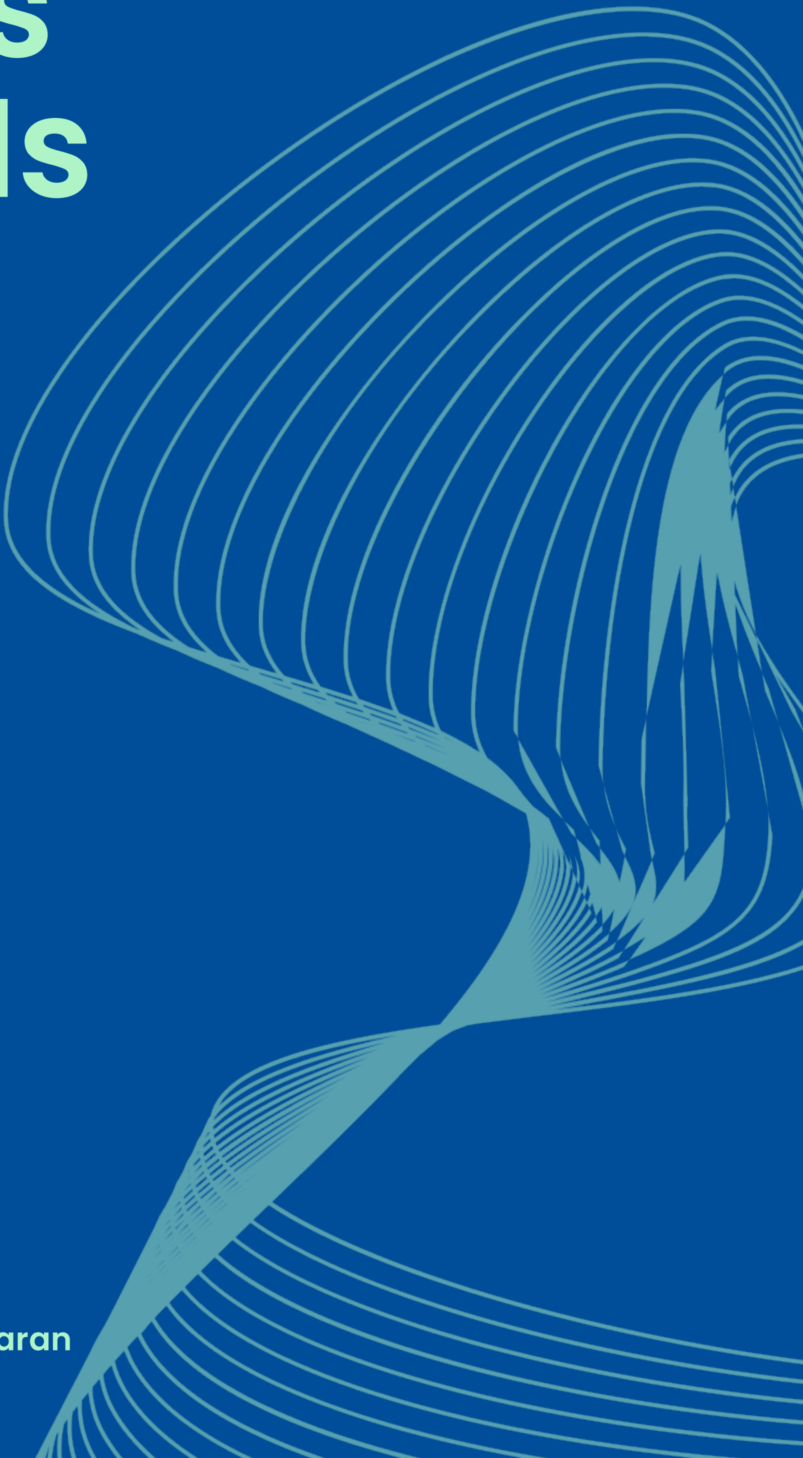
Introduction to
Machine Learning

NOVEMBER 2024

Building Neural Networks and CNNs



Prepared by
Jagrit Sharma & Pavithran Gnanasekaran
UBIT: jagritsh, pgnanase
Person#: 50606727, 50604192

Table of Contents

Teammate Contribution

Team Member	Assignment Part	Contribution
Jagrit Sharma	Part 1, 2, 3, 4 & Bonus	50%
Pavithran Gnanasekaran	Part 1, 2, 3, 4 & Bonus	50%

Part 1: Building a Basic NN

Introduction

Objective: The purpose of this experiment is to implement a basic neural network model using the PyTorch framework to predict a binary target variable based on seven input features. We will also be using the built-in libraries for data visualization, mathematical calculations and data processing operations. We will utilize the '*dataset.csv*' dataset, consisting of several features, then use this neural network model to predict the target feature of our dataset.

Importance of a Neural Network: Neural networks are powerful machine learning models capable of learning complex patterns in data. For this binary classification, we will be using a neural network as it can capture non-linear relationships between the input features and the target feature effectively with higher accuracies.

Features of the dataset used to predict the target: f1, f2, f3, f4, f5, f6, f7

Data Preparation

Data Statistics

1. Begin with analyzing our dataset and look for anomalies which we can preprocess before training our model on it.

```
df.describe()
```

	f3	target
count	766	766
mean	69.118799	0.349869
std	19.376901	0.47724
min	0	0
25%	62.5	0
50%	72	0

75%	80	1
max	122	1

2. Use unique() function to find any anomalies in each column.

Ex:
```
print('f1: ', df['f1'].unique())
```

```
f1 :  ['6' '1' '8' '0' '5' '3' '10' '2' '4' '7' '9' '11' '13' '15' '17' '12' '14' 'c']
```

Data Preprocessing

3. Remove the anomalies by replacing the unwanted characters to NaN.

```
df['f1'] = pd.to_numeric(df['f1'], downcast= 'float', errors= 'coerce')
```

Ex:

```
f1 :  [ 6.  1.  8.  0.  5.  3. 10.  2.  4.  7.  9. 11. 13. 15. 17. 12. 14. nan]
```

4. Find the count of null values in each column.

```
df.isnull().sum()
```

```
f1      1
f2      1
f3      0
f4      1
f5      1
f6      1
```

5. Replace the null values with the mean of their respective columns. Use Simple Imputer from sklearn.

```
imputer = SimpleImputer(strategy='mean') # Replacing the null values using SimpleImputer
df = pd.DataFrame(imputer.fit_transform(df), columns = df.columns)
```

Repeating step 4 again, we get:

```
f1      0
f2      0
f3      0
f4      0
f5      0
f6      0
```

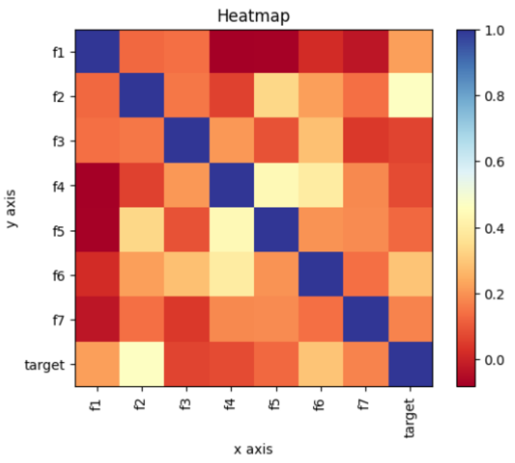
Data Visualization

6. Find the correlation matrix of our dataset.

```
df_correlation_matrix = df.corr()
print("Correlation matrix",df_correlation_matrix)

plt.imshow(df_correlation_matrix, cmap='RdYlBu', interpolation='nearest')
plt.colorbar()
plt.title("Heatmap")
plt.xlabel("x axis")
plt.ylabel("y axis")
plt.xticks(np.arange(len(df.columns)), df.columns, rotation=90)
```

Correlation matrix	f1	f2	f3	f4	f5	f6	f7	target
f1	1	0.125133	0.14099	-0.082371	-0.07543	0.017205	-0.03118	0.221211
f2	0.125133	1	0.151813	0.057729	0.332425	0.217905	0.137664	0.463623
f3	0.14099	0.151813	1	0.20714	0.088694	0.281532	0.041543	0.064623
f4	-0.082371	0.057729	0.20714	1	0.437818	0.392922	0.182141	0.078163
f5	-0.075434	0.332425	0.088694	0.437818	1	0.197967	0.184416	0.128889
f6	0.017205	0.217905	0.281532	0.392922	0.197967	1	0.139827	0.292430
f7	-0.031183	0.137664	0.041543	0.182141	0.184416	0.139827	1	0.173636
target	0.221211	0.463623	0.064623	0.078163	0.128889	0.29243	0.173636	1

Correlation Matrix

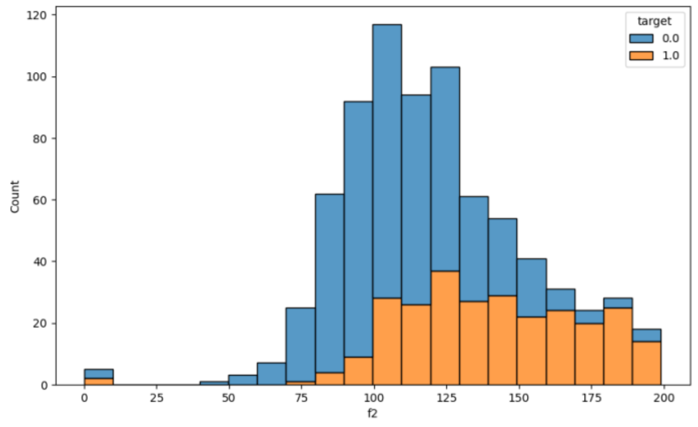


a. The Correlation matrix tells us that ‘f2’ is the most optimal feature to predict the target. Therefore, we will plot the histogram between ‘f2’ and ‘target’.

7. Plot a histogram between ‘f2’ and ‘target’.

```
plt.figure(figsize=(10,6))
sns.histplot(data=df, x="f2", hue="target", multiple="stack", kde=False, bins=20)
plt.show()
```

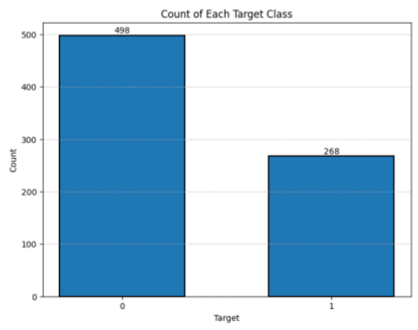
Histogram



8. Plot a bar graph to visualize the count of target values.

```
target_count = df.groupby("target")["target"].count()
plt.figure(figsize=(8, 6))
bars = plt.bar(target_count.index, target_count.values, edgecolor='black',linewidth=1.5,width=0.6)
plt.title('Count of Each Target Class')
plt.xlabel('Target')
plt.ylabel('Count')
plt.xticks([0, 1])
plt.grid(axis='y', linestyle='--', alpha=0.7)
for bar in bars:
   yval = bar.get_height()
   plt.text(bar.get_x() + bar.get_width()/2, yval + 1, int(yval), ha='center', va='bottom')
```

Bar Graph



Splitting the dataset

9. Before splitting the data, we normalize it using Min-Max scaler. We used Min-Max scaler because our target values are either 0 or 1.

```
scaler = MinMaxScaler() #Normalizing values using Min-Max Scaling
df = pd.DataFrame(scaler.fit_transform(df), columns= df.columns)
```

10. Divide our dataset into X and Y.

```
X = df.drop("target", axis=1)
Y = df.target
```

11. Further split X and Y into X_train, Y_train, X_test, Y_test, X_val and Y_val, where the train:val:test ratio is 80:10:10. This can be achieved by splitting X and Y into (X_train, Y_train) and (X_test, Y_test) first, then splitting our training set further to get (X_val, Y_val).

```
X_train, X_test, Y_train, Y_test  = train_test_split(X, Y, test_size=0.1, random_state=42)
X_train, X_val, Y_train, Y_val  = train_test_split(X_train, Y_train, test_size=0.11, random_state=42)
# Given Ratio is 80:10:10
print("X_train:", X_train.shape)
print("Y_train:", Y_train.shape)
print("X_val:", X_val.shape)
print("Y_val:", Y_val.shape)
print("X_test:", X_test.shape)
print("Y_test:", Y_test.shape)
```

```
X_train: (613, 7)
Y_train: (613,)
X_val: (76, 7)
Y_val: (76,)
```

12. Our final step is to convert the datasets into tensors. We will be using a built-in package called ‘torch’ to achieve this.

```
X_train = torch.from_numpy(X_train.values)
Y_train = torch.from_numpy(Y_train.values)
X_test = torch.from_numpy(X_test.values)
Y_test = torch.from_numpy(Y_test.values)
X_val = torch.from_numpy(X_val.values)
```

Model Implementation

We use a class structure to implement our basic neural network. This model has been implemented from scratch using torch package. The forward method defines the forward pass of the neural network, specifying how input data flows through the layers to produce an output. It's where we implement the actual computation performed by your neural network. Our class contains 2 methods, and it is structured as follows:

Methods used:

__init__():

This method is used to initialize variables and take input parameters while accessing the model.

forward():

This method defines the forward pass of the neural network, basically defining how the input data should flow through the layers to deduce the output.

Neural Network Architecture:

Input Neurons	7
Output Neurons	1
Activation function used for hidden layers	ReLU (Rectified Linear Unit)
Activation function used for output layer	Sigmoid
Hidden layers	2
Size of each hidden layer	(64, 64)
Dropout value	0.5

Sigmoid and ReLU are commonly used for binary classification, hence, they make the perfect choice for our neural network architecture.

Model Summary:

```
==========================================================================================
Layer (type:depth-idx)          Output Shape        Param #
==========================================================================================
SimpleNN                        [1, 1]              --
├─Linear: 1-1                   [1, 64]             512
├─ReLU: 1-2                     [1, 64]             --
├─Dropout: 1-3                  [1, 64]             --
├─Linear: 1-4                   [1, 64]             4,160
├─ReLU: 1-5                     [1, 64]             --
├─Linear: 1-6                   [1, 1]              65
├─Sigmoid: 1-7                  [1, 1]              --
==========================================================================================
Total params: 4,737
```

Experimentation:

We experiment with our model by passing the following values has the hyperparameters:

1. Learning Rate: 0.001
2. Number of Iterations: 50
3. Batch Size: 32
4. Loss Function: Binary Cross Entropy Loss function
5. Optimizer: Adam

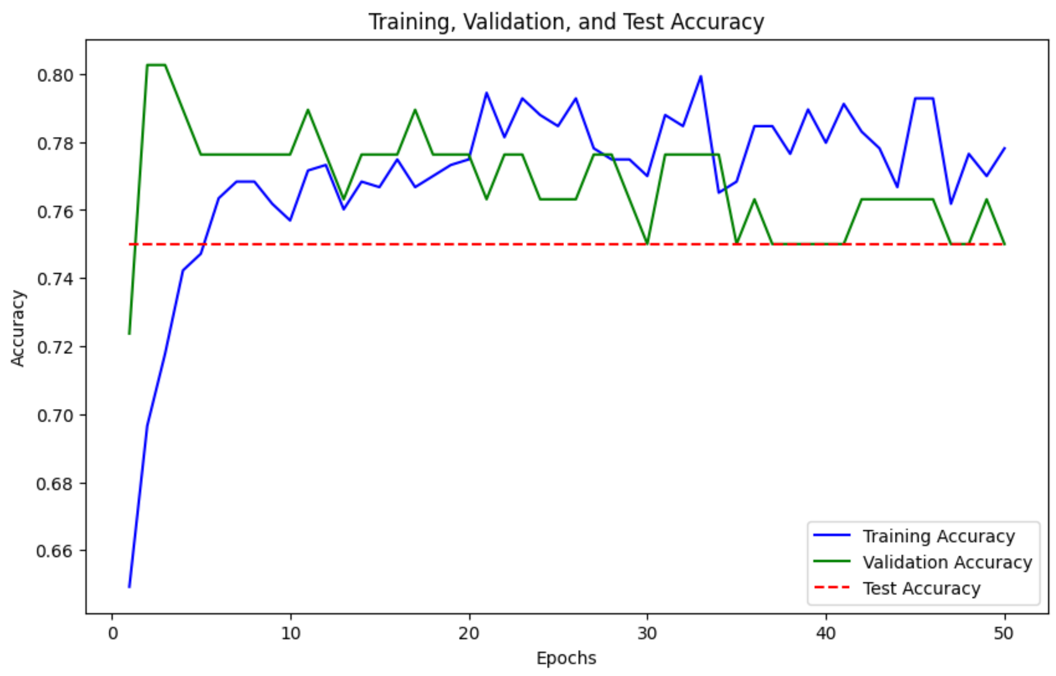
Training our model on the given hyperparameters:

```
for epoch in range(num_epochs):
    model.train()
    epoch_loss = 0
    correct_train = 0
    total_train = 0
    for inputs, labels in train_loader:
        outputs = model(inputs).squeeze()
        loss = loss_function(outputs, labels)
        epoch_loss += loss.item()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        predicted_train = torch.round(outputs)
        correct_train += (predicted_train == labels).sum().item()
        total_train += labels.size(0)
    avg_train_loss = epoch_loss / len(train_loader)
    training_losses.append(avg_train_loss)
    training_accuracies.append(correct_train / total_train)
    model.eval()
    val_loss = 0
    correct_val = 0
    total_val = 0
    with torch.no_grad():
        for val_inputs, val_labels in val_loader:
            val_ = model(val_inputs).squeeze()
            val_loss += loss_ outputs function(val_outputs, val_labels).item()
            predicted_val = torch.round(val_outputs)
            correct_val += (predicted_val == val_labels).sum().item()
            total_val += val_labels.size(0)
    avg_val_loss = val_loss / len(val_loader)
    validation_losses.append(avg_val_loss)
    validation_accuracies.append(correct_val / total_val)
    print(f"Epoch {epoch+1}/{num_epochs} | Training Loss: {avg_train_loss:.4f} | Validation Loss: {avg_val_loss:.4f} | "
          f"Training Accuracy: {training_accuracies[-1]:.4f} | Validation Accuracy: {validation_accuracies[-1]:.4f}")
    if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
```

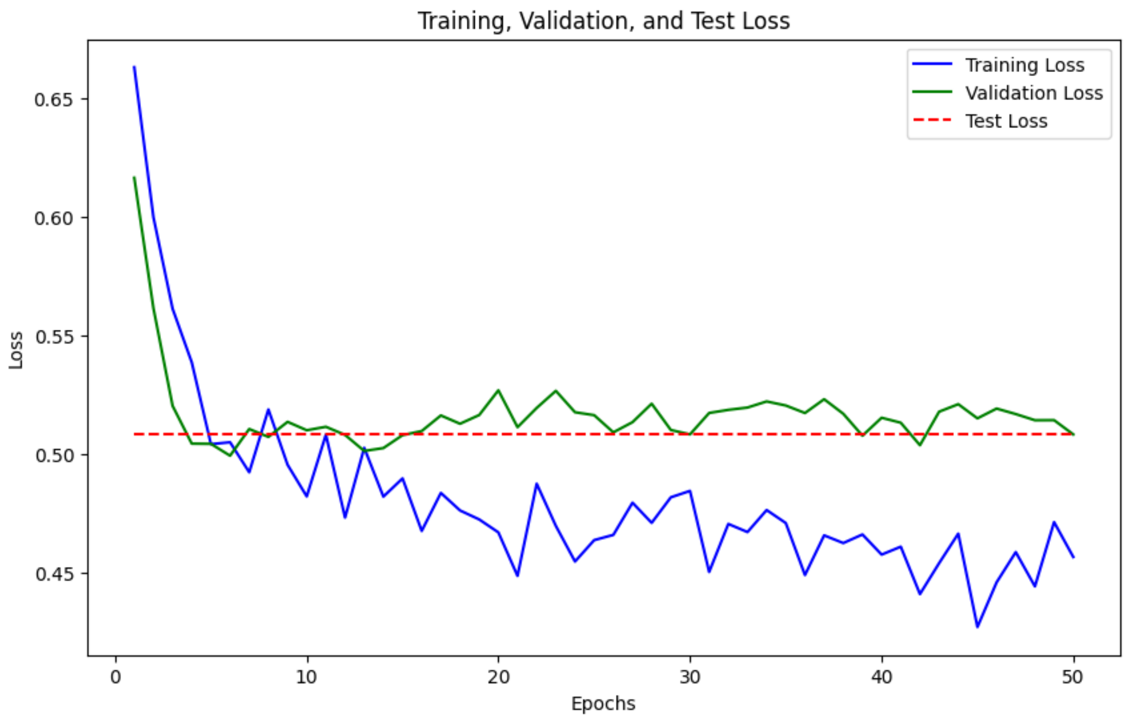
Observation:

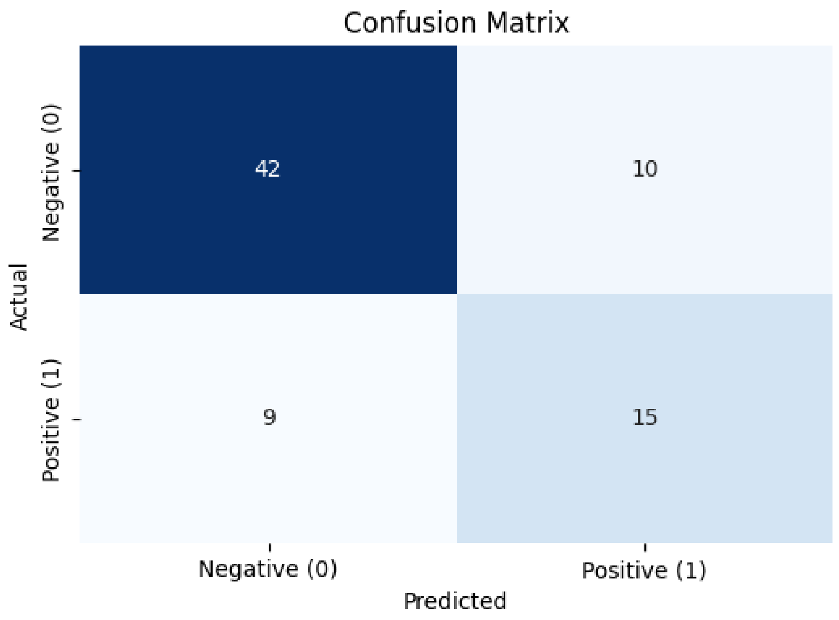
Our model took 0.49 seconds to train on the given dataset and was able to provide the following performance metrics:

- Accuracy: 75.00%

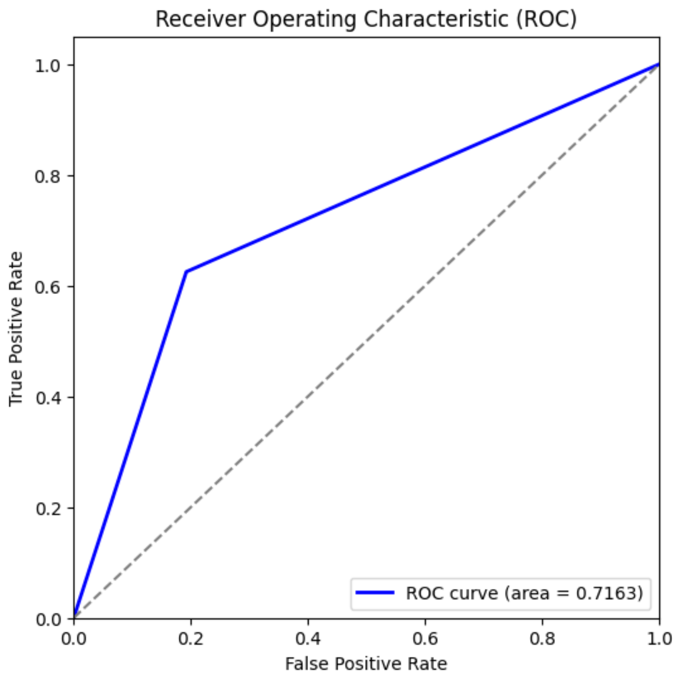


- Precision: 60.00%
- Recall: 0.6250





- F1 Score: 0.6122



Exporting the optimal weight

We'll export our optimal weights as 'jagritsh_pgnanase_assignment2_part1_2.pt' using the following code:

```
torch.save(model.state_dict(), 'jagritsh_pgnanase_assignment2_part1_2.pt')
```

Part 2: Optimizing NN [20 points]

Observations of hyperparameter setups:

DROPOUTS

	VALUE	Test Accuracy
Setup 1	0.1	76.32%
Setup 2	0.3	72.37%
Setup 3	0.6	77.63%

NUMBER OF HIDDEN LAYERS

	VALUE	Test Accuracy
Setup 1	1	76.32%
Setup 2	3	77.63%
Setup 3	4	78.95%

ACTIVATION FUNCTION

	VALUE	Test Accuracy
Setup 1	Leaky ReLU	77.63%
Setup 2	ELU	78.95%
Setup 3	SELU	76.32%

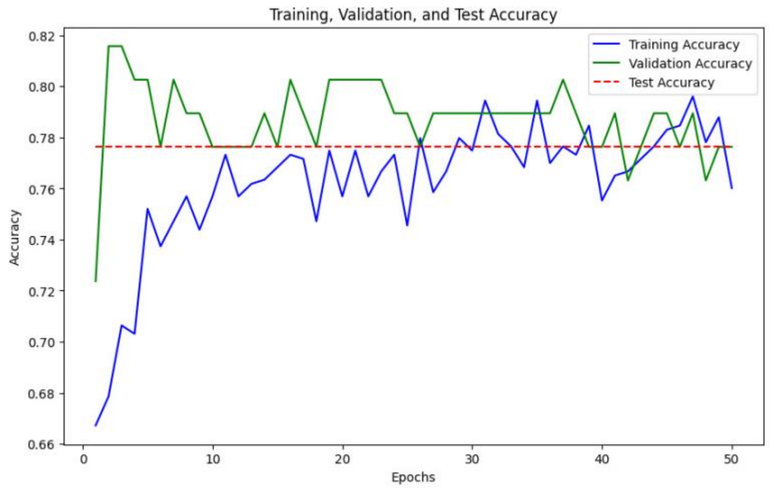
Analysis:

1. As we can see the accuracy increases as we increase our dropout, indicating that there are excessive nodes which are confusing the model and reducing the accuracy. Hence the higher the dropout value, the higher the accuracy.
2. From the above table, we can observe that the accuracy increases as we add the number of hidden layers. This is because adding each layer will make the model compute complex pattern which helps the model to predict the output accurately.
3. We have got the highest accuracy using ELU followed by Leaky ReLU and SELU as 78.95%, 77.63% and 76.32% respectively. ELU smoothens the negative input curve giving faster convergence and better gradient flow. Therefore, it is the better option to choose. Leaky ReLU performs better than SELU because it prevents inactive neurons while SELU doesn’t perform that well due to differences in model architecture or data characteristics.

Interesting Observations:

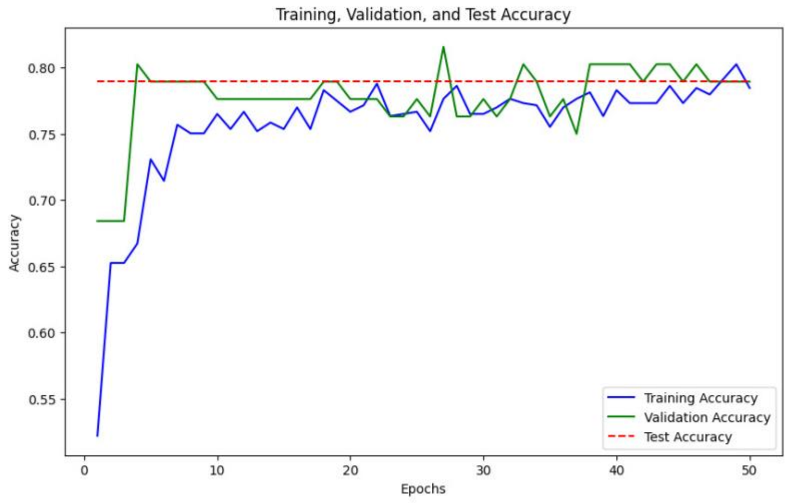
From the following graphs we get to know the Accuracy has changed due to change in parameter.

Dropout = 0.6



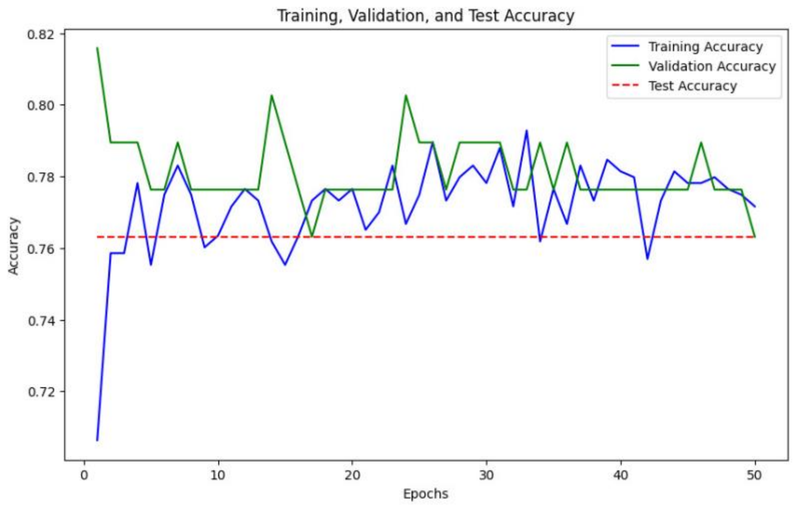
This is the accuracy analysis graph for dropout = 0.6 which has a high accuracy. We see that the accuracy is converging to 78%. ∴ Increased dropout increases the performance of the model.

Number of Layers = 4



When the number of layers is 4, we get the accuracy of around 79%. This indicates that increasing the number of hidden layers increases the efficiency in computing complex pattern which helps in predicting the output.

SELU Activation Function



SELU being a good activation function has given a lesser accuracy indicates that there are differences in model architecture and data characteristics. Yet this has also given a reasonable accuracy. Initially, the validation accuracy reached 82% and then dropped to 77%.

Methods used to optimize the model:

We have used 4 methods to increase our accuracy. The methods are listed as follows:

- Early Stopping
- Learning Rate scheduler
- Batch Normalization
- K Fold method.

Early Stopping:

We can avoid overfitting during training by using the early stopping regularization strategy. When the model's performance on a validation set begins to deteriorate instead of improving, it stops training. Early stopping can reduce computation time and prevent the model from overtraining and performing poorly when applied to new data by tracking a parameter such as validation loss or accuracy. During training, if the validation performance doesn't improve after a certain number of iterations, we stop the training. This has given us an accuracy of **76.32%**.

Learning Rate Scheduler

Learning rate scheduler modifies the learning rate to facilitate more effective model convergence. While a low learning rate can result in overly slow training, a high rate may cause the model to miss the ideal weights. Fast convergence and accuracy can be balanced by the model with the aid of the scheduler, which adjusts the learning rate according to the iterations or performance. Typically, the learning rate is reduced as the training progresses, often by a factor once the validation loss becomes constant. This has given us an accuracy of **77.63%**.

Batch Normalization

By standardizing the inputs to a layer across the mini-batch, batch normalization guarantees that each input to a layer has a mean of 0 and a standard deviation of 1. By stabilizing the activation distribution across layers, this enables the model to learn more effectively. By standardizing the inputs during training, it lowers the internal covariate shift and facilitates the model's learning process. This has given us an accuracy of **71.05%.**

K-fold cross-validation

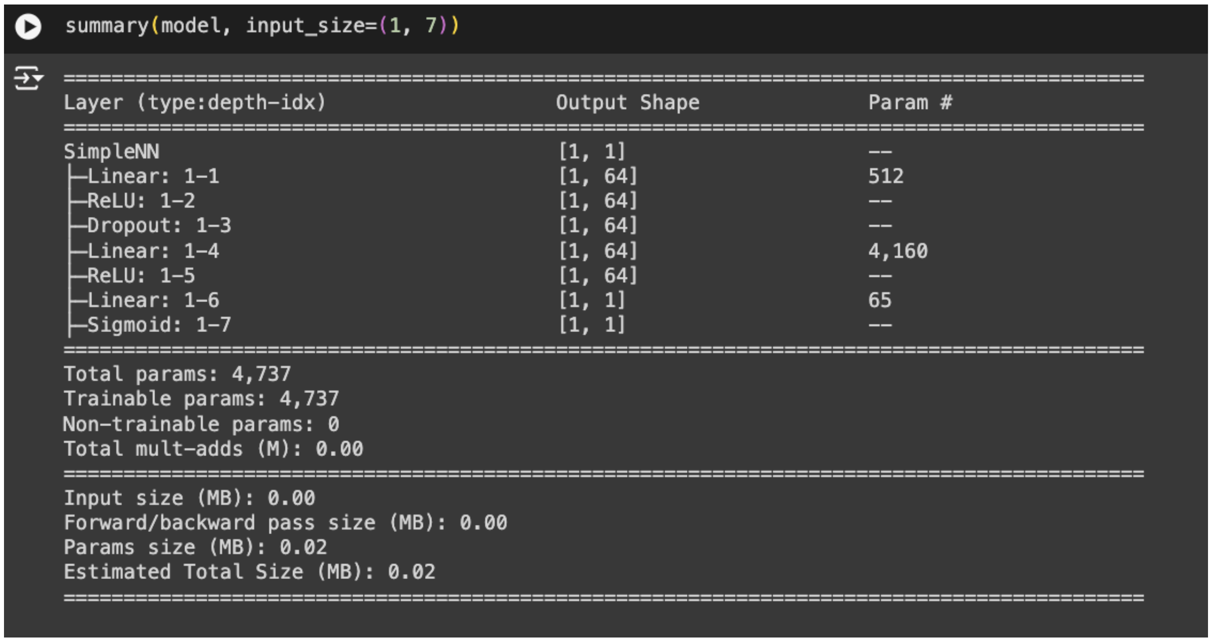
The K-fold cross-validation method helps in assessing the model's performance more robustly. To train the model, the data is divided into K subsets. The validation set is always one subset, and the training set is the remaining K−1 subsets. Overall K iterations, the ultimate performance is averaged. For each one of the K folds that make up the dataset, the model is trained on K−1 folds and validated on the remaining fold. This has given us an accuracy of **76.95%.**

Detailed Description of the **Best Model**

Hyperparameters

Hyperparameters of the best model	
Number of Layers	3
Input Layer	1 (Dimensions: (7,64))
Hidden Layer	2 (Dimensions: (64,64) and (64,1))
Dropout	0.5
Activation Functions	ReLU, sigmod
Loss Function	Binary Cross Entropy Loss
Learning Rate	0.001
Iterations	50
Batch Size	32
Optimizer	Adam

Optimization Technique	Learning Rate Scheduler

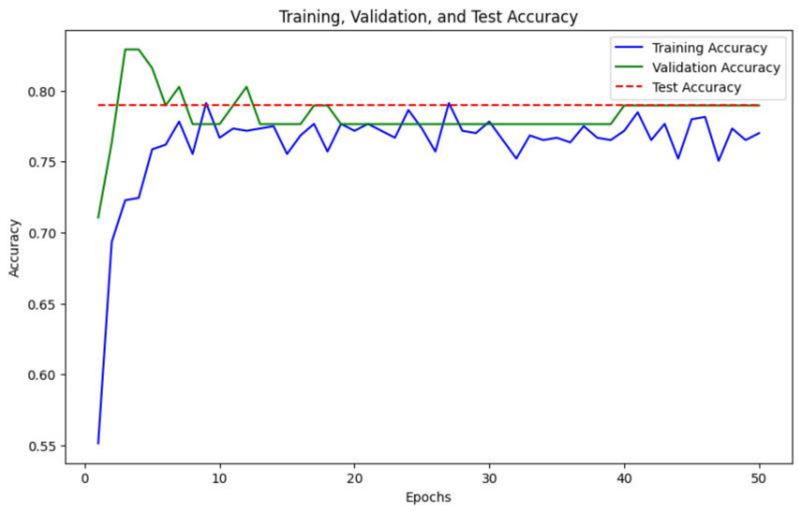


```
summary(model, input_size=(1, 7))

==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
SimpleNN                                 [1, 1]                    --
├─Linear: 1-1                            [1, 64]                   512
├─ReLU: 1-2                              [1, 64]                   --
├─Dropout: 1-3                           [1, 64]                   --
├─Linear: 1-4                            [1, 64]                   4,160
├─ReLU: 1-5                              [1, 64]                   --
├─Linear: 1-6                            [1, 1]                    65
├─Sigmoid: 1-7                           [1, 1]                    --
==========================================================================================
Total params: 4,737
Trainable params: 4,737
Non-trainable params: 0
Total mult-adds (M): 0.00
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.00
Params size (MB): 0.02
Estimated Total Size (MB): 0.02
==========================================================================================
```

Figure 1: Summary of the optimized model

Performance Metrics

- Accuracy: 78.95%
- Precision: 66.67%
- Recall: 0.6667
- F1Score: 0.6667



The accuracy of best model is around 79% which is higher than other models. The Model has shown higher validation accuracy initially, indicating that it has learnt very well. This tells us that it is a perfect fit. The Best Model has also shown great accuracy in predicting the output on test data.

Part 3: Building a CNN [20 points]

Introduction

Objective: The purpose of this experiment is to develop a Convolutional Neural Network (CNN) from scratch and then train it on our image dataset. This model will then be used for image classification using a dataset consisting of 36 categories (i.e. 10 digits and 26 alphabets). Each category has 2800 example files for our model to train on. Our goal is to create a model which can accurately identify and categorize images into the 36 classes. In this experiment, we will be performing data preprocessing, implementing a CNN model from scratch using torch and experiment with various optimization techniques to finally enhance the performance of our base model.

Importance of a Convolutional Neural Network: Convolutional Neural Networks (CNNs) play a crucial role in the field of image processing and computer vision. They automatically learn from the hierarchical features from the image dataset provided to efficiently recognize images and detect objects and faces. CNNs utilize convolutional hidden layers, which divides the task of processing, making it relatively faster and less resource intensive while being able to recognize images with higher accuracies. CNN models can be very helpful in the fields of medical imaging, autonomous driving and facial recognition.

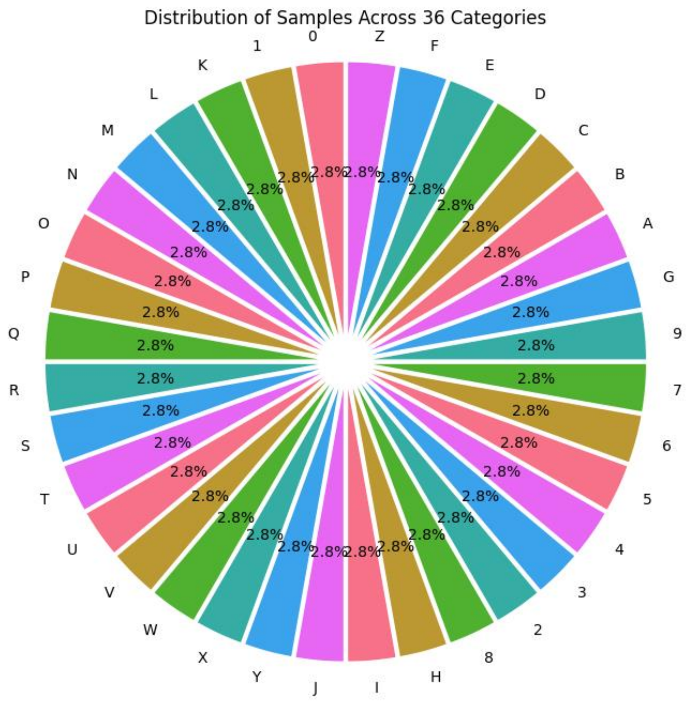
Features of the dataset used to predict the target: Our dataset (cnn_dataset.zip) consists of sample handwritten image files of 36 categories (10 digits: 0-9 and 26 letters: A-Z) with approximately 2800 images with of dimensions 28x28 pixels & a size of about 484 bytes each for each category to train the model, making it 100,837 image files in total.

Data Preparation

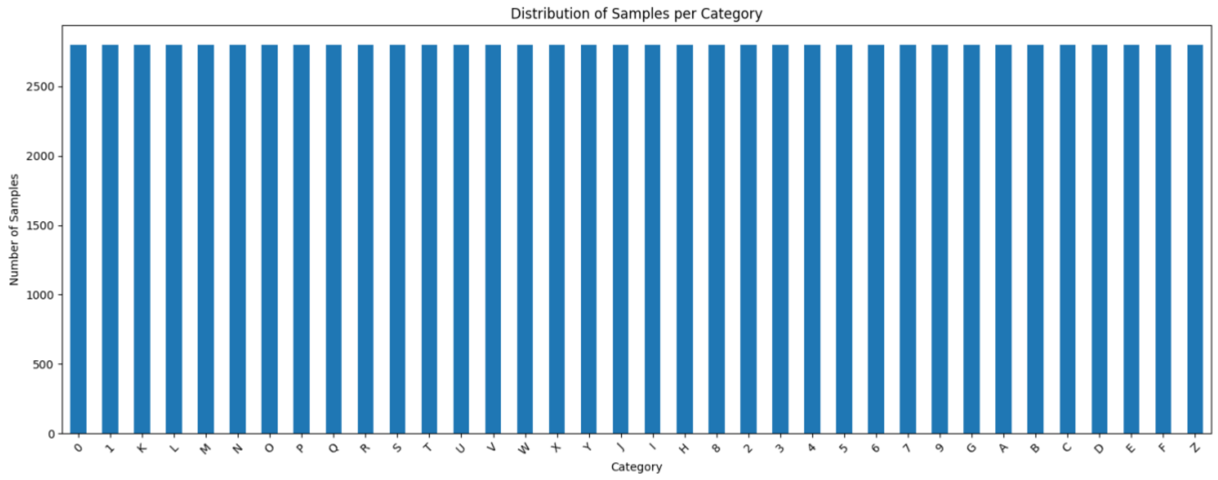
Data Statistics

The following graphs describe some important insights on our image dataset:

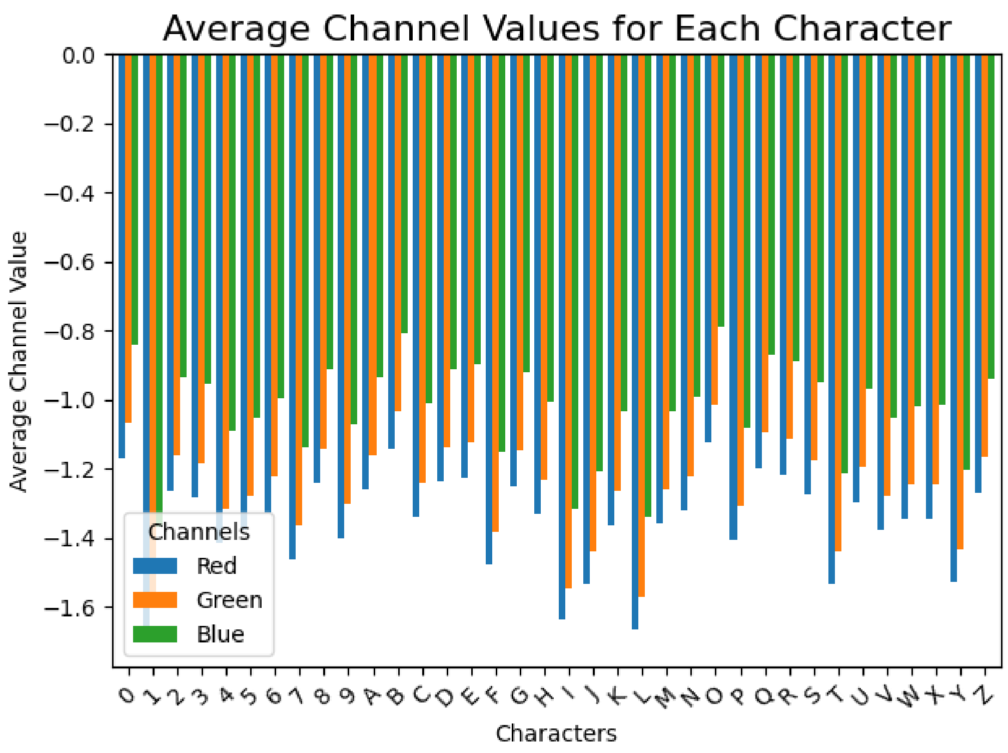
1. Pie Chart: The following graph describes that the samples images are equally distributed into 36 categories (10 digits and 26 letters) with each character comprising of an equal share of 2.8% of the total samples.



2. Bar Chart: The following bar chart represents that the samples are equally distributed among 36 categories with each category comprising of approximately 2800 samples, making a total of 100,837 images.



3. Histogram: The following histogram depicts the average channel values for each character in the dataset. The x-axis represents the characters, and the y-axis represents the average channel value. There are 3 bars for each character representing their respective values for red, green and blue channels.



Data Preprocessing

1. We will begin with our image folder using the following function:

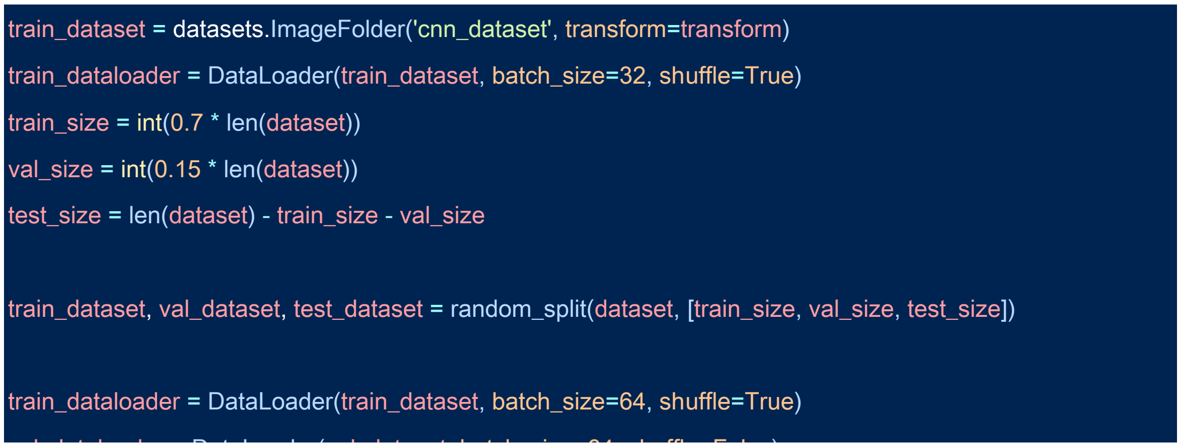
```
dataset = datasets.ImageFolder('cnn_dataset', transform=transform)
```

2. Transform the dataset using transform function. Transform function randomly flips, rotates and normalizes the images.

```
transform = transforms.Compose([
    transforms.Resize((28, 28)),
    transforms.ToTensor(),
    transforms.RandomRotation(10),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
```

Splitting the dataset

1. Divide the dataset into train, validation and test sets.



```
train_dataset = datasets.ImageFolder('cnn_dataset', transform=transform)
train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True)
train_size = int(0.7 * len(dataset))
val_size = int(0.15 * len(dataset))
test_size = len(dataset) - train_size - val_size

train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size, val_size, test_size])

train_dataloader = DataLoader(train_dataset, batch_size=64, shuffle=True)
```

∴ Total samples: 100,800

Training samples: 70560

Validation samples: 15120

Test samples: 15120

Model Implementation

We use a class structure to implement our basic convolutional neural network. This model has been implemented from scratch using torch package. The forward method defines the forward pass of the neural network, specifying how input data flows through the layers to produce an output. It's where we implement the actual computation performed by our neural network. Our class contains 2 methods, and it is structured as follows:

Methods used:

__init__():

This method is used to initialize the hyperparameters used in our CNN architecture.

forward():

This method defines the forward pass of the neural network, basically defining how the input data should flow through the layers to deduce the output.

CNN Architecture:

Input Neurons	3 (RGB channels)
Output Neurons	36
Activation function used for hidden layers	ReLU (Rectified Linear Unit)
Activation function used for output layer	None (raw logits)
Hidden layers	5

Size of each hidden layer	(32, 32, 64, 128, 512)
Dropout value	0.5

Model Summary:

```
==========================================================================
Layer (type:depth-idx)              Output Shape          Param #
==========================================================================
BaseCNN                          [1, 36]                --
├─Conv2d: 1-1                      [1, 32, 28, 28]        896
├─MaxPool2d: 1-2                     [1, 32, 14, 14]        --
├─Conv2d: 1-3                      [1, 64, 14, 14]        18,496
├─MaxPool2d: 1-4                     [1, 64, 7, 7]          --
├─Conv2d: 1-5                      [1, 128, 7, 7]         73,856
├─MaxPool2d: 1-6                     [1, 128, 3, 3]         --
├─Linear: 1-7                      [1, 512]               590,336
├─Dropout: 1-8                       [1, 512]               --
├─Linear: 1-9                      [1, 36]                18,468
==========================================================================
Total params: 702,052
```

Performance Metrics on the base model:

Training completed in 1060.49 seconds.

-----------------------------BASE MODEL-----------------------------

Best Validation Accuracy: 0.8839

Final Metrics on Validation Set:

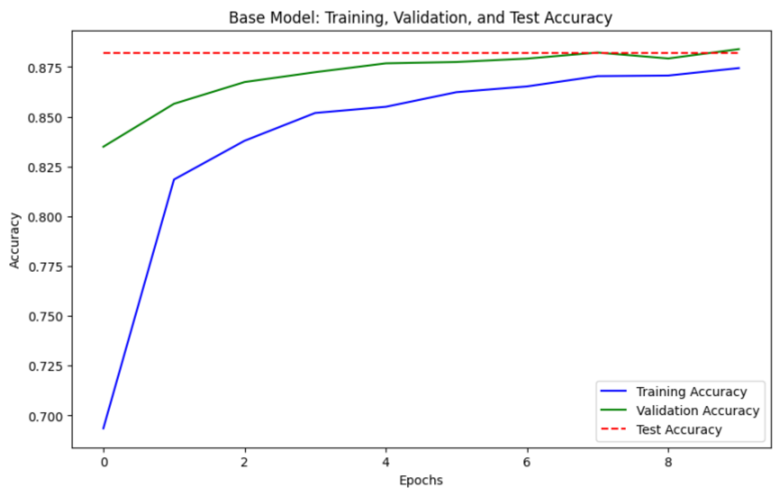
Test Accuracy: 0.8819

Test Precision: 0.8829

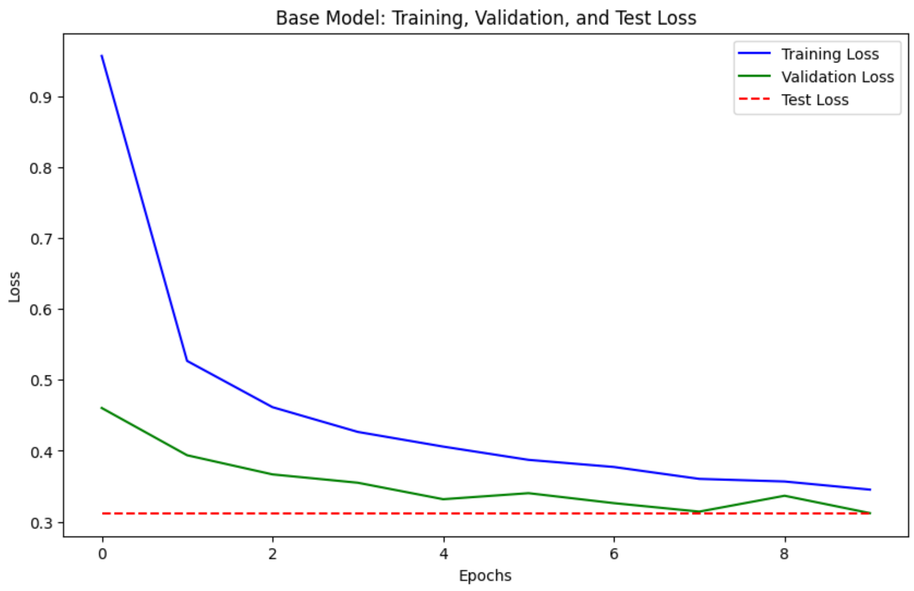
Test Recall: 0.8819

Test F1 Score: 0.8819

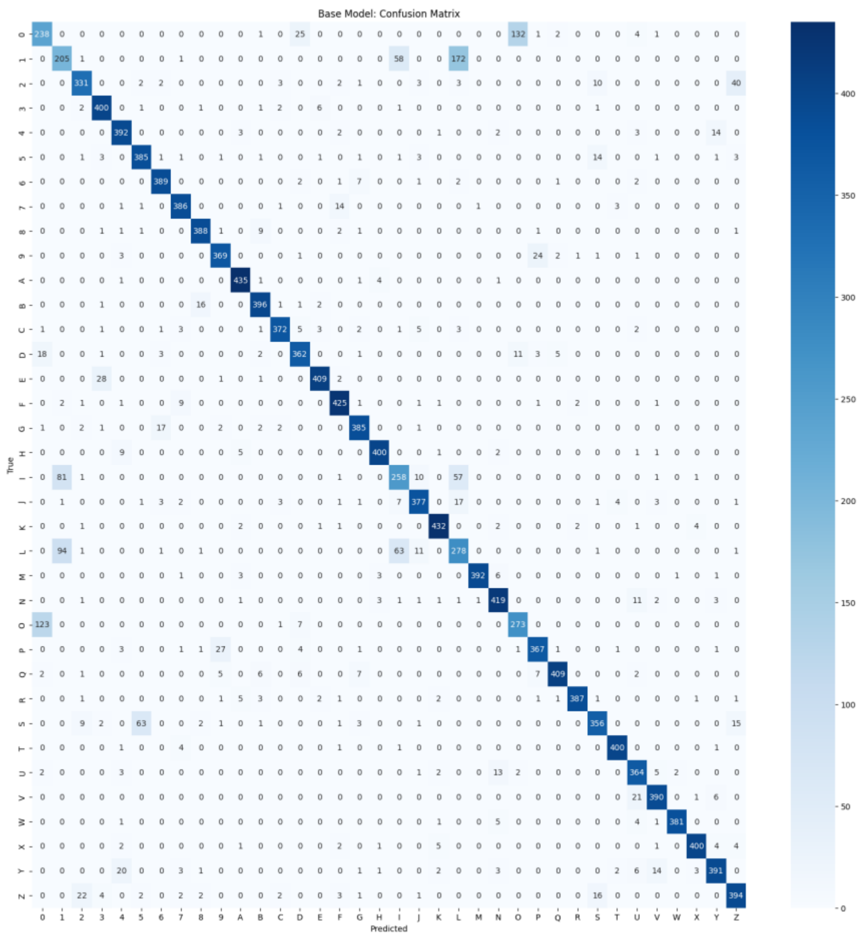
Accuracy



Loss



Correlation Matrix



Optimizing Convolutional Neural Network (CNN) model

We will be optimizing our CNN model by experimenting with the following 3 optimization techniques:

1. Batch Normalization
2. Learning Rate Scheduler
3. K-Fold Cross-Validation

Optimization Techniques:

Batch Normalization:

Optimizing our model using batch normalization technique gives us the following performance metrics:

```
Training completed in 1102.99 seconds.
------------------------------BATCH NORMALIZED MODEL----------------------
Best Validation Accuracy: 0.8930
```

Final Metrics on Validation Set:

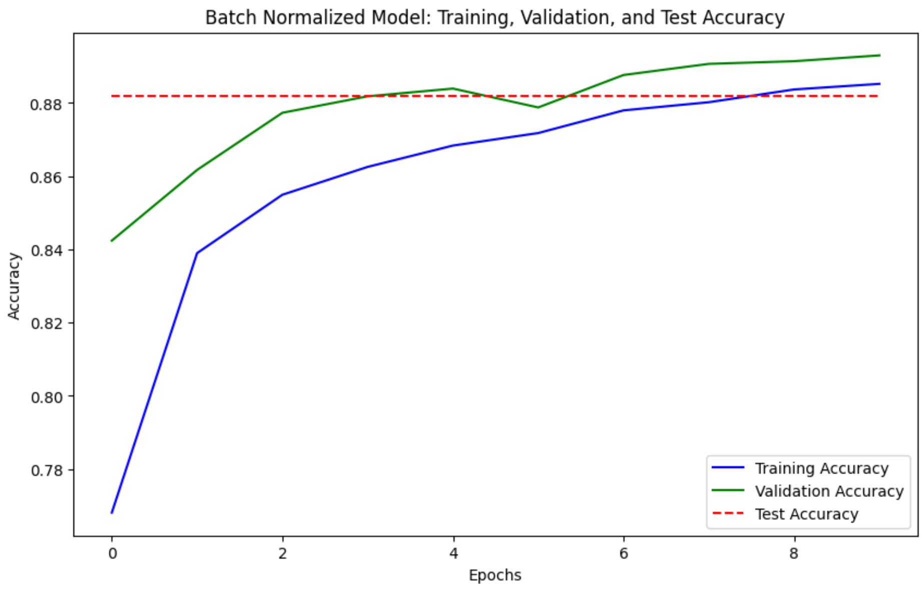
Test Accuracy: 0.8932

Test Precision: 0.8945

Test Recall: 0.8932

Test F1 Score: 0.8930

Accuracy using Batch Normalization



Learning Rate Scheduler:

Optimizing our model using Learning rate scheduler technique gives us the following performance metrics:

Training completed in 1051.56 seconds.

----------LEARNING RATE SCHEDULER OPTIMIZED MODEL----------

Best Validation Accuracy: 0.8941

Final Metrics on Validation Set:

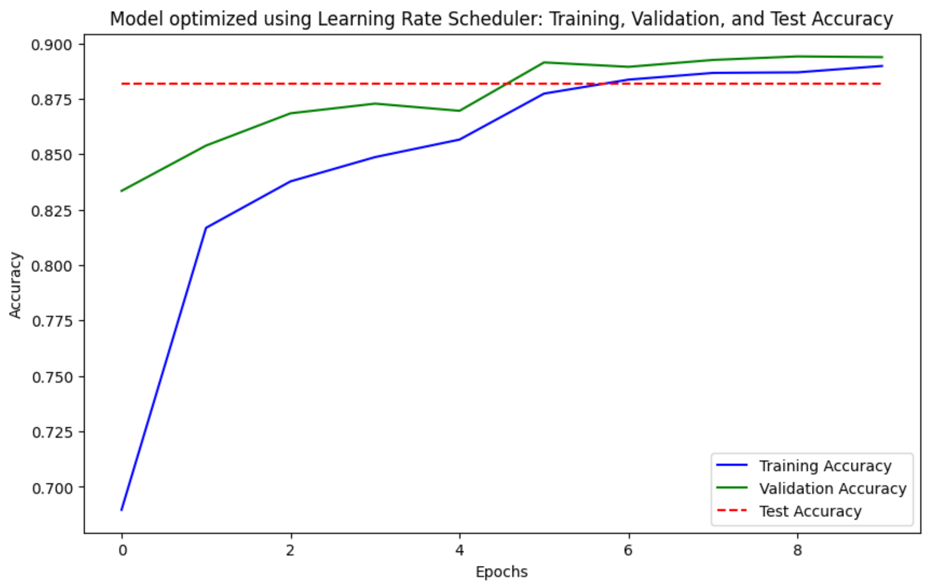
Test Accuracy: 0.8929

Test Precision: 0.8948

Test Recall: 0.8929

Test F1 Score: 0.8925

Accuracy using Learning Rate Scheduler



K-Fold Cross-Validation:

Optimizing our model using K-Fold cross-validation technique gives us the following performance metrics:

Training completed in 6630.09 seconds.

-----------------------K-FOLD OPTIMIZED MODEL-----------------------

Average performance across all 5 folds:

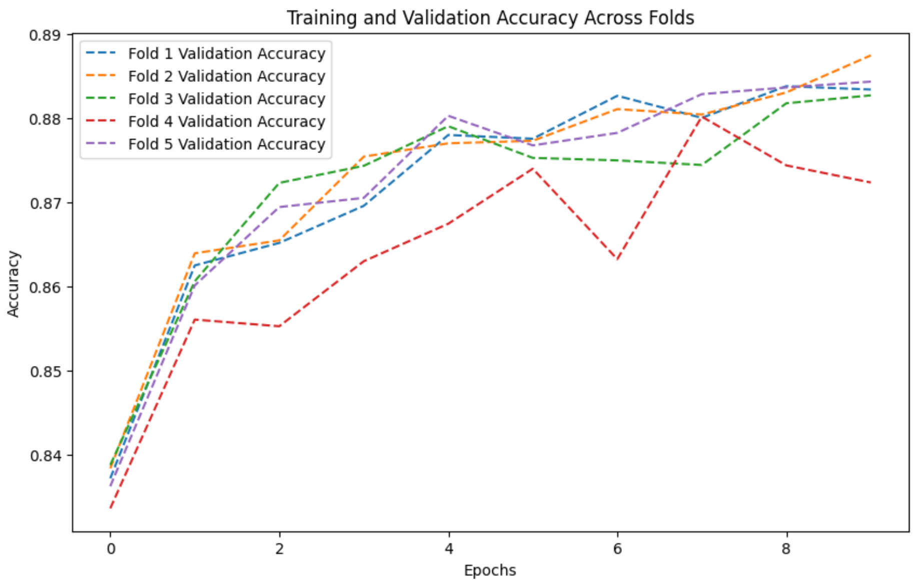
Final Train Loss: 0.2883

Final Validation Loss: 0.0650

Final Train Accuracy: 0.8705

Final Validation Accuracy: 0.8821

Accuracy using K-Fold Cross-Validation



∴ After analyzing and comparing the performance metrics,

```
---------COMPARING ACCURACIES FOR EACH OPTIMIZATION METHOD---------
Batch Normalization Optimization Accuracy: 0.8931878306878307
Learning Rate Scheduler Optimization Accuracy: 0.8928571428571429
K-Fold Optimization Accuracy: 0.8820833333333334
```

We achieve the best model by Batch Normalization technique with an accuracy of 0.8932.

Batch Normalized Base Model:

Since, batch normalization gave consistently high accuracies on our base model, we will be moving forward with it for our experiment. The CNN architecture for the optimized model is given as follows:

Input Neurons	3 (RGB channels)
Output Neurons	36
Activation function used for hidden layers	ReLU (Rectified Linear Unit)
Activation function used for output layer	None (raw logits)
Hidden layers	4
Size of each hidden layer	(32, 64, 128, 512)
Dropout value	0.5

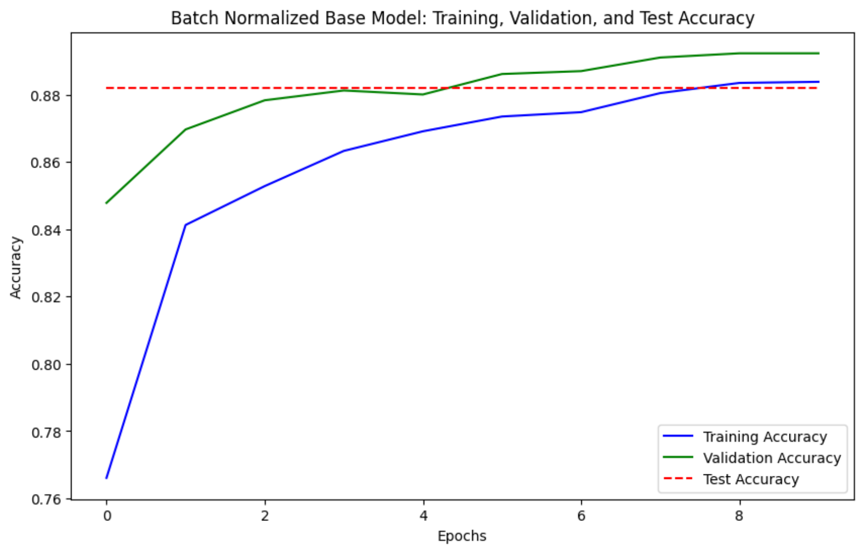
```
class BNModel(BaseCNN):
    def __init__(self):
        super(BNModel, self).__init__()
        self.bn1 = nn.BatchNorm2d(32)
        self.bn2 = nn.BatchNorm2d(64)
        self.bn3 = nn.BatchNorm2d(128)
        self.bn4 = nn.BatchNorm1d(512)

    def forward(self, x):
        x = self.pool(torch.relu(self.bn1(self.conv1(x))))
        x = self.pool(torch.relu(self.bn2(self.conv2(x))))
        x = self.pool(torch.relu(self.bn3(self.conv3(x))))
        x = x.view(-1, 128 * 3 * 3)
```

Performance Metrics of our Batch Normalized Base Model:

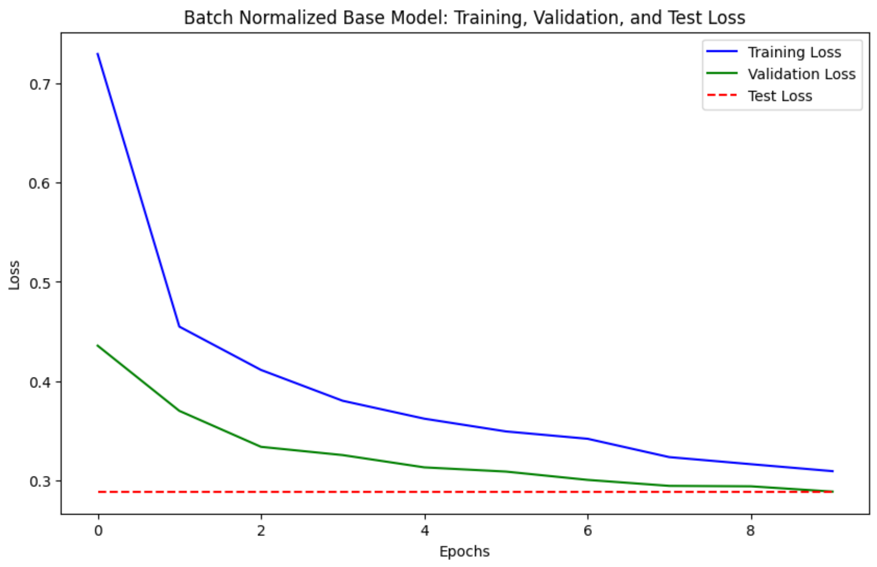
```
Training completed in 1130.43 seconds.
--------------------------BATCH NORMALIZED BASE MODEL---------------------
Best Validation Accuracy: 0.8923
Final Metrics on Validation Set:
Test Accuracy: 0.8927
Test Precision: 0.8950
Test Recall: 0.8927
Test F1 Score: 0.8924
```

Visualizing the results on our Optimized Base Model:

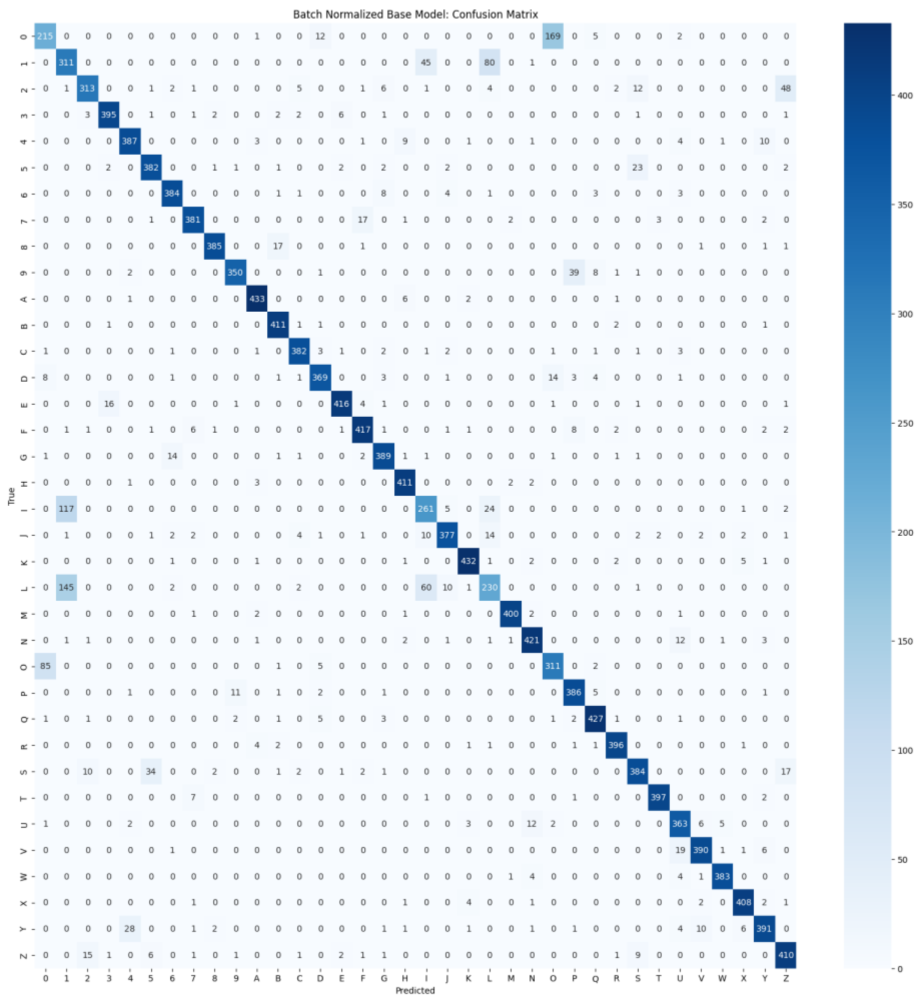
Accuracy of our Batch Normalized Base Model



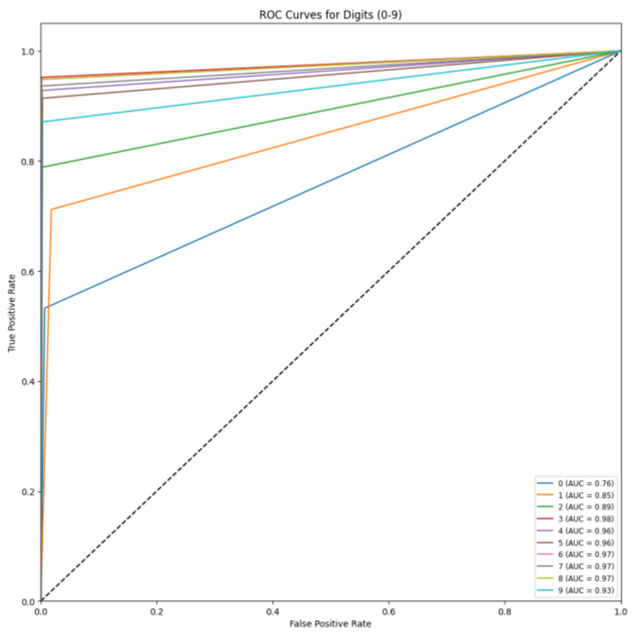
Loss of our Batch Normalized Base Model

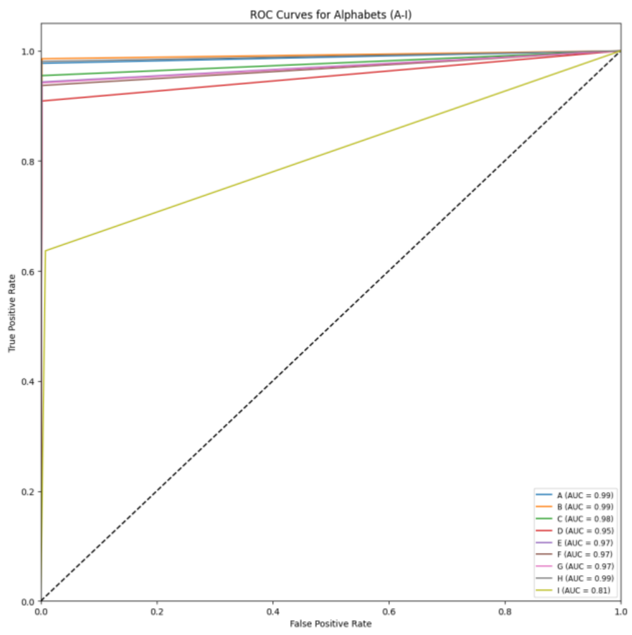


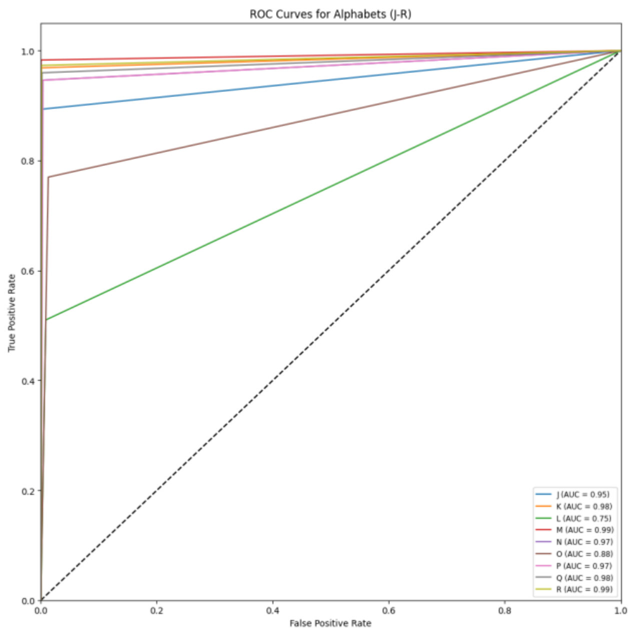
Confusion matrix of our Batch Normalized Base Model

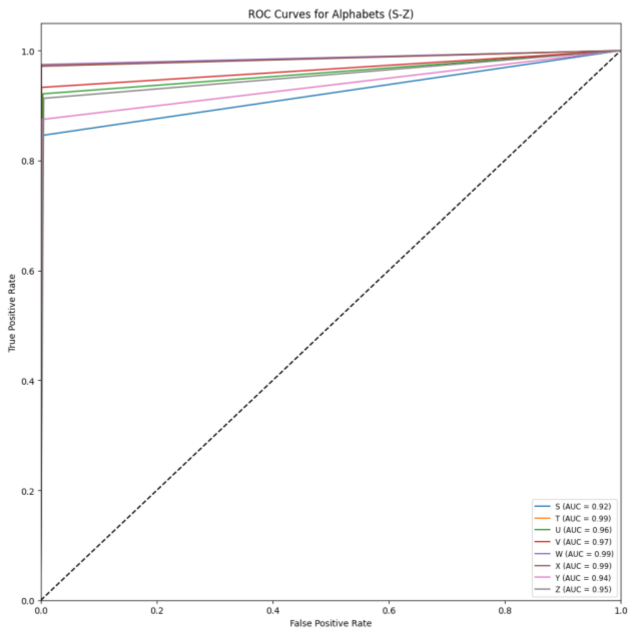


ROC Curves for each character:









The above graphs describe how the accuracy of the model increases, and the loss of the model decreases by each iteration, finally leading to give us the optimal results. The diagonal in the confusion matrix depicts that most of the images were successfully recognized as their respective labeled character. We finally conclude our observation by plotting a Receiver Operating Character (ROC) curve for the 36 characters giving an average AUC score of 0.9450.

Part 4: VGG-13 Implementation [20 points]

Introduction

Objective: The purpose of this experiment is to develop a VGG Neural Network (CNN) from scratch and then train it on our image dataset. This model will then be used for image classification using a dataset consisting of 36 categories (i.e. 10 digits and 26 alphabets). Each category has 2800 example files for our model to train on. Our goal is to create a model which can accurately identify and categorize images into the 36 classes. In this experiment, we will be performing data preprocessing, implementing a VGG model from scratch using torch , train the model with our EMIST dataset and evaluate the model with accuracy matrix.

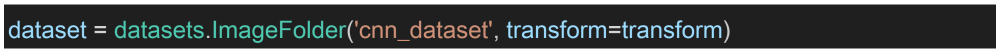
Importance of a VGG Convolutional Neural Network: The VGG network is a type of Convolutional Neural Network (CNN) that has significantly impacted image recognition and computer vision. Its deep structure, consisting of many small 3x3 convolutional filters, enables it to capture fine-grained details in images, resulting in high accuracy. This architecture is both effective and relatively efficient in processing, making it widely used in areas like medical imaging, self-driving cars, and facial recognition where high accuracy is essential.

Features of the dataset used to predict the target: Our dataset (cnn_dataset.zip) consists of sample handwritten image files of 36 categories (10 digits: 0-9 and 26 letters: A-Z) with approximately 2800 images with of dimensions 28x28 pixels & a size of about 484 bytes each for each category to train the model, making it 100,837 image files in total

Data Preparation

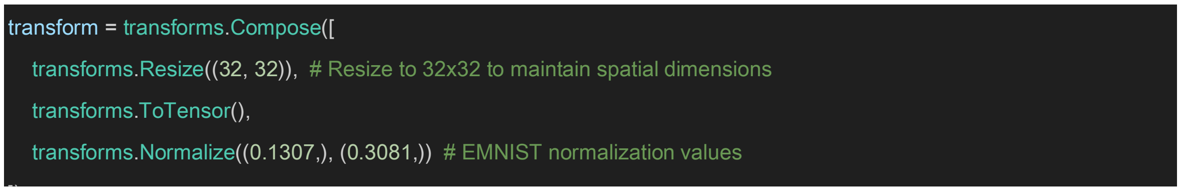
Data Preprocessing

1. We will begin with our image folder using the following function:



```
dataset = datasets.ImageFolder('cnn_dataset', transform=transform)
```

2. Transform the dataset using transform function. Transform function resizes and normalizes the images.



```
transform = transforms.Compose([
    transforms.Resize((32, 32)),  # Resize to 32x32 to maintain spatial dimensions
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))  # EMNIST normalization values
```

Splitting the dataset

1. Divide the dataset into train, validation and test sets.

```
dataset = datasets.ImageFolder('cnn_dataset', transform=transform)
train_dataset = datasets.ImageFolder('cnn_dataset', transform=transform)

train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True)
train_size = int(0.7 * len(dataset))
val_size = int(0.15 * len(dataset))
test_size = len(dataset) - train_size - val_size

train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size, val_size, test_size])

train_dataloader = DataLoader(train_dataset, batch_size=64, shuffle=True)
```

∴ Total samples: 100,800

Training samples: 70560

Validation samples: 15120

Test samples: 15120

Model Implementation

Methods used:

__init__():

This method is used to initialize the hyperparameters used in our VGG architecture.

forward():

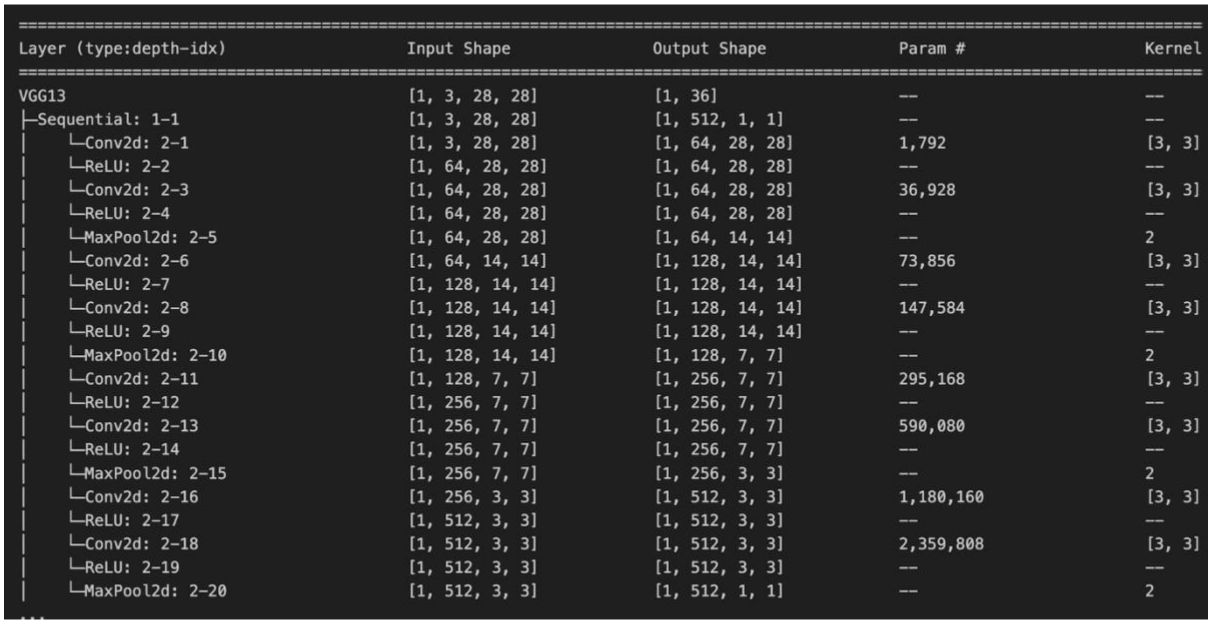
This method defines the forward pass of the neural network, basically defining how the input data should flow through the layers to deduce the output.

VGG Architecture:

Input Neurons	3 (RGB channels)
Output Neurons	36
Activation function used for hidden layers	ReLU (Rectified Linear Unit)
Activation function used for output layer	None (raw logits)

Hidden layers	5 convolutional blocks, each with 2-3 convolutional layers followed by max-pooling layers
Size of each hidden layer	(32, 32, 64, 128, 512)
Dropout value	0.5

Model Summary:



```
==========================================================================================================================
Layer (type:depth-idx)                   Input Shape               Output Shape              Param #                   Kernel
==========================================================================================================================
VGG13                                    [1, 3, 28, 28]            [1, 36]                   --                        --
├─Sequential: 1-1                        [1, 3, 28, 28]            [1, 512, 1, 1]            --                        --
│    └─Conv2d: 2-1                       [1, 3, 28, 28]            [1, 64, 28, 28]           1,792                     [3, 3]
│    └─ReLU: 2-2                         [1, 64, 28, 28]           [1, 64, 28, 28]           --                        --
│    └─Conv2d: 2-3                       [1, 64, 28, 28]           [1, 64, 28, 28]           36,928                    [3, 3]
│    └─ReLU: 2-4                         [1, 64, 28, 28]           [1, 64, 28, 28]           --                        --
│    └─MaxPool2d: 2-5                    [1, 64, 28, 28]           [1, 64, 14, 14]           --                        2
│    └─Conv2d: 2-6                       [1, 64, 14, 14]           [1, 128, 14, 14]          73,856                    [3, 3]
│    └─ReLU: 2-7                         [1, 128, 14, 14]          [1, 128, 14, 14]          --                        --
│    └─Conv2d: 2-8                       [1, 128, 14, 14]          [1, 128, 14, 14]          147,584                   [3, 3]
│    └─ReLU: 2-9                         [1, 128, 14, 14]          [1, 128, 14, 14]          --                        --
│    └─MaxPool2d: 2-10                   [1, 128, 14, 14]          [1, 128, 7, 7]            --                        2
│    └─Conv2d: 2-11                      [1, 128, 7, 7]            [1, 256, 7, 7]            295,168                   [3, 3]
│    └─ReLU: 2-12                        [1, 256, 7, 7]            [1, 256, 7, 7]            --                        --
│    └─Conv2d: 2-13                      [1, 256, 7, 7]            [1, 256, 7, 7]            590,080                   [3, 3]
│    └─ReLU: 2-14                        [1, 256, 7, 7]            [1, 256, 7, 7]            --                        --
│    └─MaxPool2d: 2-15                   [1, 256, 7, 7]            [1, 256, 3, 3]            --                        2
│    └─Conv2d: 2-16                      [1, 256, 3, 3]            [1, 512, 3, 3]            1,180,160                 [3, 3]
│    └─ReLU: 2-17                        [1, 512, 3, 3]            [1, 512, 3, 3]            --                        --
│    └─Conv2d: 2-18                      [1, 512, 3, 3]            [1, 512, 3, 3]            2,359,808                 [3, 3]
│    └─ReLU: 2-19                        [1, 512, 3, 3]            [1, 512, 3, 3]            --                        --
│    └─MaxPool2d: 2-20                   [1, 512, 3, 3]            [1, 512, 1, 1]            --                        2
```

Performance Metrics on the VGG model:

Training completed in 4668.23 seconds.

----------------------------BASE MODEL----------------------------

Test Loss: 0.2778

Accuracy: 90.2910%

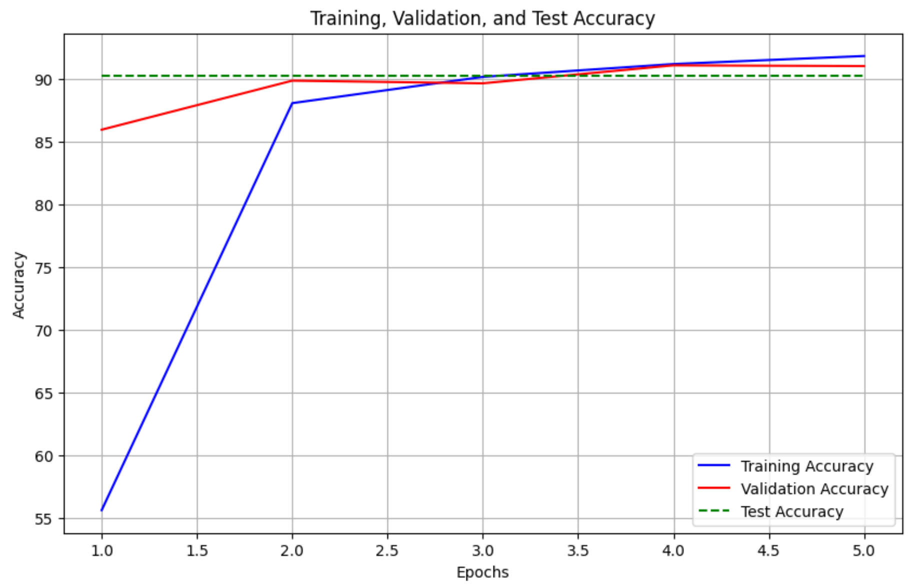
Precision: 0.9083

Recall: 0.9029

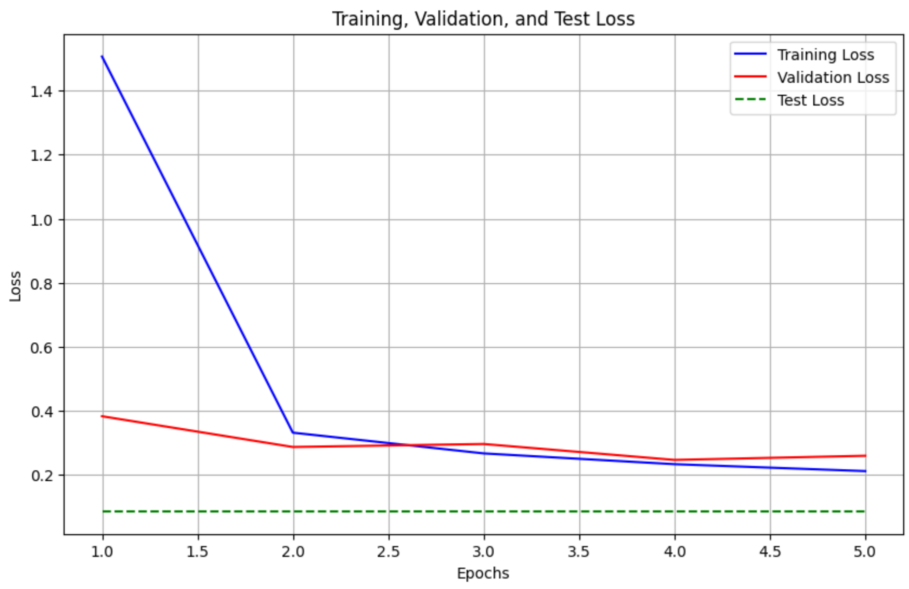
F1 Score: 0.9026

Visualizing the performance metrics on the VGG model:

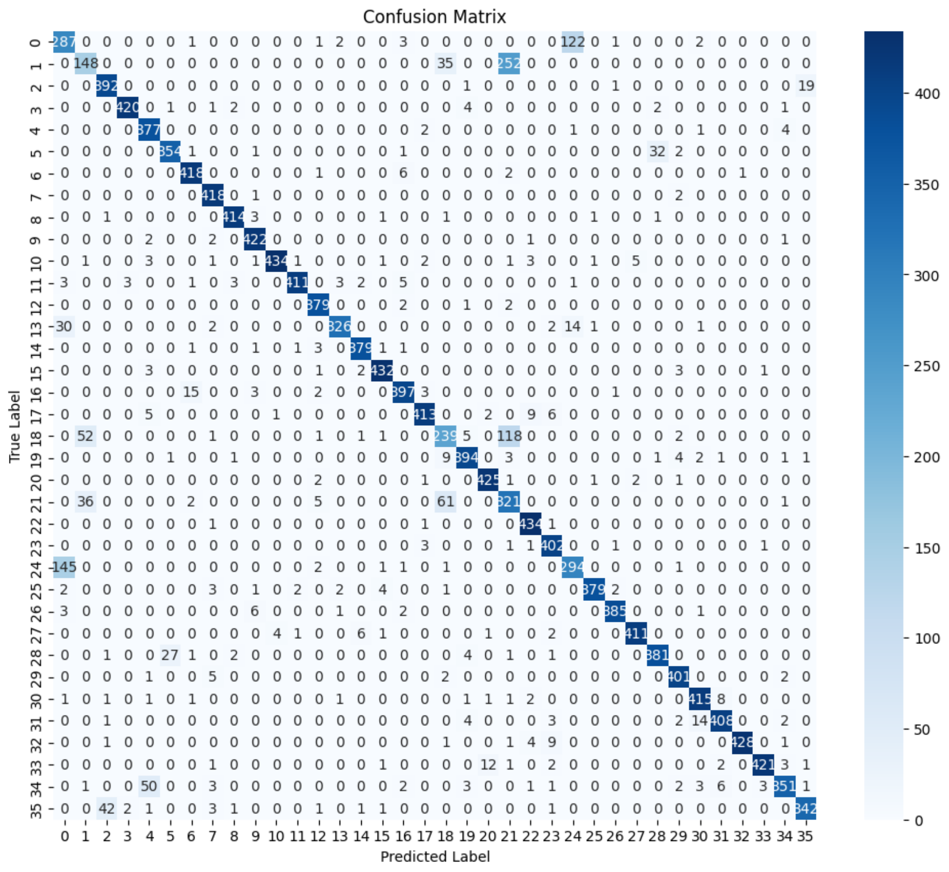
Accuracy of our VGG Model



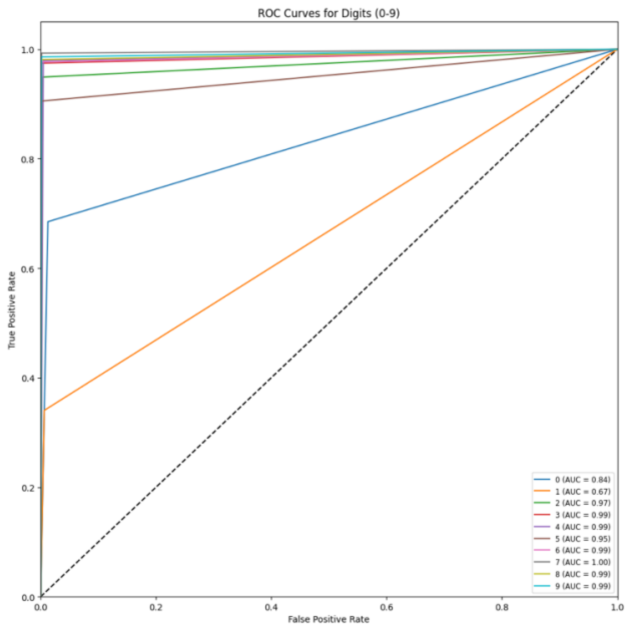
Loss of our VGG Model

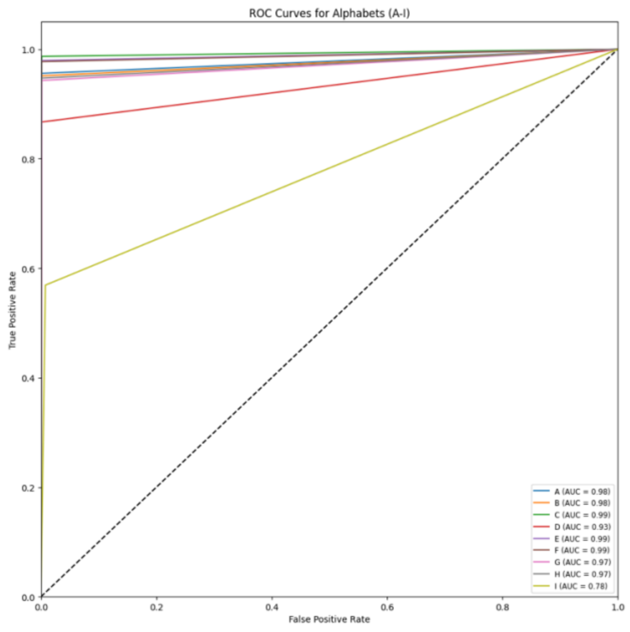


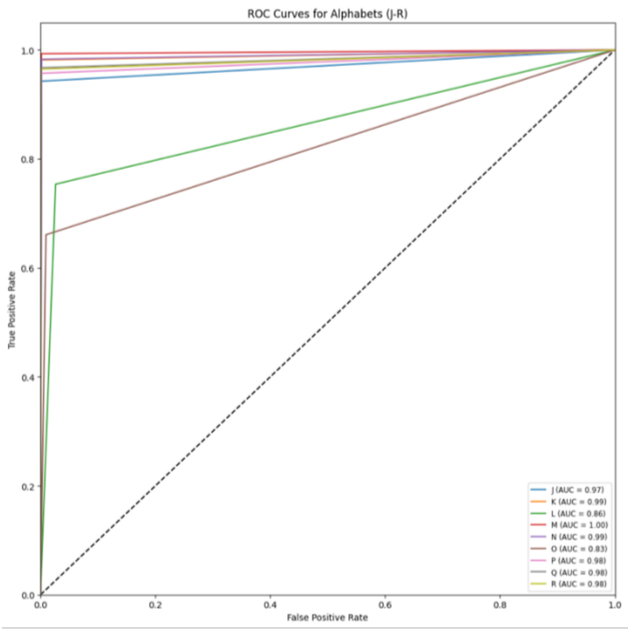
Confusion matrix of our VGG Model

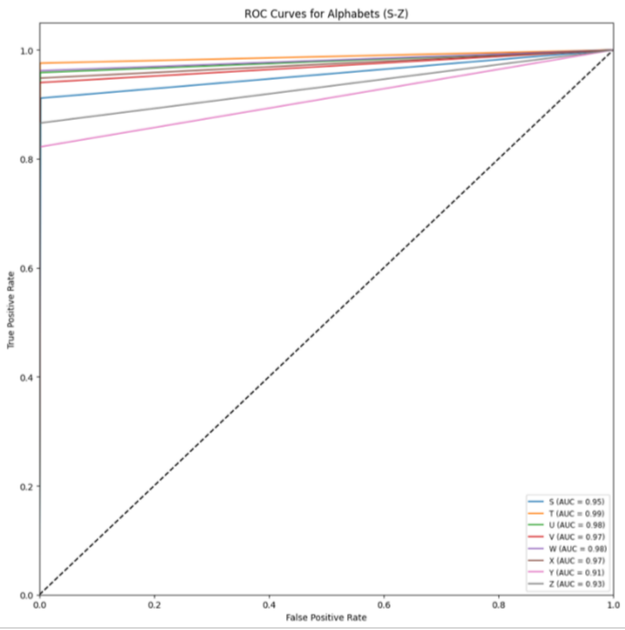


ROC Curves for each character:






ROC Curves for Alphabets (J-R)
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
J (AUC = 0.97)
K (AUC = 0.99)
L (AUC = 0.86)
M (AUC = 1.00)
N (AUC = 0.99)
O (AUC = 0.83)
P (AUC = 0.98)
Q (AUC = 0.98)
R (AUC = 0.98)


ROC Curves for Alphabets (S-Z)
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
S (AUC = 0.95)
T (AUC = 0.99)
U (AUC = 0.98)
V (AUC = 0.97)
W (AUC = 0.98)
X (AUC = 0.97)
Y (AUC = 0.91)
Z (AUC = 0.93)

Reference:

- Assignment 1
- https://www.geeksforgeeks.org/ml-handle-missing-data-with-simple-imputer/
- https://www.geeksforgeeks.org/data-pre-processing-wit-sklearn-using-standard-and-minmax-scaler/
- https://pytorch.org/docs/stable/tensors.html
- https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html
- https://pytorch.org/docs/stable/nn.html
- https://github.com/TylerYep/torchinfo
- https://docs.python.org/3/library/time.html
- https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html
- https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc
- https://pytorch.org/ignite/generated/ignite.handlers.early_stopping.EarlyStopping.html
- https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html
- https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html#torch.optim.lr_scheduler.ReduceLROnPlateau
- https://geeksforgeeks.org/batch-normalization-implementation-in-pytorch/
- https://medium.com/latinxinai/convolutional-neural-network-from-scratch-6b1c856e1c07
- https://pytorch.org/tutorials/beginner/basics/data_tutorial.html
- https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html
- https://www.geeksforgeeks.org/vgg-16-cnn-model/