ASSIGNMENT 2

Full Name	UBIT Name	UB Number
Dharma Acha	Dharmaac	50511275
Charan Kumar	cnara	50545001

Part 1

Domain:

The given dataset is a binary classification whose target variable is either “0” or “1” . The dataset mostly consists of integers and float values. From the dataset we can infer that “f1”,”f2”, “f3”,”f4”,”f5”,”f6”,”f7” are the features and one “target” variable.

Number of samples and features in the dataset:

Number of Samples: 767

Number of Features: 7, 1 target variable

Dataset Overview:

columns	**Column Type**	**Missing values**
f1	Numerical	0
f2	Numerical	0
f3	Numerical	0
f4	Numerical	0
f5	Numerical	0
f6	Numerical	0
f7	Numerical	0

target	Numerical	0

Basic Statistics of Numerical Columns:

Column	Mean	Median	Min	Max	Standard Deviation
f1	3.849	3.00	0.00	17.00	3.37
f2	120.909	117.00	0.00	199	31.947
f3	69.118	72.00	0.00	122	19.37
f4	20.542	23.00	0.00	99	15.960
f5	80.091	36.00	0.00	846	115.37
f6	31.998	32.00	0.00	67.10	7.89
f7	0.472	0.374	0.078	2.42	0.33
target	0.349	0.00	0.00	1.00	0.477

Data Cleaning

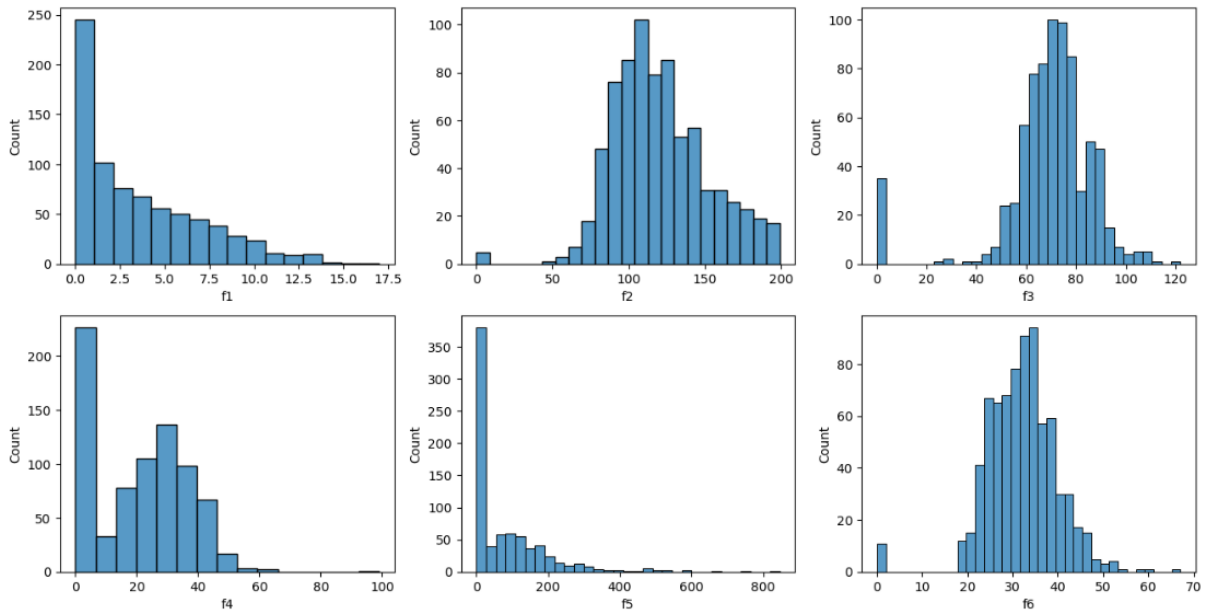
The columns [f1,f2,f3,f4,f5,f6,f7] have some noise i.e they have some categorical values. These are removed as described

1. Every column is first converted into numeric type with the help of “pd.to_numeric” when this method encounters alphabets it converts them into null values.
2. Further these null values are replaced with the median of the respective columns using df.fillna() method.

Visualisations

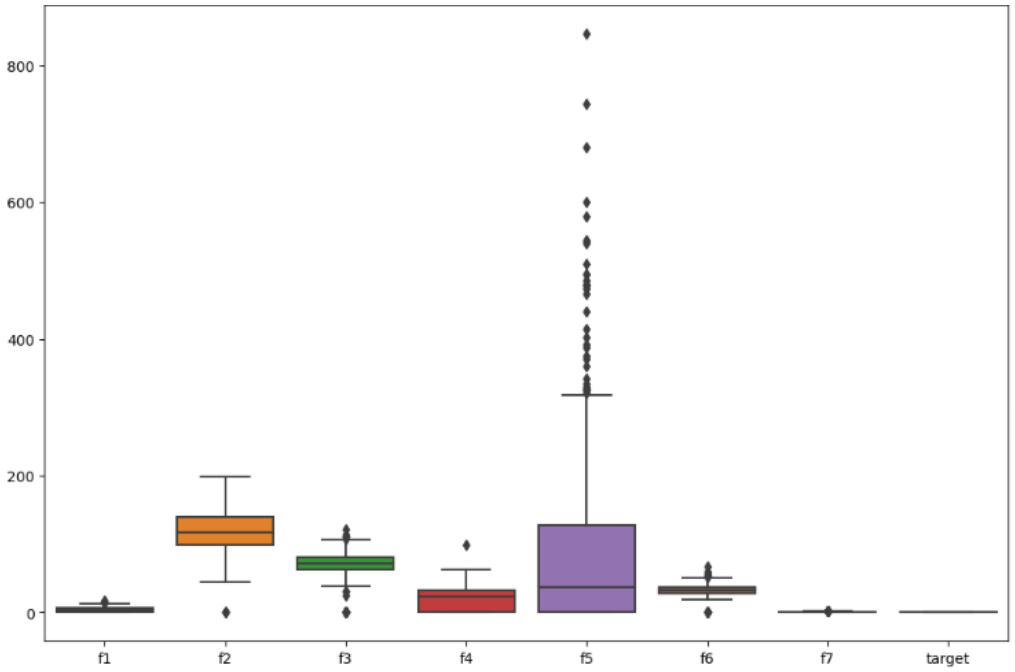
We plotted the hist plots of every column to understand the distribution of data points. The below plot represents the distribution of every point.

Plot-1



From the graphs we can say that some of the data points are skewed. The columns f1, f4, f5 distributions are completely towards the left . On the other hand, columns f2, f3, f6 distributions are normal.

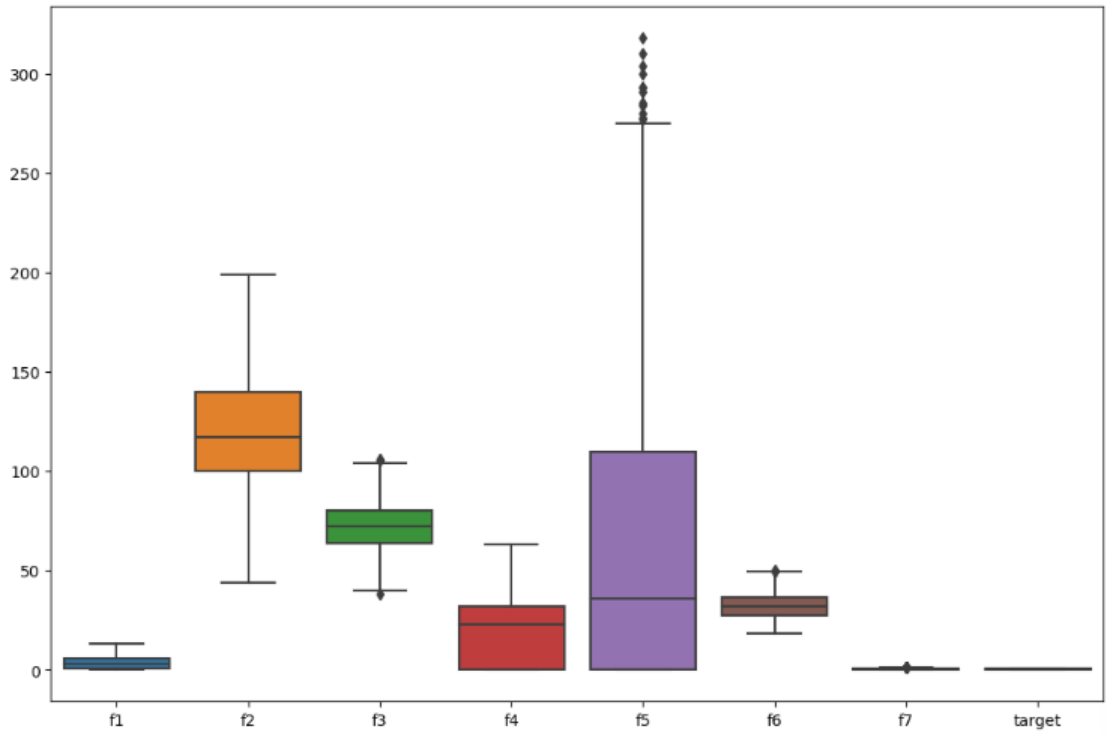
With the help of box plot we figured out the outliers of the dataset as shown below.



By using interquartile range we set the upper and lower range. And finally replaced the points which fall under lower bound or higher upper with median of the respective column.

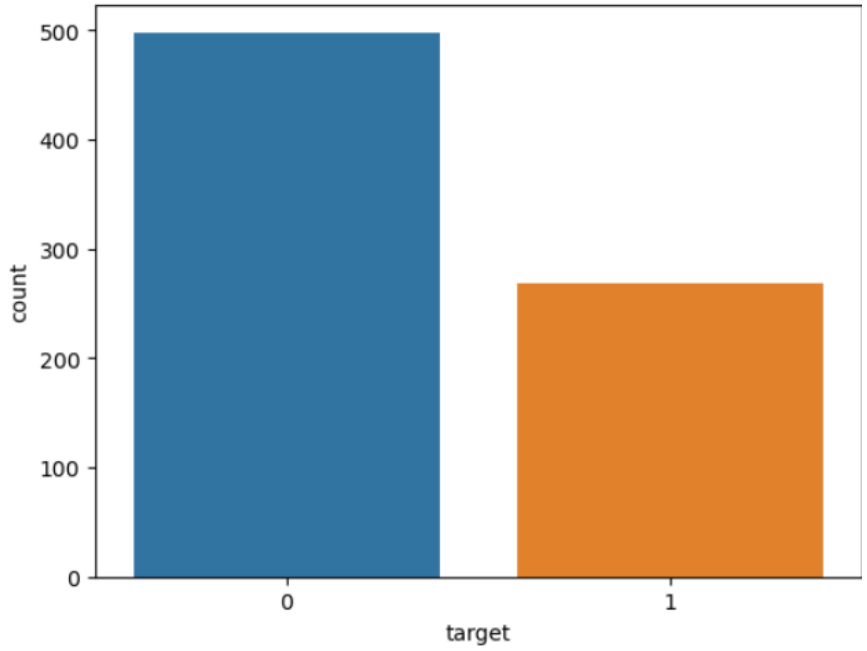
The below graph represents the box plot of every column after removing outliers. As we can see the column f2 has the highest median which roughly equals to 117. The columns f7 and target has more or less equal medians. And all other columns median lies between 0 - 75

Plot-2



Plot-3

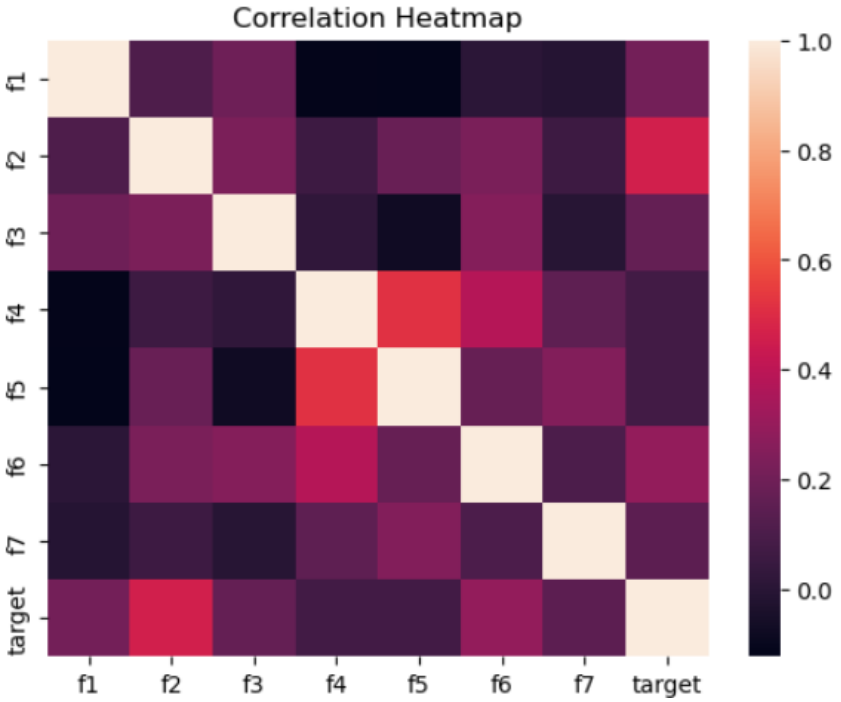
The graph depicts the frequency of values of the target variable.



From the plot we can clearly say that the count of “0” is twice that of “1” . That is nearly equal to 500. However the count of “ 1“ settled at 1.

Plot-4

The correlation heat map defines the fraction of correlation among the columns.



From the heat map it is clear that no two distinct columns have higher correlations. All the columns fall under moderate to lower correlation scale.

Splitting and Scaling the dataset

As we have already observed, the given dataset contains imbalanced classes. We used the Smote library to handle the minority class.
Oversampling using SMOTE:
oversample = SMOTE()
X, y = oversample.fit_resample(X, y)

The SMOTE stands for Synthetic Minority Over-sampling Technique. Basically it is used to handle the class imbalance in the dataset by producing samples to the minority class.
Here we are oversampling the feature matrix and target vector “y” to generate samples to the minority class.

scaler = StandardScaler(). This method created the StandardScalar class. It is used to change feature values to have “0” or “1”.

X = scaler.fit_transform(X). It scales the feature values to have 0 or 1.

Summary of the Neural network model:

Train_test_split function splits the feature variables and target variables into two subsets based on assigned test_size.

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3, random_state=42)

The X_temp data allocates 30% of data and the remaining data is used for training.

X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42). Here test size is set to 0.5. Therefore, 50 % data is set as validation data and remaining 50% data is set to testing data.

We set the batch size to 32. It means at each iteration 32 data points are trained. Later we converted data to torch tensors using torch.FloatTensor()

We implemented a basic neural network architecture as follows:

Defined NeuraNetwork class. The nn.Module is the base class of all Pytorch models. The constructor class takes three arguments.

1. input_size: The number of features in the input data.
2. hidden_size1: Number of neurons in first hidden layer
3. Hidden_size2: Number of neurons in second hidden layer

nn.Linear(input_size, hidden_size1) represents a linear layer that connects input to the first hidden layer. nn.ReLU() this ReLU activation function is after the linear layer. We introduced a dropout layer with a dropout rate of ~30%. The second hidden layer also follows the same architecture. The final layer with 1 neuron is used for binary classification, here we use sigmoid activation function which is generally used for classification models. The function forward pass() calls linearAchitecture which gives the output of the neural network.

```
Out[390]: ==========================================================================
          ===========
          Layer (type:depth-idx)                   Output Shape              Param #
          ==========================================================================
          ===========
          NeuralNetwork                            [32, 1]                   --
          ├─Sequential: 1-1                        [32, 1]                   --
          │    └─Linear: 2-1                       [32, 128]                 1,024
          │    └─ReLU: 2-2                         [32, 128]                 --
          │    └─Dropout: 2-3                      [32, 128]                 --
          │    └─Linear: 2-4                       [32, 64]                  8,256
          │    └─ReLU: 2-5                         [32, 64]                  --
          │    └─Dropout: 2-6                      [32, 64]                  --
          │    └─Linear: 2-7                       [32, 1]                   65
          │    └─Sigmoid: 2-8                      [32, 1]                   --
          ==========================================================================
          ===========
          Total params: 9,345
          Trainable params: 9,345
          Non-trainable params: 0
          Total mult-adds (Units.MEGABYTES): 0.30
          ==========================================================================
          ===========
          Input size (MB): 0.00
          Forward/backward pass size (MB): 0.05
          Params size (MB): 0.04
          Estimated Total Size (MB): 0.09
          ==========================================================================
          ===========
```

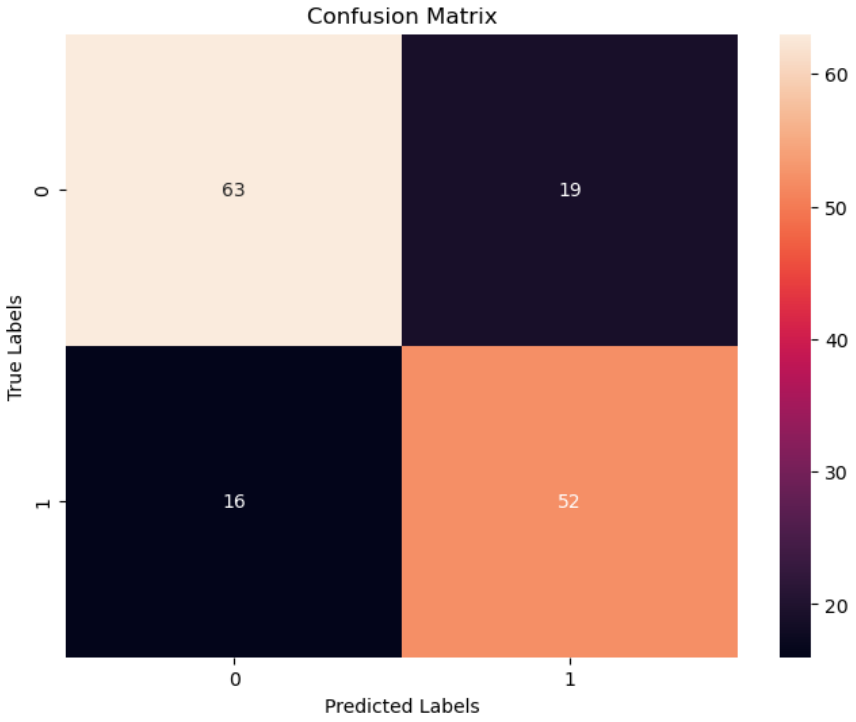
BCELoss():

nn.BCELoss() stands for binary cross-entropy loss which is used for binary classification problems. And we use an optimizer to update the model's parameters during the training. Here we Stochastic Gradient (SGD). model.parameters retrieve the learned params and updates.
In the training step, it iterates over the training data in batches. For each step it extracts inputs and labels and the model(inputs) makes predictions. Finally after evaluating the test data we obtained:

Time to train: 0.53 seconds
Test Loss: 0.50
Test Accuracy: 76.67%
Precision: 0.73
Recall: 0.76
F1 Score: 0.75
ROC AUC: 0.77

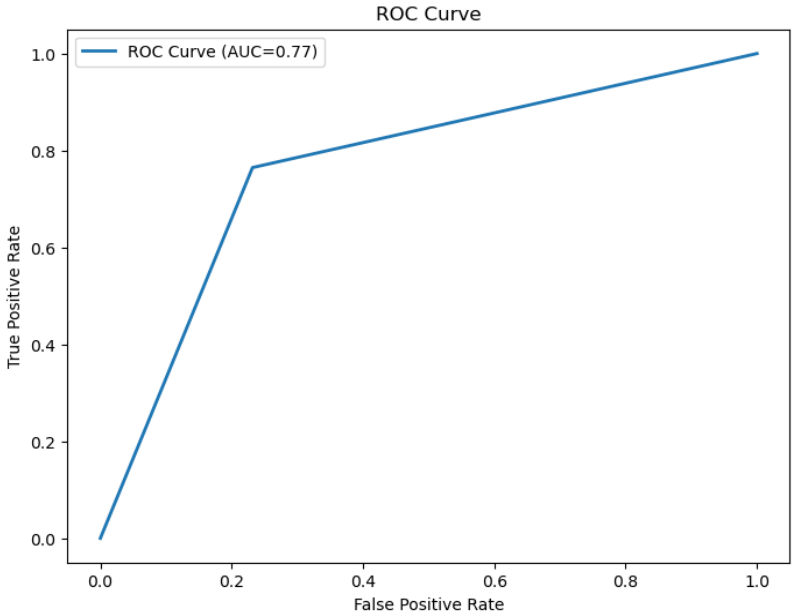
Confusion matrix:

The plot represents the confusion matrix of the model. It describes the total number of correct predictions and also wrong predictions.



From the plot we can say the model correctly predicted 52 1's and 63 0's and wrongly predicted 16 1's and 19 0's

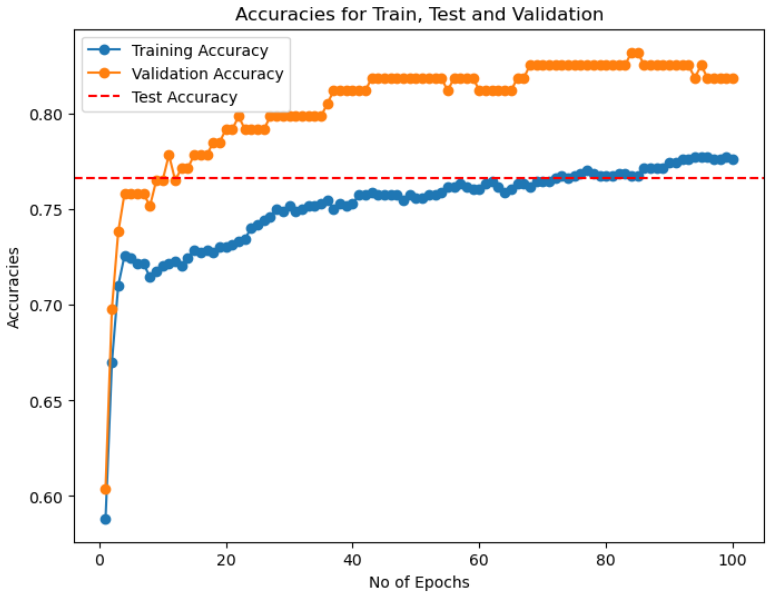
ROC Curve: It stands for Receiver Operating Characteristic curve, It plots the graph between True positive rate and False positive rate.



Roc_auc can be calculated by using the method **roc_auc_score(Actual values, predictions). So** calculating the **ROC** we obtained 0.77.

Training, Validation and test accuracy plot:

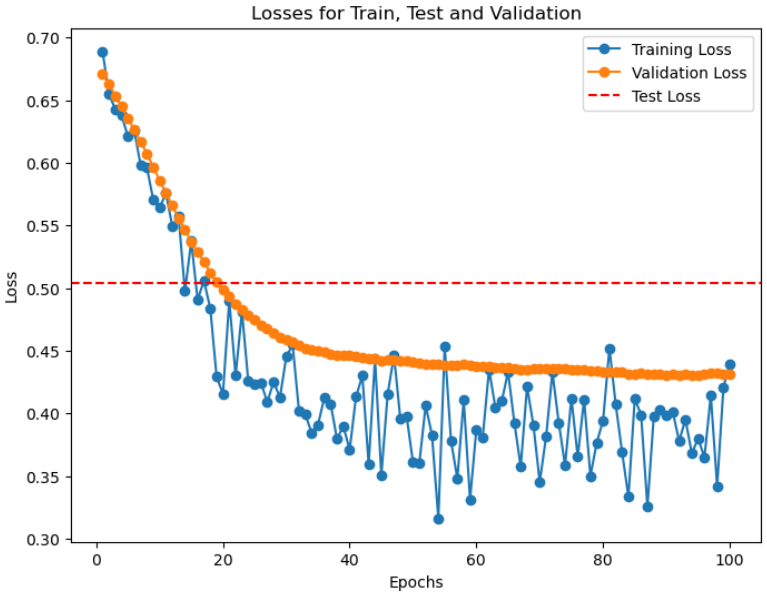
The plot is between accuracy and number of Epochs.



From the graph The test accuracy is 76.67 % and training accuracy is increasing for every epoch which is greater than 70% and validation accuracy is greater than 80%.

Training, Validation and test loss plot

The plot is between loss and number of Epochs.



From the graph we can infer that increasing the number of epochs the validation loss is decreasing drastically and test loss is constant whereas training loss is fluctuating as we increase the number of epochs.

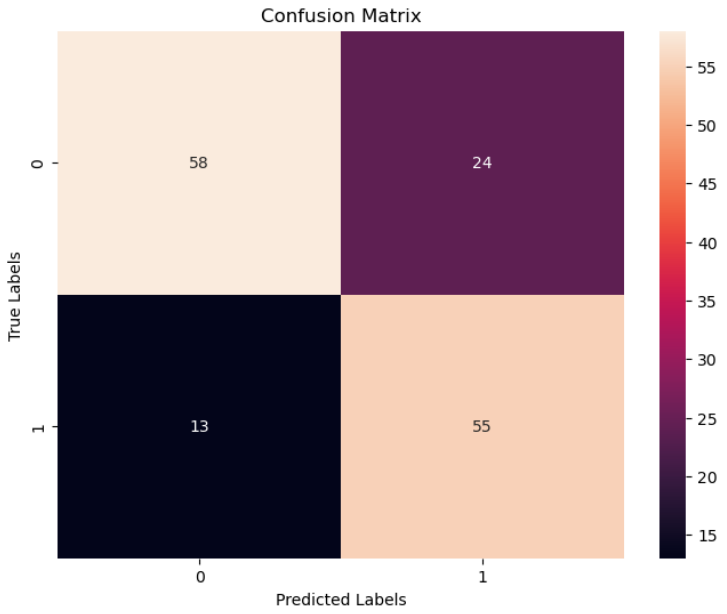
Part II

Hyperparameter Tuning -Dropout -0.1

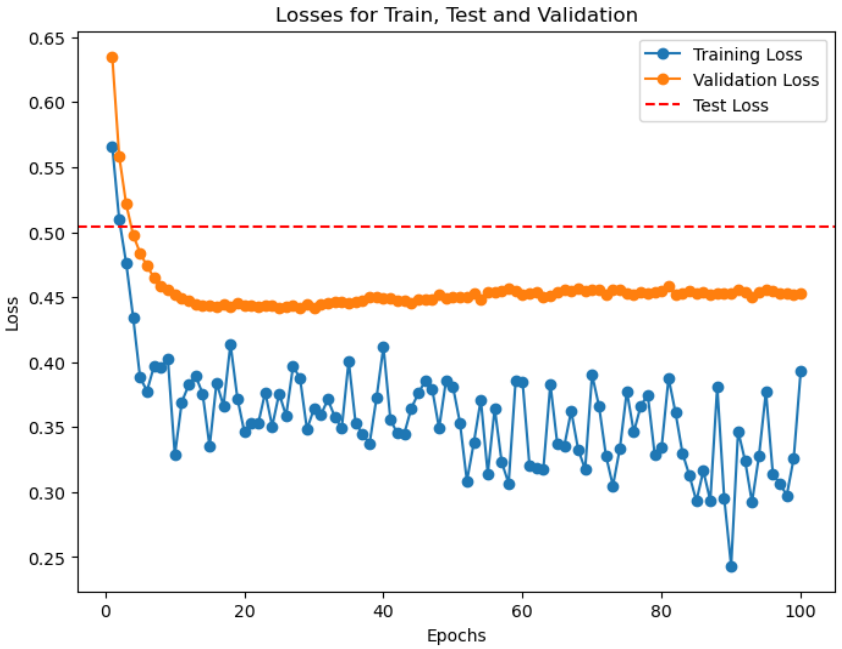
Training with Dropout Rate: 0.1
Time to train model: 1.63 seconds
ROC: 0.76
Test Loss: 0.57
Precision: 0.70
F1 Score: 0.75
Recall: 0.81
Test Accuracy: 75.33%

	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Step3	Test Accuracy
Dropout	0.1	75.33	0.3	77.33	0.5	76
Optimizer	SGD		SGD		SGD	
Activation Function	Relu		Relu		Relu	
Initializer						

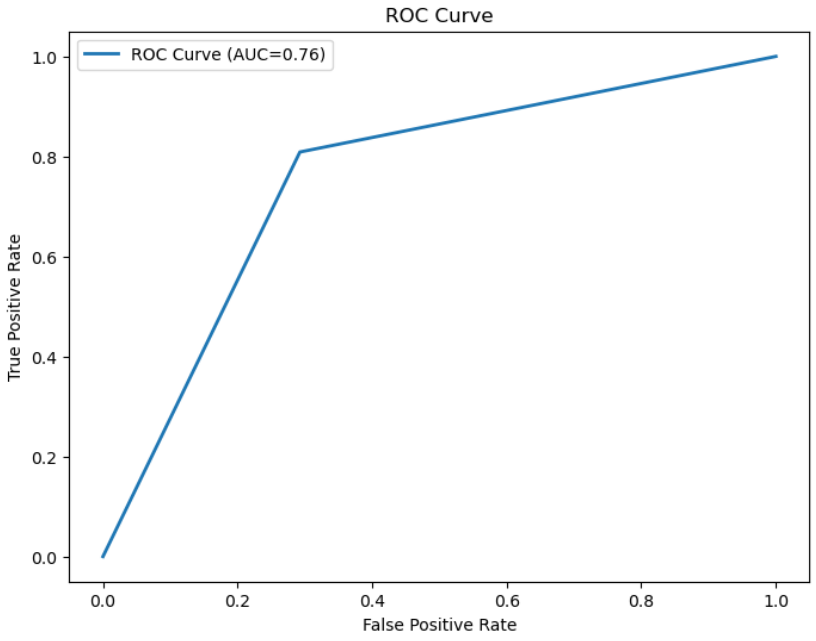
Visualisation for Dropout -0.1:



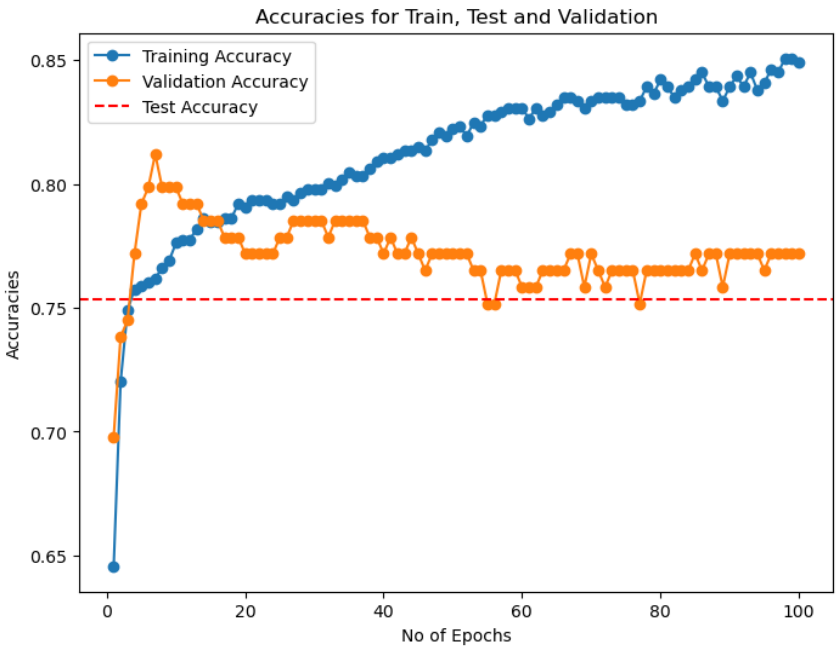
Losses for Train, Test and validation.



ROC Curve



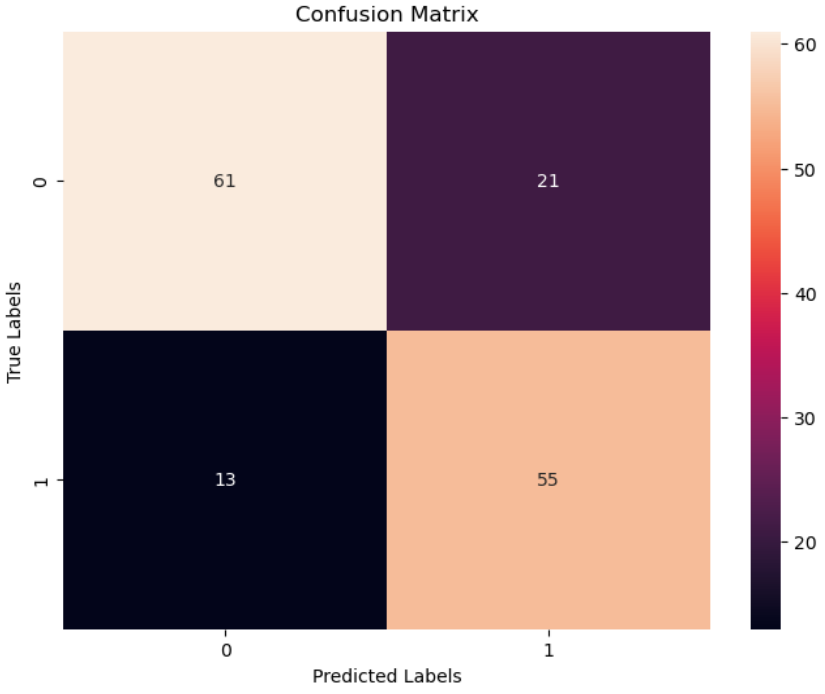
Accuracies for Train, Test and validation



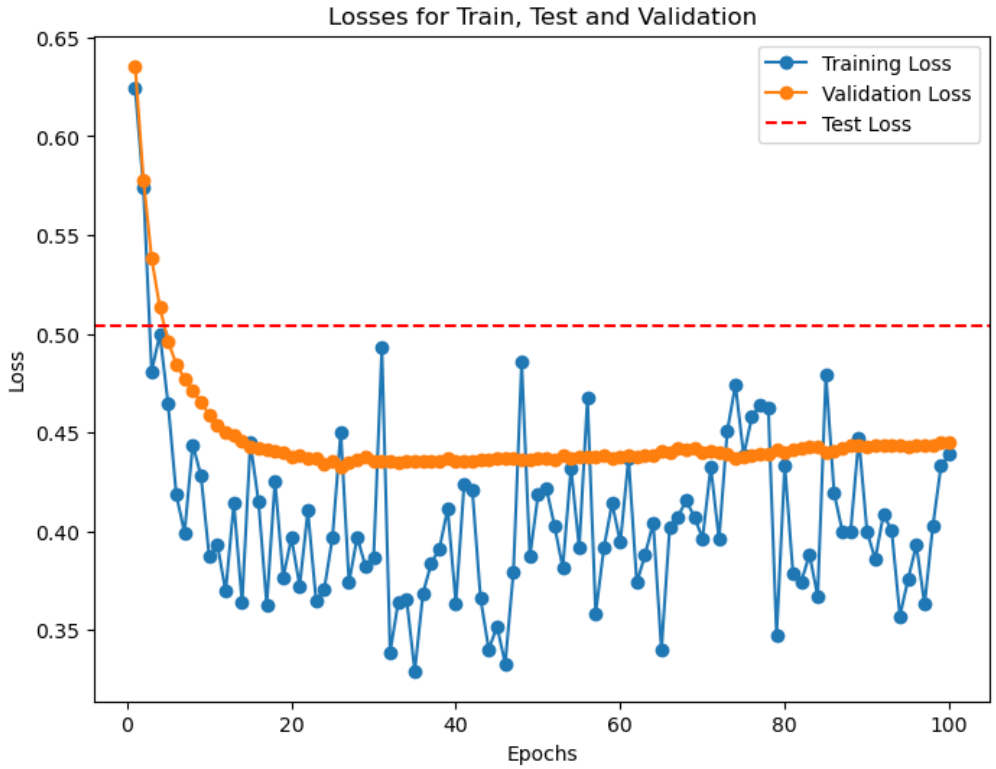
Hyperparameter Tuning -Dropout -0.3

Training with Dropout Rate: 0.3
Time to train model: 1.24 seconds
ROC: 0.78
Test Loss: 0.53
Precision: 0.72
F1 Score: 0.76
Recall: 0.81
Test Accuracy: 77.33%

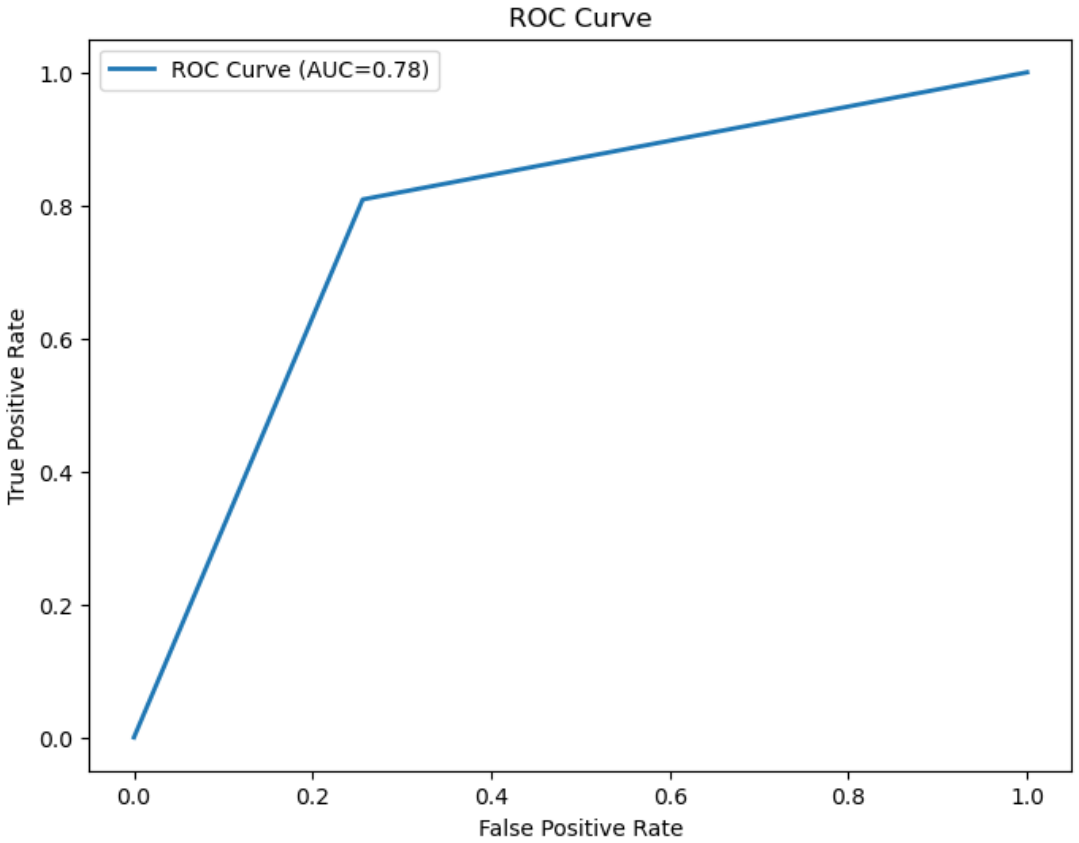
Visualisation graphs



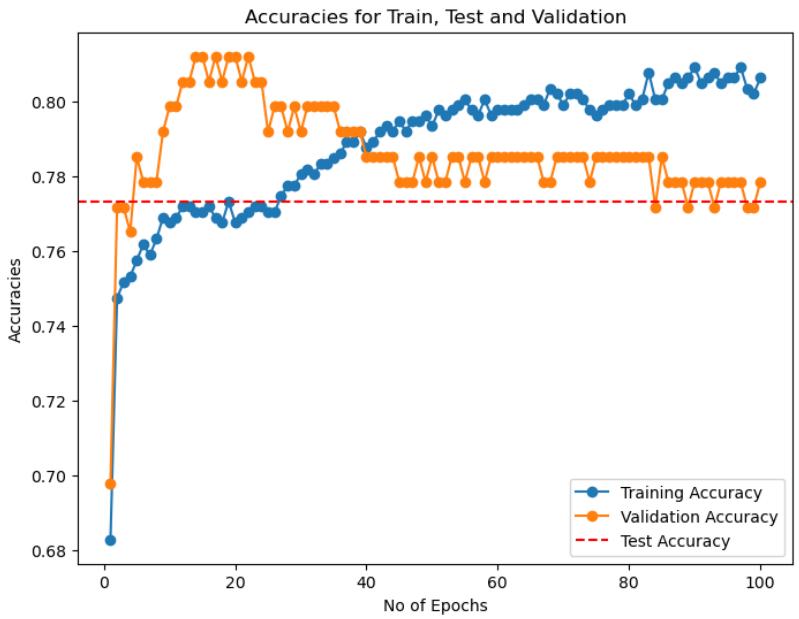
Losses for train, test and validation



ROC Curve



Accuracies for Train, Test and validation



Hyperparameter Tuning -Dropout -0.5

Training with Dropout Rate: 0.5

Time to train model: 1.47 seconds

ROC: 0.76

Test Loss: 0.51

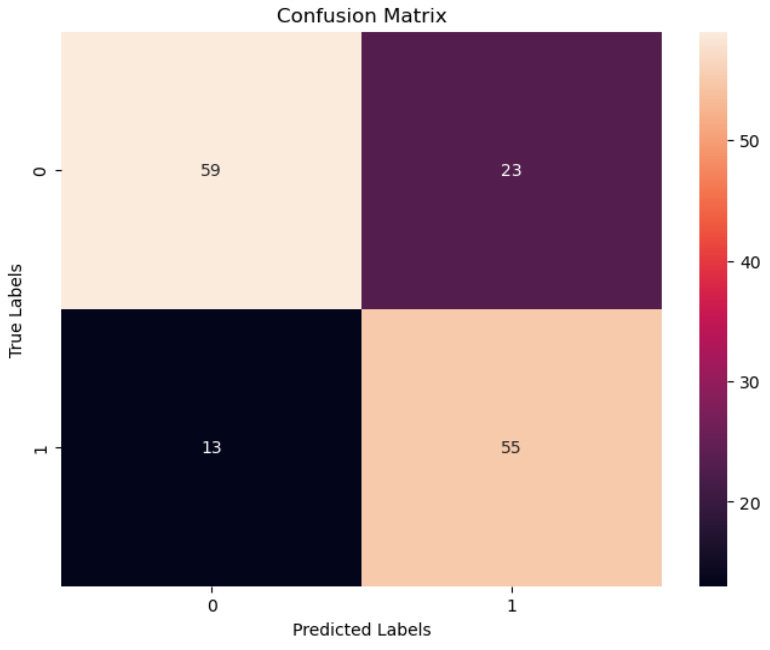
Precision: 0.71

F1 Score: 0.75

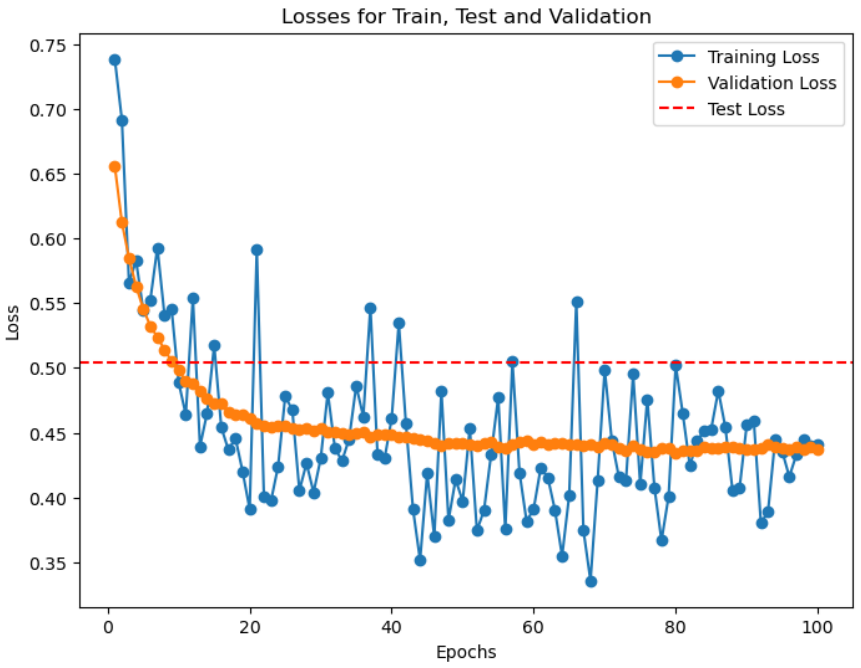
Recall: 0.81

Test Accuracy: 76.00%

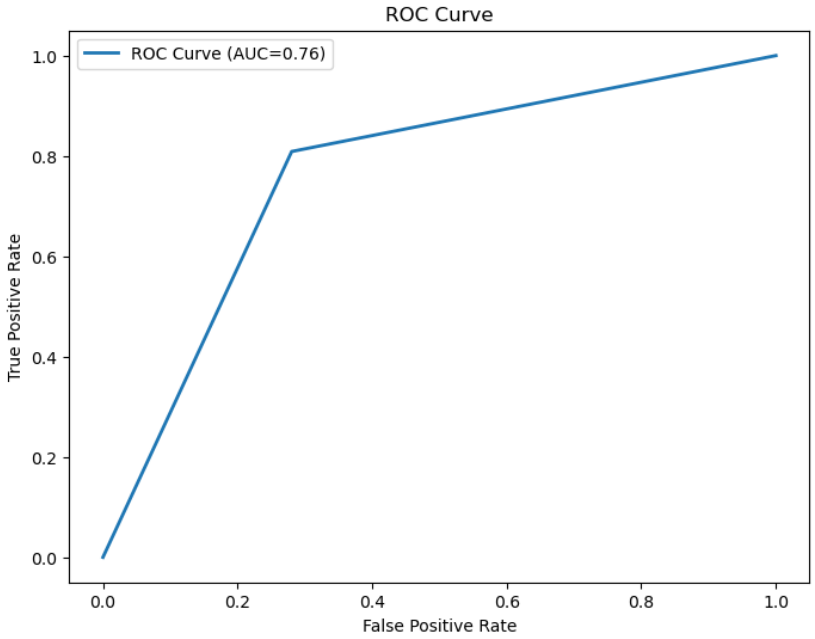
Visualisation graphs



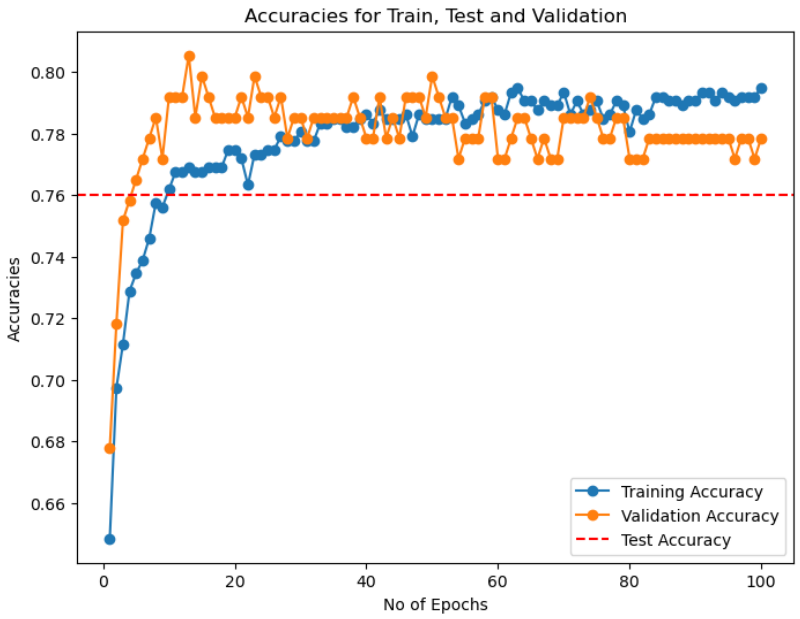
Losses for Train, Test and validation



ROC curve



Accuracies fr train, test and validation

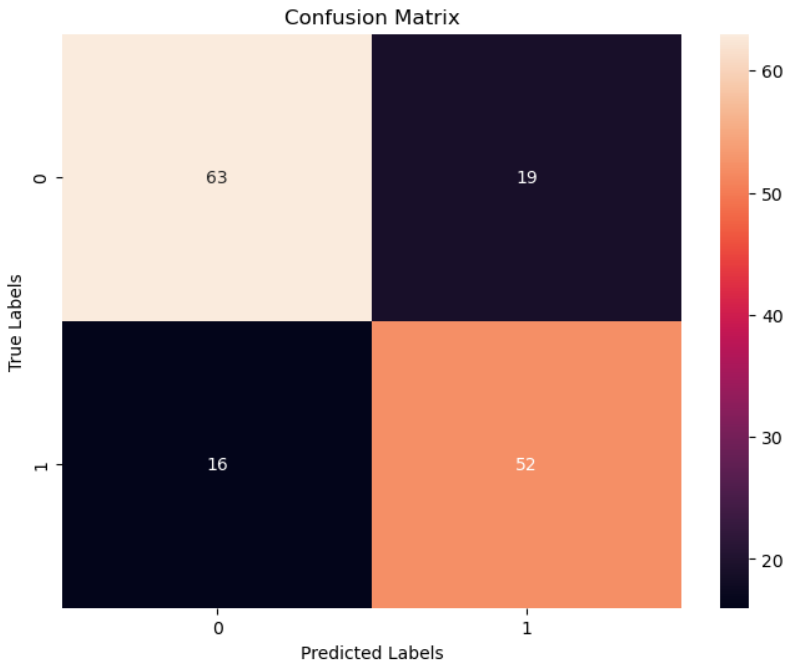


Hyperparameter Tuning -Initializers

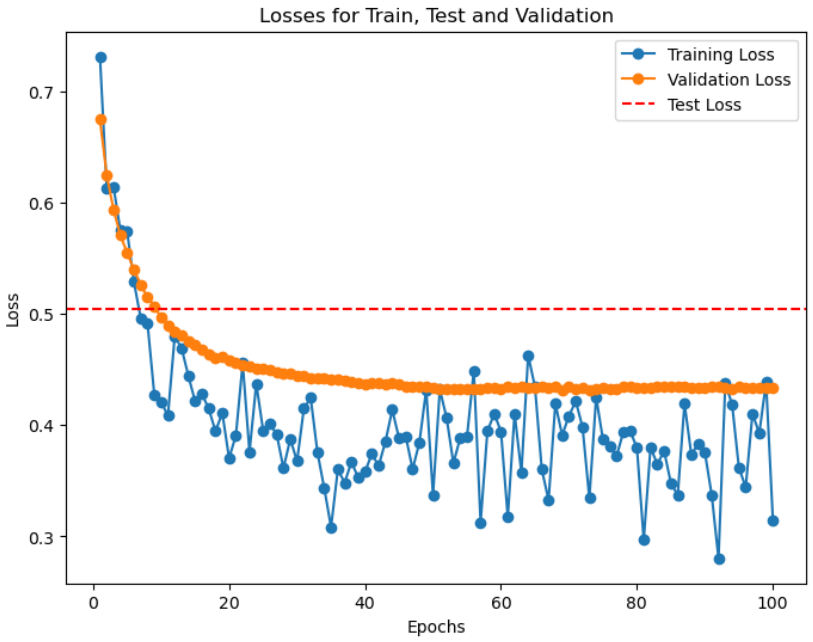
	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Step3	Test Accuracy
Dropout	0.31	76.67	0.31	78		76.67
Optimizer	SGD		SGD		SGD	
Activation Function	Relu		Relu		Relu	
Initializer	Xavier		He		uniform	

Training with Initializers : **Xavier**
Time to train model: 1.58 seconds
ROC: 0.77
Test Loss: 0.50
Precision: 0.73
F1 Score: 0.75
Recall: 0.76
Test Accuracy: 76.67%

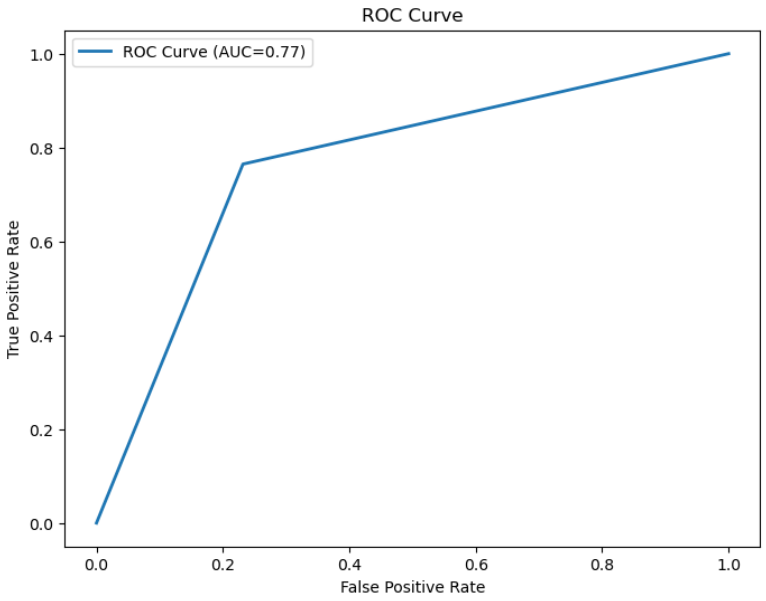
Confusion Matrix:



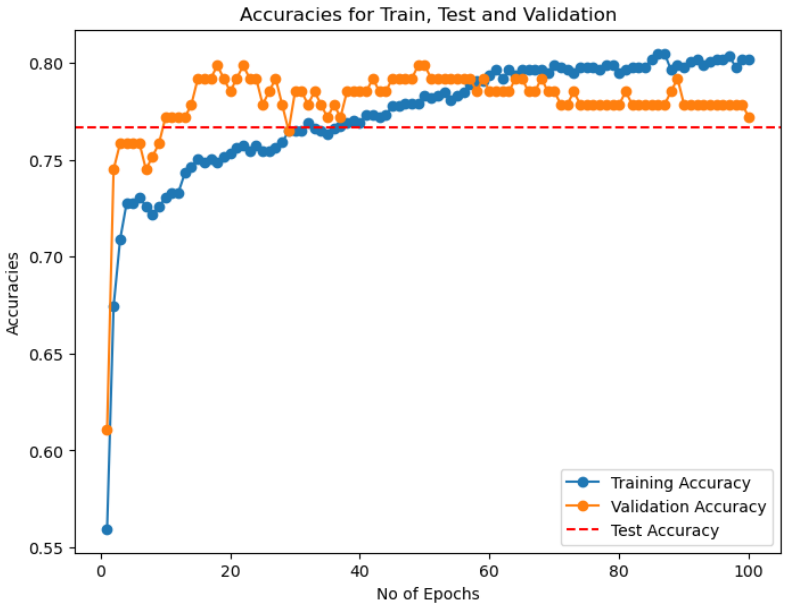
Losses for Train, test and validation



ROC Curve

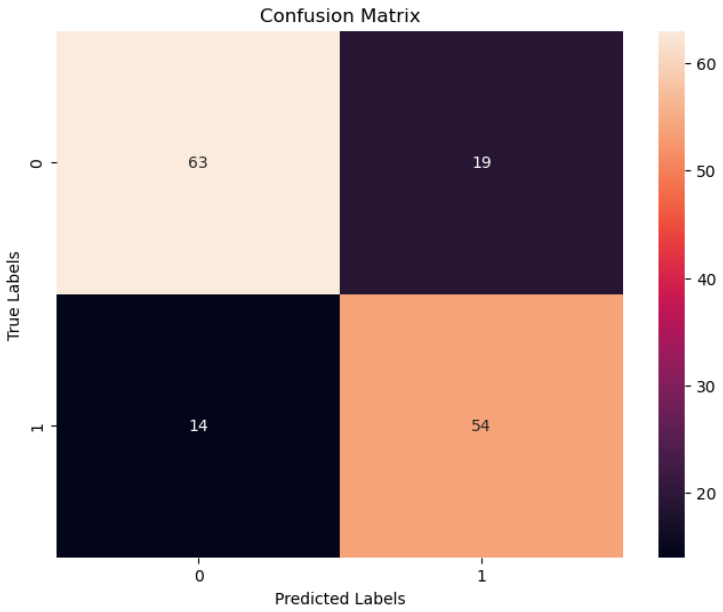


Accuracies for Train, Test and validation

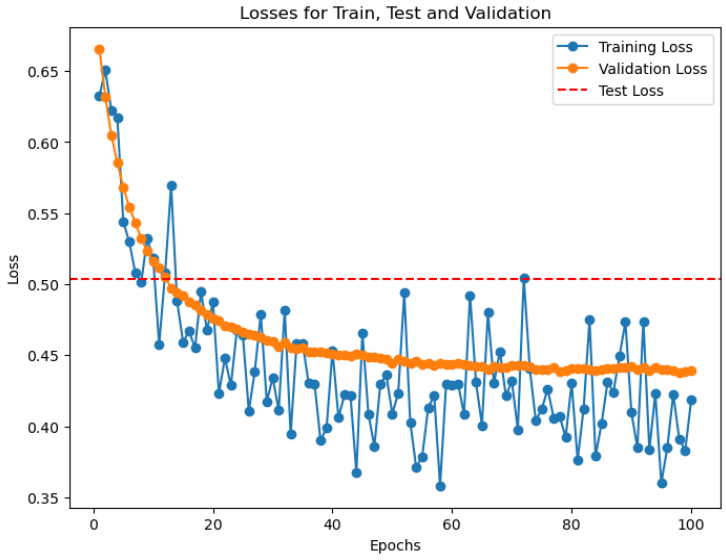


Training with Initializers : **He**
Time to train model: 1.67 seconds
ROC: 0.78
Test Loss: 0.50
Precision: 0.74
F1 Score: 0.77
Recall: 0.79
Test Accuracy: 78.00%

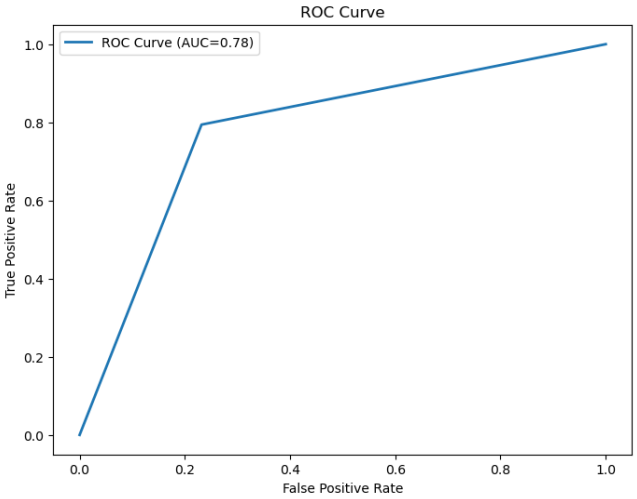
Confusion Matrix



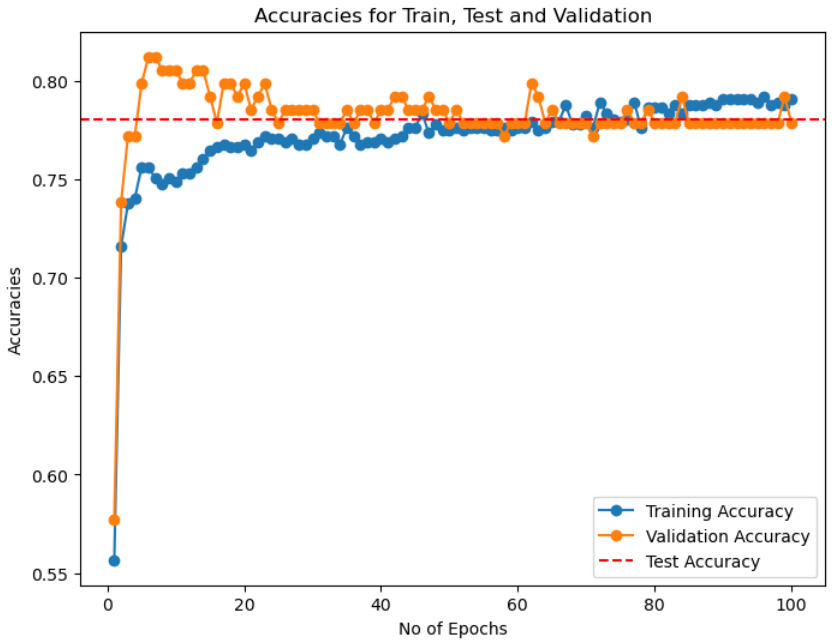
Losses for Train, Test and validation



ROC curve

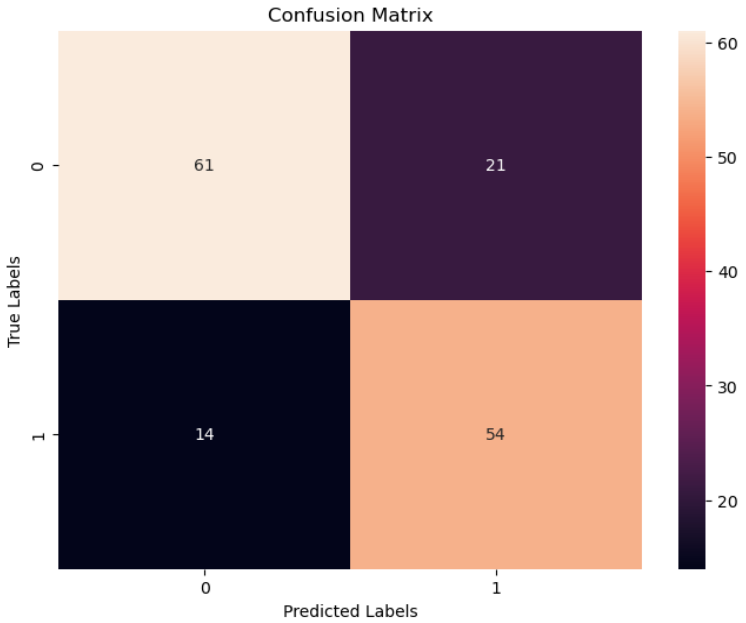


Accuracies for train,test and validation

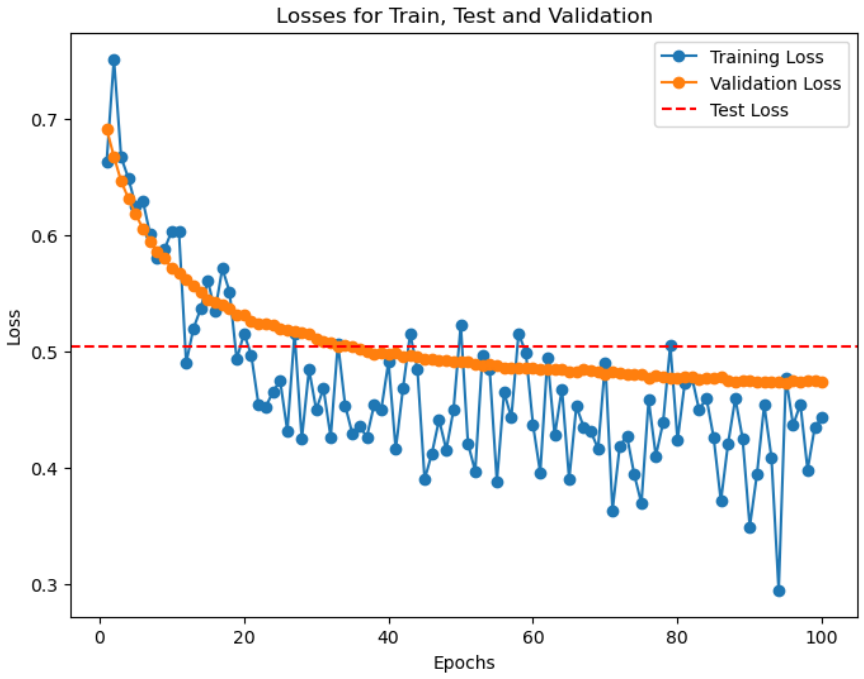


Training with Initializers : **uniform**
Time to train model: 1.55 seconds
ROC: 0.77
Test Loss: 0.50
Precision: 0.72
F1 Score: 0.76
Recall: 0.79
Test Accuracy: 76.67%

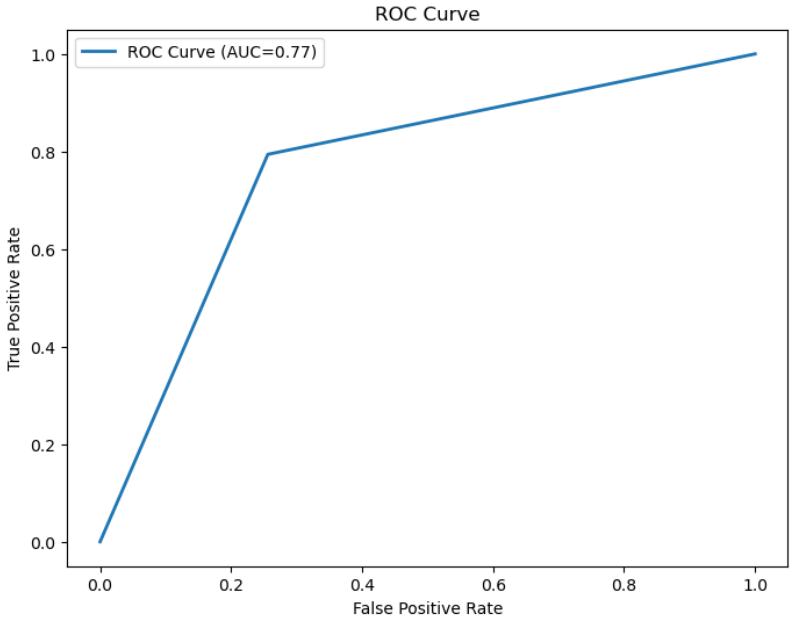
Confusion matrix



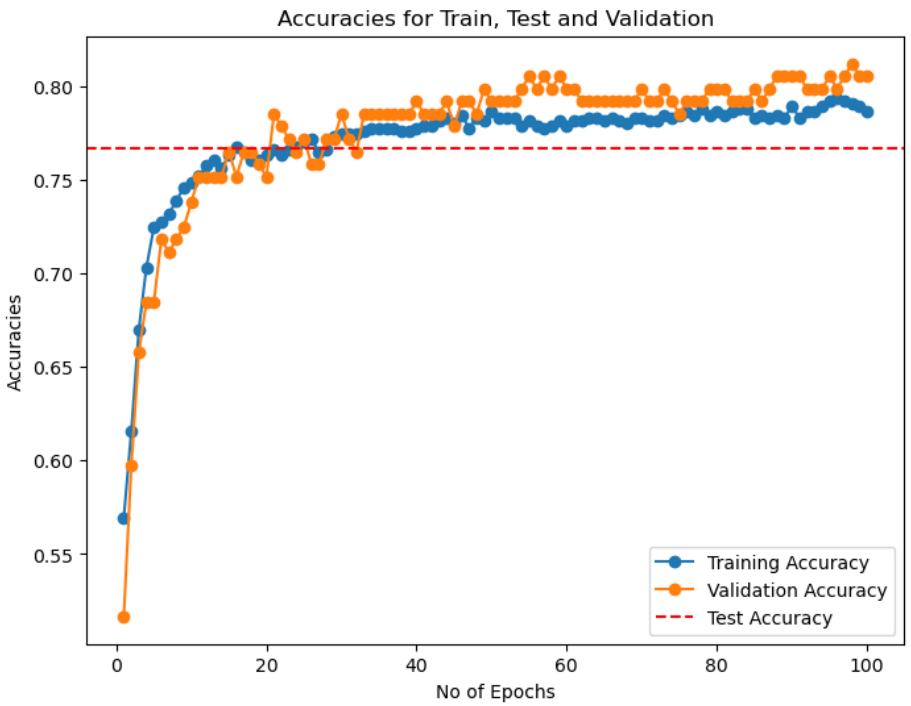
Losses for Train, Test and validation



ROC Curve



Accuracies for Train,test and validation

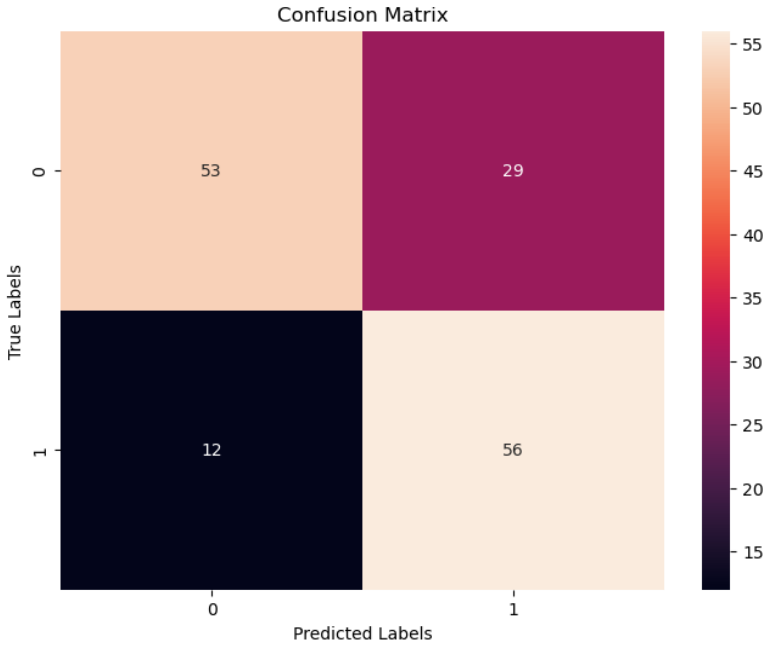


Hyperparameter Tuning -Activation function

	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Step3	Test Accuracy
Dropout	0.31	72.67	0.31	73.33	0.31	76
Optimizer	SGD		SGD		SGD	
Activation Function	ReLu		Leaky Relu		Elu	
Initializer						

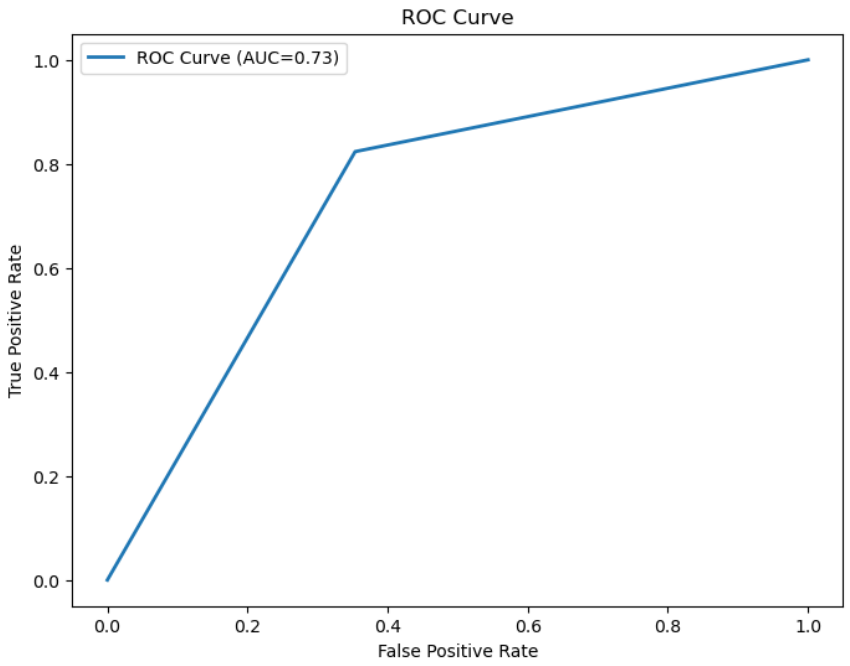
Training with Activation Function: **ReLU**
Time to train model: 2.04 seconds
ROC: 0.73
Test Loss: 0.84
Precision: 0.66
F1 Score: 0.73
Recall: 0.82
Test Accuracy: 72.67%

Confusion matrix

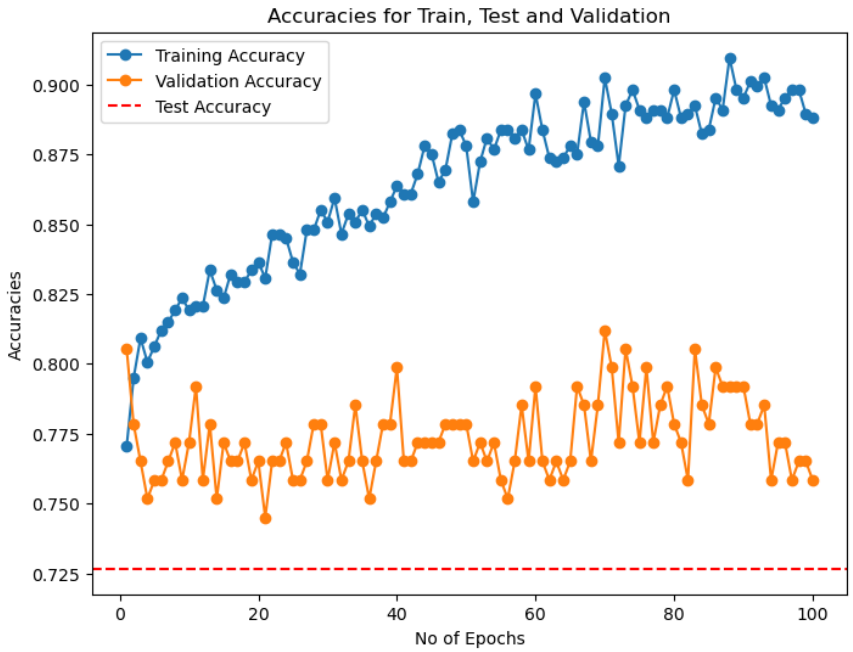


Losses for Train,Test and validation

ROC curve

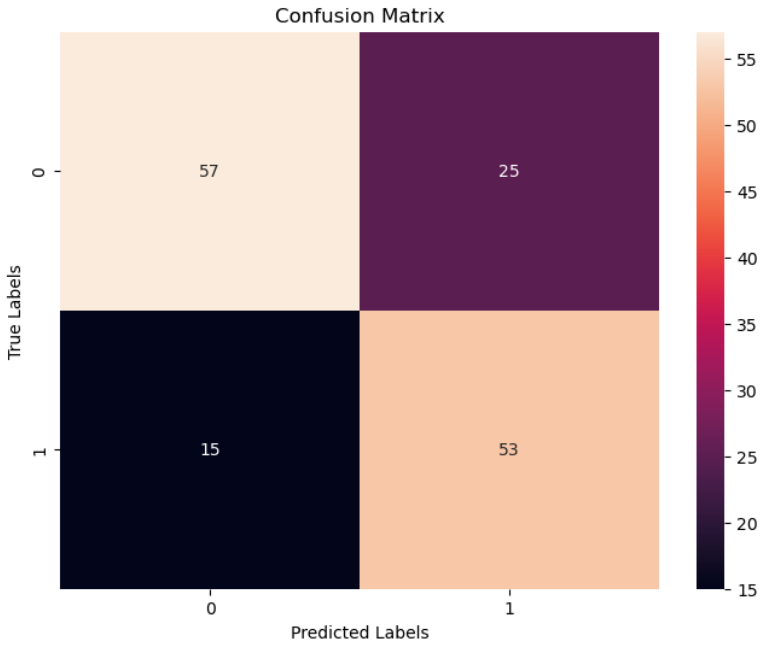


Accuracies for Train,Test and validation

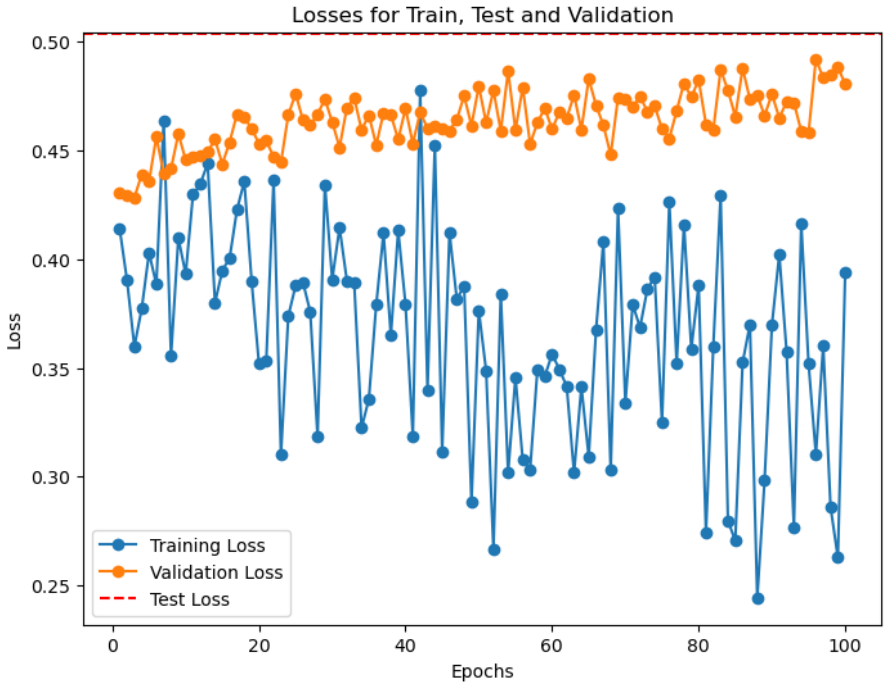


Training with Activation Function: **LeakyReLU**
Time to train model: 1.92 seconds
ROC: 0.74
Test Loss: 0.61
Precision: 0.68
F1 Score: 0.73
Recall: 0.78
Test Accuracy: 73.33%

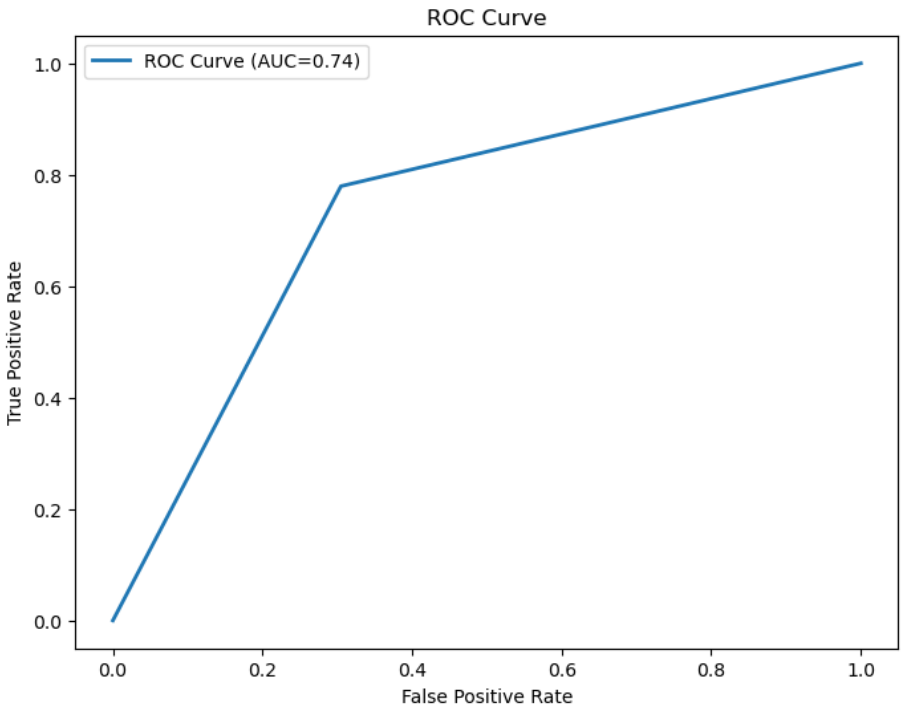
Confusion Matrix



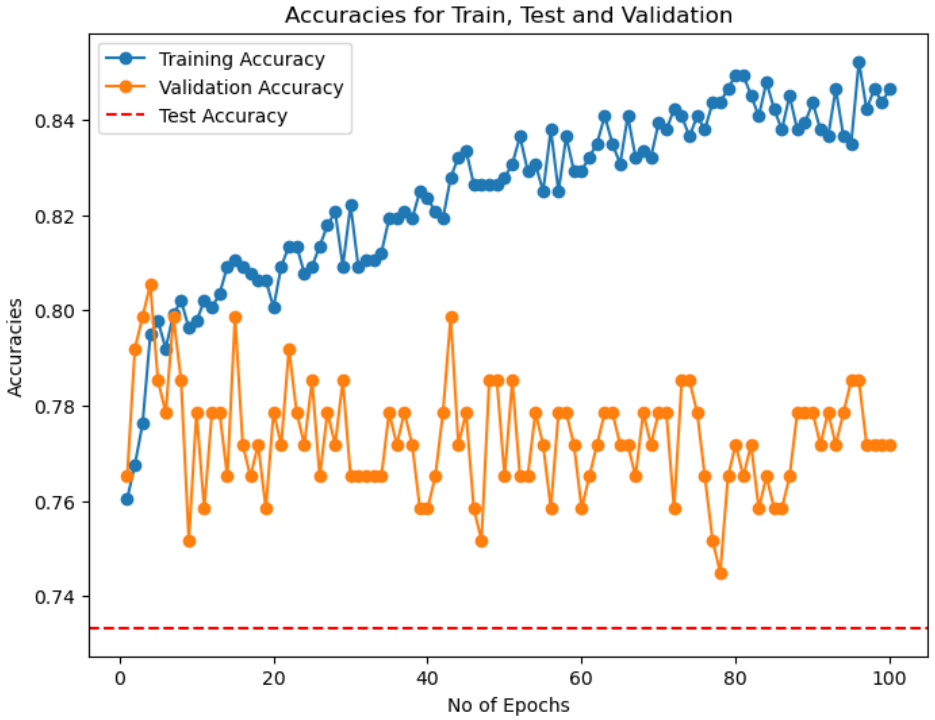
Losses for Train,Test and validation



ROC curve

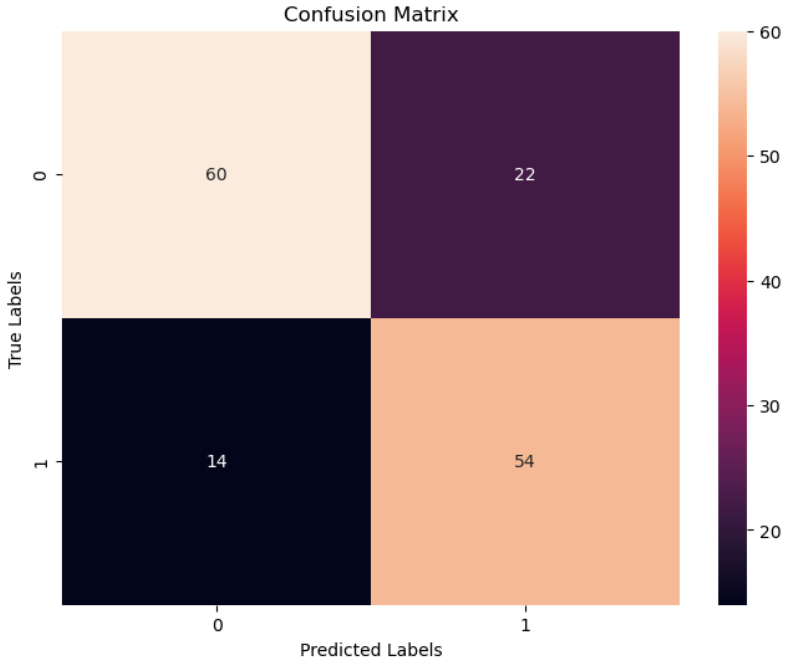


Accuracies for Train , test and validation

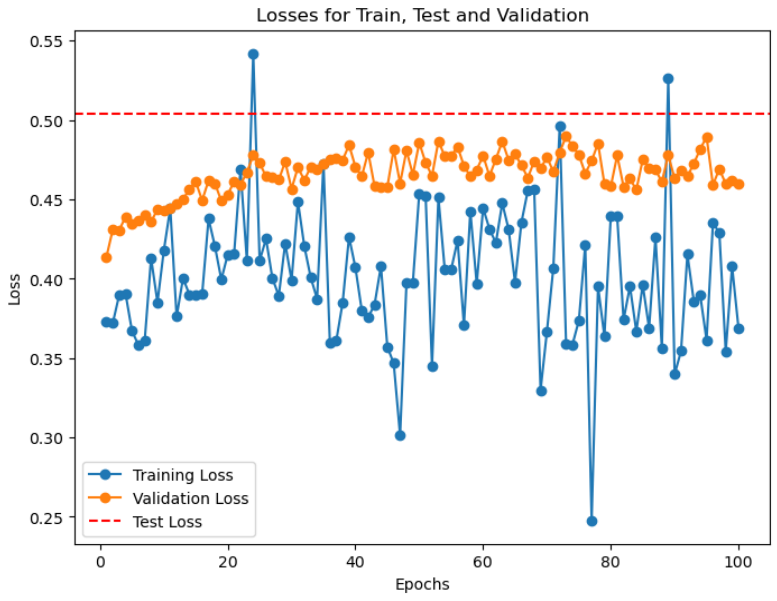


Training with Activation Function: **ELU**
Time to train model: 2.11 seconds
ROC: 0.76
Test Loss: 0.57
Precision: 0.71
F1 Score: 0.75
Recall: 0.79
Test Accuracy: 76.00%

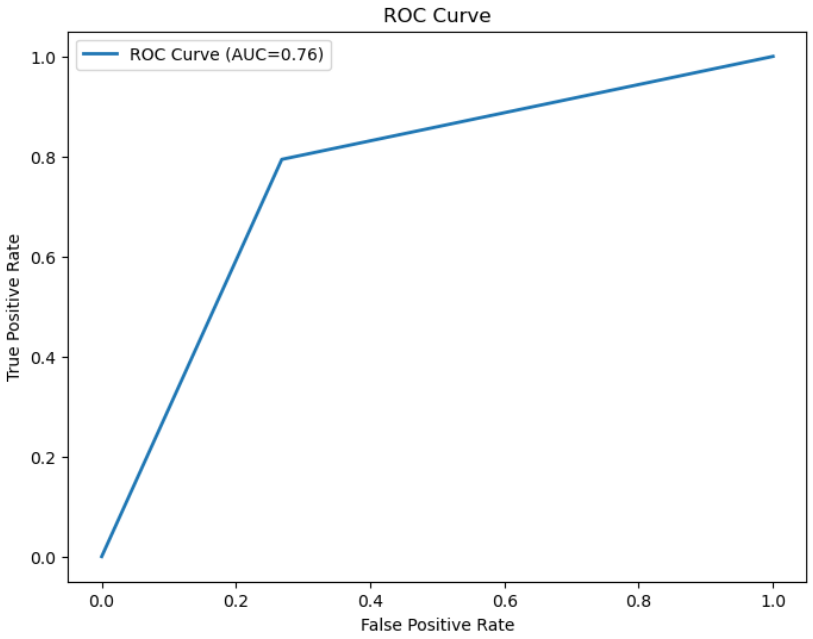
Confusion Matrix



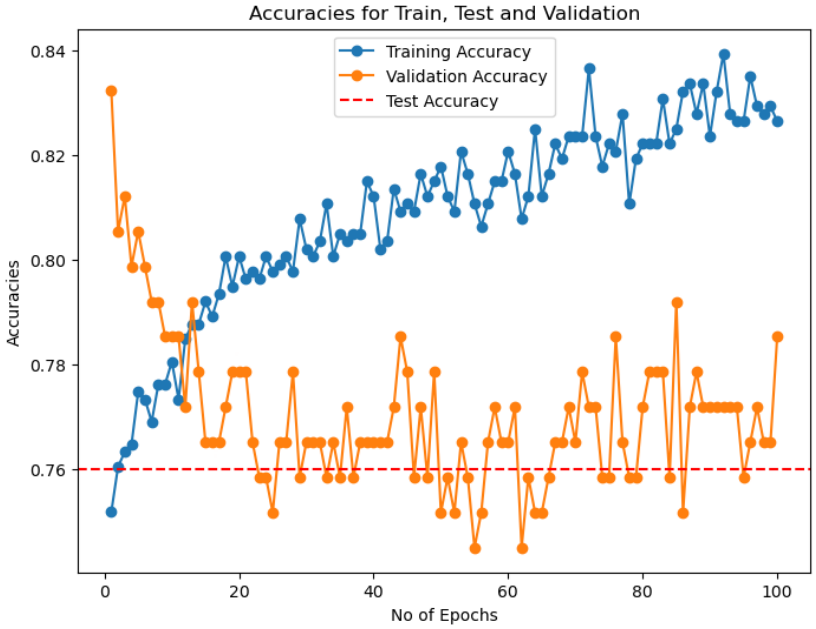
Losses for Train, Test and validation



ROC Curve



Accuracies for Train, Test and validation



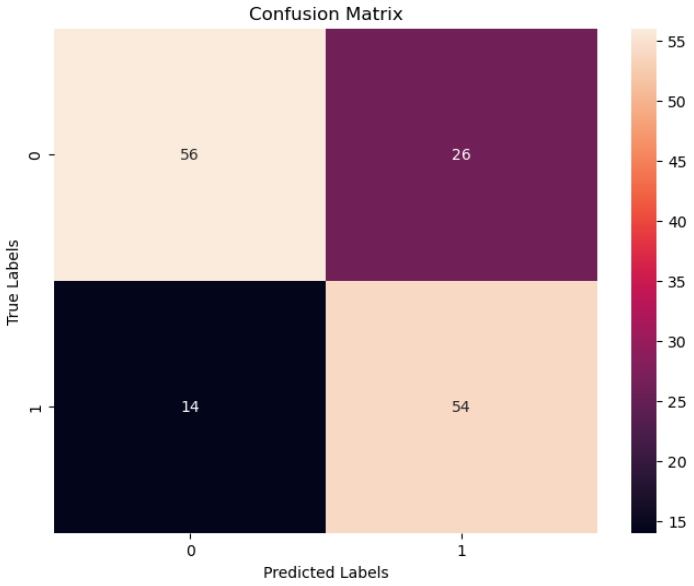
Hyperparameter Tuning -Optimizer

	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Step3	Test Accuracy
Dropout	0.31	73.33	0.31	69.33	0.31	74.67
Optimizer	Adam		Adamax		RMSProp	

Activation Function	ReLu		Relu		Relu	
Initializer						

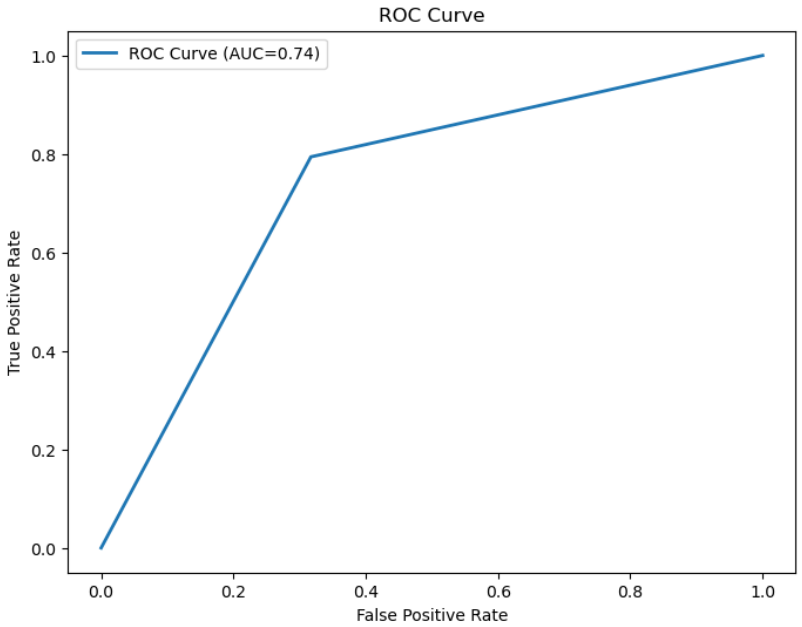
Training with Optimizers **: Adam**
Time to train model: 2.05 seconds
ROC: 0.74
Test Loss: 1.00
Precision: 0.68
F1 Score: 0.73
Recall: 0.79
Test Accuracy: 73.33%

Confusion Matrix

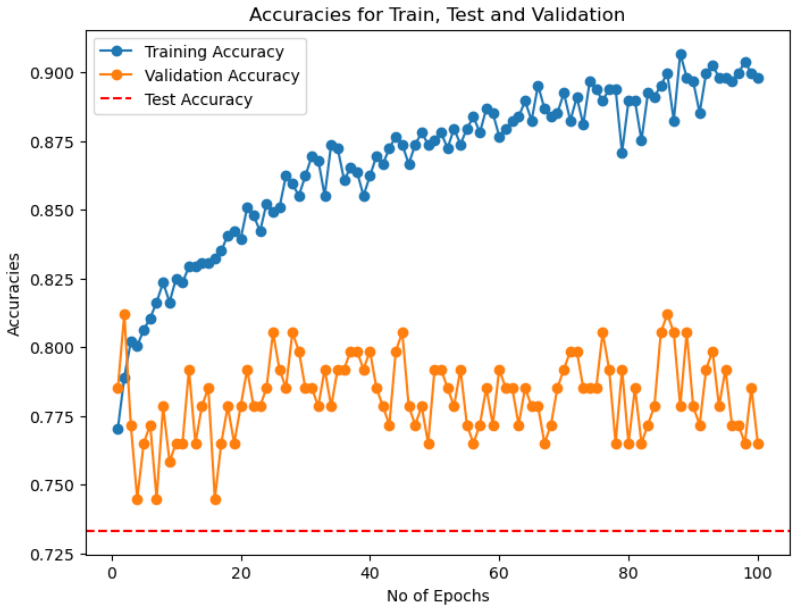


Losses for Train, Test and validation

ROC curve

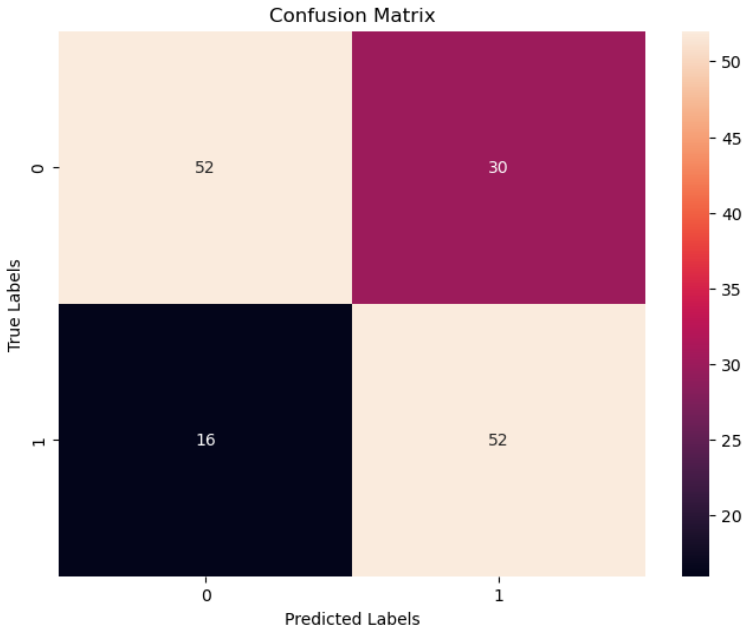


Accuracies for Train, Test and validation

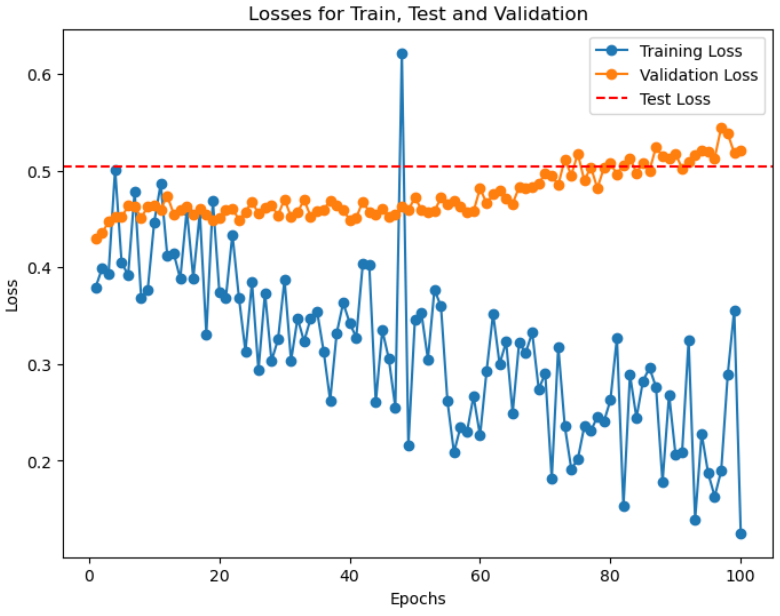


Training with Optimizers : **Adamax**
Time to train model: 1.98 seconds
ROC: 0.70
Test Loss: 0.72
Precision: 0.63
F1 Score: 0.69
Recall: 0.76
Test Accuracy: 69.33%

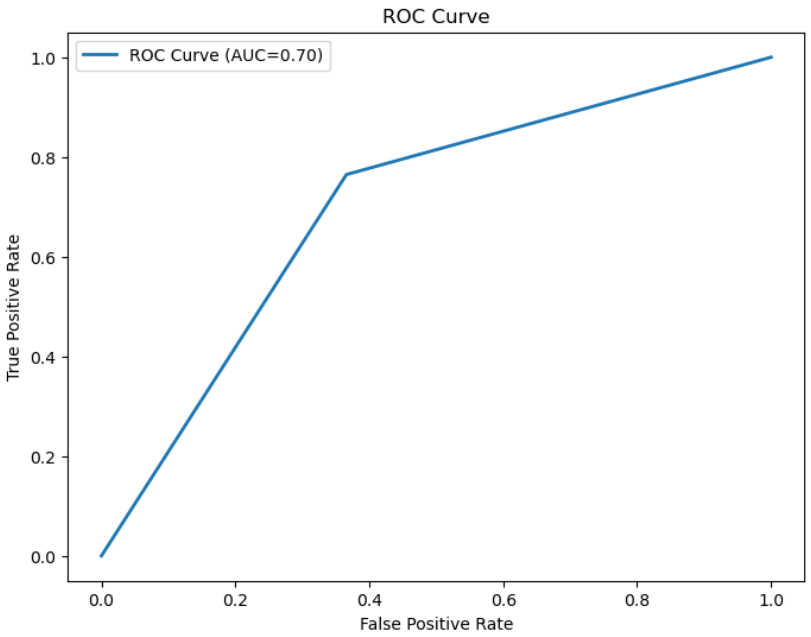
Confusion matrix



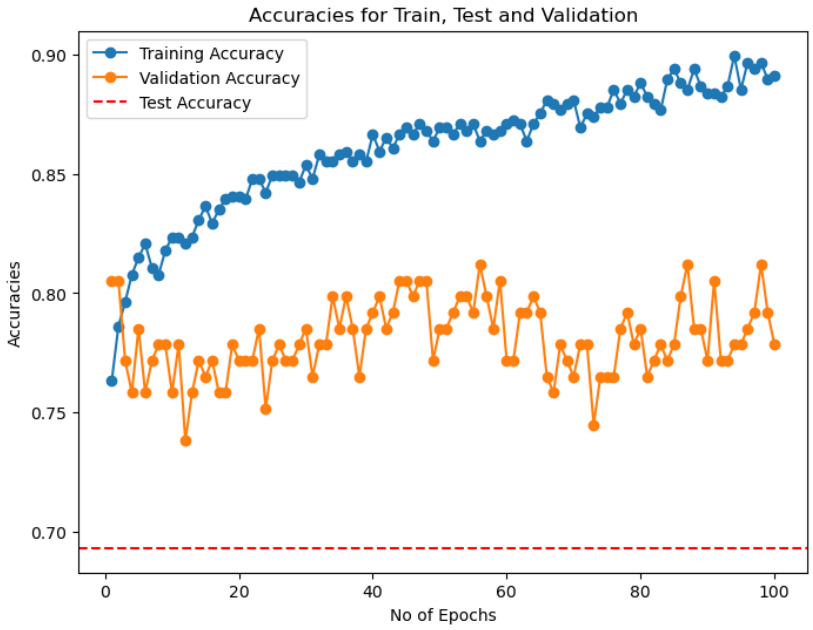
Losses for Train, Test and validation



ROC curve

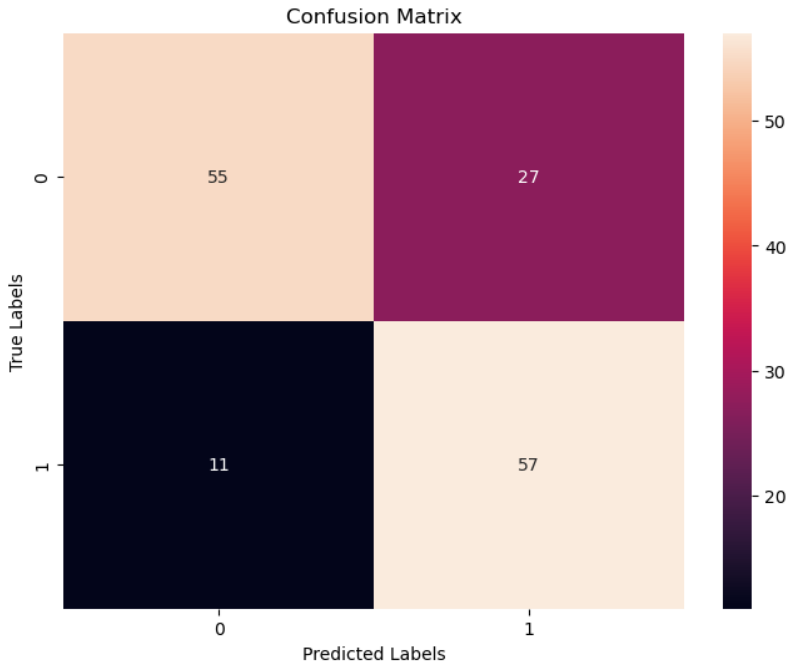


Accuracies for Train, Test and validation

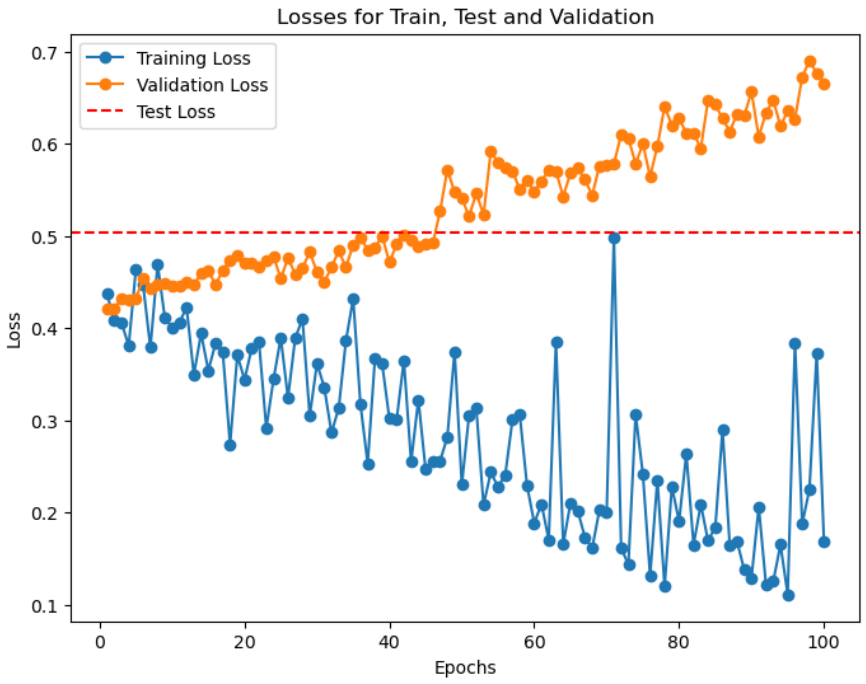


Training with Optimizers : **RMSProp**
Time to train model: 1.74 seconds
ROC: 0.75
Test Loss: 1.01
Precision: 0.68
F1 Score: 0.75
Recall: 0.84
Test Accuracy: 74.67%

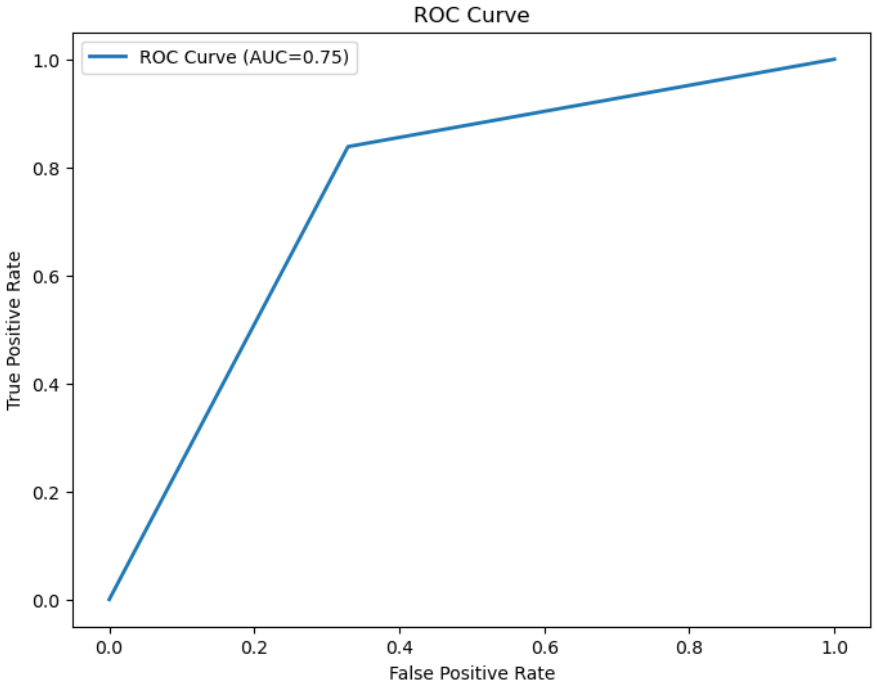
Confusion matrix:



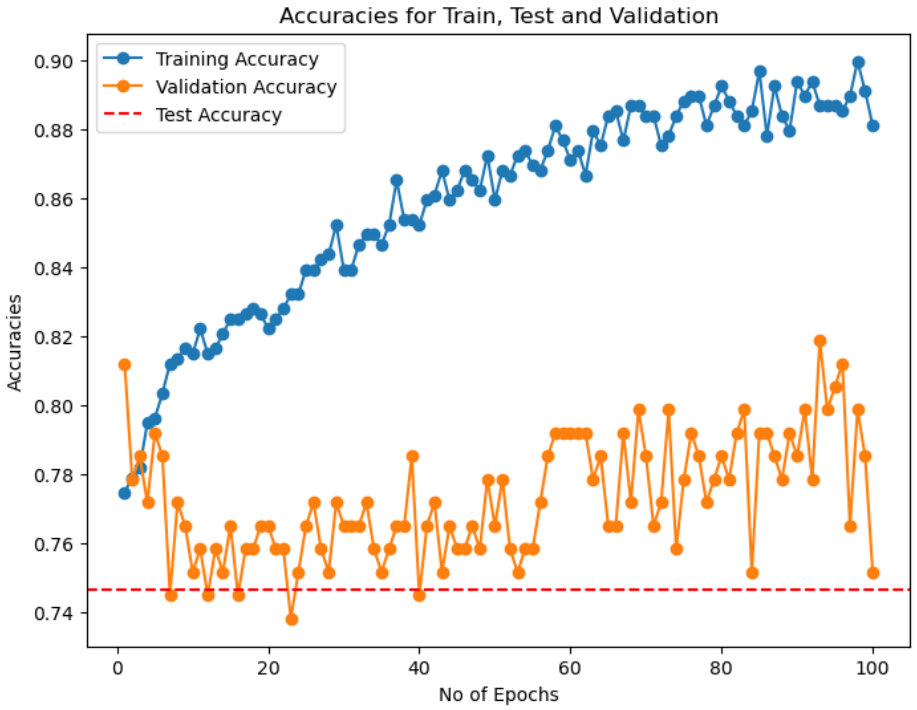
Losses for Train, Test and validation



ROC Curve



Accuracies for Train, Test and validation



Analysis and Reasoning about the 12 setups we tried:

In the first three setups, we have tuned the hyperparameter ‘dropout’ with the values 0.1, 0.3 and 0.5. Dropout is basically used to drop some nodes in the neural network, in order to make the deep learning model robust. The best accuracy we achieved is by keeping the dropout rate as 0.3, with accuracy being 77.33 percent.

In the next three setups, we have tuned the hyperparameter ‘initializers’ considering Xavier, He, and uniform as the initialising methods. Initializers try to initialise the starting weights

for the neural network model, in order to achieve the global maxima or minima faster when compared to a model without initializer. The best accuracy we achieved is with 'He' as the initializer having accuracy 78 percent.
In the further three setups, we have tuned the hyperparameter 'activation function', considering Relu, LeakyRelu and Elu for the linears defined in the neural network architecture. Activation functions are used to introduce non linearity to the output of a neuron and it either activates it or deactivates the neuron based on the result of a weighted sum. The best accuracy we have achieved is by keeping ELU as the activation function, with accuracy being 76 percent.
In the last three setups, we have tuned the hyperparameter 'optimizer', considering Adam, Adamax and RMSProp as the optimizers for our deep learning model. Optimizer helps in improving the overall loss and accuracy of the deep learning model by updating the weights at each training step, converging the model to global minima or maxima. The best accuracy we achieved is with 'RMSProp', with accuracy being 74.67. Both in part I and part II we are using the optimizer as Stochastic Gradient Descent, with which we are getting better results.

Various methods used in improving accuracy of the model further:

1. **Early stopping:**

   It is a regularisation technique where we stop the training model when the performance on the validation set becomes worse. This leads to increased loss and decreased accuracy. Hence Early stopping helps us in achieving the better validation accuracy with which we get better results.

   Test Accuracy: 77%

2. **Batch Normalization:**

   In this technique the performance of a deep learning model is increased. And also it is used to make the training more faster and stable through normalising the layers of the input. This method changes the inner layer's output into standard format, this is called normalizing, which resets the distribution of the output of the previous layer more efficiently.

   Test Accuracy: 76%

3. **Learning rate Scheduler:**

   The Learning rate scheduler is a method which adjusts the learning rate of the deep learning model periodically to achieve the best possible performance. It describes how quickly it updates its parameters based on the gradients which are used in training. If Learning rate is high the model overshoots values and fails to converge,

if it is low the model converges slowly or there may be a possibility of settling at local minimum.

Test Accuracy: 78%

4. **L2 Regularization**

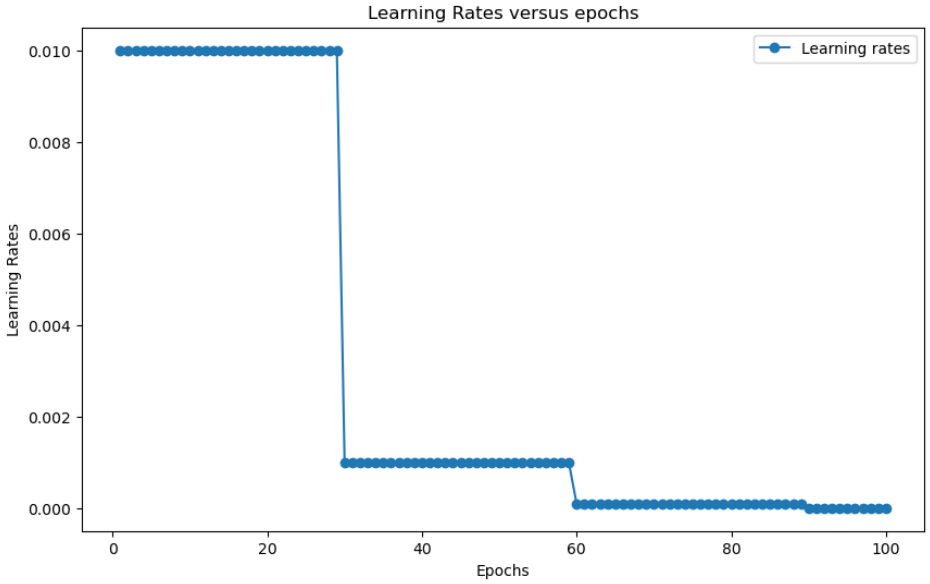
Basically L2 regularization is called ridge regression; this adds the squared magnitude of the coefficient as the term to the loss function. If the landa is very large then it will add too much weight and leads to underfitting. This method prevents overfitting issues.

Test Accuracy: 78.67%

Hence, from analyzing the loss and accuracy for each method, we identified the L2 Regularization method as a better model with more accuracy. Therefore, we are choosing it as the best model and saving its weights.

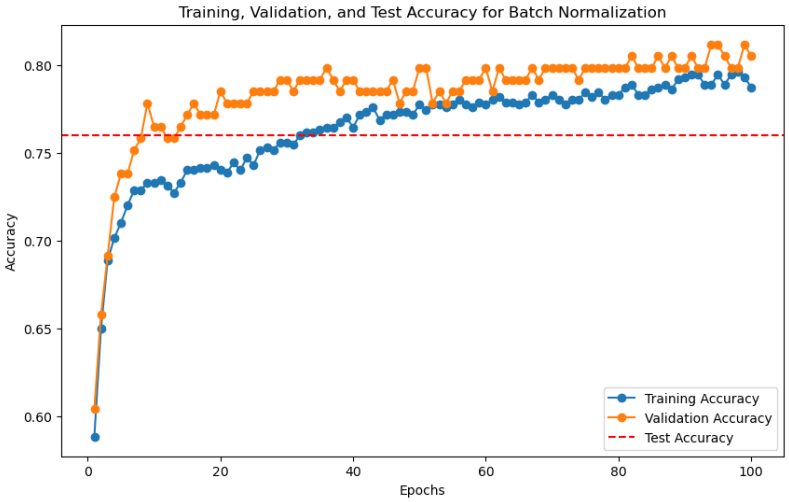
Visualization graphs for methods:

a. **Learning Rate Scheduling**



Above line plot depicts what learning rates we have used, with the help of LR scheduling, for each epoch while training the deep learning model.

b. **Batch Normalisation**



Above mentioned is the train, validation and accuracies plot for the batch normalization, with test accuracy being 76 percent.

Part III

Domain:

The given dataset is the MNIST dataset which consists of 36 classes namely 0-9 and A-Z. All the classes have an equal number of images of type png. And every class has 2800 MNIST images.

Data Preprocessing:

We first transformed the images using transforms.Compose which is a PyTorch library. These transformations are applied to input images before they are used in neural networks.

Resize(28,28): this method resizes the input image to 28*28 pixels.This is the common preprocessing step to transform all images to fixed size.

Grayscale() : this converts images to grayscale which have only a single channel.

ToTensor() : converts the image data to pytorch tensor. We need to convert into pytorch tensor because NN operate on tensors

Normalize(): normalizes the pixel values of the tensor. Normalization is the common pratice in deep learning to bring the pixel values of the input data into particular range.

Visualization graphs:

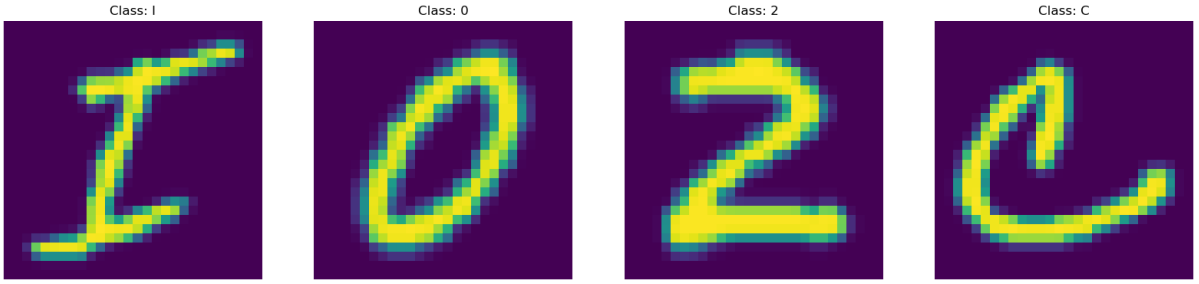
Plot-1

Plotting sample of images

We are displaying few samples from a particular class

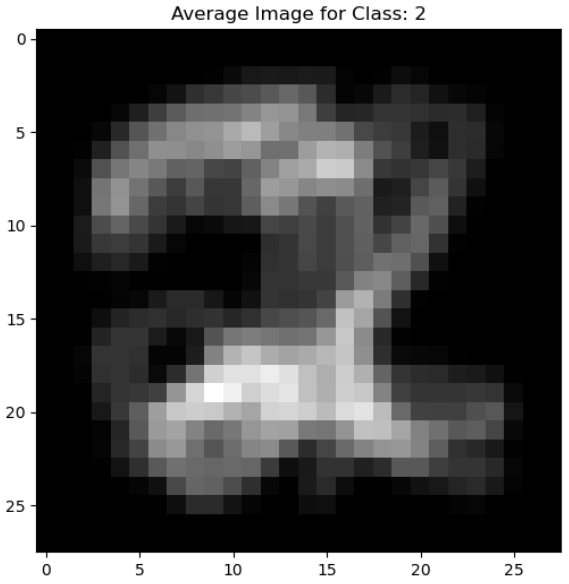
Plot -2

Here in this plot we are displaying random images by selecting random classes



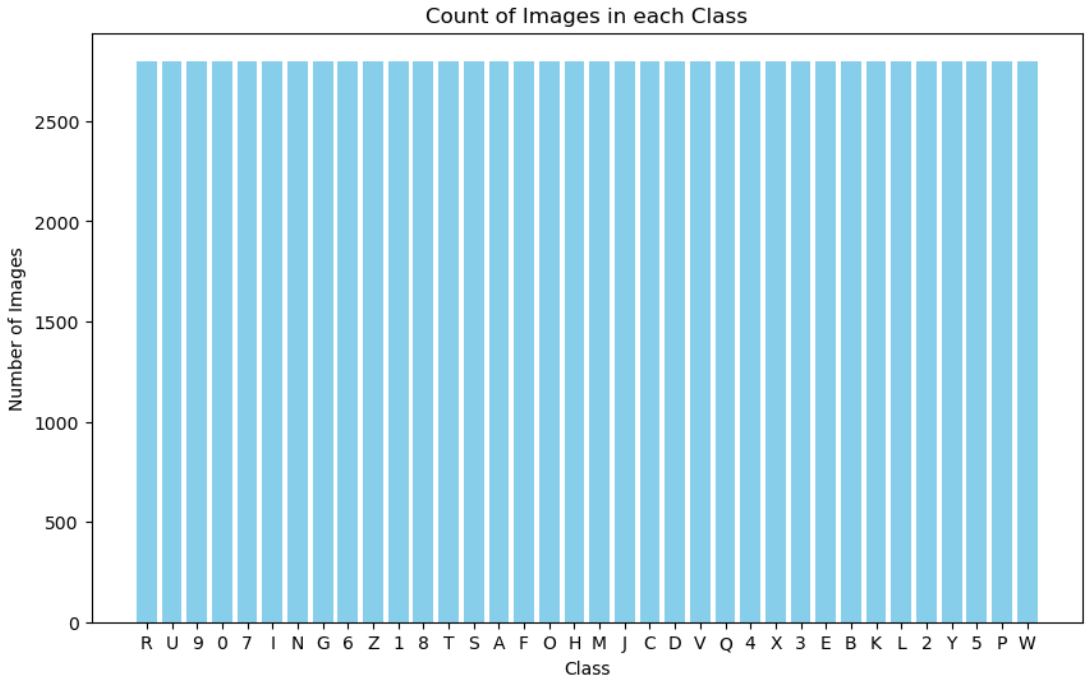
Plot-3

In this visualization we plot an average image by selecting from a class. Average image is created by taking the average pixel of corresponding pixels of multiple images. This method of visualizations are used in computer vision and image processing.



Plot-4:

This plot is referring to the number of samples present in each class.



Since every class has equal number of samples all the bar graph plot is settled at 2800

Summary of CNN model:

First we implemented the basic CNN model. Below screenshots represent model details of our CNN model.

```
Out[195]: SimpleCNN(
            (convlayers): Sequential(
              (0): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (1): ReLU()
              (2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
              (3): ReLU()
            )
            (fclayers): Sequential(
              (0): Linear(in_features=25088, out_features=128, bias=True)
              (1): ReLU()
              (2): Linear(in_features=128, out_features=36, bias=True)
            )
          )
```

```
Out[196]: ==========================================================================
          Layer (type:depth-idx)                   Param #
          ==========================================================================
          SimpleCNN                                --
          ├─Sequential: 1-1                        --
          │    └─Conv2d: 2-1                       160
          │    └─ReLU: 2-2                         --
          │    └─Conv2d: 2-3                       4,640
          │    └─ReLU: 2-4                         --
          ├─Sequential: 1-2                        --
          │    └─Linear: 2-5                       3,211,392
          │    └─ReLU: 2-6                         --
          │    └─Linear: 2-7                       4,644
          ==========================================================================
          Total params: 3,220,836
          Trainable params: 3,220,836
          Non-trainable params: 0
          ==========================================================================
```

To measure the difference between the model's predictions and actual labels during training we used CrossEntropyLoss. And the Adam optimizer is used to adjust the model parameters based on calculated gradients.

We implemented a CNN model and defined training, validation, testing loop using pytorch.

1. The training loop and validation loop iterates through a specific number of epochs and calculates training loss, validation loss, and prints the progress.
2. And finally in testing loop the model is evaluated on separate test_loader and calculator test accuracy, average loss, precision, recall and f1 score

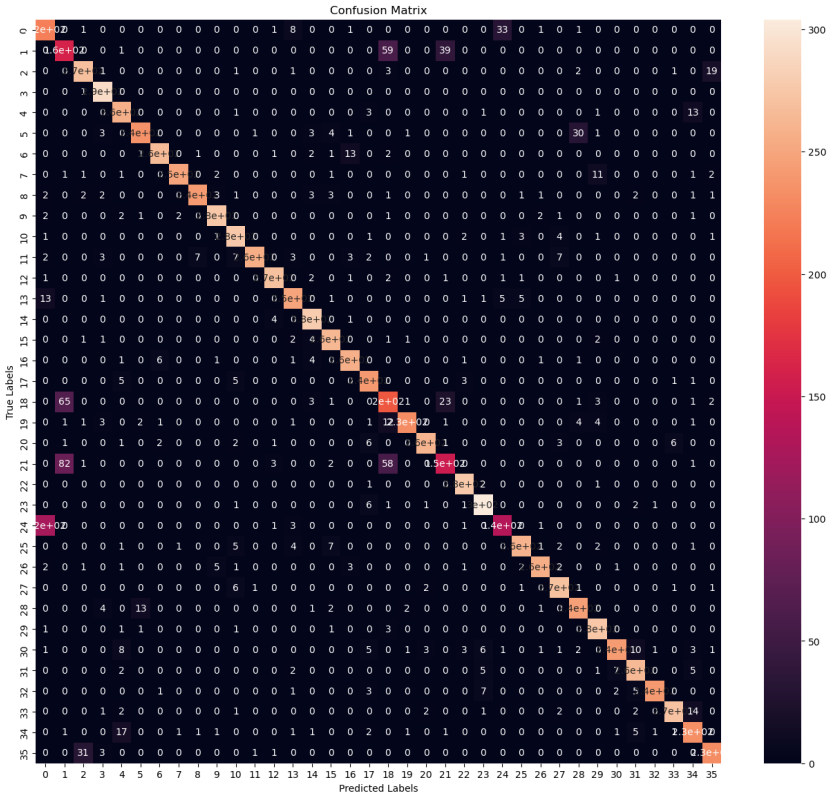
See the below confusion matrix to observe the model predictions.

Train Accuracy for epoch 3/3: 90.31%
Validation Accuracy after training the model: 88.46%

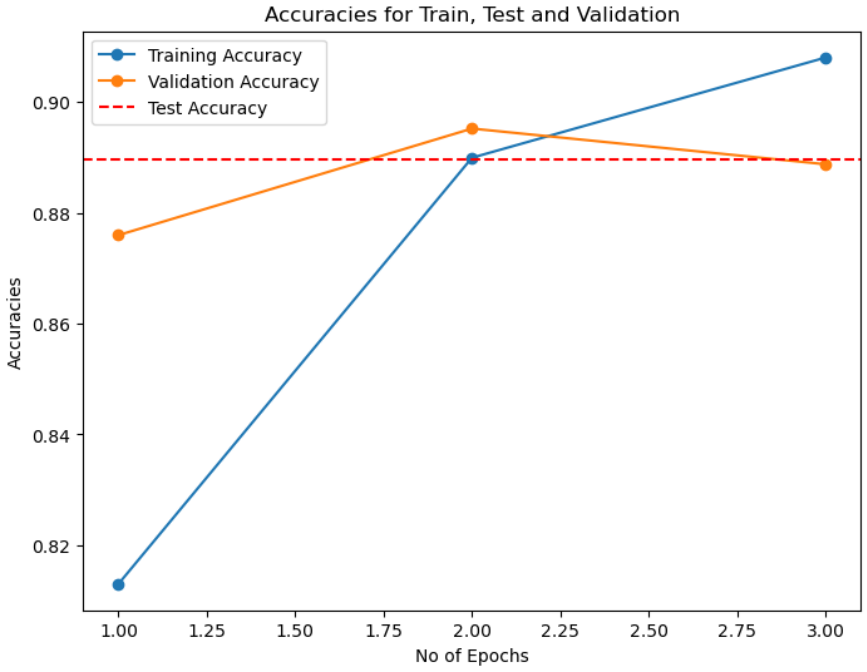
Performance Metrics:
Time to train model: 87.37 seconds
Test Accuracy after training and validation: 88.38
Average Test Loss after training and validation: 0.3421
Recall: 0.88
Precision: 0.89
F1 Score: 0.88

Confusion Matrix:

By observing below confusion matrix the model has made mostly all predictions correctly and very few are wrongly predicted



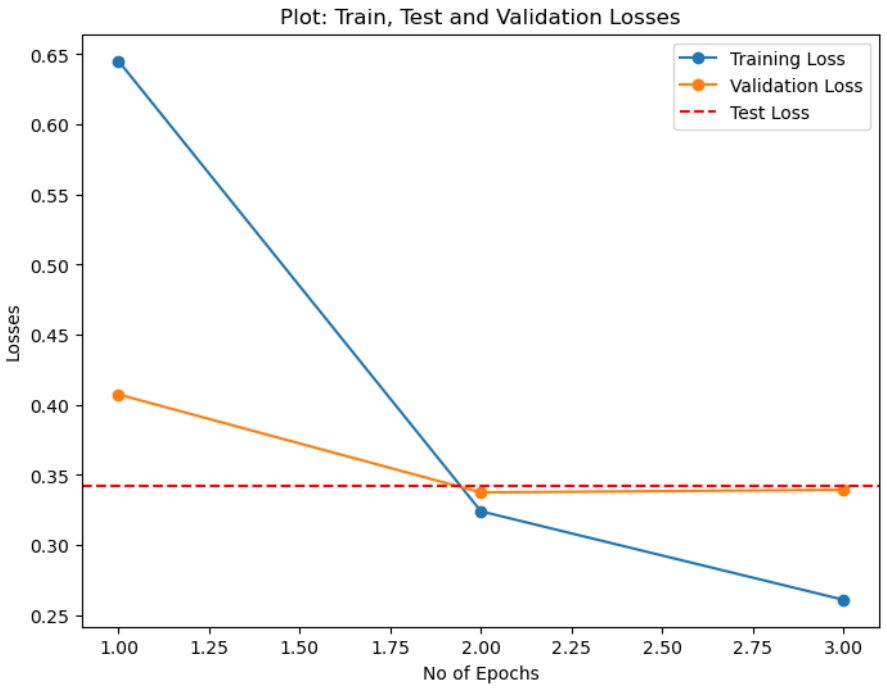
Accuracies for Train, test and validation:



By observing from the graph with the increase of the number of epochs the training accuracy and validation accuracy keeps on increasing. Where as test accuracy remained same which slightly less than 0.90

Train, Test and validation losses

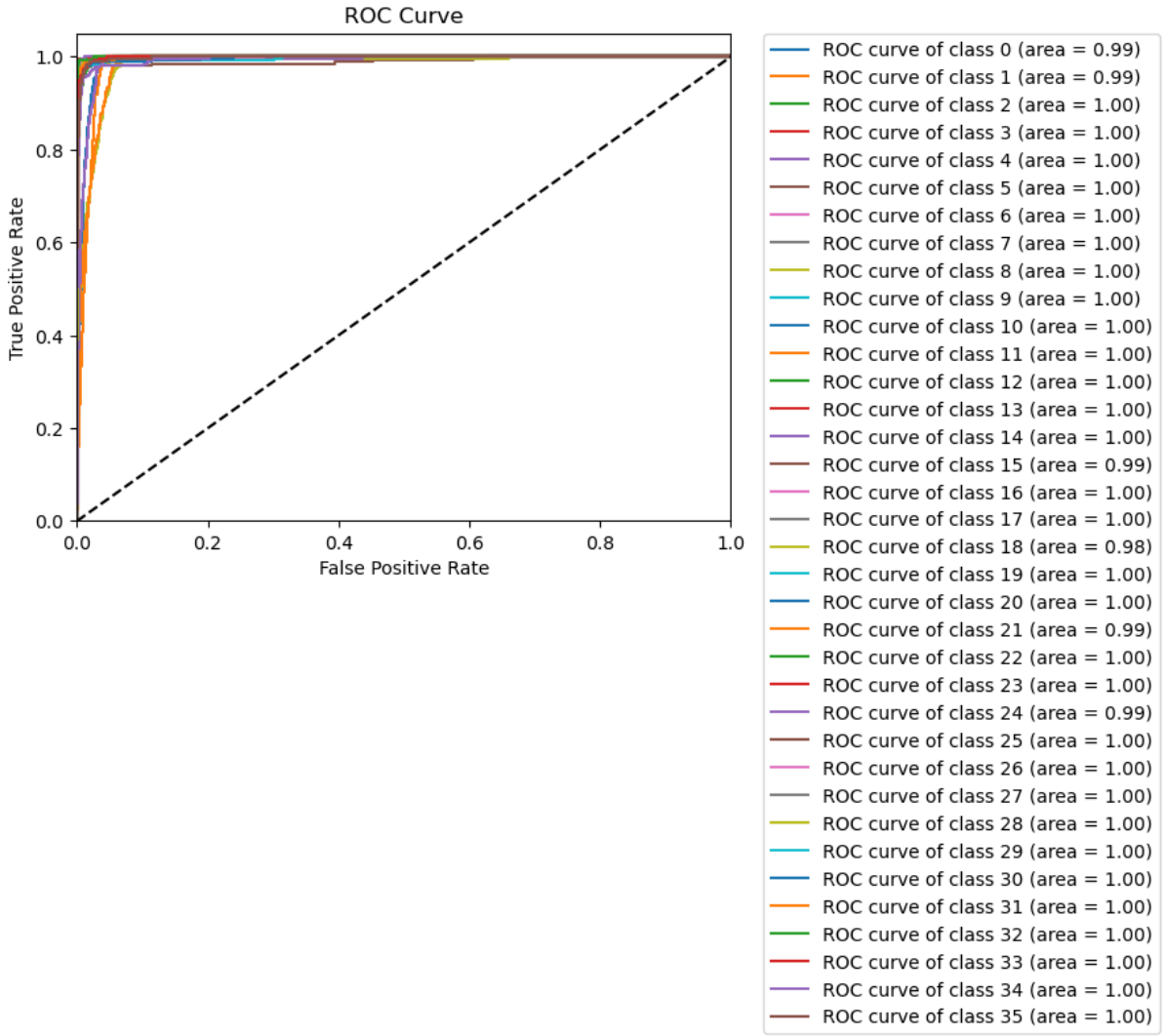
From the graph we can infer that with the increase of the number of epochs the train and validation loss is decreasing.



After, we evaluated the trained CNN network model using test data.

1. The the model is evaluated using model.eval()
2. It iterates through test_loader and collects true labels and predicted probabilities for every images
3. And later true labels and predicted are flattened into numpy array
4. For each class we calculated we plotted the graph of true positive rate against false positive rate. This is the ROC curve. Thai results are stored in fpr, tpr, roc_auc.

See the below screenshot for ROC curve



By observing the ROC curve we can say that all the classes are approaching 1. So the model is predicting perfectly since the area under the ROC curve is approximately equals to 1

The improvement tools that we used on CNN architecture

Early Stopping:

Train Accuracy for epoch 3/3: 0.93%
Validation Accuracy for early stopping: 88.47%

Performance Metrics:
Time to train model: 89.34 seconds
Test Accuracy for Early stopping: 88.22
Average Loss: 0.3475
Recall: 0.88
Precision: 0.89

Observation:

1. The time to train the model decreased.
2. After implementing early stopping in CNN model, the train accuracy increased from 90% to 93% and Validation accuracy is increased slightly
3. And also test accuracy is increased
4. The overall average loss is decreased.
5. The precision values also increased by 0.01 whereas recall and F1 Score remained the same.

L2 Regularization

Train Accuracy for epoch 3/3: 94.41%
Validation Accuracy after training the model: 88.47%

Performance Metrics:
Time to train model: 76.18 seconds
Test Accuracy after training and validation: 88.73
Average Test Loss after training and validation: 0.3762
Recall: 0.89
Precision: 0.89
F1 Score: 0.89

Observation:

1. L2 regularization adds a penalty term to the loss function that decreases the large weights in the model. Thai penalty term is proportional to the square of the weights
2. This prevents overfitting of the model
3. Due to this there is an increase of test accuracy about 0.25%
4. however the average loss has slight decrement
5. The recall, Precision, F1 score increased slightly

Learning rate Scheduler:

Train Accuracy for epoch 3/3: 0.94%
Validation Accuracy for LR scheduler: 88.47%

Performance Metrics:
Time to train model: 72.72 seconds
Test Accuracy for LR Scheduler: 88.73
Average Test Loss after training and validation: 0.3762
Recall: 0.89

Precision: 0.89
F1 Score: 0.89

Observation:

1. This method is used during the training of CNN to adjust the learning rate over time. By adjusting the learning rate during the training the converges faster.
2. This decreases the divergence of the during the training.
3. Using the Learning rate scheduler the model took less time to train than before
4. The test accuracy has improved by 0.4%
5. The average loss remained same
6. The precision, recall, F1 score has also increased

Batch Normalisation

```
BatchNN(
  (conv1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu1): ReLU()
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu2): ReLU()
  (fc1): Linear(in_features=25088, out_features=128, bias=True)
  (batch): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu3): ReLU()
  (fc2): Linear(in_features=128, out_features=36, bias=True)
)
```

Train Accuracy for epoch 3/3: 95.88%
Validation Accuracy after training the model: 88.55%

Performance Metrics:
Time to train model: 93.34 seconds
Test Accuracy after training and validation: 89.08
Average Test Loss after training and validation: 0.3530
Recall: 0.89
Precision: 0.89
F1 Score: 0.89

Observation:

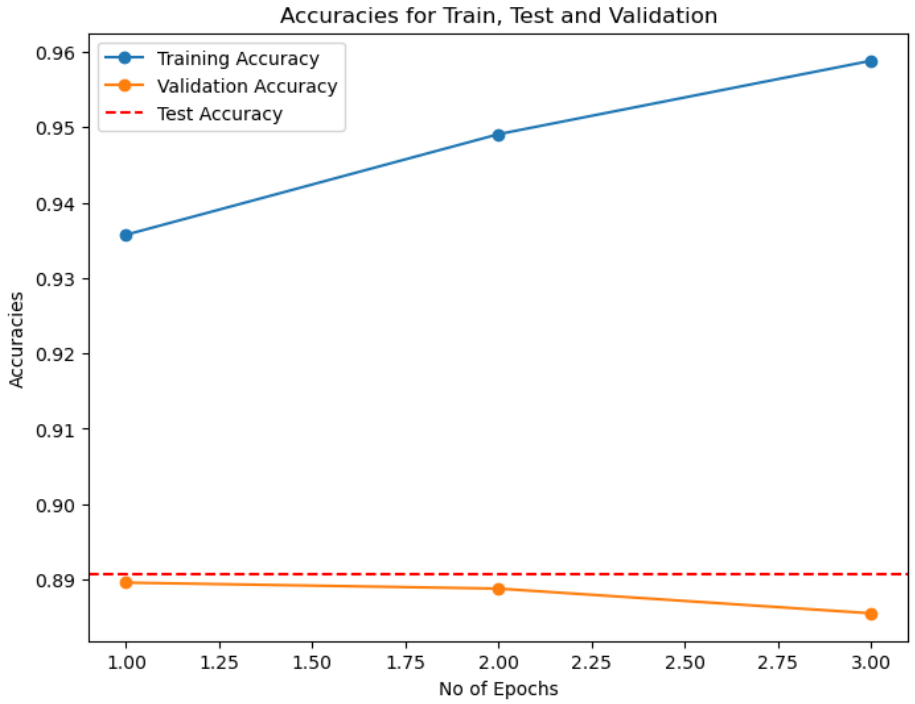
1. Batch Normalization stabilises the training process by reducing variance shift.
2. Due to this it enables the use of higher learning rates without risk of divergence
3. The test accuracy is increased by 1.1 %
4. The overall average loss also decreased when compared with initial setup

5. There is slight increment in F1 score, precision and recall

From all the methods above, we are choosing the best method as Batch Normalization with highest accuracy.

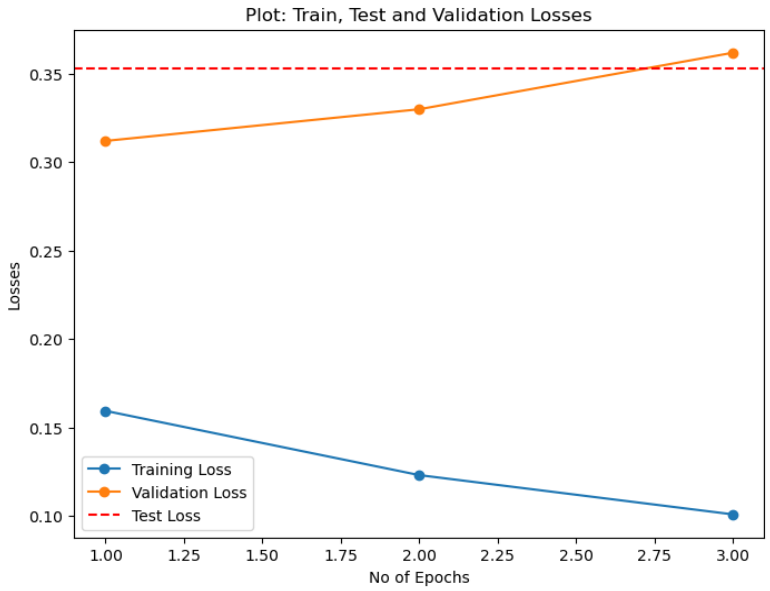
Performance metrics graph for Batch Normalization

Accuracy Plot:



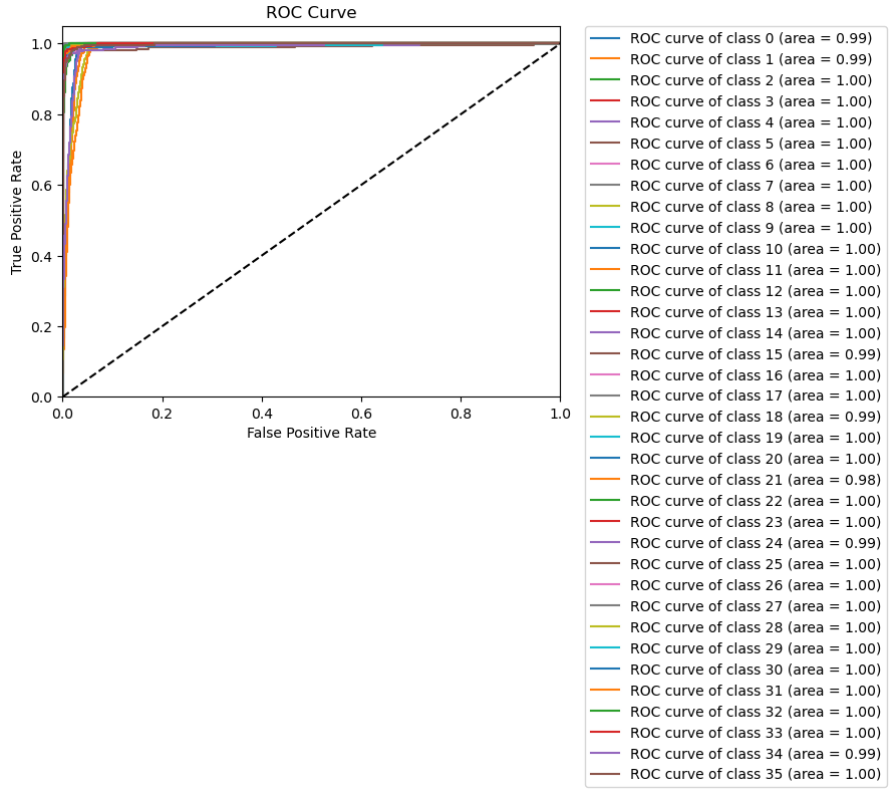
The graph shows accuracies for Train, Test and validations. From the graph we can infer that increasing number of epochs the training accuracy is increasing and validation accuracy maintained same over the period whereas test accuracy also remained constant

Losses Plot



The Loss plot is plotted between the number of epochs and Losses. The graph depicts The training loss keeps on decreasing as the number of epochs increased. While test loss remained same.

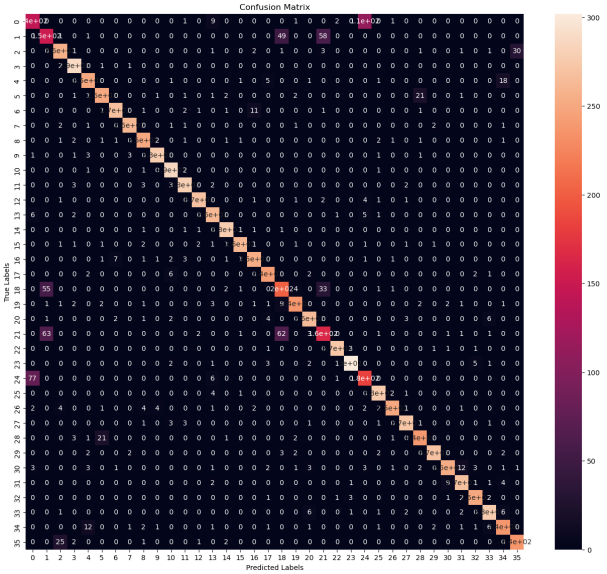
ROC curve:



From the ROc plot we can see that every class is approaching 1. So the area under the curve approximately becomes equal to 1. So by this we can conclude that the model is trying to predict correct predictions.

Confusion Matrix:

The confusion matrix below indicates, the model is predicting most of the classes correctly and only values are wrongly predicted. So we are choosing this model as the best model.



Part IV

VGG

VGG11Network(
  (features): Sequential(
    (0): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (11): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): ReLU(inplace=True)
    (13): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): ReLU(inplace=True)
    (15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (16): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): ReLU(inplace=True)
    (20): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

```
  )
  (classifier): Sequential(
    (0): Linear(in_features=512, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=36, bias=True)
  )
)
```

Summary of the Model

```
Layer (type:depth-idx)                   Param #

VGG11Network                             --
├─Sequential: 1-1                        --
│    └─Conv2d: 2-1                       640
│    └─ReLU: 2-2                         --
│    └─MaxPool2d: 2-3                    --
│    └─Conv2d: 2-4                       73,856
│    └─ReLU: 2-5                         --
│    └─MaxPool2d: 2-6                    --
│    └─Conv2d: 2-7                       295,168
│    └─ReLU: 2-8                         --
│    └─Conv2d: 2-9                       590,080
│    └─ReLU: 2-10                        --
│    └─MaxPool2d: 2-11                   --
│    └─Conv2d: 2-12                      1,180,160
│    └─ReLU: 2-13                        --
│    └─Conv2d: 2-14                      2,359,808
│    └─ReLU: 2-15                        --
│    └─MaxPool2d: 2-16                   --
│    └─Conv2d: 2-17                      2,359,808
│    └─ReLU: 2-18                        --
│    └─Conv2d: 2-19                      2,359,808
│    └─ReLU: 2-20                        --
│    └─MaxPool2d: 2-21                   --
├─Sequential: 1-2                        --
│    └─Linear: 2-22                      2,101,248
│    └─ReLU: 2-23                        --
│    └─Dropout: 2-24                     --
│    └─Linear: 2-25                      16,781,312
│    └─ReLU: 2-26                        --
```

```
│    └─Dropout: 2-27                --
│    └─Linear: 2-28                 147,492
```

Total params: 28,249,380
Trainable params: 28,249,380
Non-trainable params: 0

Differentiating **VGG11Network** with **SimpleCNN** network defined in Part 3:
Depth:
VGG architecture is deep when compared to Simple CNN network.
Number of layers in VGG: 11 layers (including convolutional and fully connected)

Convolutional Layers:
VGG11 primarily uses 3X3 convolutional layers with stride 1 and padding 1 along with max pooling layers (which are not considered as additional layers. Similarly our SimpleCNN architecture also uses 3X3 convolutional layers with stride and padding.
We have around 8 convolutional layers in VGG and 2 convolutional layers in SimpleCNN.

Pooling Layers:
Pooling layers are mainly used to reduce the dimension of the feature maps, preserving important features.
VGG uses max pooling layers with a 2X2 window size and a stride of 2.
Whereas in our SimpleCNN model, we did not use any max pooling layers.

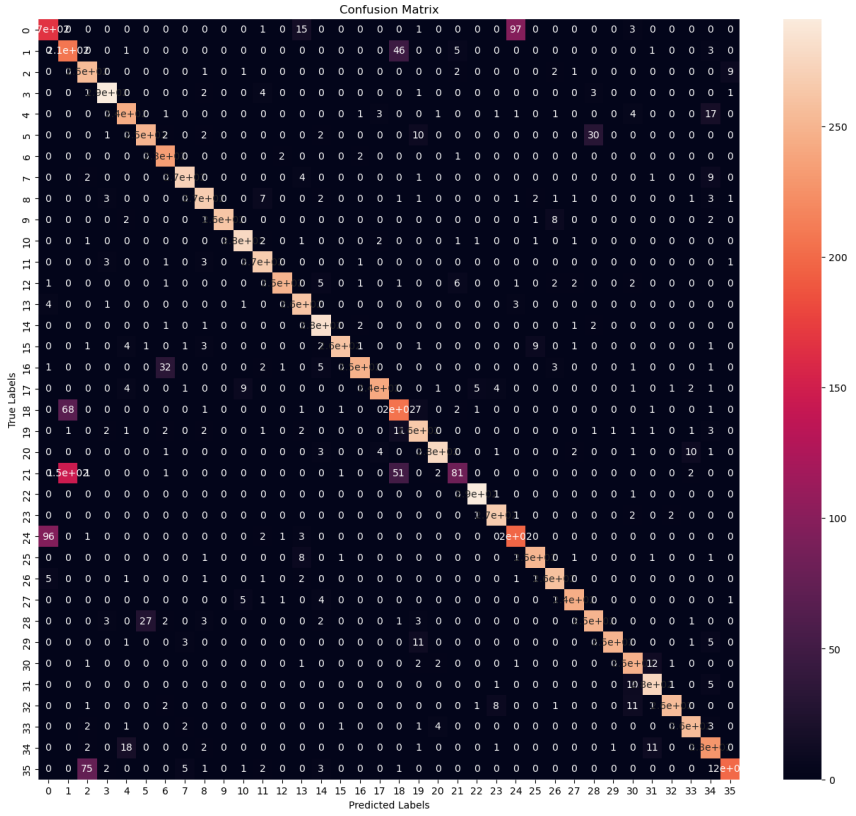
Fully Connected Layers:
The fully connected layers are placed after the convolutional and max pooling layers, i.e. at the end of the network.
In VGG, there are 3 fully connected layers, followed by ReLU activation functions after each layer, with output of the last fully connected layer being 36.
In our SimpleCNN architecture, there are 2 fully connected layers, followed by a ReLU activation function after each layer.

Train Accuracy for epoch 3/3: 0.87%
Validation Accuracy after training the model: 87.69%

Performance Metrics:
Time to train model: 269.90 seconds
Test Accuracy after training and validation: 87.59
Average Test Loss after training and validation: 0.3650
Recall: 0.88
Precision: 0.88
F1 Score: 0.87

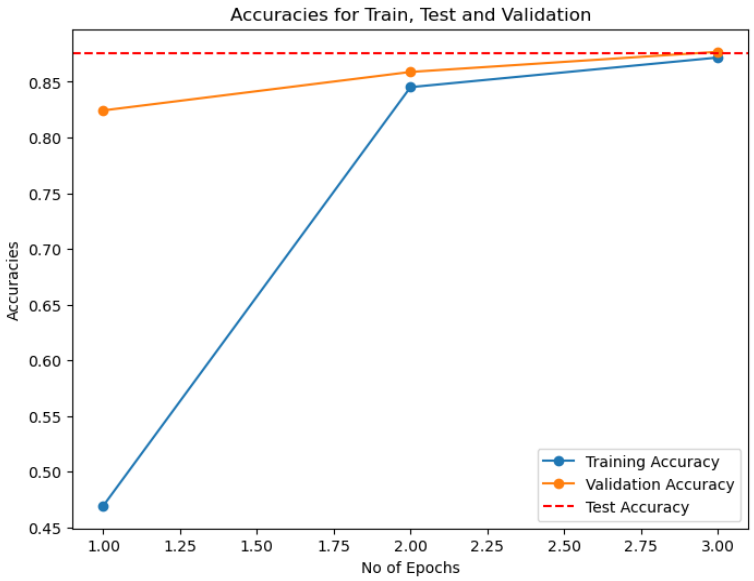
Confusion Matrix:

From the confusion matrix we can say that the VGG model is predicting most of the values correctly however it is not predicting correctly for few values.



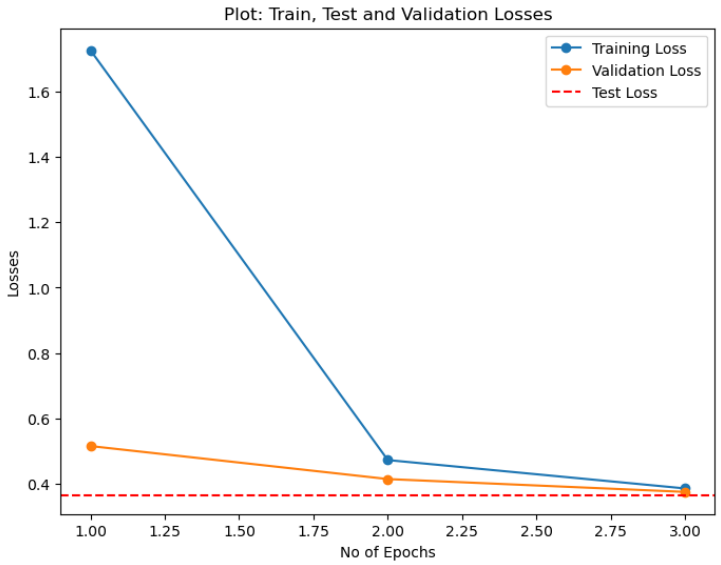
Accuracy:

The accuracy plots of the graph states that the training accuracy and Validation accuracy keeps on increasing with the increase of the number of epochs, while testing accuracy remains the same.



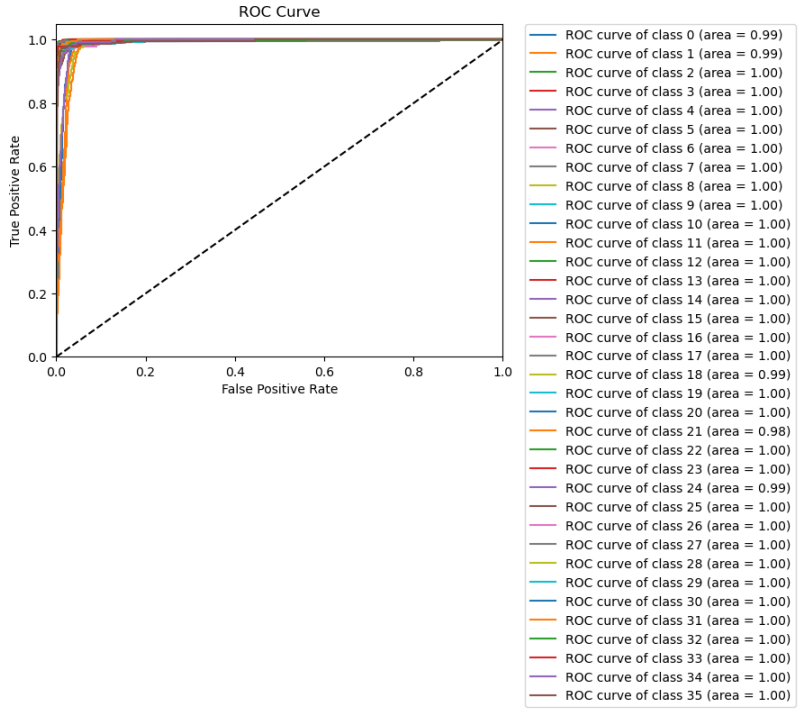
Loss:

The loss keeps on decreasing as number of epochs are increased so the model is trying to predict correct values for increasing number of epochs.



ROC Curve

The ROC graph represents that all classes are approaching 1, The area under the curve is becoming approximately equal to 1. Hence we can say that the model is trying to predict correct values.



References:

https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

https://pytorch.org/vision/main/models/vgg.html

Contribution:

Charan Kumar Nara: 50%

Dharma Acha: 50%