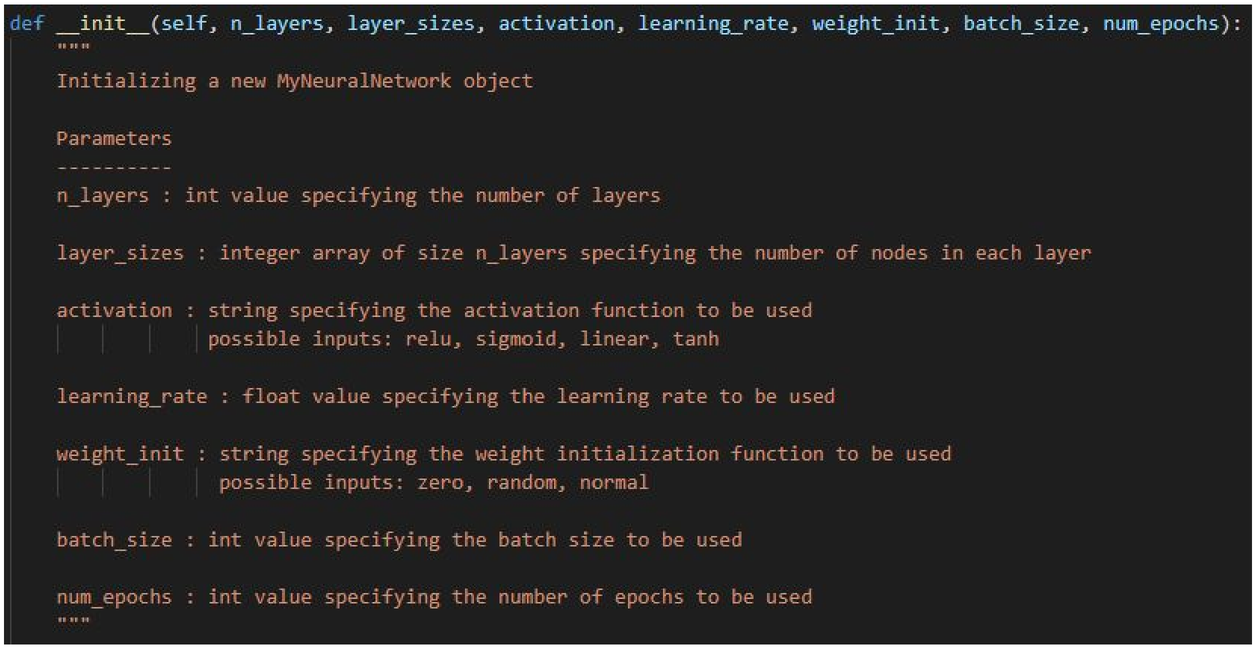
Q1. (55 points) You have to implement a general algorithm for Neural Networks. You can only use the NumPy library. Use the attached Q1.py for implementing the algorithm.

1. **(6) The network should have the following parameters**
   a. n layers: Number of Layers (int)
   b. layer sizes: An array of size n layers that contains the number of nodes in each layer. (array of int)
   c. activation: Activation function to be used (string)
   d. learning rate: the learning rate to be used (float)
   e. weight init: initialization function to be used
   f. batch size: batch size to be used (int)
   g. num epochs: number of epochs to be used (int)



```python
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
    """
```

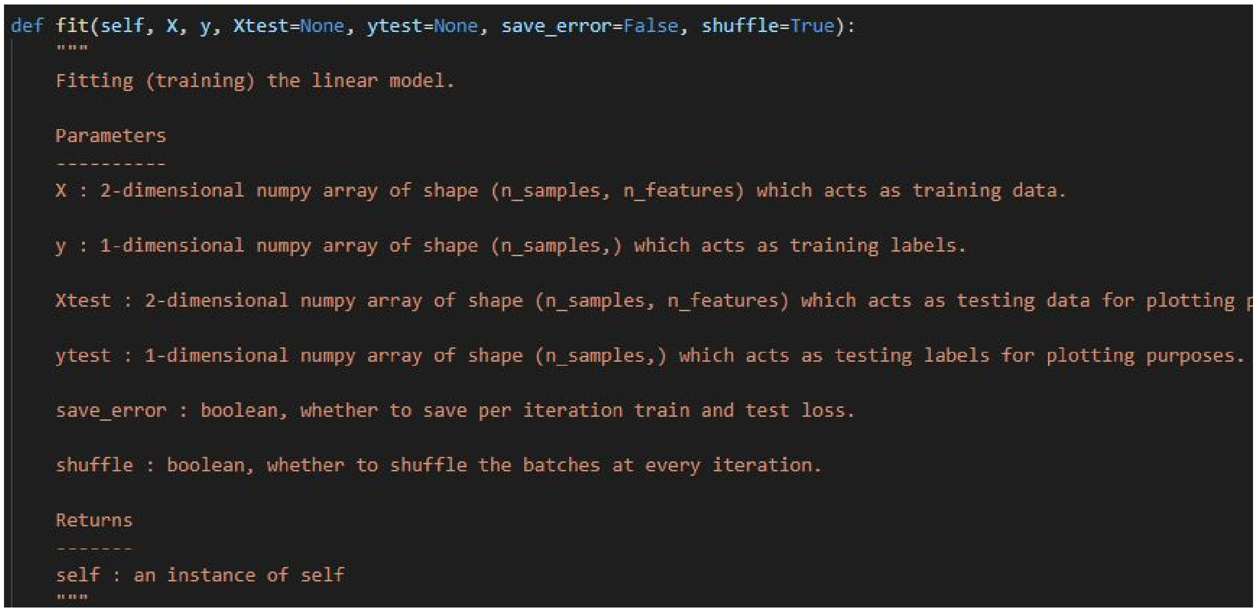
2. **(3+3+3+3+3) Implement the following activation functions with their gradient calculation too:**
   a. ReLU
   b. Sigmoid
   c. Linear
   d. Tanh
   e. Softmax

   DONE

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

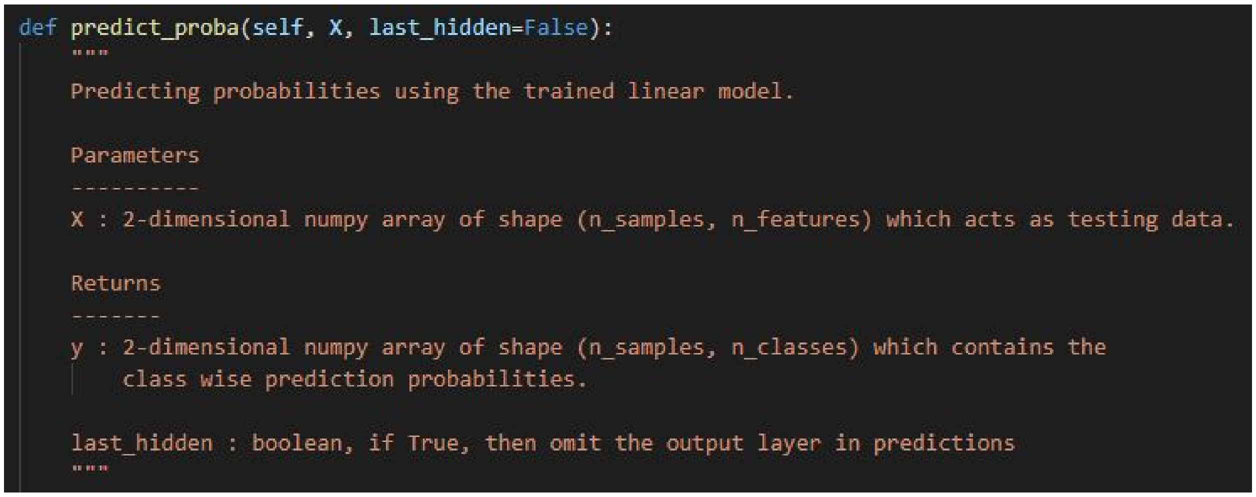
DONE

4. **(10+5+5+5) Implement the following functions with bias=0 and cross-entropy loss as the loss function. You can create other helper functions too.**
    - **a.** fit(): accepts input data & input labels and trains a new model



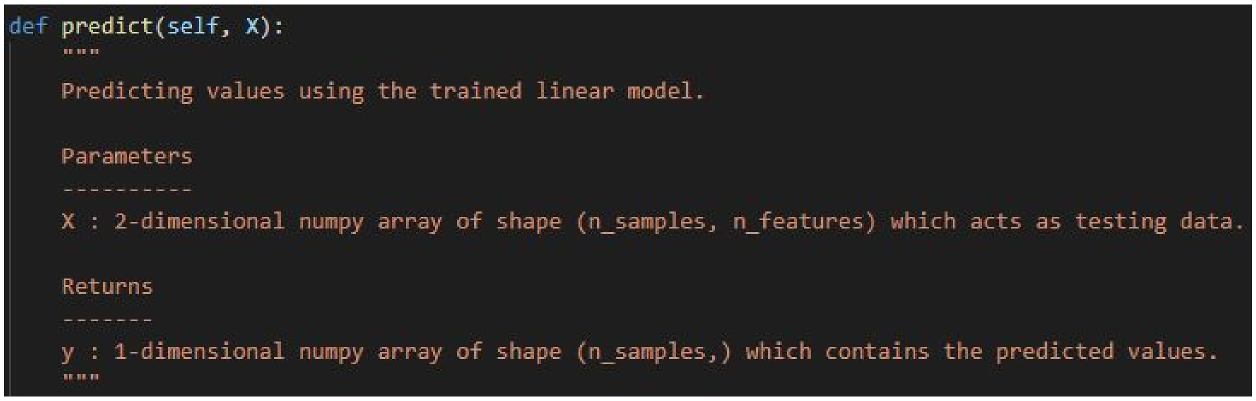
```
def fit(self, X, y, Xtest=None, ytest=None, save_error=False, shuffle=True):
    """
    Fitting (training) the linear model.

    Parameters
    ----------
    X : 2-dimensional numpy array of shape (n_samples, n_features) which acts as training data.

    y : 1-dimensional numpy array of shape (n_samples,) which acts as training labels.

    Xtest : 2-dimensional numpy array of shape (n_samples, n_features) which acts as testing data for plotting p

    ytest : 1-dimensional numpy array of shape (n_samples,) which acts as testing labels for plotting purposes.

    save_error : boolean, whether to save per iteration train and test loss.

    shuffle : boolean, whether to shuffle the batches at every iteration.

    Returns
    -------
    self : an instance of self
    """
```

- **b.** predict_proba(): accepts input data and returns the class-wise probability



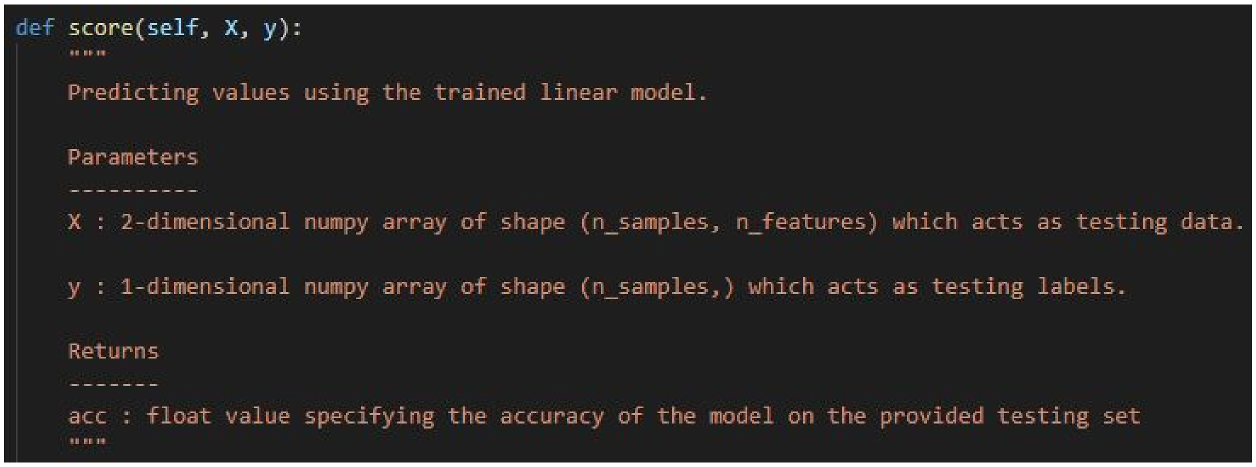
```
def predict_proba(self, X, last_hidden=False):
    """
    Predicting probabilities using the trained linear model.

    Parameters
    ----------
    X : 2-dimensional numpy array of shape (n_samples, n_features) which acts as testing data.

    Returns
    -------
    y : 2-dimensional numpy array of shape (n_samples, n_classes) which contains the
        class wise prediction probabilities.

    last_hidden : boolean, if True, then omit the output layer in predictions
    """
```

- **c.** predict(): accepts input data and returns the prediction using the trained model



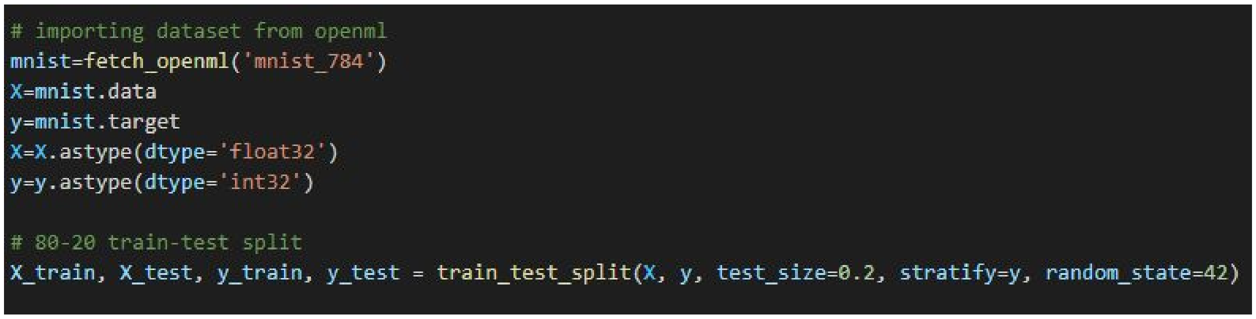
```python
def predict(self, X):
    """
    Predicting values using the trained linear model.

    Parameters
    ----------
    X : 2-dimensional numpy array of shape (n_samples, n_features) which acts as testing data.

    Returns
    -------
    y : 1-dimensional numpy array of shape (n_samples,) which contains the predicted values.
    """
```

d. score(): accepts input data and their labels and returns the accuracy of the model



```python
def score(self, X, y):
    """
    Predicting values using the trained linear model.

    Parameters
    ----------
    X : 2-dimensional numpy array of shape (n_samples, n_features) which acts as testing data.

    y : 1-dimensional numpy array of shape (n_samples,) which acts as testing labels.

    Returns
    -------
    acc : float value specifying the accuracy of the model on the provided testing set
    """
```

Q2. (35 points) Use the MNIST dataset for training and testing the neural network model created in Question 1 ONLY. Use the training dataset for calculating the training error and the test dataset for calculating the testing error.



```python
# importing dataset from openml
mnist=fetch_openml('mnist_784')
X=mnist.data
y=mnist.target
X=X.astype(dtype='float32')
y=y.astype(dtype='int32')

# 80-20 train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
```

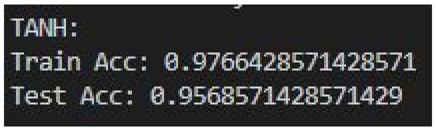
1. **(5) Use the following architecture [#input, 256, 128, 64, #output], learning rate=0.1, and number of epochs=100. Use normal weight initialization as defined in the first question.**

```
model = MyNeuralNetwork(
        5, [784, 256, 128, 64, 10],
        activation='tanh', learning_rate=0.1,
        weight_init='normal', batch_size=7000,
        num_epochs=100
    )
model = model.fit(X_train,y_train, Xtest=X_test, ytest=y_test,
save_error=True)
```

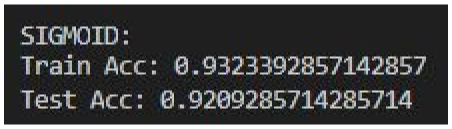
Save the weights of the trained model separately for each activation function defined above.

```
pickle.dump(model, open("Weights/tanh","wb"))
pickle.dump(model, open("Weights/relu","wb"))
pickle.dump(model, open("Weights/sigmoid","wb"))
pickle.dump(model, open("Weights/linear","wb"))
```

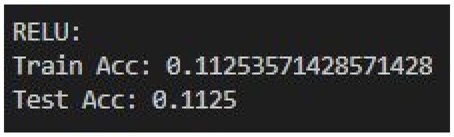
Report the test accuracy.



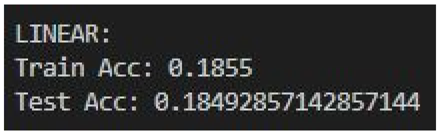
```
TANH:
Train Acc: 0.9766428571428571
Test Acc: 0.9568571428571429
```



```
SIGMOID:
Train Acc: 0.9323392857142857
Test Acc: 0.9209285714285714
```



```
RELU:
Train Acc: 0.11253571428571428
Test Acc: 0.1125
```

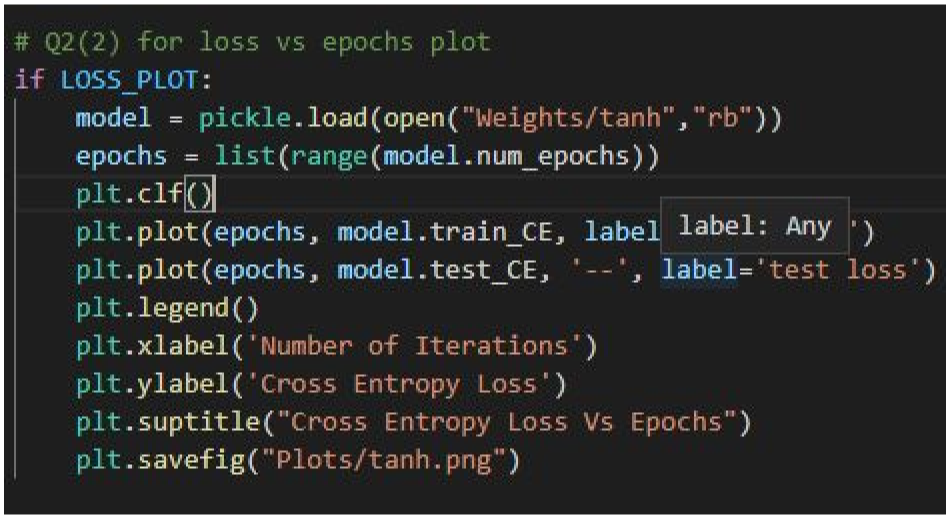


```
LINEAR:
Train Acc: 0.1855
Test Acc: 0.18492857142857144
```

MyNeuralNetwork	**tanh**	**relu**	**sigmoid**	**linear**
Train Accuracy	**0.9766**	**0.1125**	**0.9323**	**0.1855**
Test Accuracy	**0.9568**	**0.1125**	**0.9209**	**0.1849**

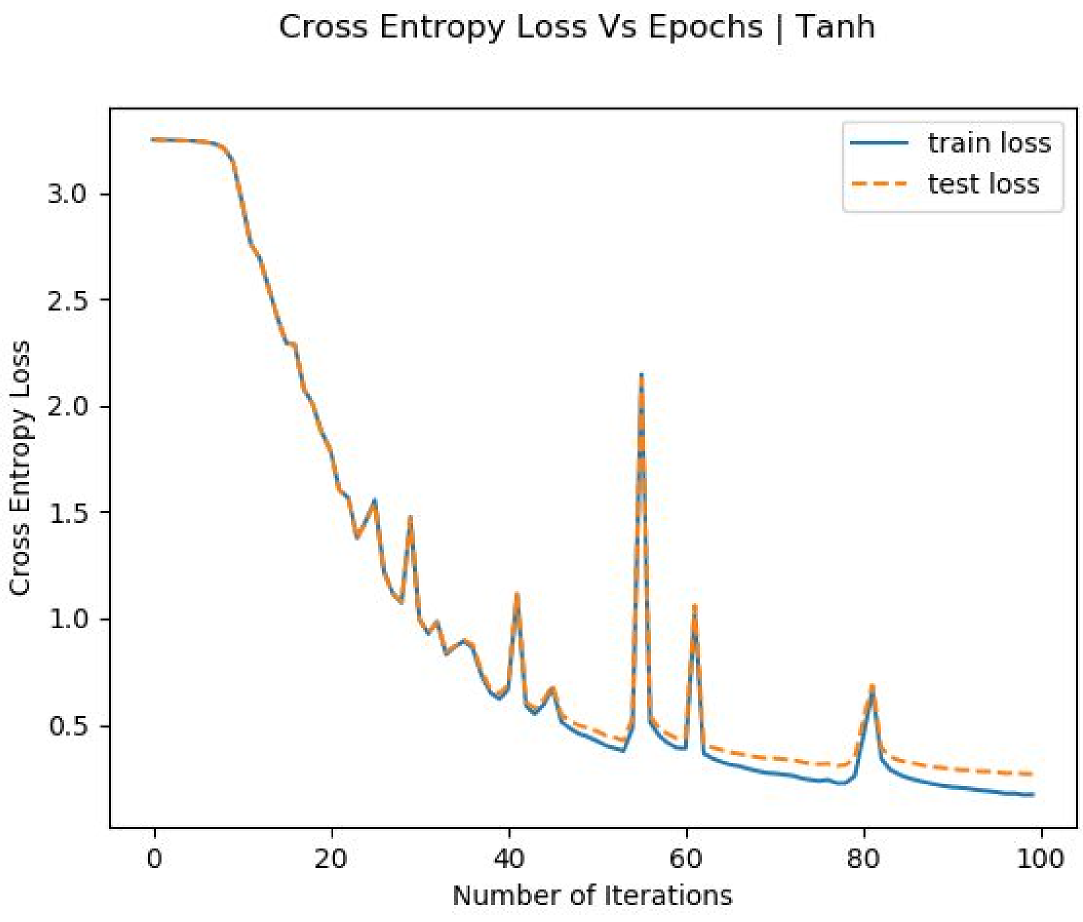
- At the given learning rates, tanh and sigmoid were found to be performing much much better than relu and linear. Tanh gave the best results with a testing accuracy of 95.68% while sigmoid also gave pretty good results with a testing accuracy of 92.09%.

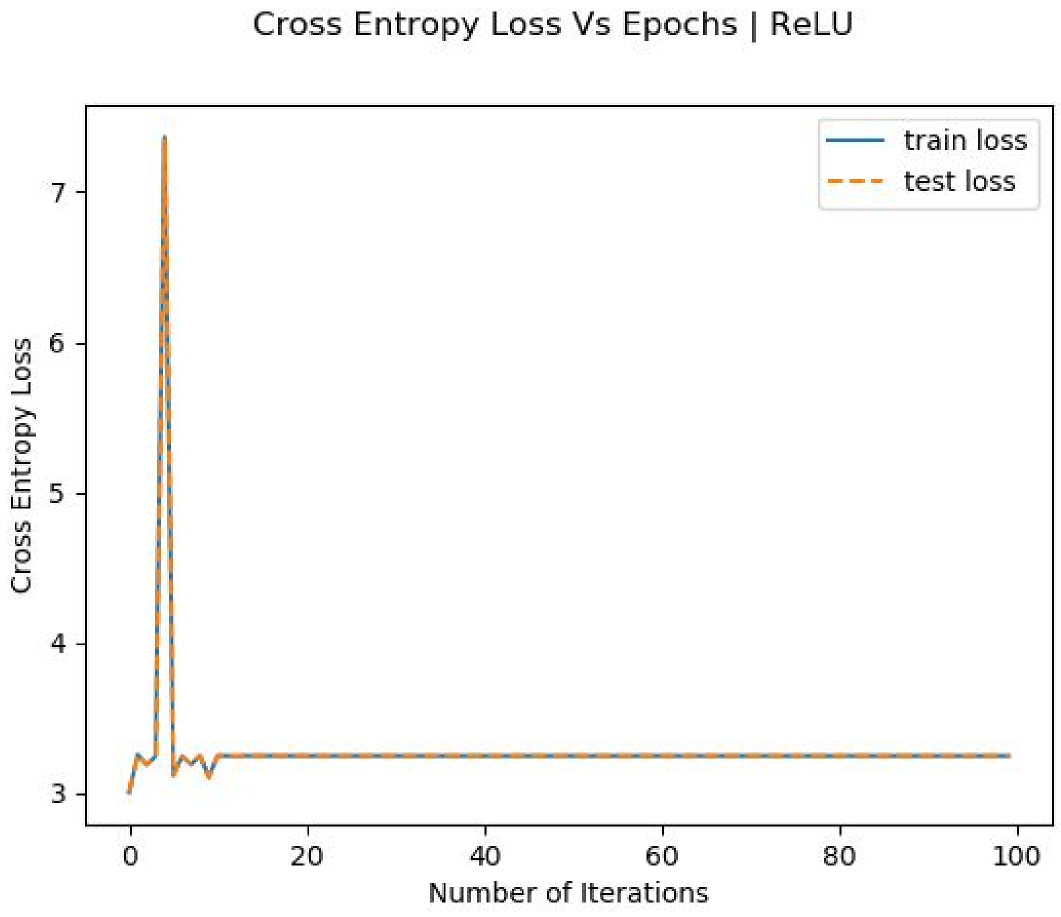
2. **(12) Plot training error vs epoch curve and testing error vs epoch curve for ReLU, sigmoid, linear, and tanh activation function.**

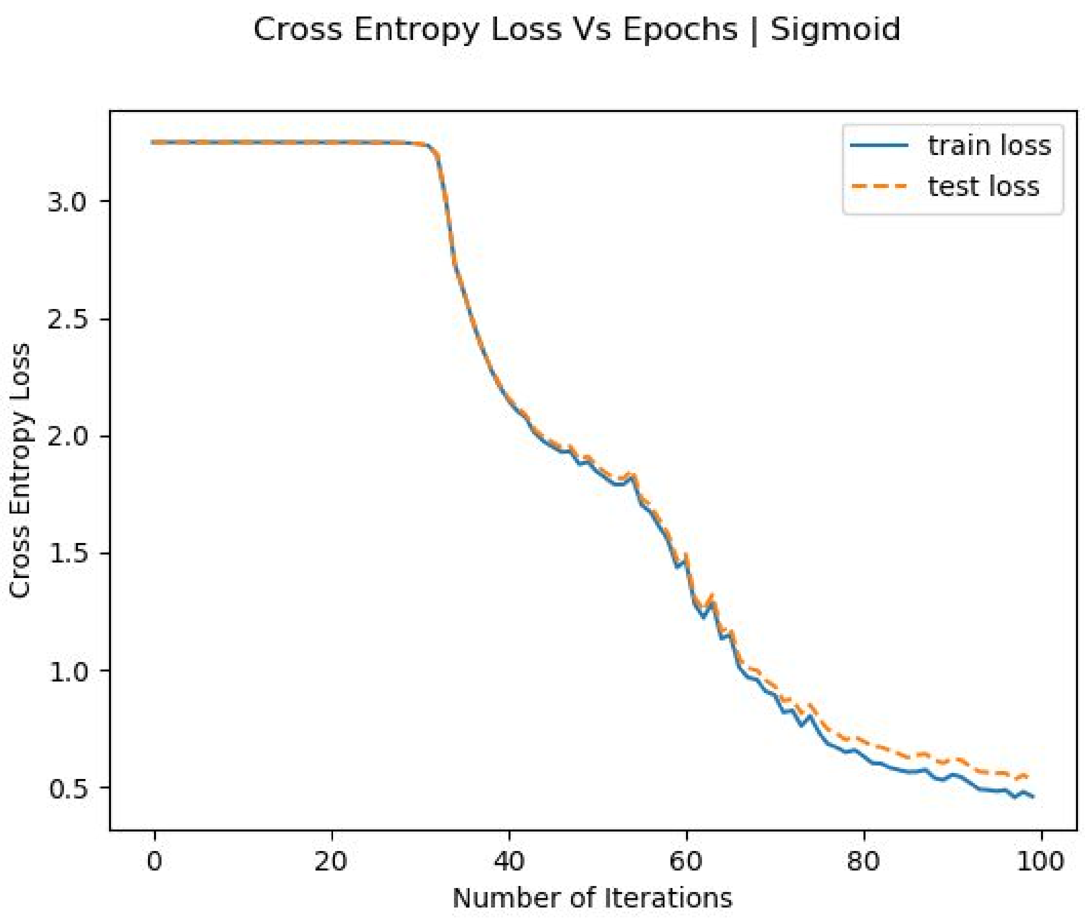


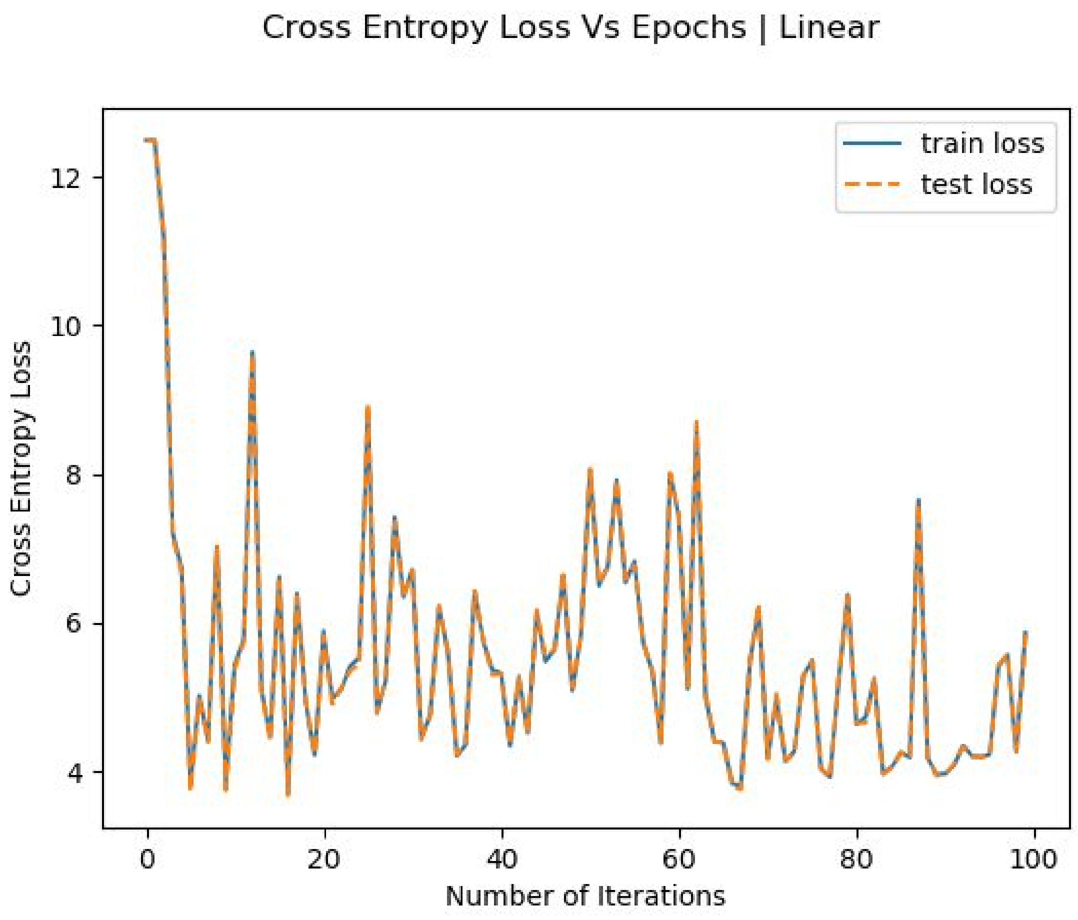
```
# Q2(2) for loss vs epochs plot
if LOSS_PLOT:
    model = pickle.load(open("Weights/tanh","rb"))
    epochs = list(range(model.num_epochs))
    plt.clf()
    plt.plot(epochs, model.train_CE, label label: Any ')
    plt.plot(epochs, model.test_CE, '--', label='test loss')
    plt.legend()
    plt.xlabel('Number of Iterations')
    plt.ylabel('Cross Entropy Loss')
    plt.suptitle("Cross Entropy Loss Vs Epochs")
    plt.savefig("Plots/tanh.png")
```

Finally, you should have 4 graphs for the 4 activation functions.




Cross Entropy Loss Vs Epochs | ReLU
train loss
test loss
Cross Entropy Loss
Number of Iterations
3
4
5
6
7
0
20
40
60
80
100


Cross Entropy Loss Vs Epochs | Sigmoid
train loss
test loss
Cross Entropy Loss
Number of Iterations
3.0
2.5
2.0
1.5
1.0
0.5
0
20
40
60
80
100



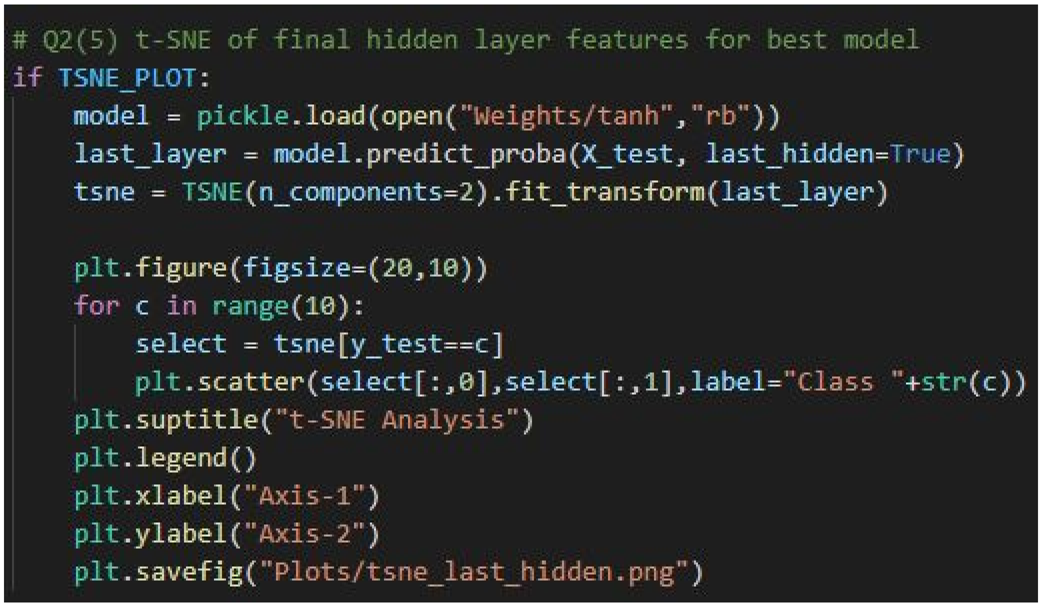
- The spikings that are visible in the plots represent the effort made by the gradient descent to escape a local minima in order to achieve a more optimal local minima. (See tanh, sigmoid curves).
- Too many jittery peaks like in the case of linear might also mean that the learning rate is too high and the parameters are thus oscillating about the local minima.
- Another observation is that while tanh starts moving towards lower loss consistently, sigmoid initially moves slowly and then suddenly starts converging starting at iteration 40.
- Moreover, ReLU does not seem to converge at all. The reason is most likely that the learning rate is too high.

3. **(5) In every case, what should be the activation function for the output layer? Give reasons to support your answer.**
   - In case of a **binary classification problem**, the output layer can work with just 1 perceptron unit (denoting the positive class, since then P(neg) = 1-P(pos)). Since the output should be depicting the probability of the sample to belong to that label, the output layer activations must be in the range [0,1]. The two activation functions that follow this rule are: **sigmoid** and **softmax**.

- In case of a **multi-class classification problem**, the output layer consists of more than one perceptron unit. In this case, the output layer activations not only need to be in the range [0,1], but should also sum up to 1. The only activation function that satisfies these two criteria is the **softmax** function.
- Thus, in order to define a generic class, **softmax** has been used as the output layer's activation function.

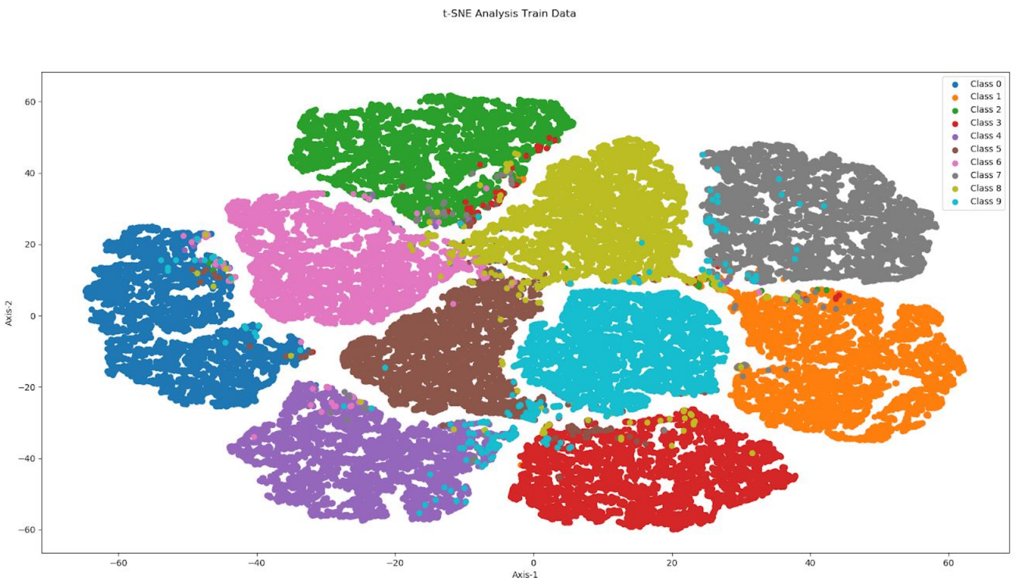
4. **(2) What are the total number of layers and the number of hidden layers in this case?**
   This neural network has 1 input layer, **3 hidden** layers, and 1 output layer making a total of **5 layers**.

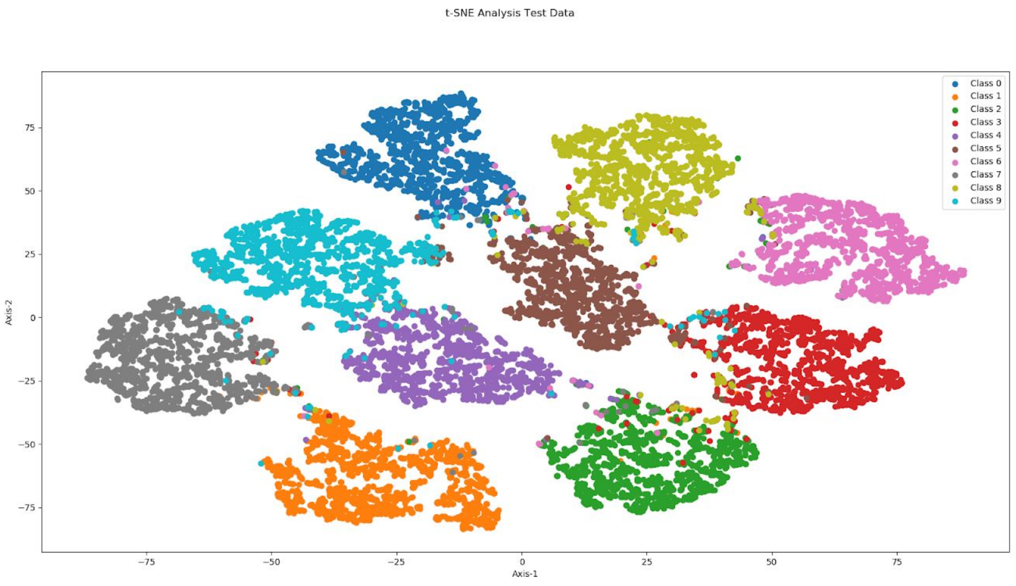
5. **(5) Visualise the features of the final hidden layer by creating tSNE plots for the model with the highest test accuracy. You can use sklearn for visualization.**



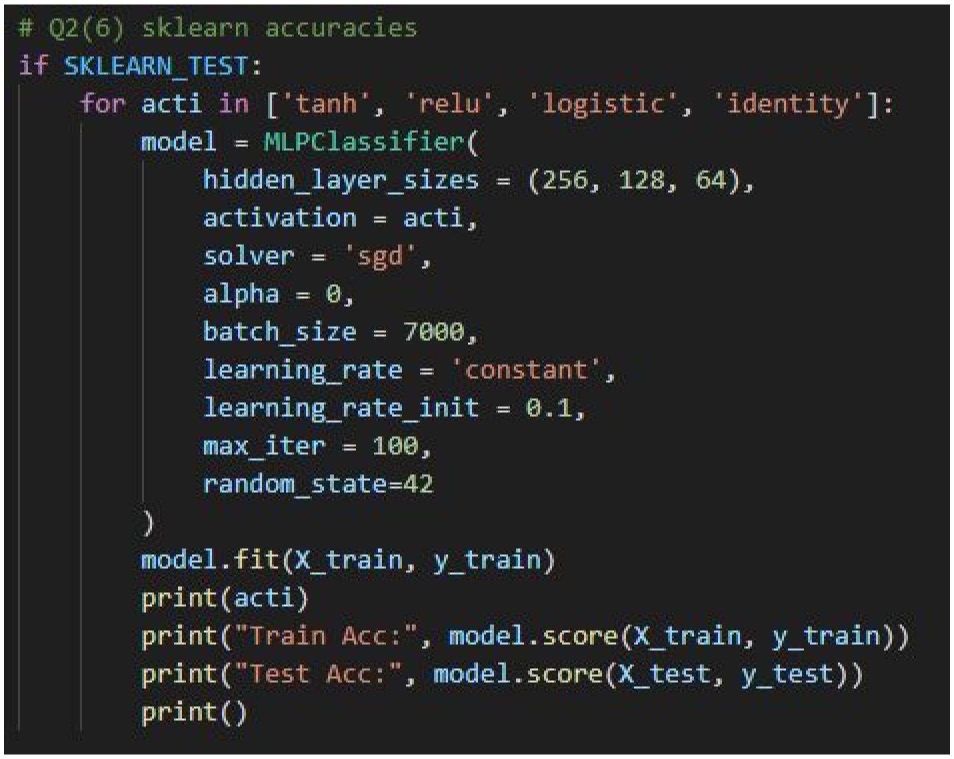
```
# Q2(5) t-SNE of final hidden layer features for best model
if TSNE_PLOT:
    model = pickle.load(open("Weights/tanh","rb"))
    last_layer = model.predict_proba(X_test, last_hidden=True)
    tsne = TSNE(n_components=2).fit_transform(last_layer)

    plt.figure(figsize=(20,10))
    for c in range(10):
        select = tsne[y_test==c]
        plt.scatter(select[:,0],select[:,1],label="Class "+str(c))
    plt.suptitle("t-SNE Analysis")
    plt.legend()
    plt.xlabel("Axis-1")
    plt.ylabel("Axis-2")
    plt.savefig("Plots/tsne_last_hidden.png")
```

- The t-SNE plots show the 10 clusters corresponding to the 10 classes. We can see that these are very well separable based on the 64 features calculated at the last hidden layer. The decision boundaries that our neural network is identifying to give such good predictions are also clearly identifiable.
- We can also see some noisy points which are very much prone to misclassification.





6. **(6) Now, use sklearn with the same parameters defined above and report the test accuracy obtained using ReLU, sigmoid, linear, and tanh activation functions.**



```python
# Q2(6) sklearn accuracies
if SKLEARN_TEST:
    for acti in ['tanh', 'relu', 'logistic', 'identity']:
        model = MLPClassifier(
            hidden_layer_sizes = (256, 128, 64),
            activation = acti,
            solver = 'sgd',
            alpha = 0,
            batch_size = 7000,
            learning_rate = 'constant',
            learning_rate_init = 0.1,
            max_iter = 100,
            random_state=42
        )
        model.fit(X_train, y_train)
        print(acti)
        print("Train Acc:", model.score(X_train, y_train))
        print("Test Acc:", model.score(X_test, y_test))
        print()
```

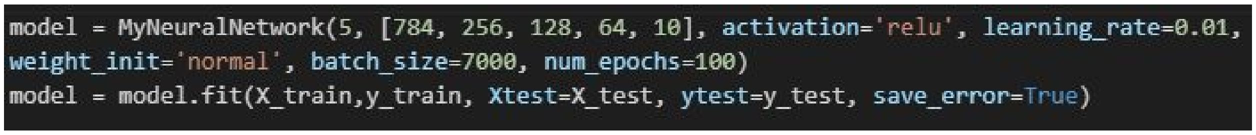
Sklearn	**tanh**	**relu**	**sigmoid**	**linear**
Train Accuracy	**0.9641**	**0.1125**	**0.9909**	**0.0986**
Test Accuracy	**0.9554**	**0.1125**	**0.9635**	**0.0986**

Comment on the differences in accuracy, if any.

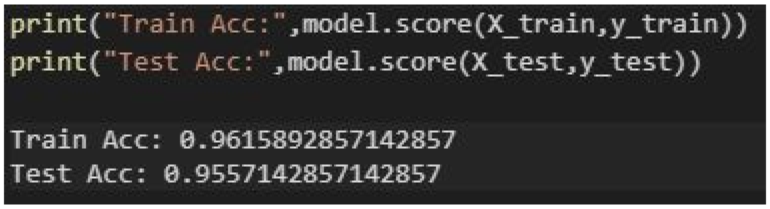
- Comparing
  - SkLearn also seems to give good results with tanh and sigmoid and not performing well for linear and relu activation functions.
  - However, sklearn gave best results for sigmoid compared to MyNeuralNetwork which gave best results on tanh.
  - Sigmoid results for sklearn are a little bit better than MyNeuralNetwork while tanh results are a little bit better in the case of MyNeuralNetwork.
  - Additionally, while relu results are exactly the same in both sklearn and MyNeuralNetwork, linear activation gives almost twice the accuracy in case of MyNeuralNetwork as compared to the sklearn implementation.
- Analysis
  - In the linear case, for sklearn, setting verbose=True reveals that we're getting loss=nan at every iteration. In our case, we solved this problem by clipping. But sklearn ignores this and thus such a low accuracy.
  - Sklearn implementation includes some additional advanced hyperparameters such as momentum, variable learning rate, etc, which our implementation does not take into account.

  - Some variations also appear due to randomly assigned weights and randomly created batches for mini-bgd optimizer.
  - **Another reason for poor accuracy in case of ReLU and Linear could be that the learning rate we used is inappropriate for the case**.

Note: To support the above fact, using a learning rate of 0.01, gave surprisingly good results on ReLU.



```
model = MyNeuralNetwork(5, [784, 256, 128, 64, 10], activation='relu', learning_rate=0.01,
weight_init='normal', batch_size=7000, num_epochs=100)
model = model.fit(X_train,y_train, Xtest=X_test, ytest=y_test, save_error=True)
```

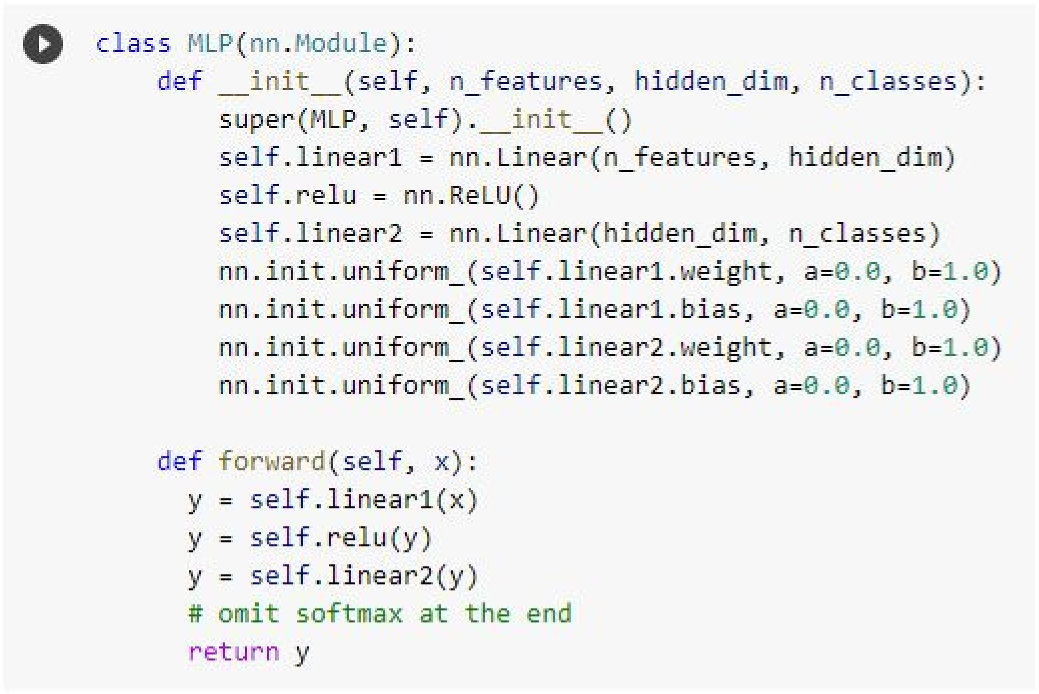


```
print("Train Acc:",model.score(X_train,y_train))
print("Test Acc:",model.score(X_test,y_test))

Train Acc: 0.9615892857142857
Test Acc: 0.9557142857142857
```

Q3. (20 points) For the DATASET provided, use the following hyperparameter settings to train each neural network (using PyTorch) described below.

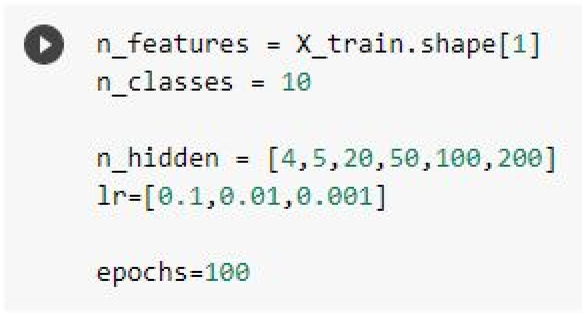
4 Hidden units
Random Weight Initialization
Learning Rate=0.01
Epochs=100

- Defining a custom PyTorch model class



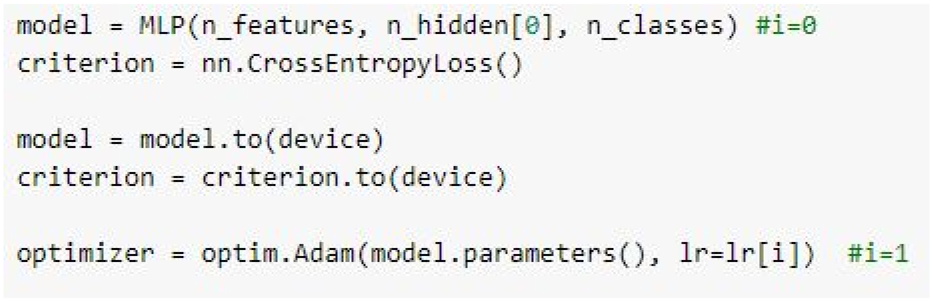
```
class MLP(nn.Module):
    def __init__(self, n_features, hidden_dim, n_classes):
        super(MLP, self).__init__()
        self.linear1 = nn.Linear(n_features, hidden_dim)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(hidden_dim, n_classes)
        nn.init.uniform_(self.linear1.weight, a=0.0, b=1.0)
        nn.init.uniform_(self.linear1.bias, a=0.0, b=1.0)
        nn.init.uniform_(self.linear2.weight, a=0.0, b=1.0)
        nn.init.uniform_(self.linear2.bias, a=0.0, b=1.0)

    def forward(self, x):
      y = self.linear1(x)
      y = self.relu(y)
      y = self.linear2(y)
      # omit softmax at the end
      return y
```

- Hyperparameters



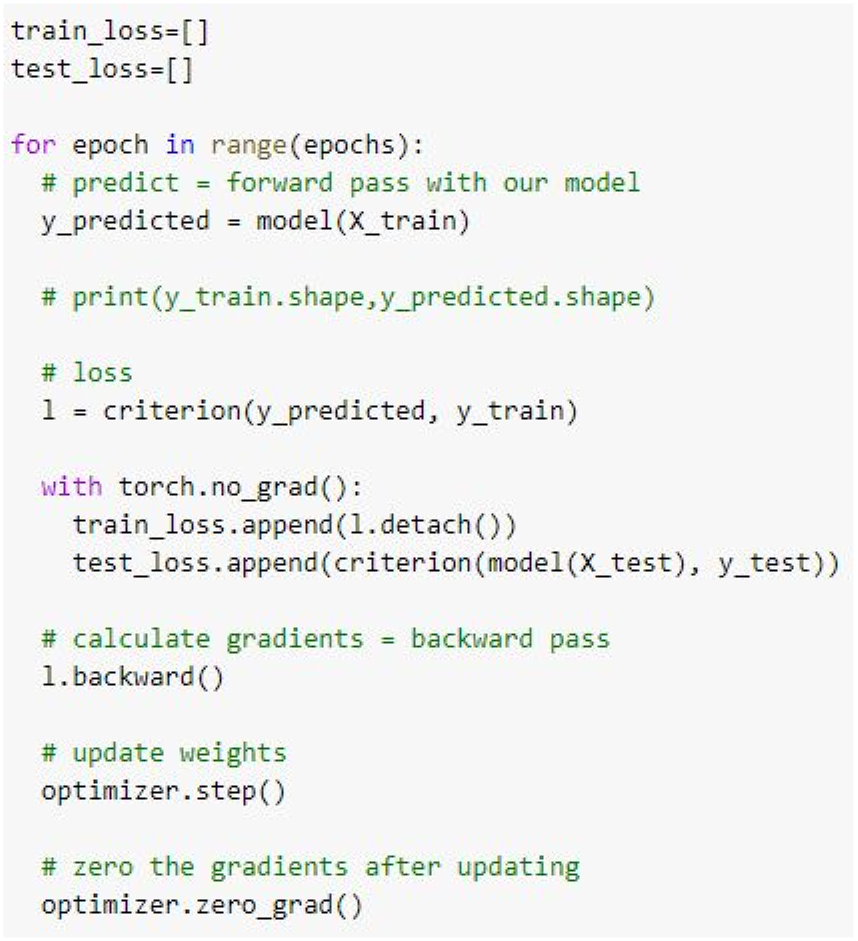
```
n_features = X_train.shape[1]
n_classes = 10

n_hidden = [4,5,20,50,100,200]
lr=[0.1,0.01,0.001]

epochs=100
```

- Defining loss, optimizer, and creating a model object



```
model = MLP(n_features, n_hidden[0], n_classes) #i=0
criterion = nn.CrossEntropyLoss()

model = model.to(device)
criterion = criterion.to(device)

optimizer = optim.Adam(model.parameters(), lr=lr[i])  #i=1
```

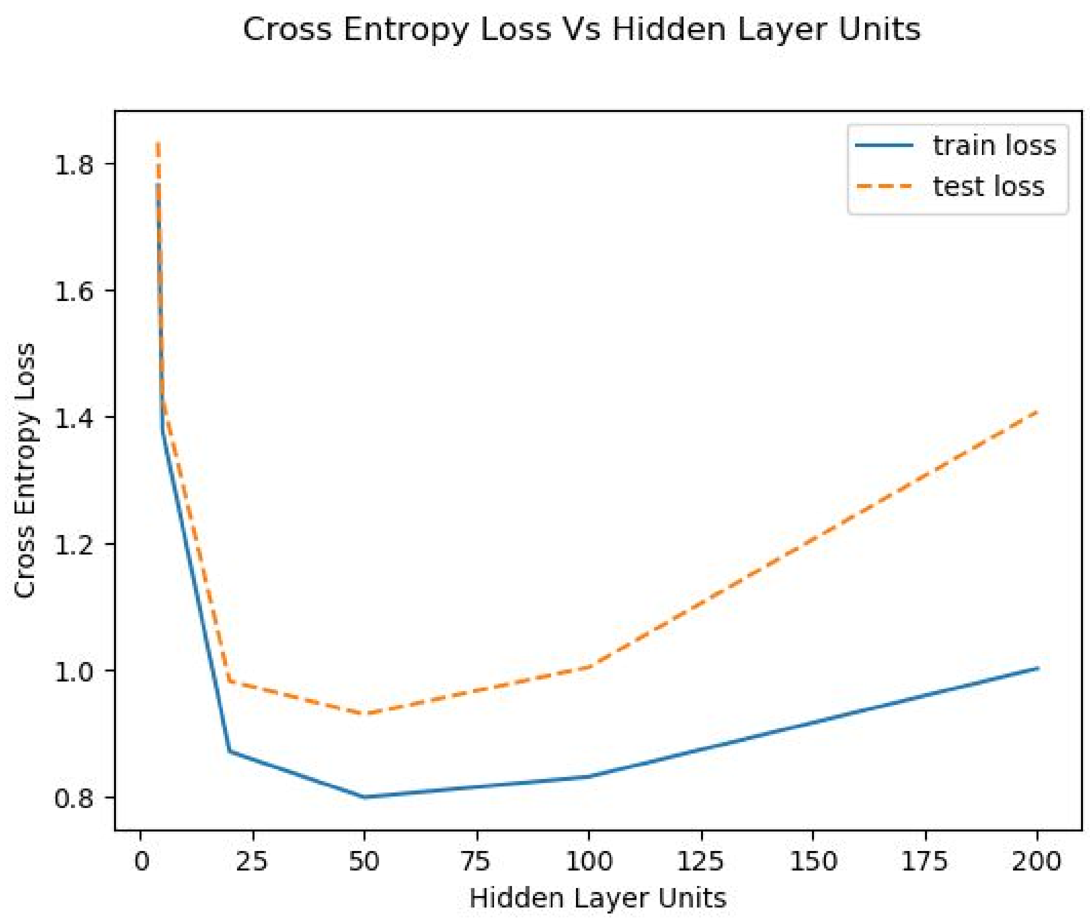
- Training Loop



```python
train_loss=[]
test_loss=[]

for epoch in range(epochs):
  # predict = forward pass with our model
  y_predicted = model(X_train)

  # print(y_train.shape,y_predicted.shape)

  # loss
  l = criterion(y_predicted, y_train)

  with torch.no_grad():
    train_loss.append(l.detach())
    test_loss.append(criterion(model(X_test), y_test))

  # calculate gradients = backward pass
  l.backward()

  # update weights
  optimizer.step()

  # zero the gradients after updating
  optimizer.zero_grad()
```

1. **Hidden Units:** [4, 5, 20, 50, 100, 200]

   (a) **(6) Plot the average training and validation cross-entropy vs the number of hidden units on the x-axis.**

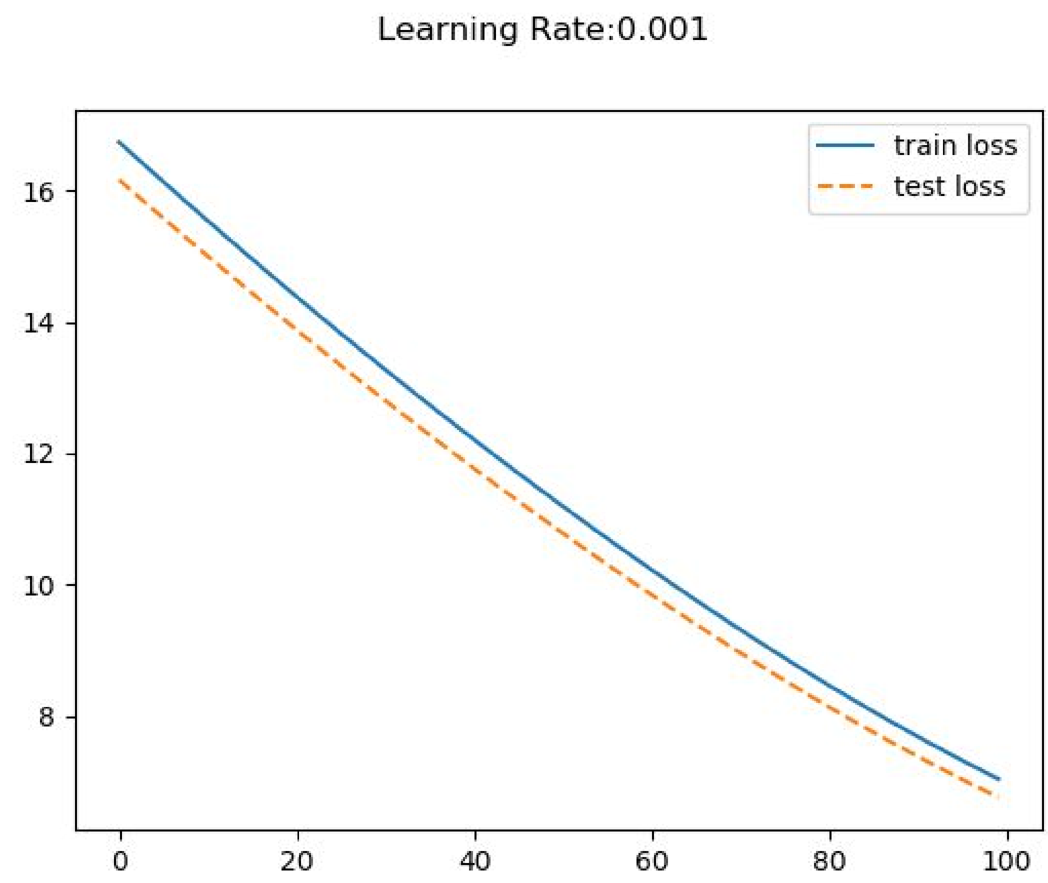


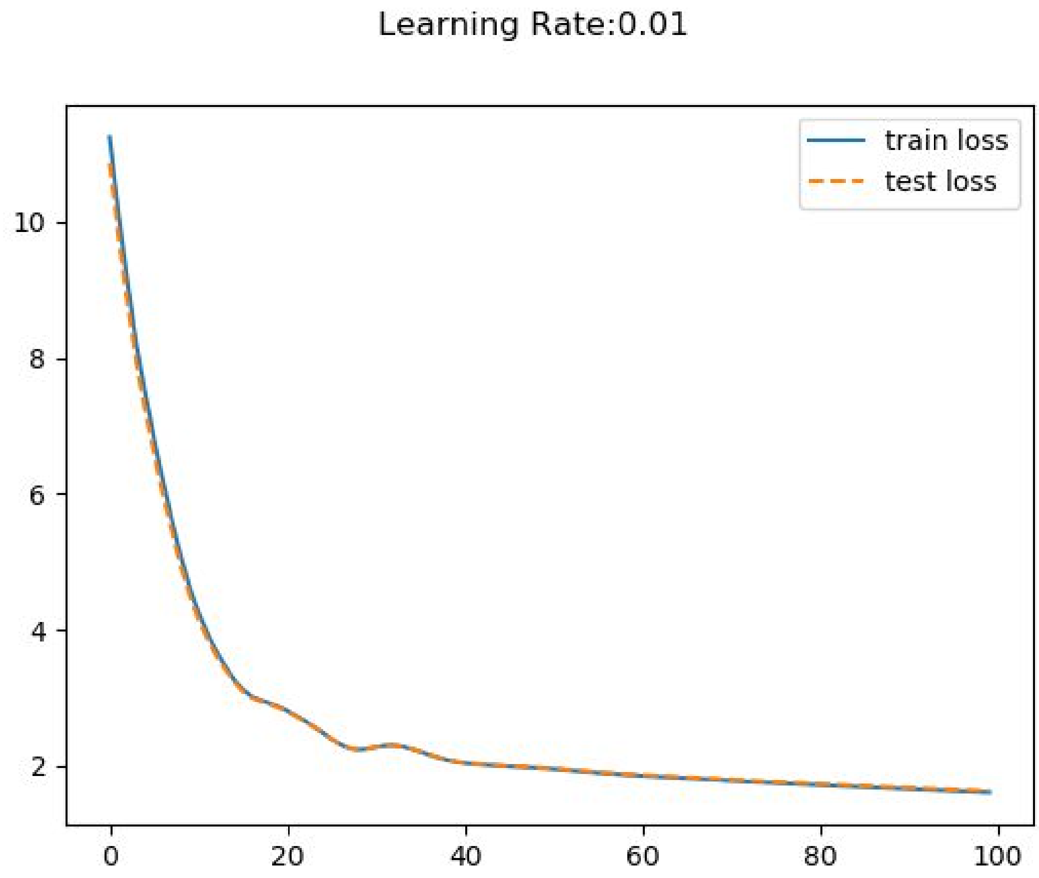
(b) (4) Examine and comment on the plots of training and validation cross-entropy. What is the effect of changing the number of hidden units?

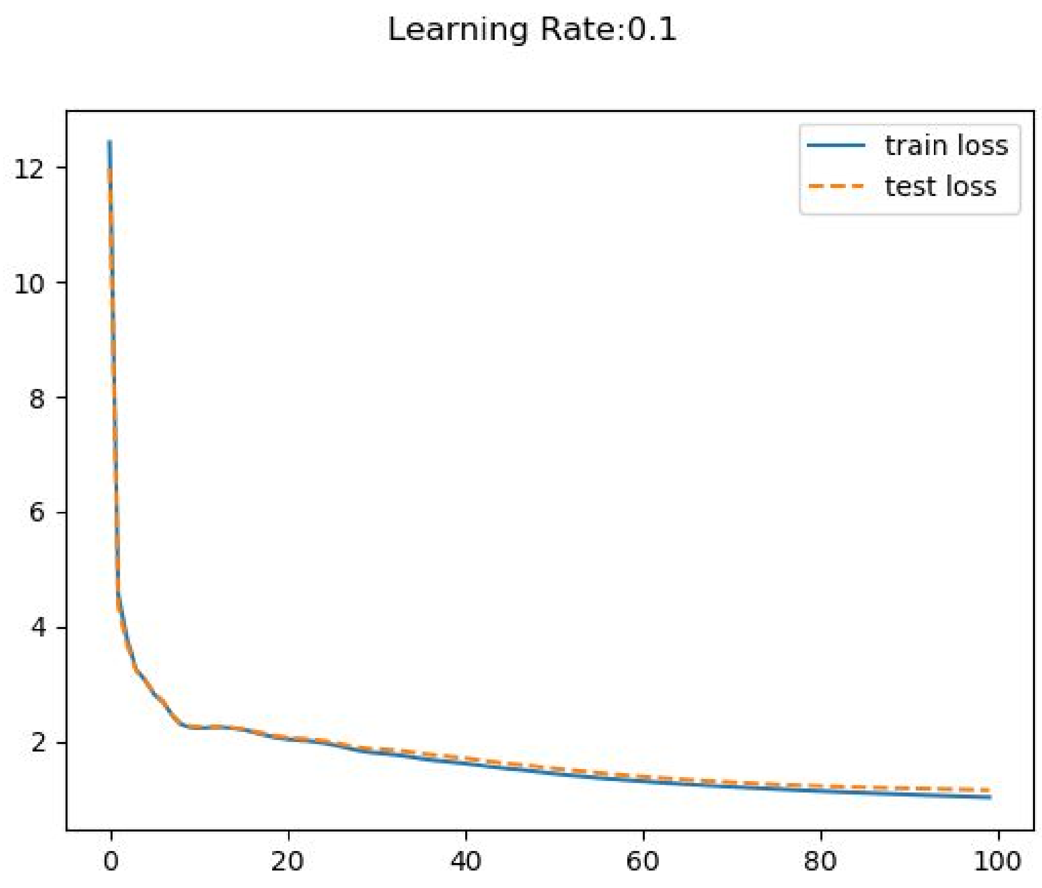
- We can see that at 50 hidden units, we are getting the lowest training as well as validation loss.
- Increasing the hidden units beyond 50 further increases the model complexity and the model hence tends to overfit. This is also evident from the increasing gap between training and validation loss. While training loss is only increasing a little bit, validation loss is increasing drastically with the increase in the number of perceptron units in the only hidden layer.
- Below 20 hidden units, the model seems to be underfit since both the training and validation loss are pretty high.
- A wise choice would be to select a model between 20 to 50 hidden units, preferably away from the 50 limit and more towards 20.
- *Thus changing the number of hidden units affects the model behaviour, starting from underfit, to just perfect fit, to overfit. Choosing the right number is thus very important.*

2. Learning Rate: [0.001, 0.01, 0.1]

(a) (6) Plot the average training and validation cross-entropy vs the number of epochs for different learning rates.


Learning Rate:0.001
train loss
test loss
16
14
12
10
8
0
20
40
60
80
100


Learning Rate:0.01
train loss
test loss
10
8
6
4
2
0
20
40
60
80
100



(b) (4) 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?

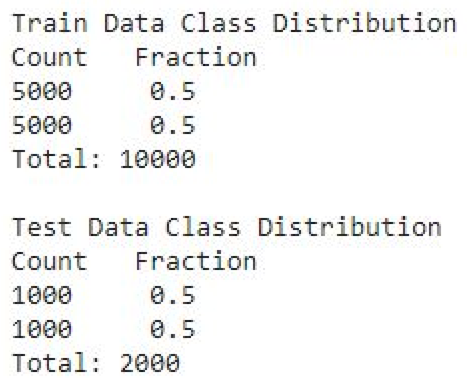
- Evident from the first plot, that learning rate 0.001 is too low for convergence in 100 iterations.
- At 0.01 learning rate, the training and validation loss are dropping together and the plot suggests that the model is close to convergence.
- At 0.1 learning rate, the convergence is much more likely also evident from the plot as well.
- *The learning rate determines how fast the model will converge, however if it is set to be too high, the model is likely to miss the minima and thus not converge. Too low learning rate leads to very slow convergence thus requiring too many iterations to convergence.*

Q4. (20 points) Use the binary CIFAR 10 subset for this part.

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

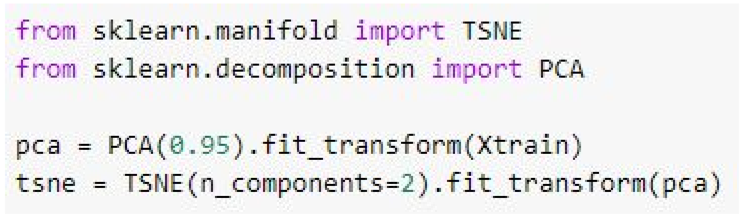
The dataset consists of car and aeroplane images. This is a binary class dataset where label 0 represents an airplane and label 1 represents a car. Each image consists of 3072 features, which is just pixel values for an rgb image of size 32x32. The pixel values are 8 bit, i.e. 0-255.

- **Class Distribution**

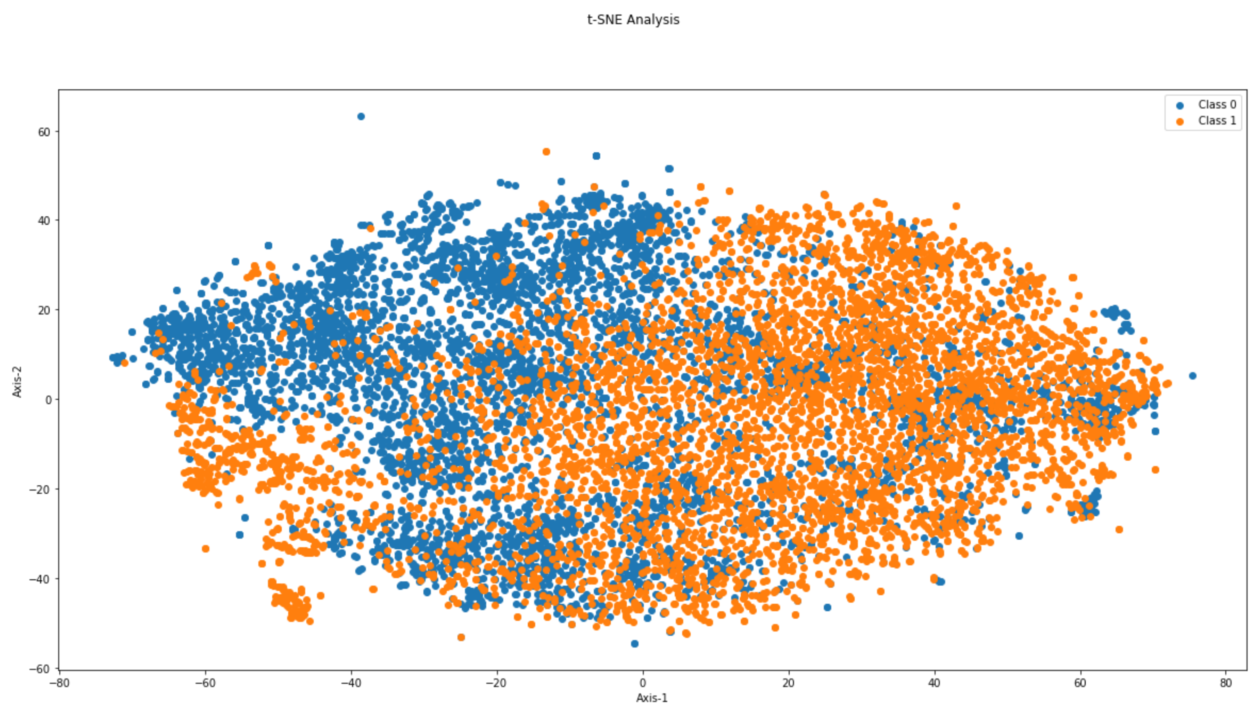


```
Train Data Class Distribution
Count   Fraction
5000     0.5
5000     0.5
Total: 10000

Test Data Class Distribution
Count   Fraction
1000     0.5
1000     0.5
Total: 2000
```

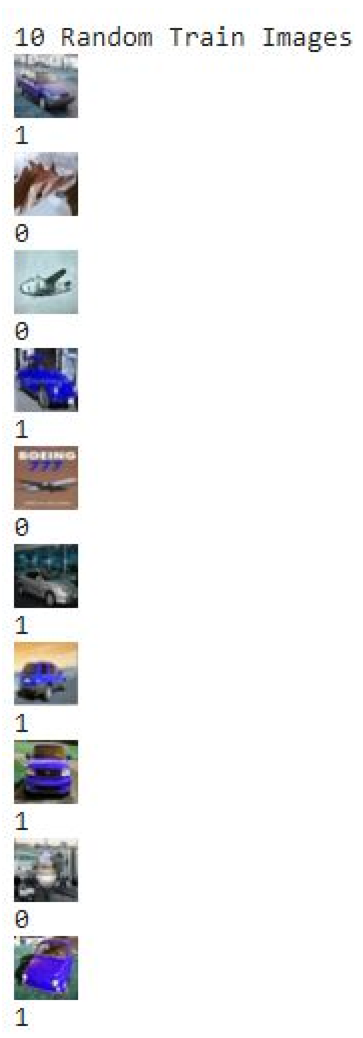
- **t-SNE Analysis after using PCA for Dimensionality Reduction**



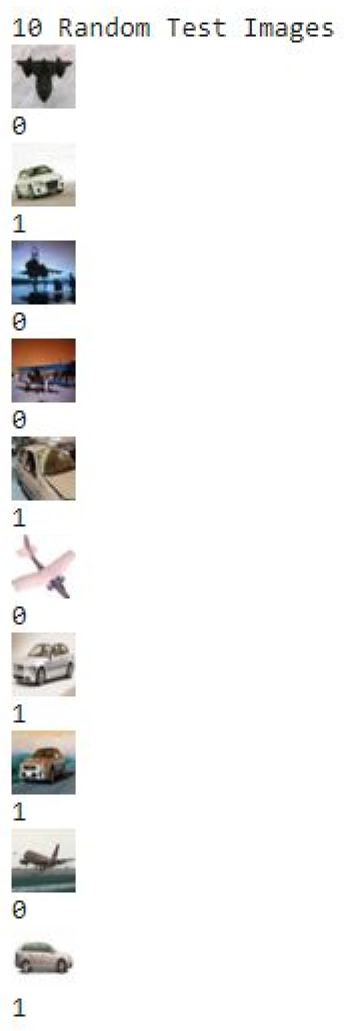
```python
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA

pca = PCA(0.95).fit_transform(Xtrain)
tsne = TSNE(n_components=2).fit_transform(pca)
```



- **Random Sample Images Visualization**



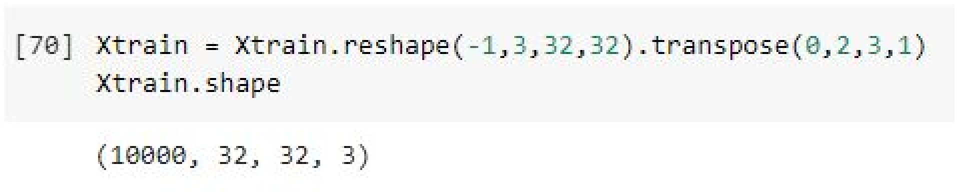
```
10 Random Train Images
1
0
0
1
0
1
1
1
0
1
```



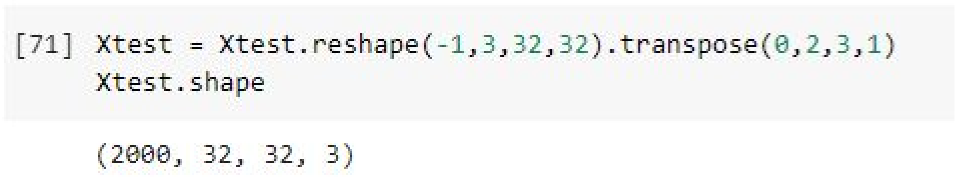
```
10 Random Test Images
0
1
0
0
1
0
1
1
0
1
```

2. **(10) 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.**

    - **Reshaped** given flattened NumPy feature arrays to the appropriate dimension to be displayed as an image. (This was done before the cv2.imshow based sample image visualization.)

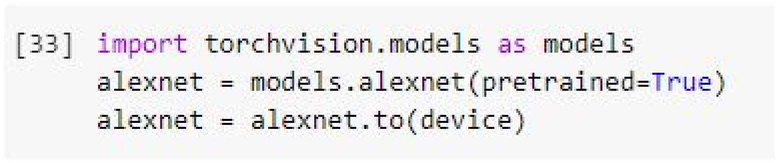


```
[70] Xtrain = Xtrain.reshape(-1,3,32,32).transpose(0,2,3,1)
     Xtrain.shape

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



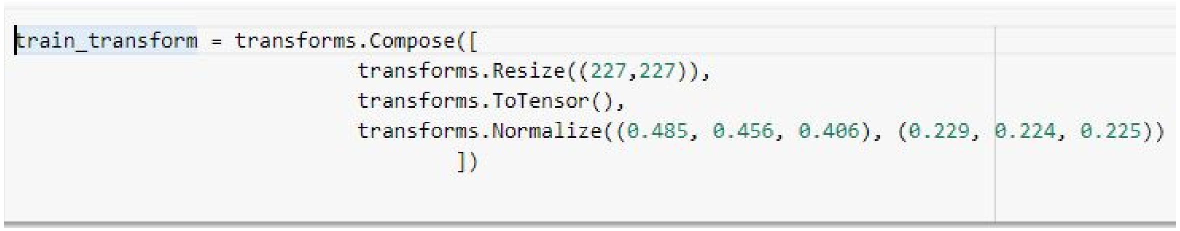
```
[71] Xtest = Xtest.reshape(-1,3,32,32).transpose(0,2,3,1)
     Xtest.shape

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

- **Imported** pre-trained AlexNet Model from PyTorch



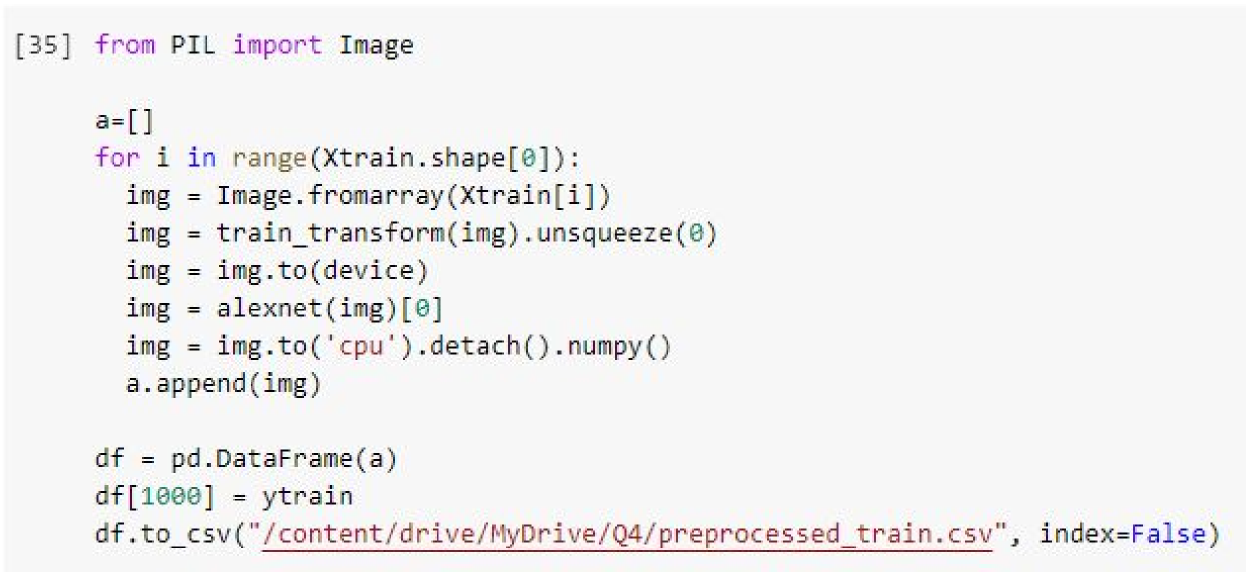
```
[33] import torchvision.models as models
     alexnet = models.alexnet(pretrained=True)
     alexnet = alexnet.to(device)
```

- **Preprocessing** the given CIFAR dataset using torch transforms to make it compatible with the AlexNet model which requires (227,227) size RGB images whereas our dataset consists of (32,32) size RGB images.



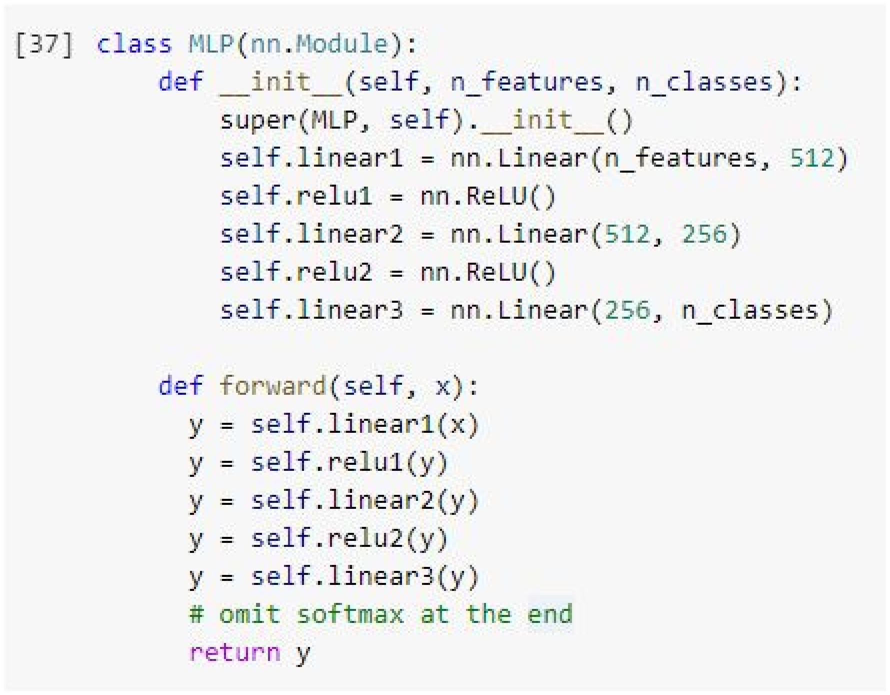
```
train_transform = transforms.Compose([
                                transforms.Resize((227,227)),
                                transforms.ToTensor(),
                                transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
                                      ])
```

- Since the given dataset was huge, the images were transformed one by one followed by extracting 1000 features per image using the AlexNet model, and stored in a new CSV file on Google Drive while using Google Colab. Both train and test data were transformed this way creating a new CSV for either, in order to eliminate this calculation during model training.



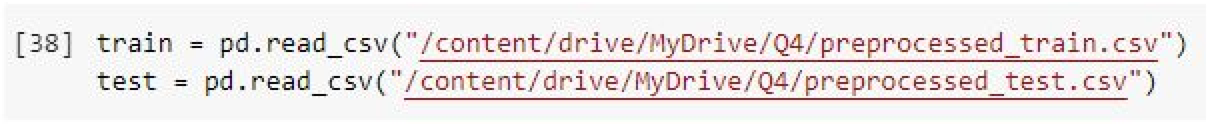
```
[35] from PIL import Image

     a=[]
     for i in range(Xtrain.shape[0]):
       img = Image.fromarray(Xtrain[i])
       img = train_transform(img).unsqueeze(0)
       img = img.to(device)
       img = alexnet(img)[0]
       img = img.to('cpu').detach().numpy()
       a.append(img)

     df = pd.DataFrame(a)
     df[1000] = ytrain
     df.to_csv("/content/drive/MyDrive/Q4/preprocessed_train.csv", index=False)
```

3. **(5) 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.**
   - Defined a neural network similar to Q3



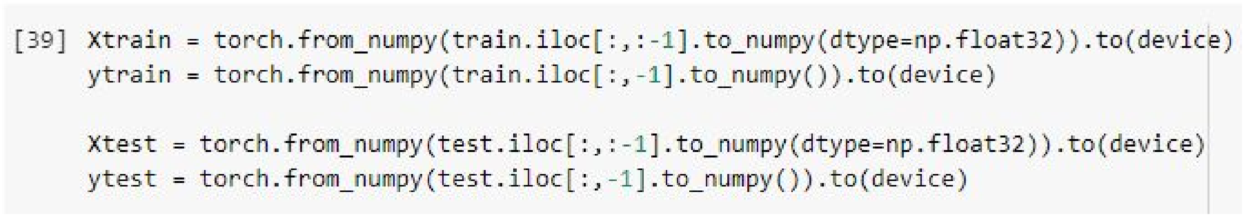
```
[37] class MLP(nn.Module):
         def __init__(self, n_features, n_classes):
             super(MLP, self).__init__()
             self.linear1 = nn.Linear(n_features, 512)
             self.relu1 = nn.ReLU()
             self.linear2 = nn.Linear(512, 256)
             self.relu2 = nn.ReLU()
             self.linear3 = nn.Linear(256, n_classes)

         def forward(self, x):
           y = self.linear1(x)
           y = self.relu1(y)
           y = self.linear2(y)
           y = self.relu2(y)
           y = self.linear3(y)
           # omit softmax at the end
           return y
```

   - Import transformed data



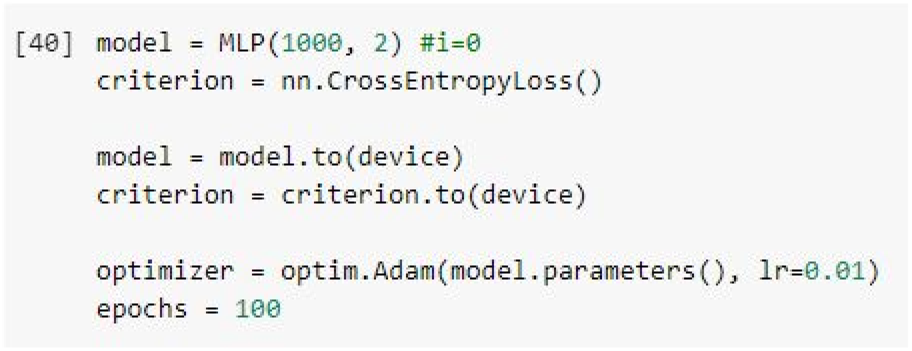
```
[38] train = pd.read_csv("/content/drive/MyDrive/Q4/preprocessed_train.csv")
     test = pd.read_csv("/content/drive/MyDrive/Q4/preprocessed_test.csv")
```

   - Convert to torch.tensor format and copy to GPU



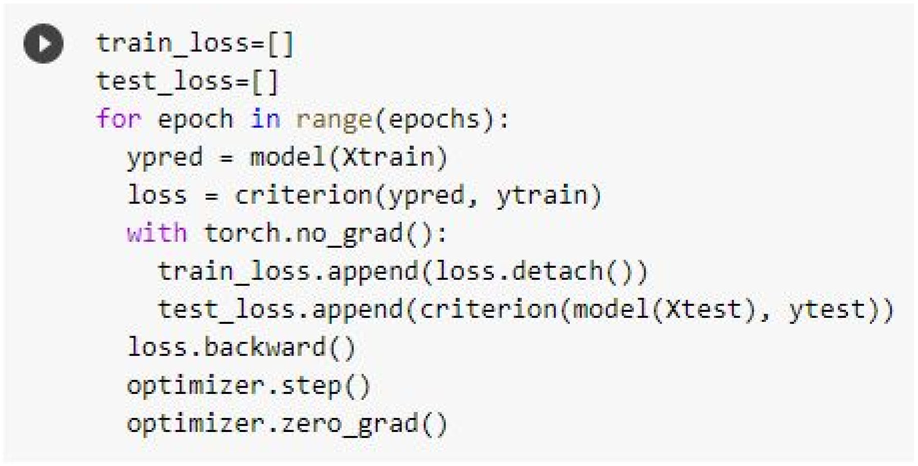
```
[39] Xtrain = torch.from_numpy(train.iloc[:,:-1].to_numpy(dtype=np.float32)).to(device)
     ytrain = torch.from_numpy(train.iloc[:,-1].to_numpy()).to(device)

     Xtest = torch.from_numpy(test.iloc[:,:-1].to_numpy(dtype=np.float32)).to(device)
     ytest = torch.from_numpy(test.iloc[:,-1].to_numpy()).to(device)
```

- Define loss, optimizer, learning rate, epochs



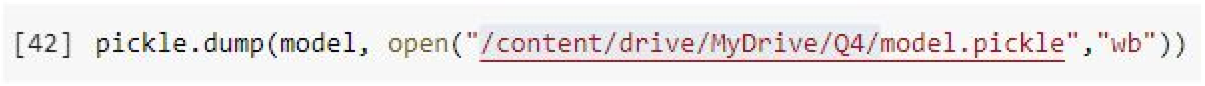
```
[40] model = MLP(1000, 2) #i=0
     criterion = nn.CrossEntropyLoss()

     model = model.to(device)
     criterion = criterion.to(device)

     optimizer = optim.Adam(model.parameters(), lr=0.01)
     epochs = 100
```

- Training the model



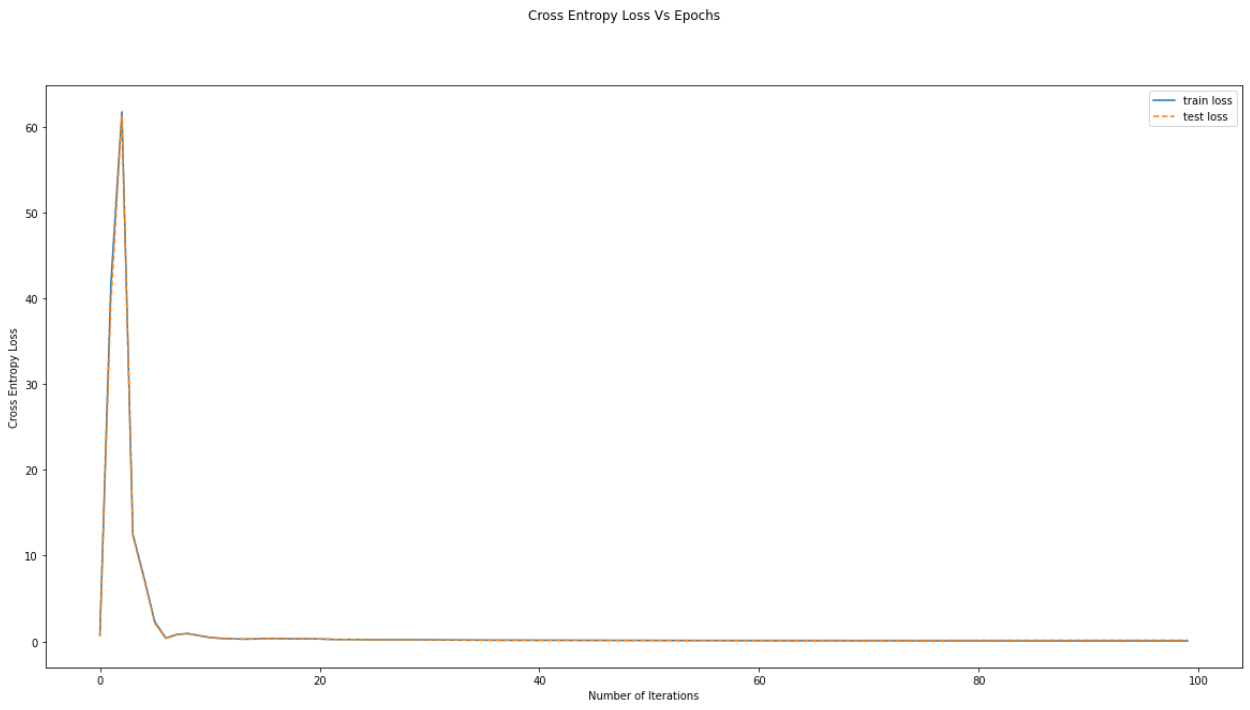
```
train_loss=[]
test_loss=[]
for epoch in range(epochs):
  ypred = model(Xtrain)
  loss = criterion(ypred, ytrain)
  with torch.no_grad():
    train_loss.append(loss.detach())
    test_loss.append(criterion(model(Xtest), ytest))
  loss.backward()
  optimizer.step()
  optimizer.zero_grad()
```

- The model has been saved in GoogleDrive



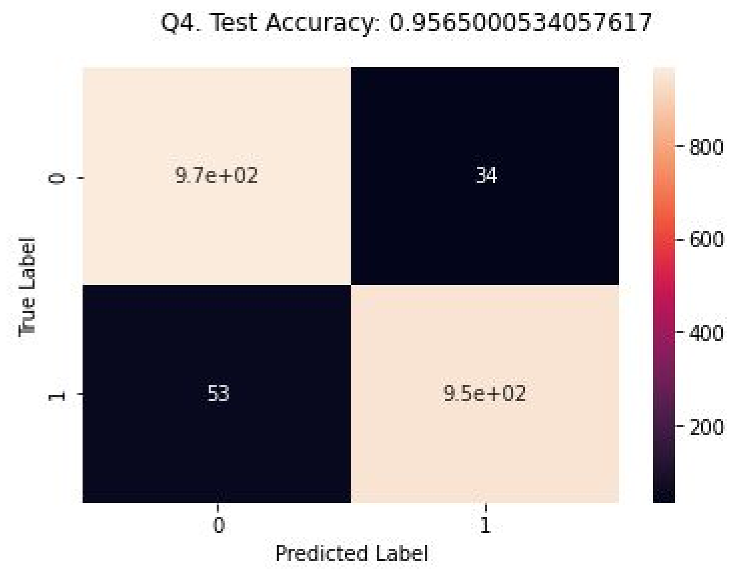
```
[42] pickle.dump(model, open("/content/drive/MyDrive/Q4/model.pickle","wb"))
```

- Loss plots during training

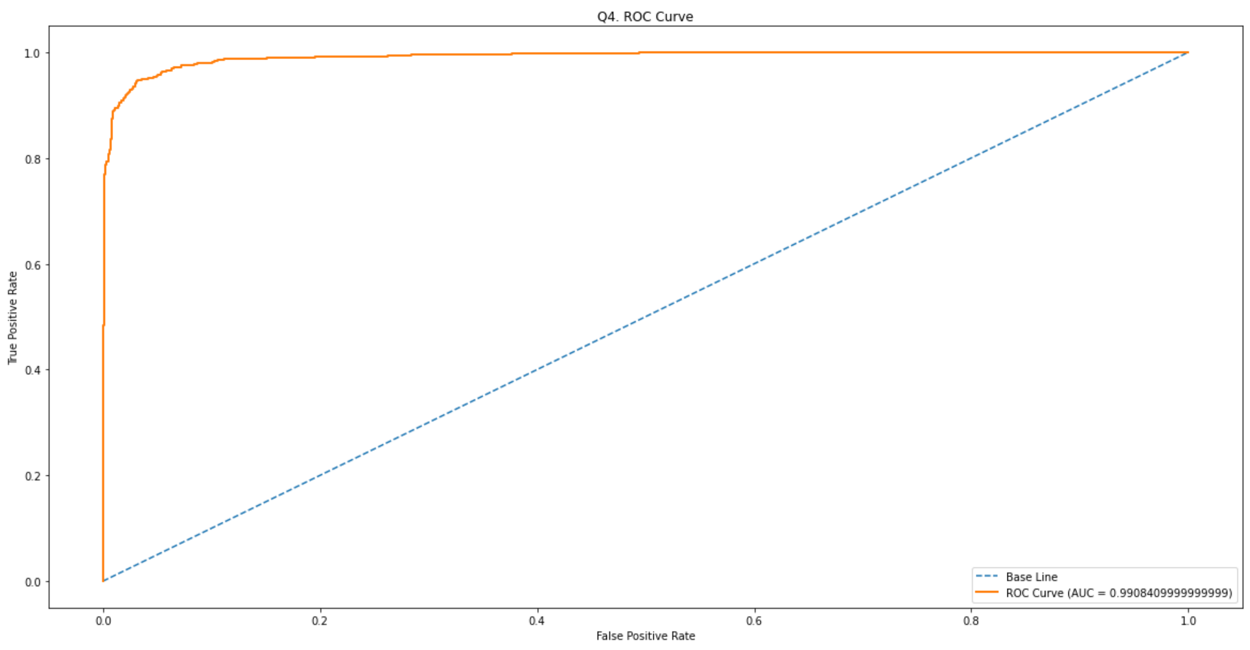


4. (2) Report the test accuracy along with the confusion matrix and the ROC curve.

- **Confusion matrix and Test Accuracy**



- **ROC Curve**


Q4. ROC Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Base Line
ROC Curve (AUC = 0.9908409999999999)