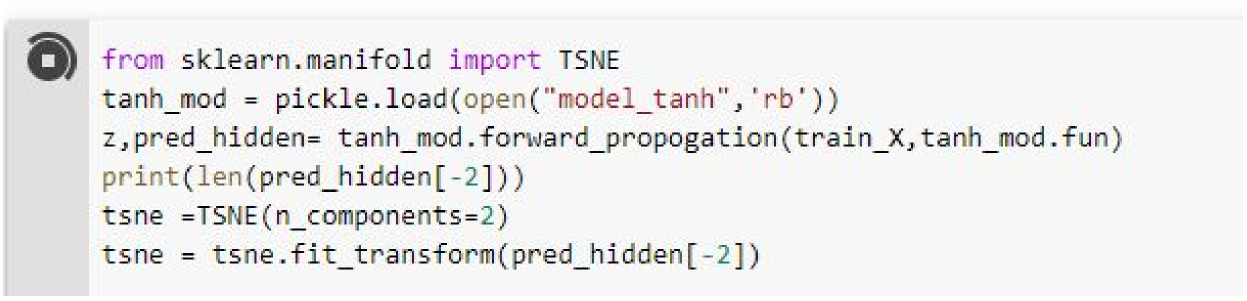
The activation function in the output layer of should be **SOFTMAX** in this case. The reason are as follows:

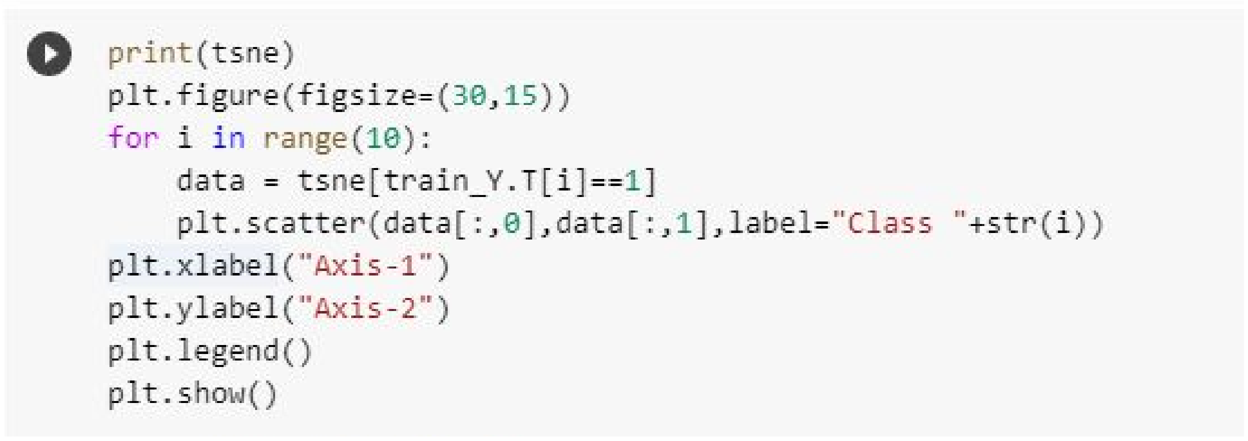
1) As we are using cross entropy loss in the classification problem, we know that the predicted values must lie between 0 and 1. Thus the best two functions which reduce the the Z from last hidden layer to between 0 and 1 are sigmoid and softmax.
2) Now as this problem, is a multiclass problem, one more rule has to be satisfied that is sum of all probabilities should be equal to 1, so oyt of the two mentioned above, only one satisfies the rule that is SOFTMAX. If the problem would be of binary classification, then sigmoid works out as well because if you have one probability, another can be found out by (1-p).
3) So, **SOFTMAX** is chosen for the output layer of the above functions.

4. What is the total number of layers and the number of hidden layers in this case?
Ans: There are total of **5 layers, in which 3 are hidden, 1 is input layer and 1 is output layer.**

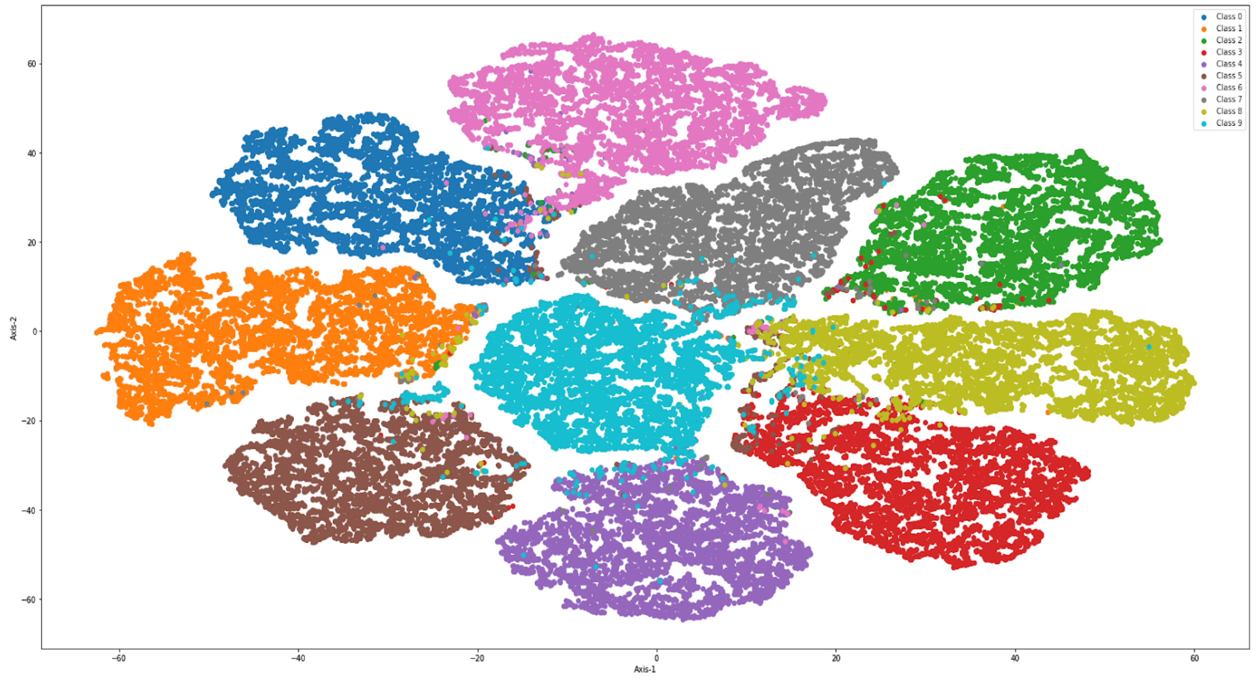
5. Visualise the final hidden layer features by creating tSNE plots for the model with the highest test accuracy. You can use sklearn for visualisation.
Ans:

Choose **tanh** because it has maximum accuracy (97%)

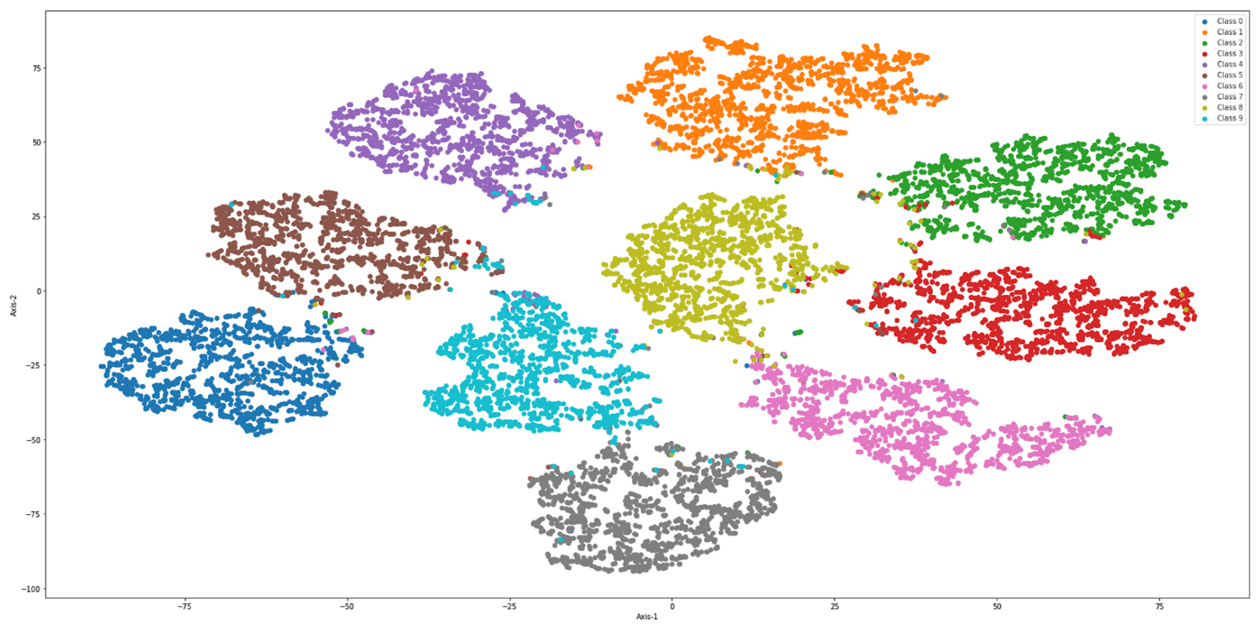

```
from sklearn.manifold import TSNE
tanh_mod = pickle.load(open("model_tanh",'rb'))
z,pred_hidden= tanh_mod.forward_propogation(train_X,tanh_mod.fun)
print(len(pred_hidden[-2]))
tsne =TSNE(n_components=2)
tsne = tsne.fit_transform(pred_hidden[-2])
```


```
print(tsne)
plt.figure(figsize=(30,15))
for i in range(10):
    data = tsne[train_Y.T[i]==1]
    plt.scatter(data[:,0],data[:,1],label="Class "+str(i))
plt.xlabel("Axis-1")
plt.ylabel("Axis-2")
plt.legend()
plt.show()
```

TRAIN_DATA

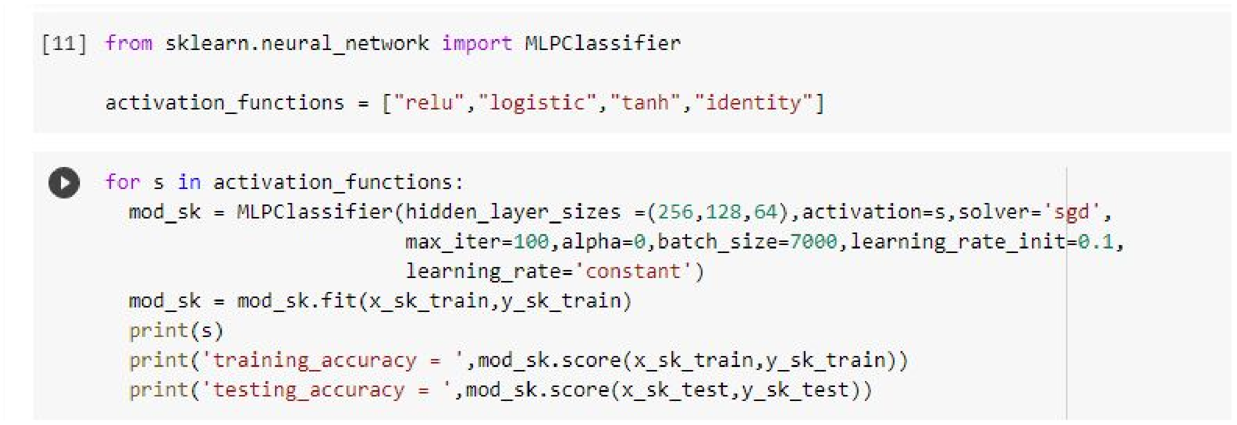


TEST_DATA



6. Now, use sklearn with the same parameters defined above and report the test

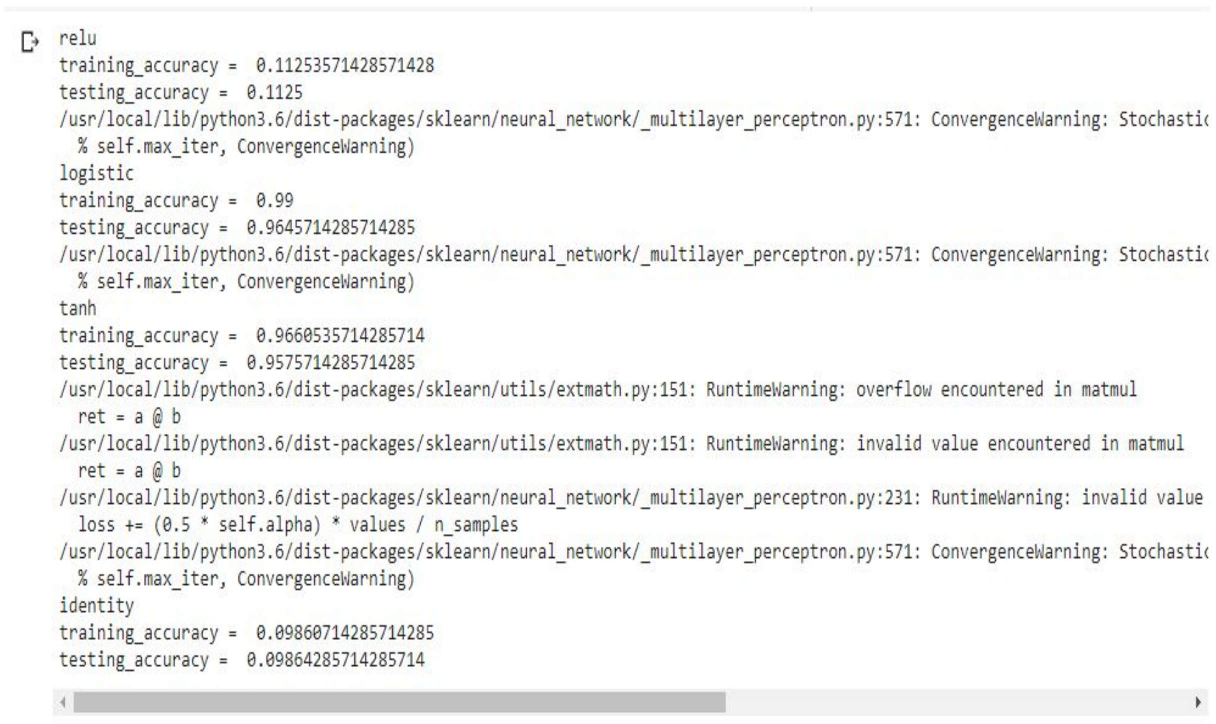
The accuracy obtained using ReLU, sigmoid, linear and tanh activation functions. Comment on the differences in accuracy, if any.



```
[11] from sklearn.neural_network import MLPClassifier

     activation_functions = ["relu","logistic","tanh","identity"]
```

```
for s in activation_functions:
  mod_sk = MLPClassifier(hidden_layer_sizes =(256,128,64),activation=s,solver='sgd',
                         max_iter=100,alpha=0,batch_size=7000,learning_rate_init=0.1,
                         learning_rate='constant')
  mod_sk = mod_sk.fit(x_sk_train,y_sk_train)
  print(s)
  print('training_accuracy = ',mod_sk.score(x_sk_train,y_sk_train))
  print('testing_accuracy = ',mod_sk.score(x_sk_test,y_sk_test))
```

Ishan Kapur 2018042



```
relu
training_accuracy =  0.11253571428571428
testing_accuracy =  0.1125
/usr/local/lib/python3.6/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:571: ConvergenceWarning: Stochastic
  % self.max_iter, ConvergenceWarning)
logistic
training_accuracy =  0.99
testing_accuracy =  0.9645714285714285
/usr/local/lib/python3.6/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:571: ConvergenceWarning: Stochastic
  % self.max_iter, ConvergenceWarning)
tanh
training_accuracy =  0.9660535714285714
testing_accuracy =  0.9575714285714285
/usr/local/lib/python3.6/dist-packages/sklearn/utils/extmath.py:151: RuntimeWarning: overflow encountered in matmul
  ret = a @ b
/usr/local/lib/python3.6/dist-packages/sklearn/utils/extmath.py:151: RuntimeWarning: invalid value encountered in matmul
  ret = a @ b
/usr/local/lib/python3.6/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:231: RuntimeWarning: invalid value
  loss += (0.5 * self.alpha) * values / n_samples
/usr/local/lib/python3.6/dist-packages/sklearn/neural_network/_multilayer_perceptron.py:571: ConvergenceWarning: Stochastic
  % self.max_iter, ConvergenceWarning)
identity
training_accuracy =  0.09860714285714285
testing_accuracy =  0.09864285714285714
```

Sk-learn	**reLU**	**tanh**	**sigmoid**	**linear**
training	11.253578	99	96.60	9.8607
testing	11.25	96.457	95.7571	9.86428

Now we see compare accuracy of ReLU:
We see that we almost receive the same results, because due to high learning rate it is over shooting. If we reduce learning rate to 0.01, ReLU give 95% accuracy, thus it is the case of high learning rate, so due to overshoot, they both predict only one class for all the inputs. Which can be different and depends on the weights assigned t initial steps randomly.

We see that in case of tanH, and sigmoid the accuracies are almost close with an error rate of 1 to 2% in training and testing, thus tanh and sigmoid works out.
For the case of linear we see that our model predicts 20% accuracy, while sklearn predicts 9.8%, its is due to overflow error. In sklearn, overflow error occurs for inear which leads to NaN weights thus all classes are predict only 1 class thus giving 9.8% accuracy, While in our linear model, to avoid overflow error we use clipping for linear to limit values from -1000 to 1000 thus our accuracy increases to 20%, though model stillo overshoot a lot and doesn't converge.

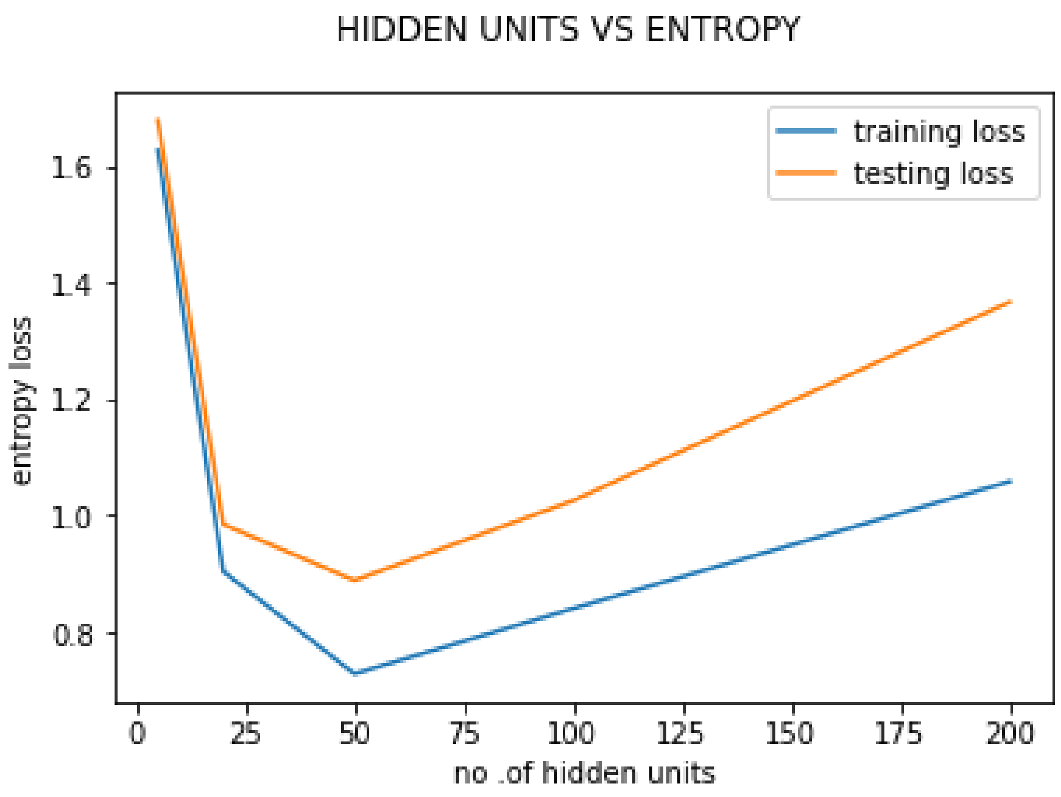
Q3.For the DATASET provided, use the following hyperparameter settings to train each neural network (using PyTorch) described below-

Parameter Value

# of Hidden units 4
Weight Initialization Random
Learning Rate 0.01

1. Hidden Units:
For the hyperparameters mentioned in the table above except the number of hidden units, train a single hidden layer neural network changing the value of the number of hidden units to 5, 20, 50, 100 and 200. Run the optimisation for 100 epochs each Time.

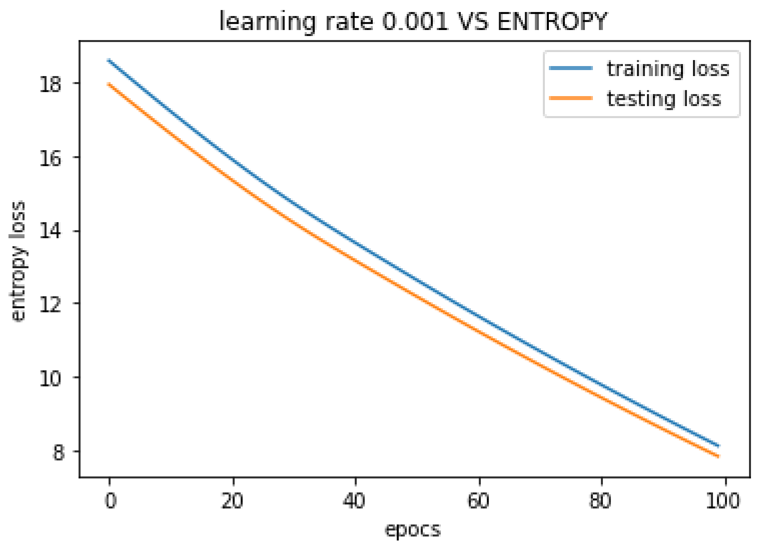
(a). (6) Plot the average training cross-entropy (sum of the cross-entropy terms over the training set divided by the total number of training examples) on the y-axis vs a number of hidden units on the x-axis. In the same figure, plot the average validation cross-entropy.

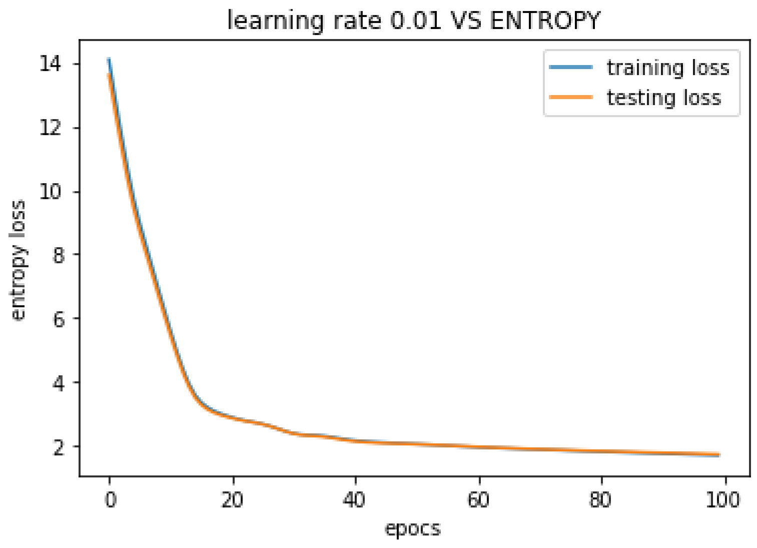


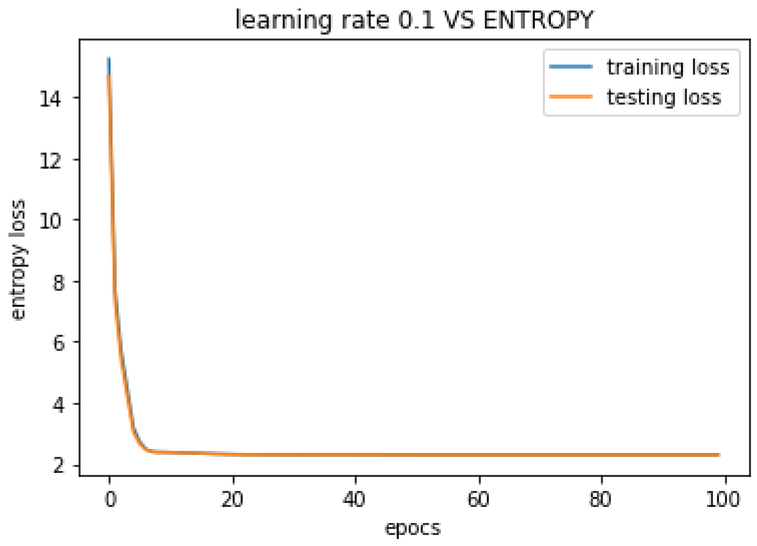
(b). Examine and comment on the plots of training and validation cross-entropy. What is the effect of changing the number of hidden units?
Ans: We can see from the graph that in the first half, hidden units from 0 to 50, the average entropy loss is decreasing as the number of hidden units is increasing. The minimum value of entropy loss achieved is at the 50 hidden units. Thus if we take hidden units less than 50, the model will be underfitting.

In the latter half, (more than 50 hidden units), we see average training and testing loss increasing, while it reaches 100 and 200 hidden units. It shows us that model is now overfitting due to the increase in the number of hidden layers.
Thus from the above examination, we see that as we increase the number of hidden layers, the model changes phases from being under fitted to perfect fit to overfitting.
We can also see that training loss is less than validation/ testing loss; we try to minimise training loss, not testing loss.

2. Learning Rate:
The hyperparameters mentioned in the table above except the learning rate, train a single hidden layer neural network changing the learning rate's value to 0.1, 0.01 and 0.001. Run the optimisation for 100 epochs each time.
(a). Plot the average training cross-entropy on the y-axis vs the number of epochs on the x-axis for the mentioned learning rates. In the same figure, plot the average validation cross-entropy loss. Make a separate figure for each learning rate.







(b). Examine and comment on the plots of training and validation cross-entropy. How does adjusting the learning rate affect the convergence of cross-entropy of each dataset?

Ans: First, as mentioned above, we can see that in all 3 graphs, training loss is less than validation/ testing loss, it is since we try to minimise training loss, not testing loss.

Now from the graph with learning rate 0.001, we see as compared to other graphs that entropy loss is 8 at final iteration as compared to graph with learning rate 0.01 which achieved that loss at 10th iteration; thus we can see that 0.001 is just so small to make the model converge. To make it converge, we will need more iterations,

Now for learning rate 0.1, we can see that the training loss of 2, which is achieved in 10th iteration is equal to at 100th iteration of 0.01 iteration, but later, there is not a change in the loss, which means that the model has converged. Thus it just a waste of computation time to run more iterations.

However, as eights are random 0.1 can become large enough to make the model overshoot, thus never achieving minimum.

So choosing 0.01 will be the optimal case of learning rate to be selected in this case.

Q4. Use the binary CIFAR 10 subset for this part.

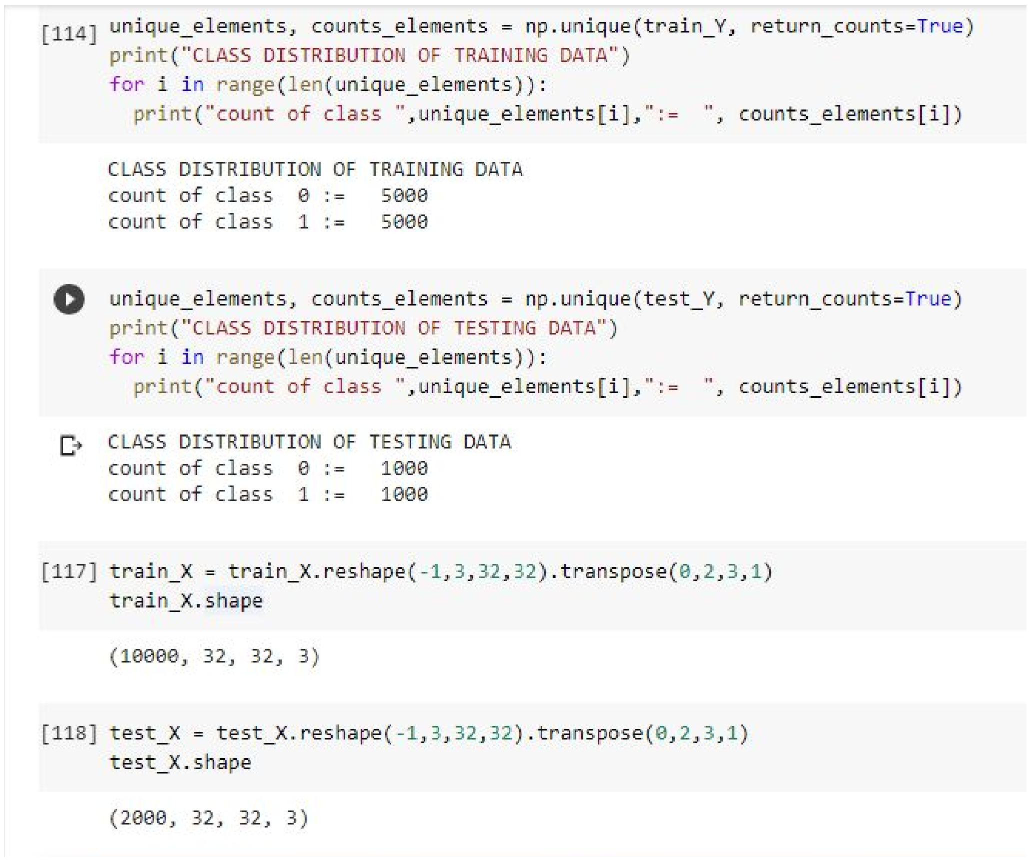
1. Conduct Exploratory Data Analysis (EDA) on the CIFAR-10 dataset. Report the class distribution.

Ans:

EDA is performed on the dataset as follows:

1) Checking the data distribution:

In the below screenshot, we had counted the number of output labels for training data. We found out that training data contains 10000 rows of data where 5000 corresponded to label 0 and 5000 to label 1.
The same has been done for test dataset, and we found out that out of 2000 samples, 1000 belongs to class 0 and 1000 to class 1. Thus, the dataset is perfectly balanced in terms of labels for both training and testing dataset; therefore, there is no need for any imbalance in data handling techniques like over or under-sampling.



```
[114] unique_elements, counts_elements = np.unique(train_Y, return_counts=True)
      print("CLASS DISTRIBUTION OF TRAINING DATA")
      for i in range(len(unique_elements)):
        print("count of class ",unique_elements[i],":=  ", counts_elements[i])
```

```
CLASS DISTRIBUTION OF TRAINING DATA
count of class  0 :=   5000
count of class  1 :=   5000
```

```
unique_elements, counts_elements = np.unique(test_Y, return_counts=True)
print("CLASS DISTRIBUTION OF TESTING DATA")
for i in range(len(unique_elements)):
  print("count of class ",unique_elements[i],":=  ", counts_elements[i])
```

```
CLASS DISTRIBUTION OF TESTING DATA
count of class  0 :=   1000
count of class  1 :=   1000
```

```
[117] train_X = train_X.reshape(-1,3,32,32).transpose(0,2,3,1)
      train_X.shape
```

```
(10000, 32, 32, 3)
```

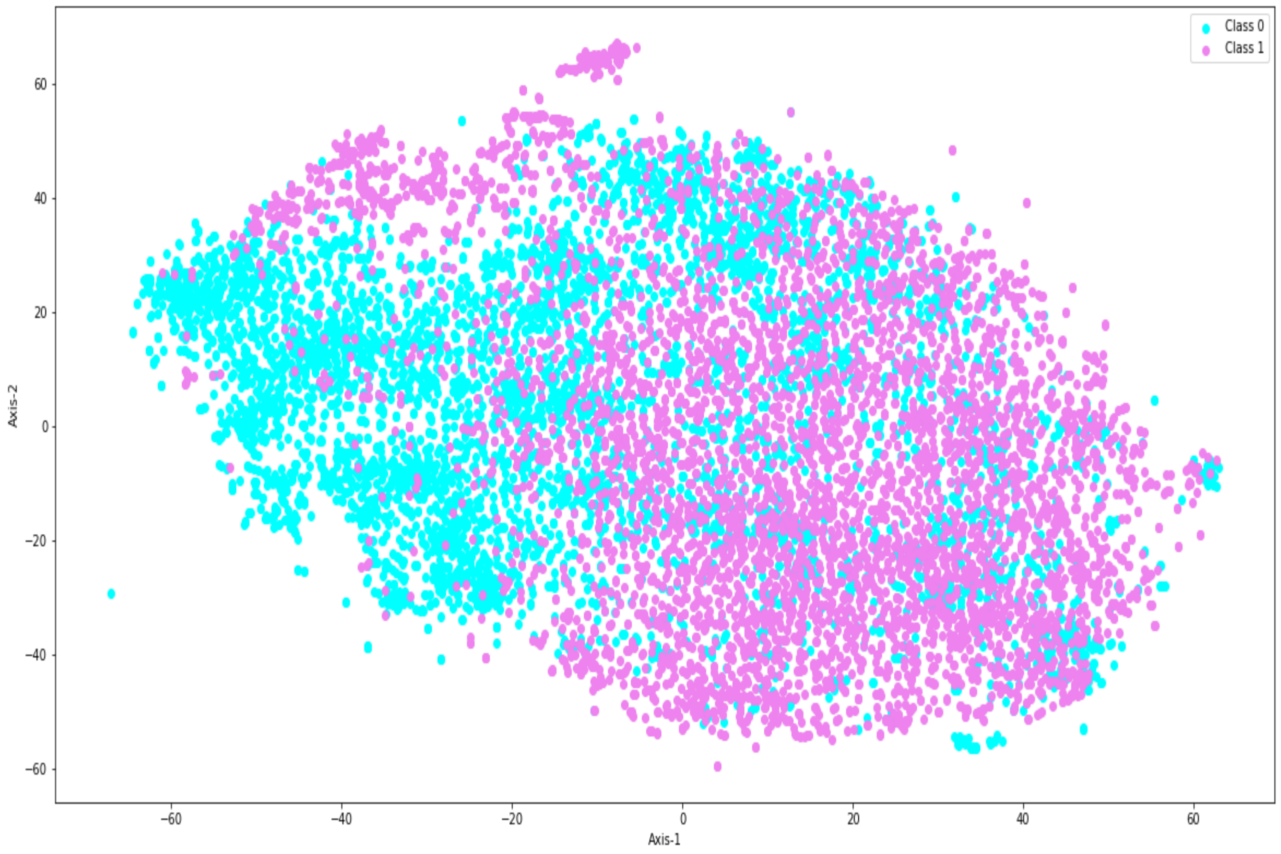
```
[118] test_X = test_X.reshape(-1,3,32,32).transpose(0,2,3,1)
      test_X.shape
```

```
(2000, 32, 32, 3)
```

After reshaping the data into images of 32X32X3 shape such that there are 32X32 pixels in the picture, each pixel corresponds to that pixel's RGB values, i.e. 3 (r,g,b) in this format.

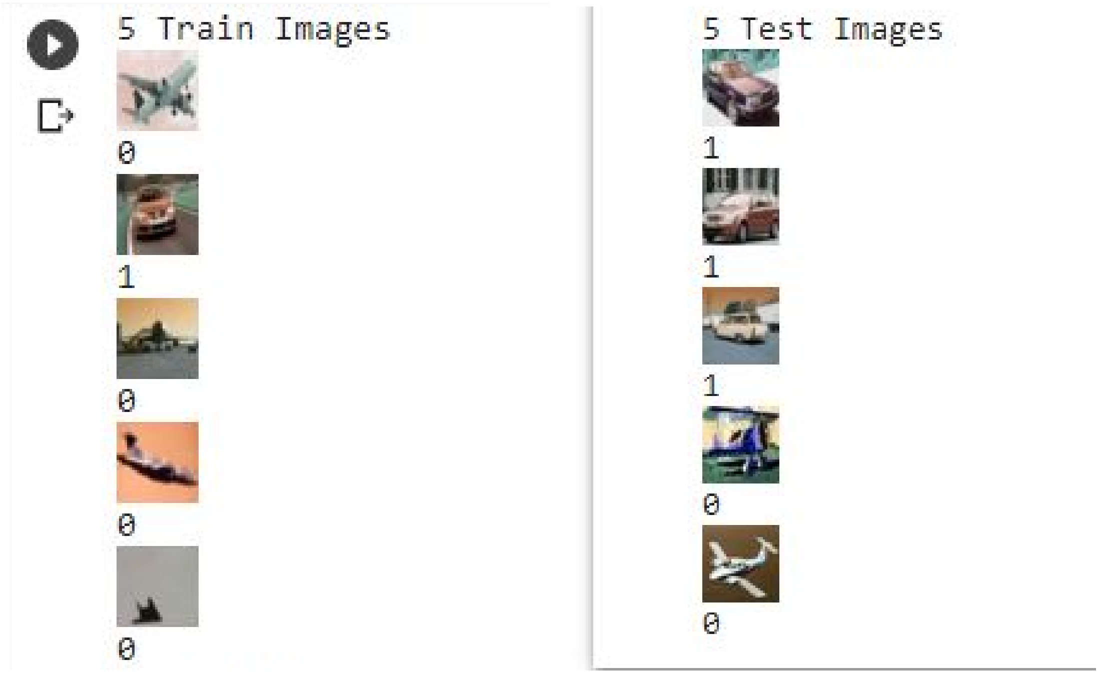
2) Visualisation of the dataset through TSNE:

Applied PCA with 0.



3) **Viewing the dataset after reshaping:**

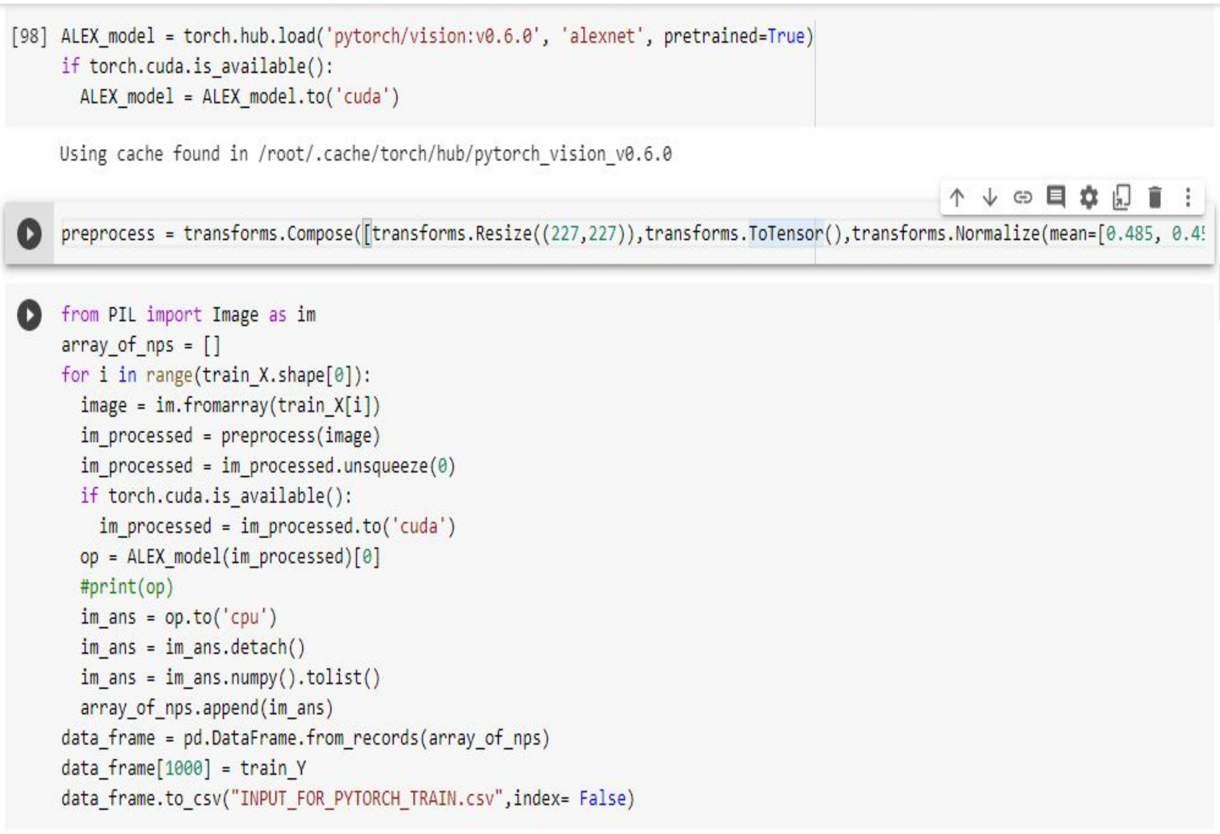
Printed 5 images from training data and test data, respectively.



```
5 Train Images
0
1
0
0
0
```

```
5 Test Images
1
1
1
0
0
```

2. Use the existing AlexNet Model from PyTorch (pre-trained on ImageNet) as a feature extractor for the images in the CIFAR subset. You should use the fc8 layer as the feature, which gives a 1000 dimensional feature vector for each image.

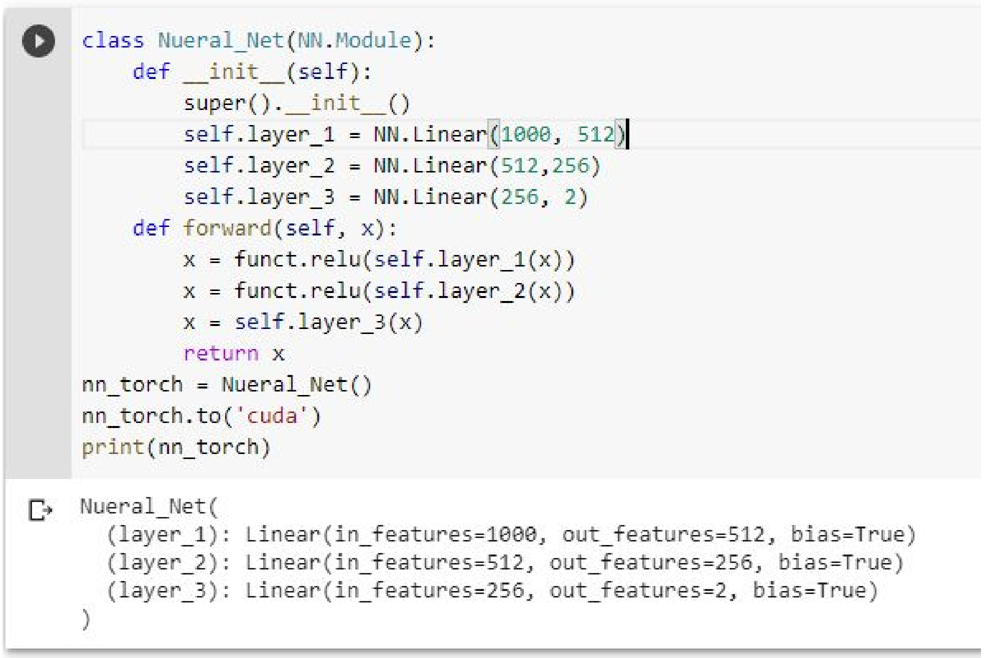


```
[98] ALEX_model = torch.hub.load('pytorch/vision:v0.6.0', 'alexnet', pretrained=True)
     if torch.cuda.is_available():
       ALEX_model = ALEX_model.to('cuda')

     Using cache found in /root/.cache/torch/hub/pytorch_vision_v0.6.0
```

```
preprocess = transforms.Compose([transforms.Resize((227,227)),transforms.ToTensor(),transforms.Normalize(mean=[0.485, 0.4
```

```
from PIL import Image as im
array_of_nps = []
for i in range(train_X.shape[0]):
  image = im.fromarray(train_X[i])
  im_processed = preprocess(image)
  im_processed = im_processed.unsqueeze(0)
  if torch.cuda.is_available():
    im_processed = im_processed.to('cuda')
  op = ALEX_model(im_processed)[0]
  #print(op)
  im_ans = op.to('cpu')
  im_ans = im_ans.detach()
  im_ans = im_ans.numpy().tolist()
  array_of_nps.append(im_ans)
data_frame = pd.DataFrame.from_records(array_of_nps)
data_frame[1000] = train_Y
data_frame.to_csv("INPUT_FOR_PYTORCH_TRAIN.csv",index= False)
```

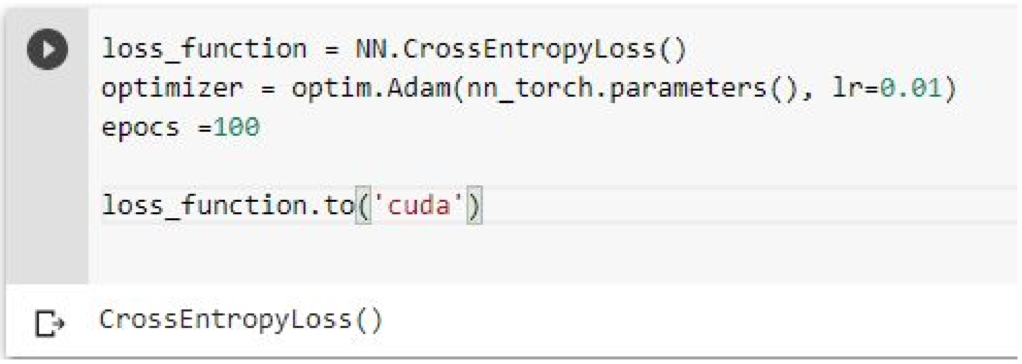
3. Train a Neural Network with 2 hidden layers of sizes 512 and 256, and use the fc8 layer as input to this Neural Network for the classification task.



```
class Nueral_Net(NN.Module):
    def __init__(self):
        super().__init__()
        self.layer_1 = NN.Linear(1000, 512)
        self.layer_2 = NN.Linear(512,256)
        self.layer_3 = NN.Linear(256, 2)
    def forward(self, x):
        x = funct.relu(self.layer_1(x))
        x = funct.relu(self.layer_2(x))
        x = self.layer_3(x)
        return x
nn_torch = Nueral_Net()
nn_torch.to('cuda')
print(nn_torch)
```

```
Nueral_Net(
  (layer_1): Linear(in_features=1000, out_features=512, bias=True)
  (layer_2): Linear(in_features=512, out_features=256, bias=True)
  (layer_3): Linear(in_features=256, out_features=2, bias=True)
)
```

Ishan Kapur 2018042



```
loss_function = NN.CrossEntropyLoss()
optimizer = optim.Adam(nn_torch.parameters(), lr=0.01)
epocs =100

loss_function.to('cuda')
```

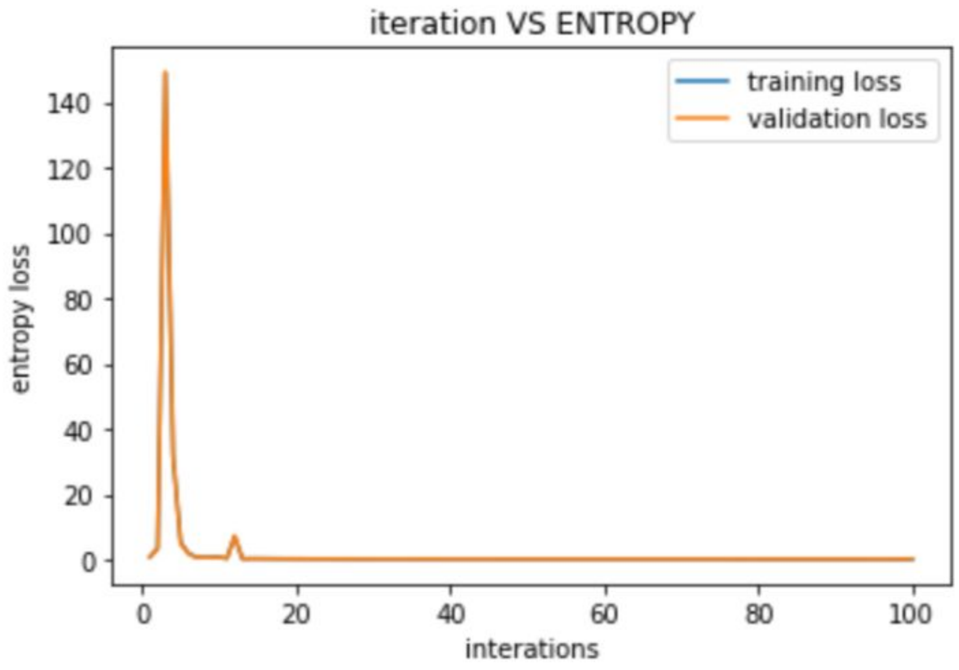
```
CrossEntropyLoss()
```

4. Report the test accuracy along with the confusion matrix and the ROC curve.

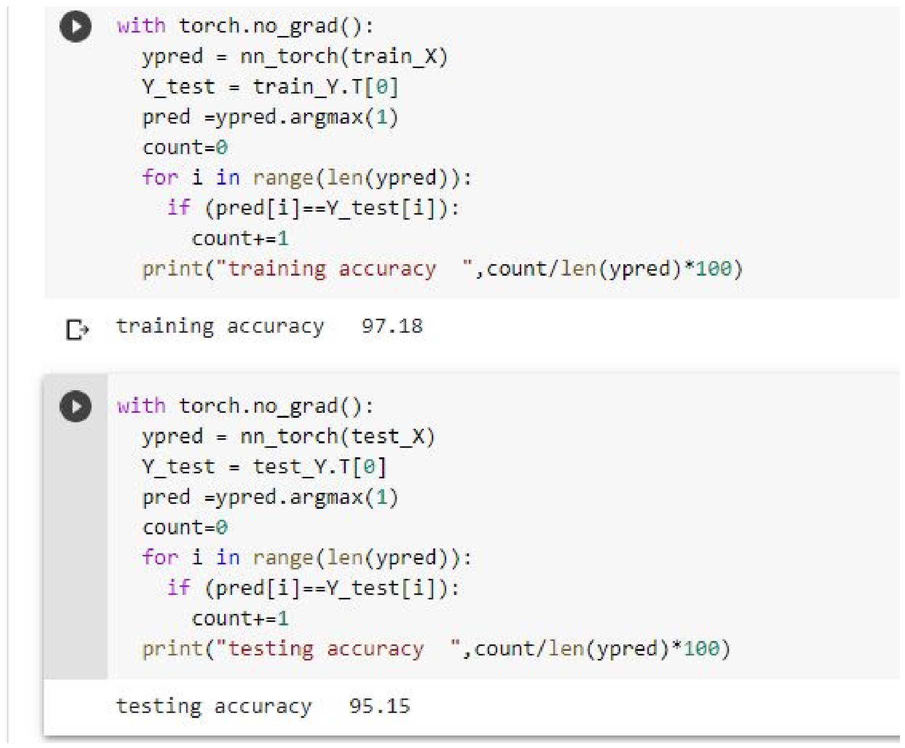
Training loss after 100 iterations: 0.0704
Test/ validation loss after 100 iterations: 0.1450

Graph of losses versus iterations are given below:

```
training loss :   tensor(0.0704, device='cuda:0')
test loss :   tensor(0.1450, device='cuda:0')
```



Training accuracy achieved: 97.18
Testing accuracy achieved: 95.15



```
with torch.no_grad():
  ypred = nn_torch(train_X)
  Y_test = train_Y.T[0]
  pred =ypred.argmax(1)
  count=0
  for i in range(len(ypred)):
    if (pred[i]==Y_test[i]):
      count+=1
  print("training accuracy  ",count/len(ypred)*100)
```

```
training accuracy   97.18
```

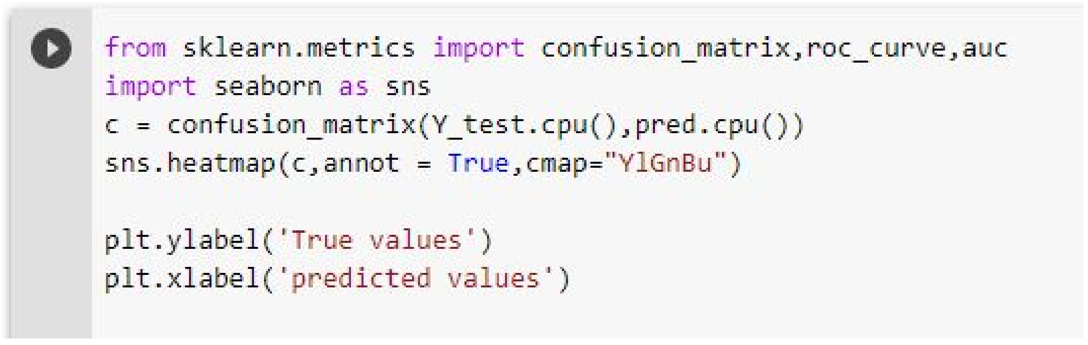
```
with torch.no_grad():
  ypred = nn_torch(test_X)
  Y_test = test_Y.T[0]
  pred =ypred.argmax(1)
  count=0
  for i in range(len(ypred)):
    if (pred[i]==Y_test[i]):
      count+=1
  print("testing accuracy  ",count/len(ypred)*100)
```

```
testing accuracy   95.15
```

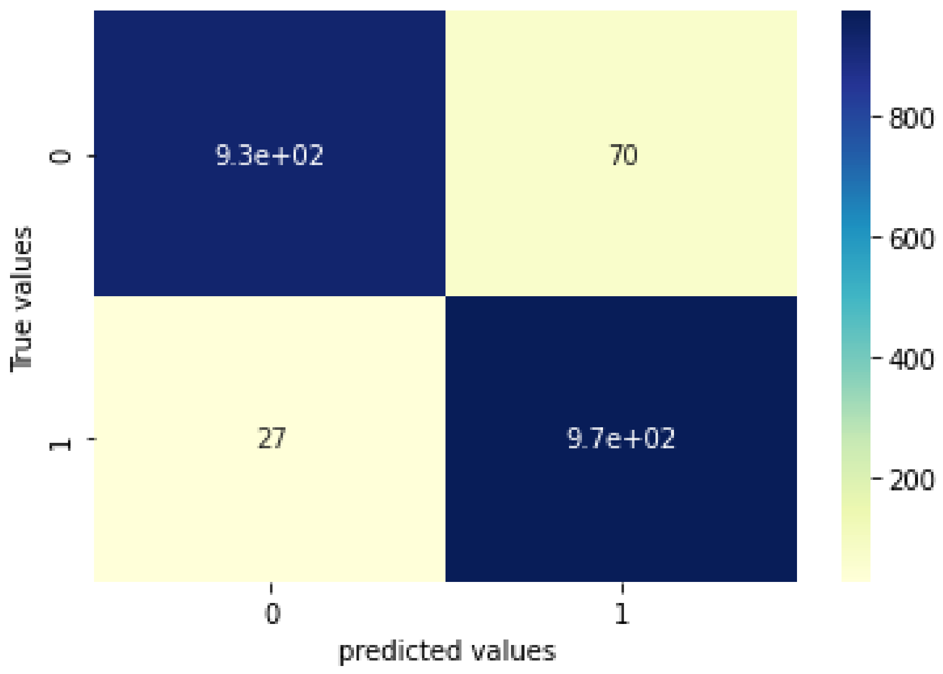
Confusion matrix in form heatmap and ROC curves are given below:

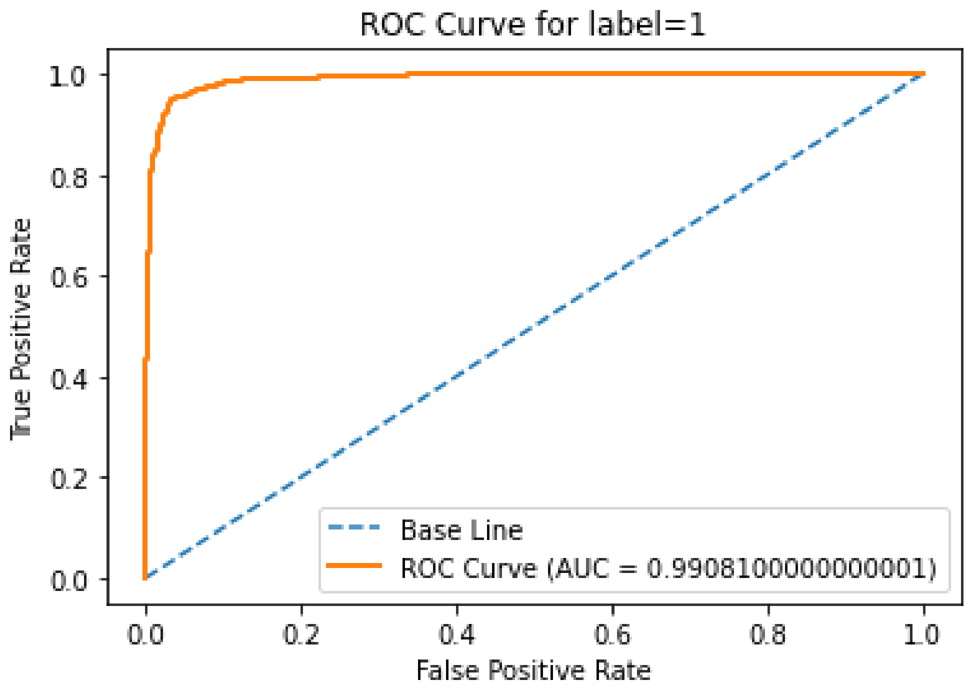
Confusion matrix:

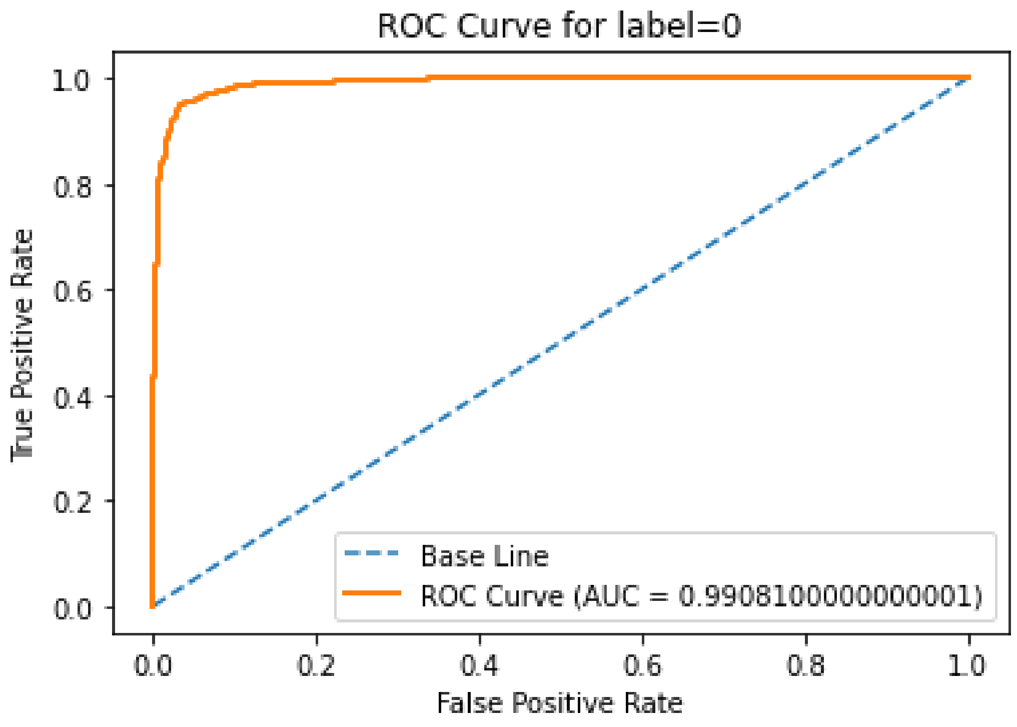
	Predicted values = 0	**Predicted values = 1**
True values = 0	932	70
True values =1	27	970



```
from sklearn.metrics import confusion_matrix,roc_curve,auc
import seaborn as sns
c = confusion_matrix(Y_test.cpu(),pred.cpu())
sns.heatmap(c,annot = True,cmap="YlGnBu")

plt.ylabel('True values')
plt.xlabel('predicted values')
```


9.3e+02
70
27
9.7e+02
True values
0
1
predicted values
800
600
400
200


ROC Curve for label=1
True Positive Rate
False Positive Rate
1.0
0.8
0.6
0.4
0.2
0.0
Base Line
ROC Curve (AUC = 0.9908100000000001)



Ishan Kapur 2018042