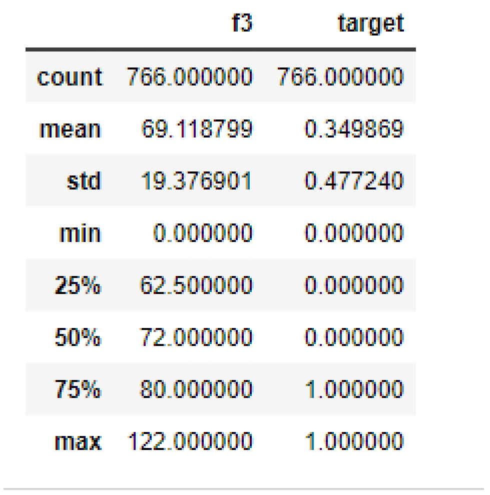
REPORT

PART I

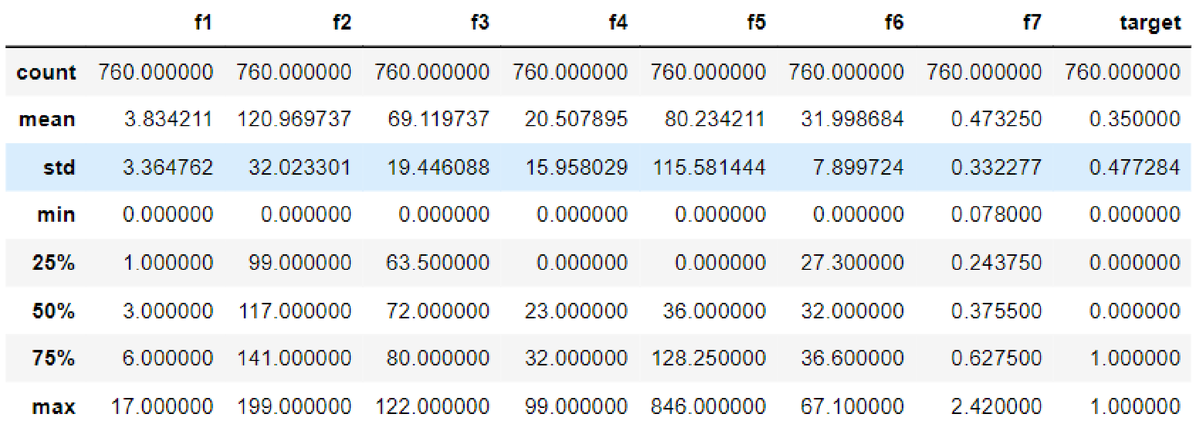
1. Nature of the dataset

The dataset given to us consists of 766 rows and 8 columns. It consists of 7 input features named f1, f2 until f7 and a target column consisting of binary values. The input features consist of continuous data and categorical data. The below figure describes the main statistics of the data.



	f3	target
count	766.000000	766.000000
mean	69.118799	0.349869
std	19.376901	0.477240
min	0.000000	0.000000
25%	62.500000	0.000000
50%	72.000000	0.000000
75%	80.000000	1.000000
max	122.000000	1.000000

Fig: Statistics of the data (Before removing non-numeric values)



	f1	f2	f3	f4	f5	f6	f7	target
count	760.000000	760.000000	760.000000	760.000000	760.000000	760.000000	760.000000	760.000000
mean	3.834211	120.969737	69.119737	20.507895	80.234211	31.998684	0.473250	0.350000
std	3.364762	32.023301	19.446088	15.958029	115.581444	7.899724	0.332277	0.477284
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	0.000000
25%	1.000000	99.000000	63.500000	0.000000	0.000000	27.300000	0.243750	0.000000
50%	3.000000	117.000000	72.000000	23.000000	36.000000	32.000000	0.375500	0.000000
75%	6.000000	141.000000	80.000000	32.000000	128.250000	36.600000	0.627500	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	1.000000

Fig: Statistics of the data (After removing non-numeric values)

2. VISUALIZATIONS

1. To determine the number of non-numeric values in a column

By plotting a histogram of the count of each value in each column, we can determine if the column consists of non-numeric values and depending on that if the number is less, we can replace them with 0 or drop. The below histogram pictorially shows the count of all the unique values in the feature f1. We can derive that the feature consists of a categorical value “c”.

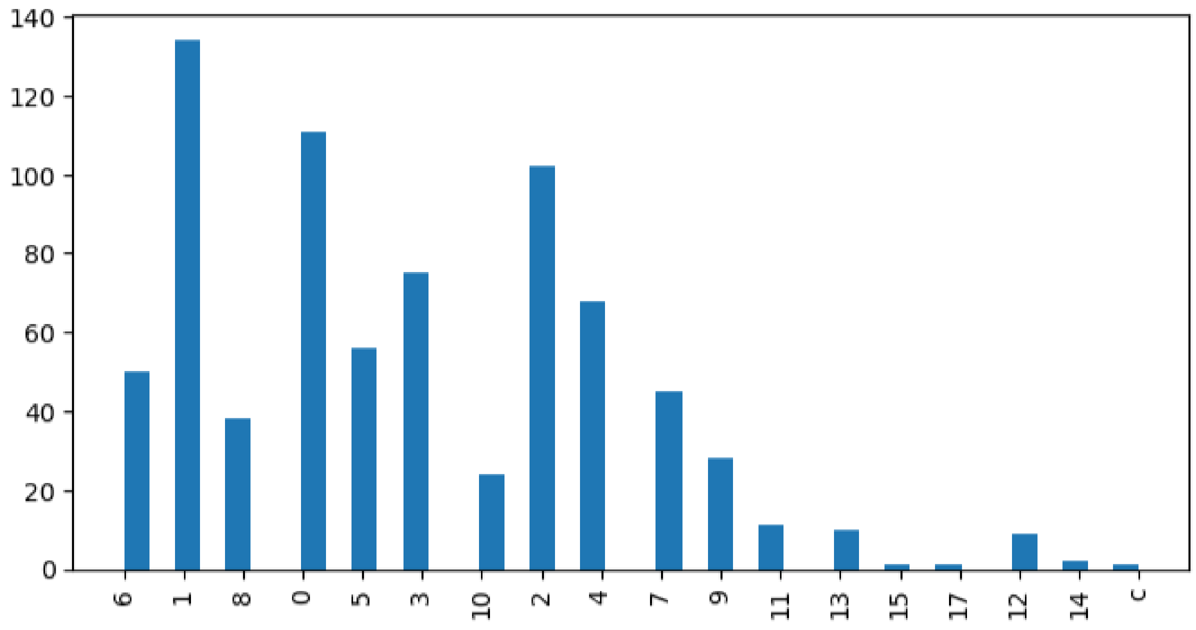


Fig: Histogram displaying the count of unique values in column f1.

2. Identifying Outliers

We have plotted boxplots to analyze if there are any outliers in the data. Removing outliers can help improve the model accuracy. From the boxplots, we inferred that there are not many outliers in the data.

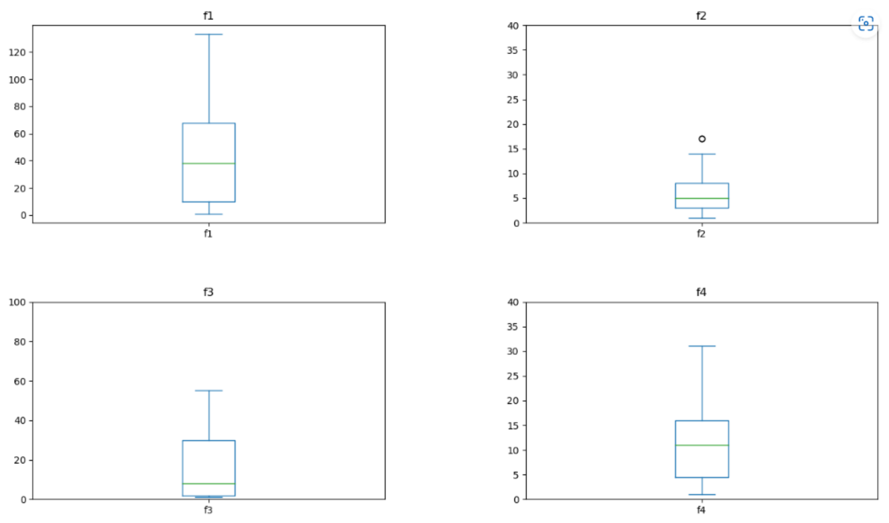


Fig: Boxplots of f1, f2, f3, f4

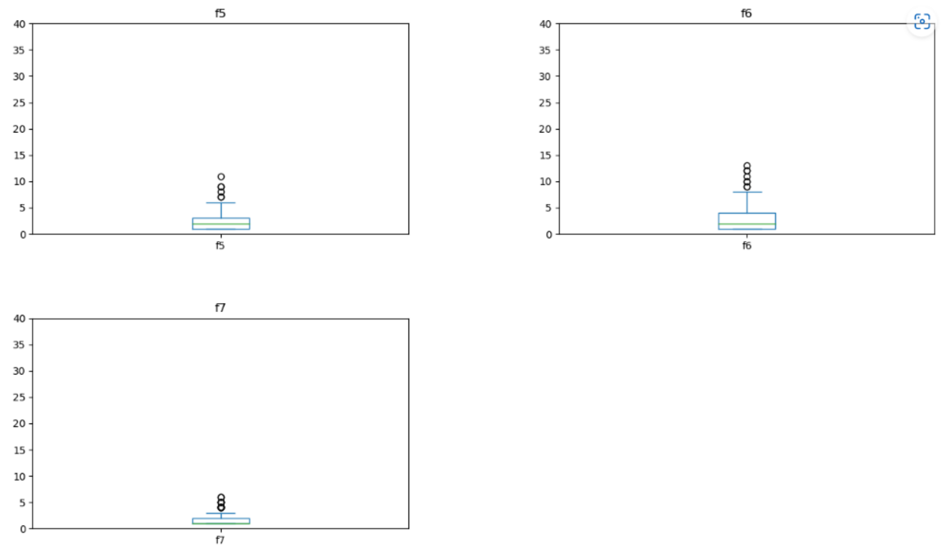


Fig: Boxplots of f5,f6,f7

3. Heatmap

Using heatmaps we can find if the various features are correlated. If a couple of input features are highly correlated, either of those can be dropped to reduce the complexity and get a better accuracy.

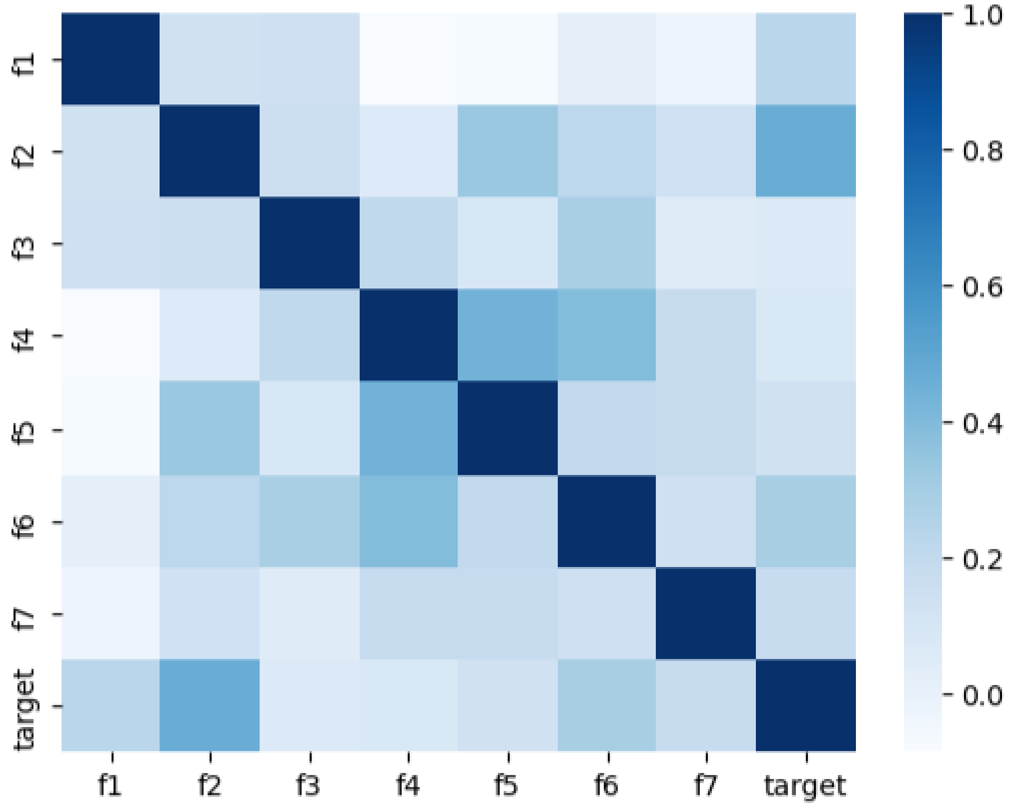


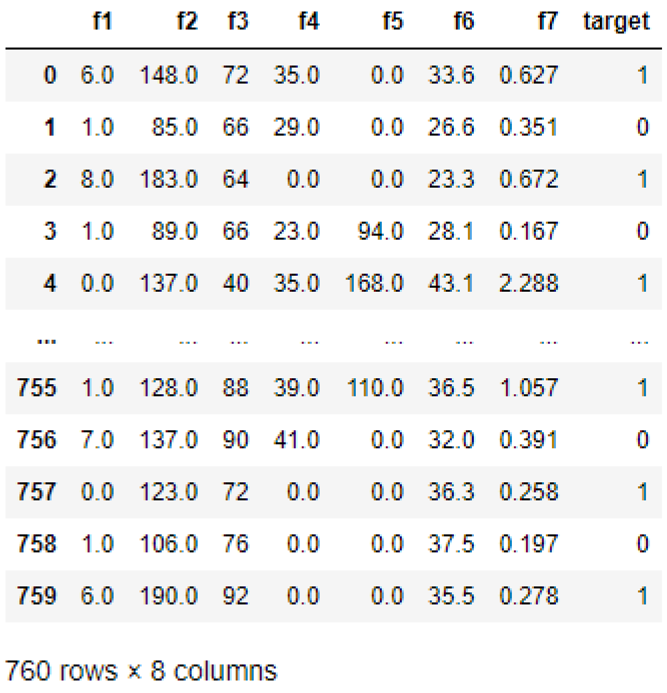
Fig: Heatmap

Observation: From the above heatmap, we can infer that no two features are highly correlated to each other. Hence, during the data preprocessing phase, we do not filter out any of the features.

3. DATA PREPROCESSING

1. Removing Non-numeric values

The dataset consisted of non-numeric values. In the 1st step of preprocessing, we remove these values, and the dataset consists of 760 rows after removing the non-numeric values.



	f1	f2	f3	f4	f5	f6	f7	target
0	6.0	148.0	72	35.0	0.0	33.6	0.627	1
1	1.0	85.0	66	29.0	0.0	26.6	0.351	0
2	8.0	183.0	64	0.0	0.0	23.3	0.672	1
3	1.0	89.0	66	23.0	94.0	28.1	0.167	0
4	0.0	137.0	40	35.0	168.0	43.1	2.288	1
...	...	...	...	...	...	...	...	...
755	1.0	128.0	88	39.0	110.0	36.5	1.057	1
756	7.0	137.0	90	41.0	0.0	32.0	0.391	0
757	0.0	123.0	72	0.0	0.0	36.3	0.258	1
758	1.0	106.0	76	0.0	0.0	37.5	0.197	0
759	6.0	190.0	92	0.0	0.0	35.5	0.278	1

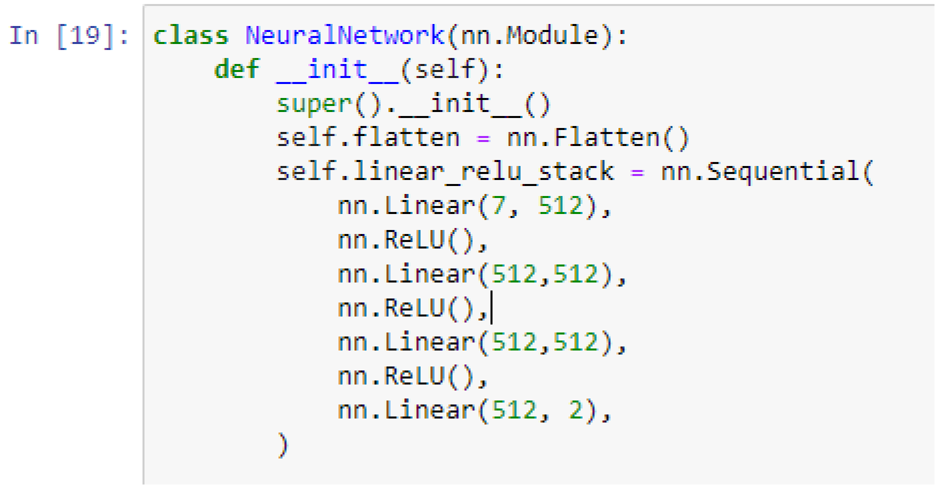
760 rows × 8 columns

Fig: Dataset after removing non-numeric values

2. SCALING

Before we train the model, it is important that the data must be scaled in order to avoid the dominance of one feature over the other. Hence, by using the "**standarScaler()**" method from the "**sklearn**" library we scale the data.

4. ARCHITECTURE STRUCTURE OF THE NN



```
In [19]: class NeuralNetwork(nn.Module):
             def __init__(self):
                 super().__init__()
                 self.flatten = nn.Flatten()
                 self.linear_relu_stack = nn.Sequential(
                     nn.Linear(7, 512),
                     nn.ReLU(),
                     nn.Linear(512,512),
                     nn.ReLU(),
                     nn.Linear(512,512),
                     nn.ReLU(),
                     nn.Linear(512, 2),
                 )
```

Fig: Code snippet defining the architecture of the NN

The neural network architecture consists of **3 hidden layers**, each of which in our base model is followed by a **ReLU activation function**. The neural network consists of **4 fully connected layers**. The input consists of 7 neurons as the number of features in our dataset after preprocessing is 7. The output layer has 2 neurons and the target variable in our dataset consists of either 0's or 1's. The input neurons are flattened suing the **nn.Flatten()** function. Out of the 4 fully connected layers, 3 layers have 512 neurons each. The output size of the network we described is 2. As we did not specify the activation function to be used after the output layer, by default "linear activation function" will be used.

5. GRAPHS

5.1 TRAINING AND TESTING LOSS

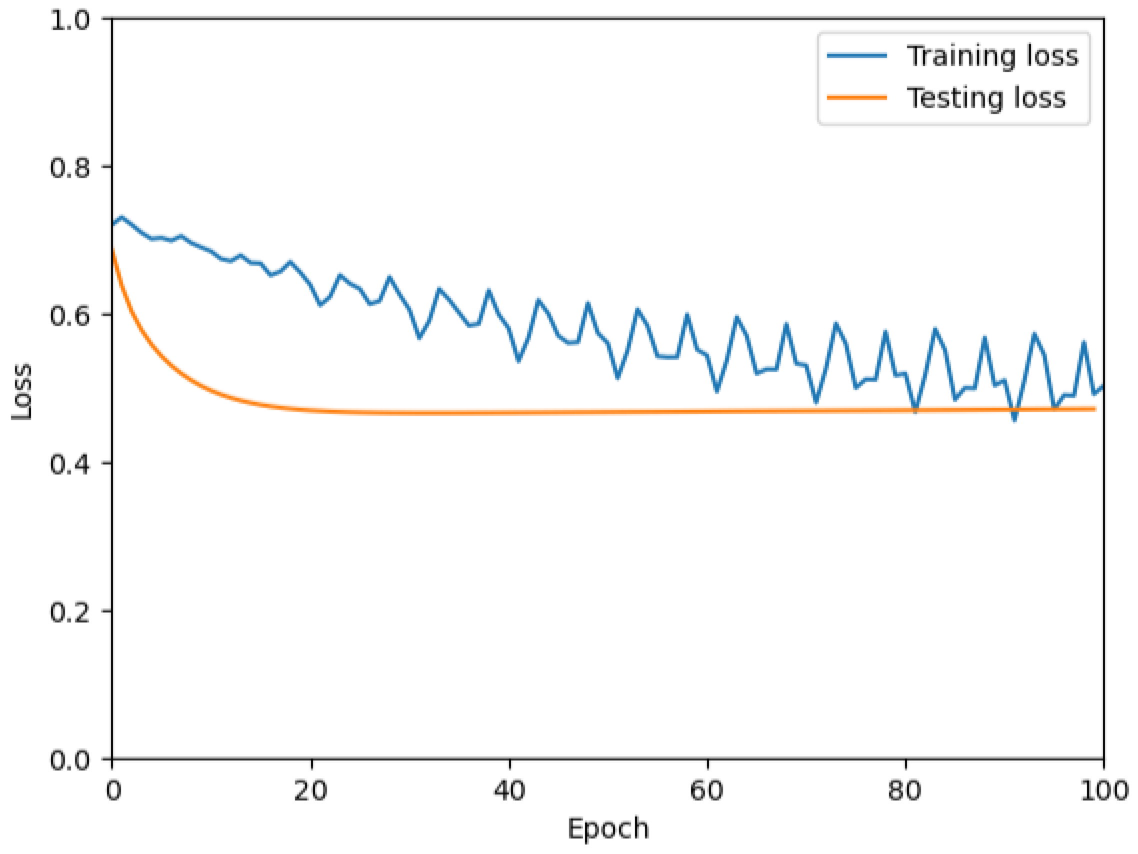


Fig: Training and Testing loss for 100 epochs

Observation: It is observed that, that both training and testing loss reduced over time which means that the model is eventually learning to make better predictions. Over the 100 epochs, the highest training loss in recorded is 0.7043 and after a certain number of epochs, the loss remained constant (0.44). For the testing, we considered the average loss and from the graph above it is evident that the loss reduced kept declining over time. The highest average loss obtained is 0.67 whereas at the end of 100 epochs, the lowest obtained is 0.47.

5.2 TRAINING AND TESTING ACCURACY

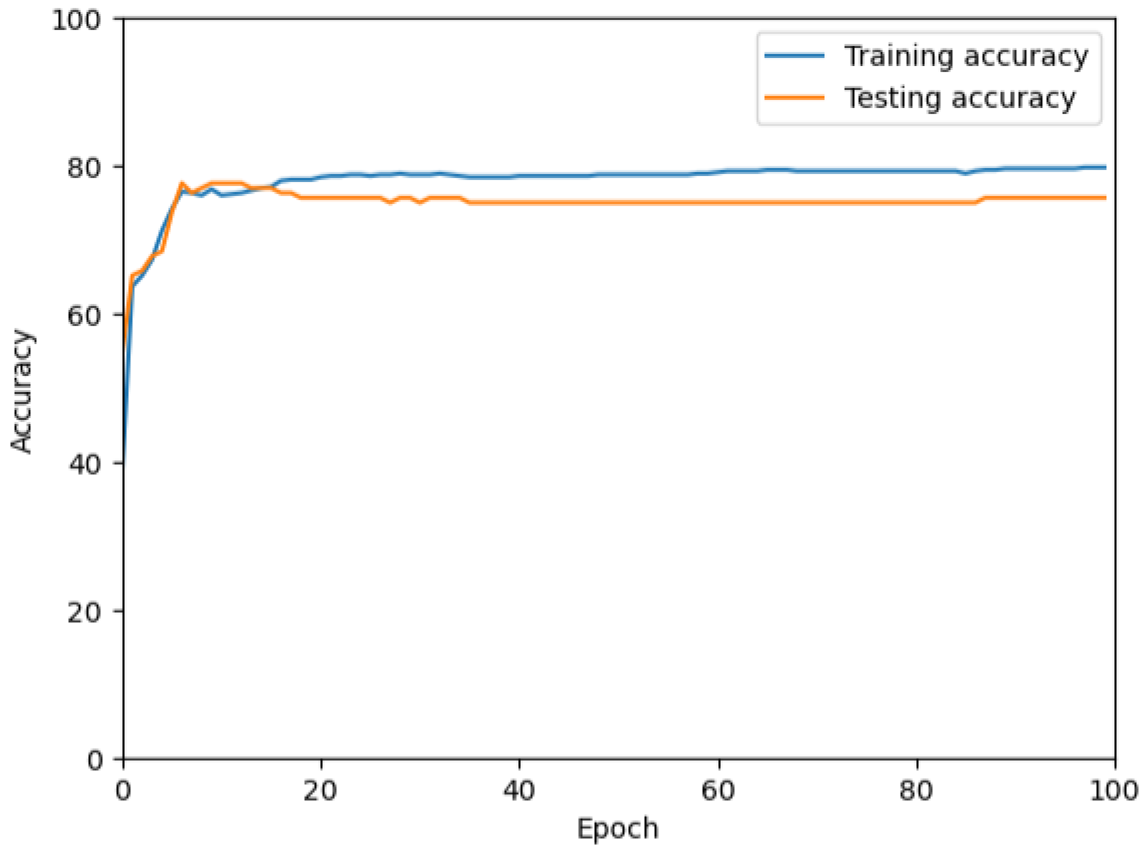


Fig: Training and Testing accuracy reported over 100 epochs

Observation: It is observed that the training and testing accuracy of the model kept increasing over time. After 20 epochs, both the training and testing accuracy remained almost constant with a very minute variation. At the end of 100 epochs, the testing accuracy achieved is 75.7%.

PART II

(1) Table of all the setups

SETUP - I

1) Changing the dropout value in the NN and keeping all the other hyperparameters constant.

Hyperparameter: Dropout	**Test Accuracy**
0.1	**74.3%**
0.2	**72.4%**
0.3	**71.7%**

Observation: From the above table, we can conclude the highest accuracy achieved is 74.3% when the dropout is 0.1 and the rest of the hyperparameters (Optimizer, activation function, initializer) are constant.

SETUP - II

2) Changing the Activation function in the NN and keeping all the hyper1parameters constant.

Hyperparameter: Activation Function	**Test Accuracy**
Tanh	**73.7%**
LeakyReLU	**75.7%**
PReLU	**75.0%**

Observation: From the above table, it can be inferred that the test accuracy achieved using LeakyReLU as the activation function is the highest.

SETUP - III

3) Changing the Optimizer in the NN and keeping all the hyperparameters constant.

Hyperparameter: Optimizer	**Test Accuracy**
Adam Optimizer	**64.5%**
Averaged Stochastic Gradient Descent (ASGD)	**67.8%**
Rprop Optimizer	**75.0%**

Observation: The highest accuracy achieved by varying the Optimizer in the neural network and keeping all the other hyperparameters constant is 75.0% for the Rprop Optimizer.

SETUP - IV

4) Changing the Initializer in the NN and keeping all the hyperparameters constant.

Hyperparameter: Initializer	**Test Accuracy**
Xavier Initializer	**65.8%**
Kaiming Uniform Initializer	**53.9%**
Orthogonal Initializer	**37.5%**

Observation: It is observed that the model achieved a better accuracy by using default initializer. Out of the above models, Xavier Initializer achieved the highest accuracy.

(2), (3)

Comparing Training and Testing Accuracy on the same plot for all the above setups.

1) Training and Testing plots obtained for setup-I (Dropout)

- **Dropout = 0.1**

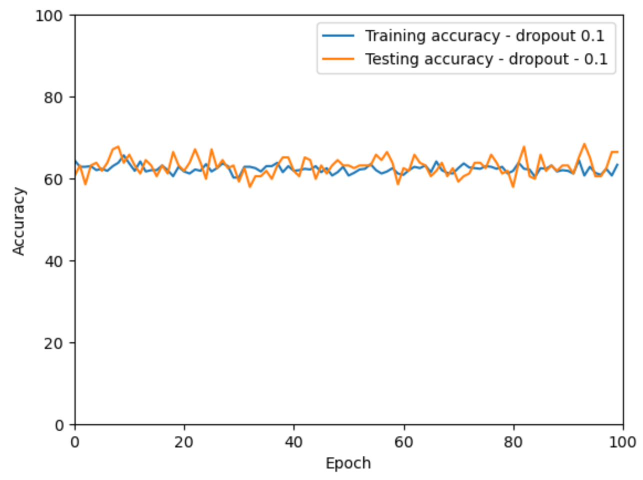


Fig: Training and Testing accuracy – Dropout = 0.1

From the above plot, we can conclude that both the training and testing accuracies have been fluctuating over the entire range of epochs. At the end of 100 epochs, the testing accuracy achieved is 74.3%

Dropout = 0.2

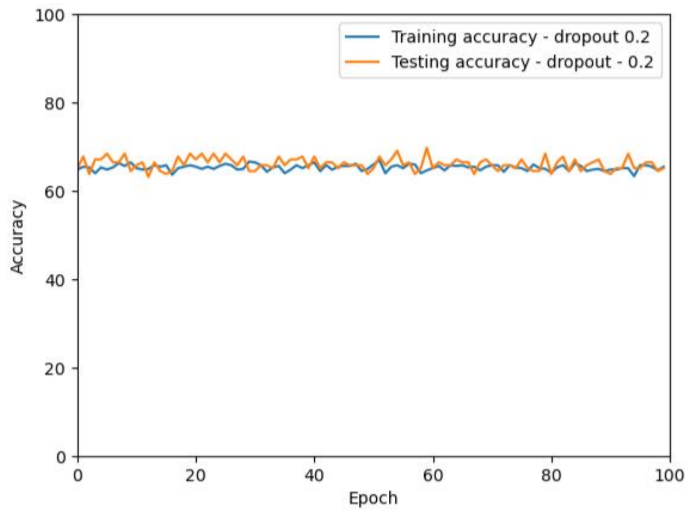


Fig: Training and Testing accuracy – Dropout = 0.2

From the above graph, it is conclusive that the testing accuracy achieved by the model when the dropout is set at 0.2 is 72.4% which is less than the accuracy achieved when the dropout was set to 0.1.

- **Dropout = 0.3**

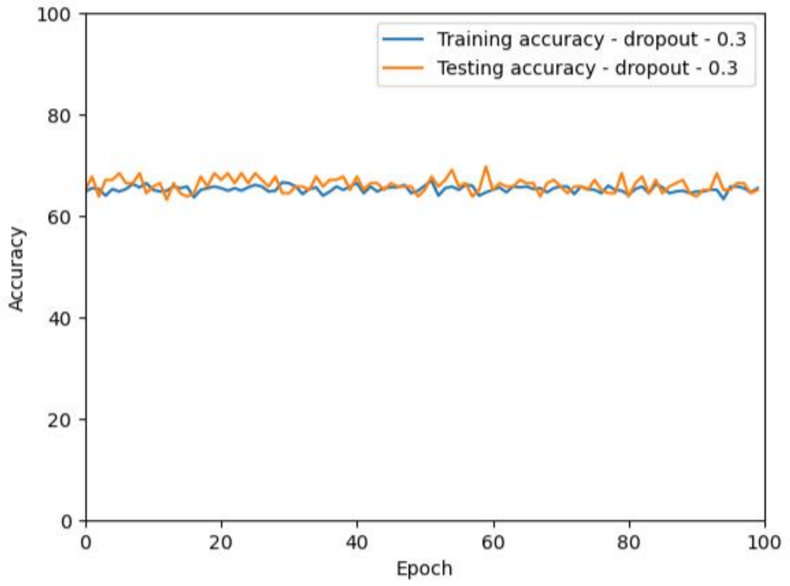


Fig: Training and Testing accuracy – Dropout = 0.3

The testing accuracy achieved when the dropout value is set to 0.3 while the rest of the hyper parameters like activation function, Optimizer, Initializer are kept constant is 71.7%.

Out of all the 3 setups of the dropout, the model that achieved the highest accuracy had a dropout value of 0.1.

2) Training and Testing plots obtained for setup-II (activation functions)

- **Activation function: Tanh**.

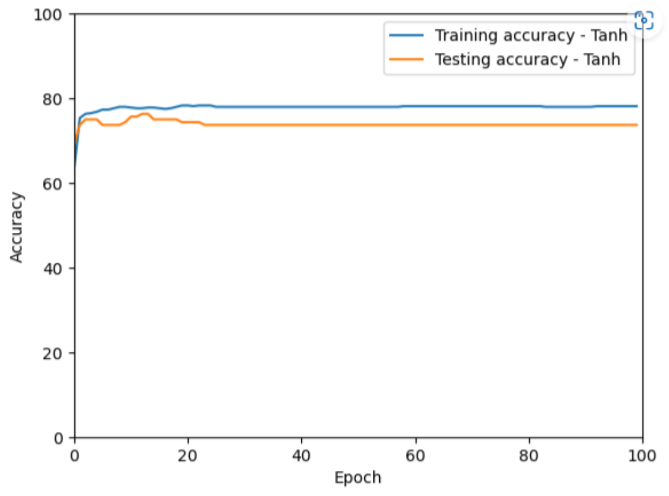


Fig: Training and Testing accuracy – Tanh

The testing accuracy achieved using the Tanh activation function is 73.7%. It is observed that the training accuracy is higher than the testing accuracy.

- **Activation function: LeakyReLU**.

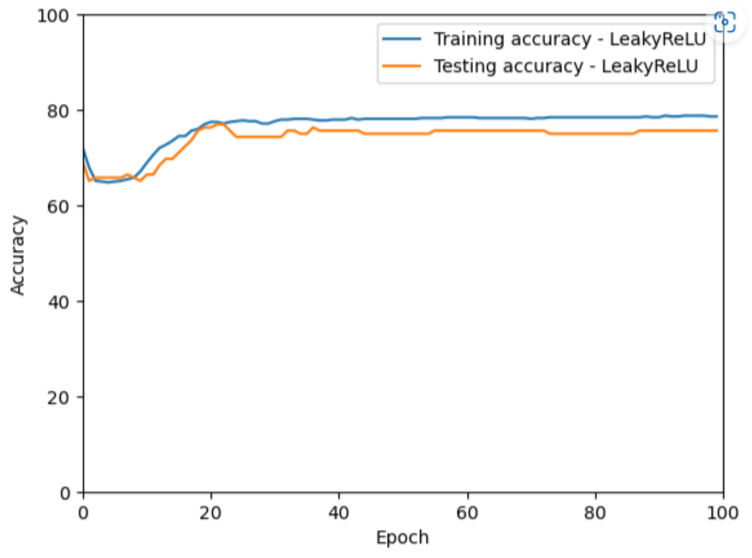


Fig: Training and Testing accuracy – LeakyReLU

The testing accuracy achieved using the LeakyReLU activation function is 75.7%. It is observed both training and testing accuracies reduced during the initial 20 epochs and later increased.

- **Activation function: PreLU**.

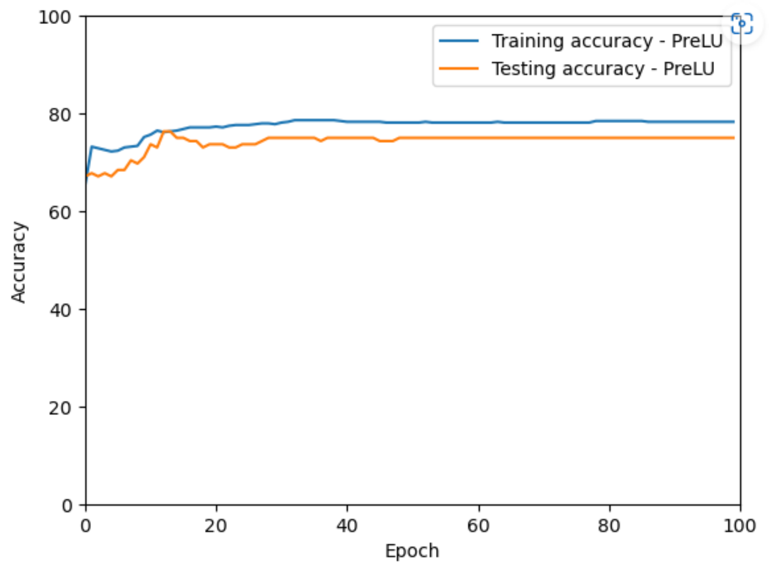


Fig: Training and Testing accuracy – PreLU

The testing accuracy achieved using the PreLU activation function is 75.0%. It is observed that the training accuracy is higher than the testing accuracy and the testing accuracy increased in between 10-20 epochs and later decreased and remained stable.

Out of all the 3 setups of the activation function, the model that achieved the highest accuracy had a LeakyReLU as its activation function.

3) Training and Testing plots obtained for setup-III (Optimizer)

- **Optimizer: Adam**

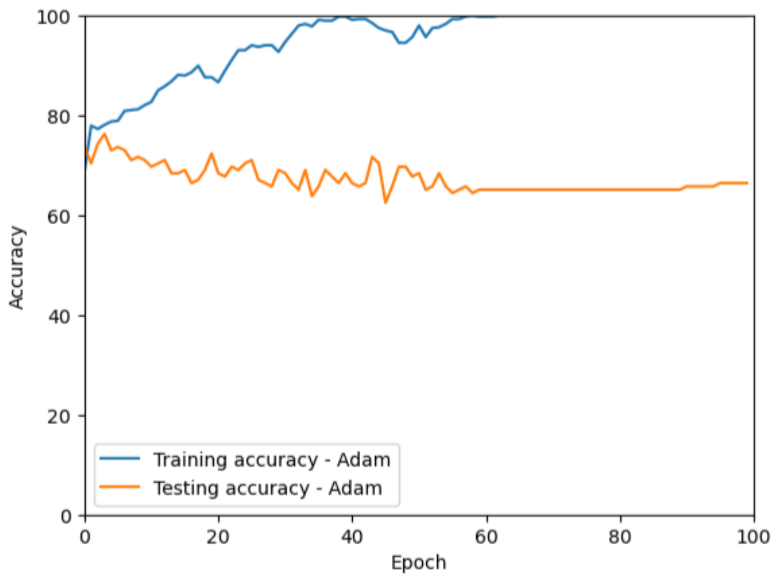


Fig: Training and Testing accuracy – Adam Optimizer

From the above graph, we can conclude that the training accuracy is 100% and over the course of 100 epochs, the testing accuracy decreased and that means the model overfit the data.

- **Optimizer: ASGD (Averaged Stochastic Gradient Descent)**

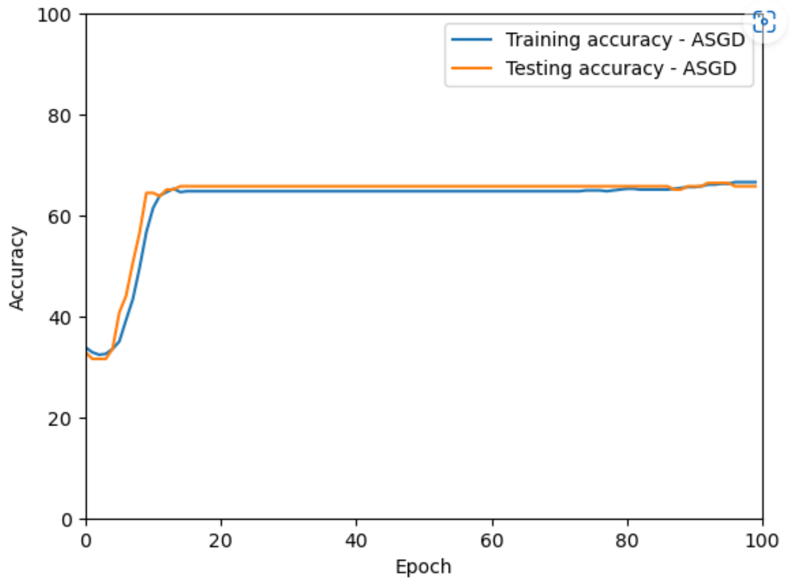


Fig: Training and Testing accuracy – ASGD Optimizer

From the above graph, it can be concluded that both the training and testing accuracy increased in a similar manner over 100 epochs. The testing accuracy achieved is 68.7%.

- **Optimizer: Rprop**

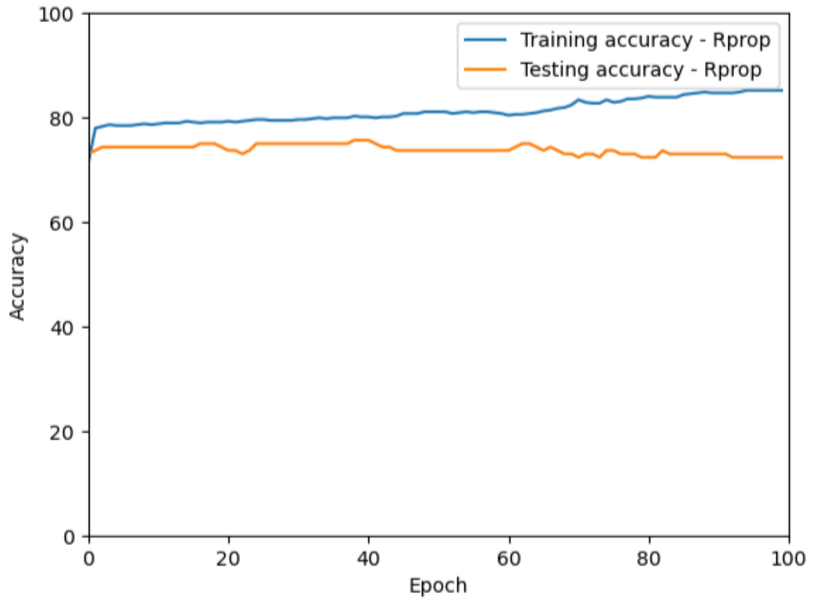


Fig: Training and Testing accuracy – Rprop Optimizer

Using the **Rprop optimizer**, the training accuracy increased over time and the testing accuracy decreased as compared to training accuracy. The testing accuracy archived is 75.0% which is the highest of the 3 optimizers used.

Out of all the 3 setups of the optimizer, the model that achieved the highest accuracy had the Rprop as the optimizer.

4) Training and Testing plots obtained for setup-IV (Initializer)

- **Xavier Initializer**

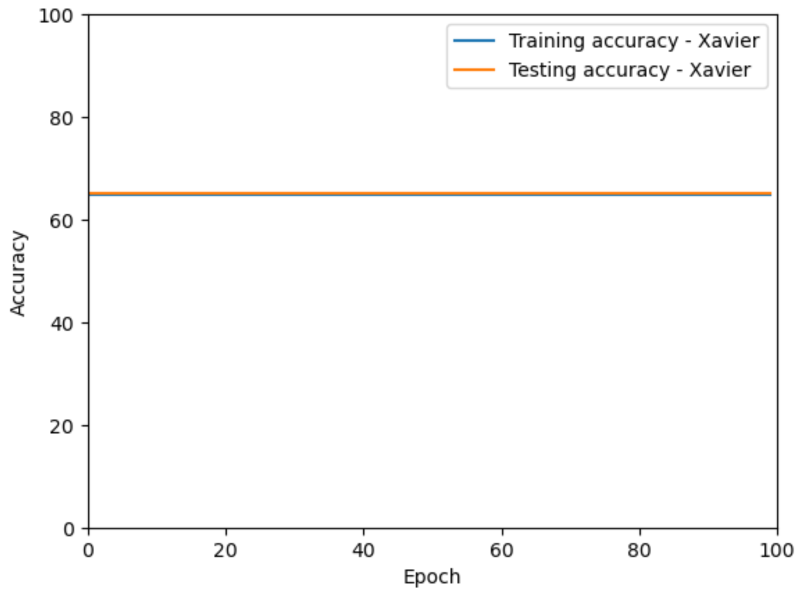


Fig: Training and Testing accuracy – Xavier Initializer

From the above plot, we can infer that both the training and testing accuracies remained constant throughout the 100 epochs. The accuracy achieved using this initializer is 65.8%.

- **Kaiming Uniform Initializer**

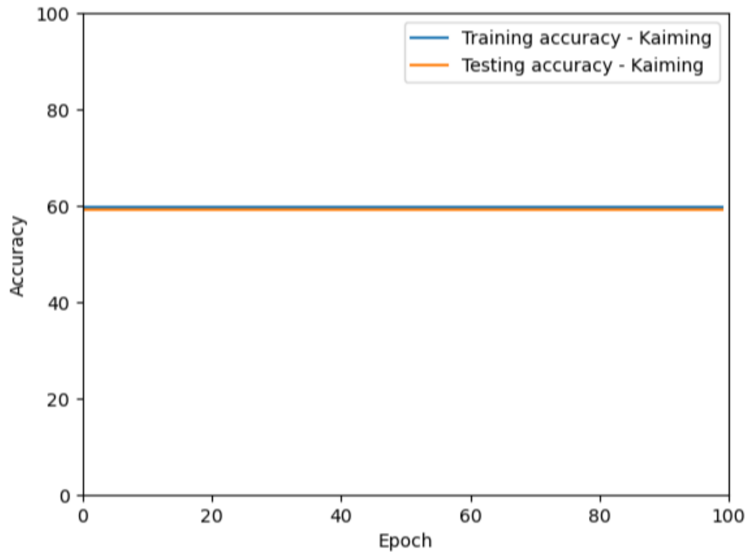


Fig: Training and Testing accuracy – Kaiming Uniform Initializer

Like the previous plot, a similar conclusion can be drawn to this plot as well. Both the training and testing accuracy remained constant throughout the 100 epochs. The accuracy achieved using this initializer is 53.9%.

- **Orthogonal Initializer**

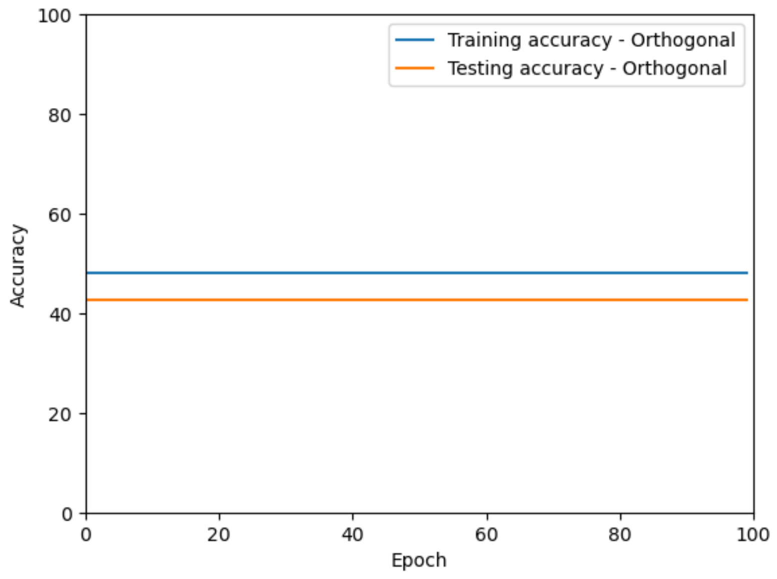


Fig: Training and Testing accuracy – Orthogonal Initializer

A similar conclusion can be made from this graph as well. Overall, it can be concluded that when the hyperparameters like activation function, optimizer, and dropout are kept constant to see the effect of weight initialization methods on the accuracy of the model and it is observed that, both training and testing accuracy remained constant throughout the 100 epochs.

Out of all the 3 setups of the Initializer, the model that achieved the highest accuracy had Xavier Initializer as the weights initializer in the model.

(4)

Methods used to improve the Testing accuracy

As an attempt to improve the accuracy of the base model, we have implemented the following techniques on our base model and to understand the variation, we have plotted graphs which depicts the testing accuracies of the base model and the base model's accuracy on application of a certain technique.

1) EARLY STOPPING

This is a technique in which the validation data set is monitored while running the model if the loss is increasing overtime, the training of the model is stopped to avoid the problem of overfitting.

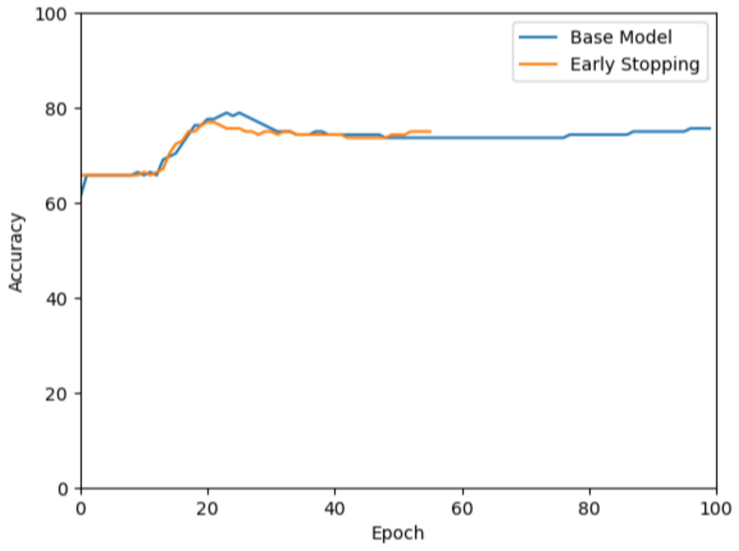


Fig: Testing accuracy comparison of Base model and the base model with the early stopping technique

Observation: It is observed that the model stopped training the validation (testing set) after 60 epochs. The accuracy achieved using the early stopping technique is 67.1%. Our model performed or achieved a better accuracy without this technique.

2) BATCH NORMALIZATION

In this technique, inputs of each layer are normalized in small batches by subtracting the mean and dividing the variance of each batch and later scaling them by learnable parameters.

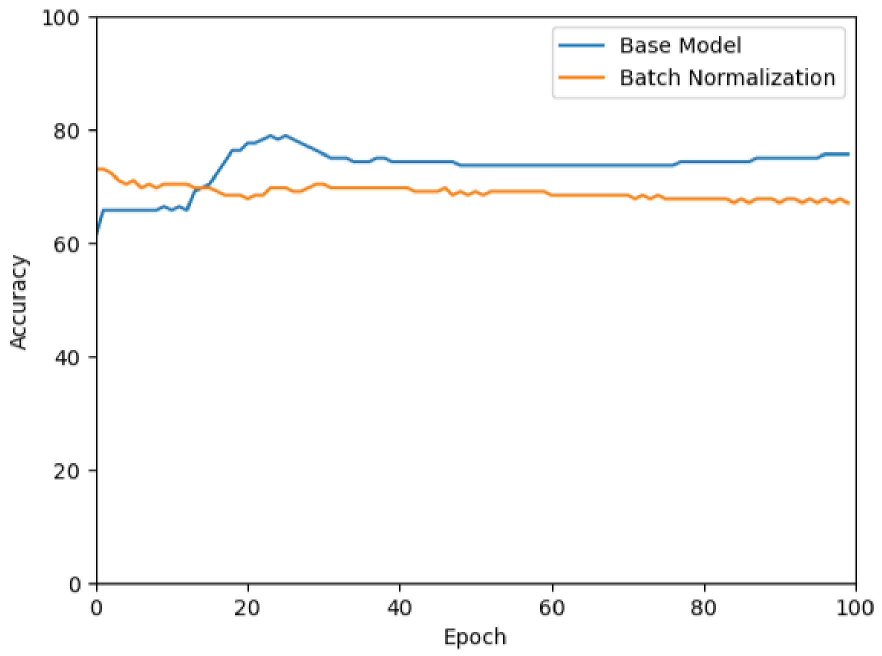


Fig: Testing accuracy comparison of Base model and the base model with the Batch Normalization Technique.

Observation: The above plot is a validation accuracy comparison between the base model and the batch normalization technique is implemented on the base model. The model achieved an accuracy of 67.1% after implementing this technique and the loss also increased over time.

3) WEIGHT DECAY

This is technique used to handle the problem of overfitting. This is done by adding a penalty term to the loss function during the training.

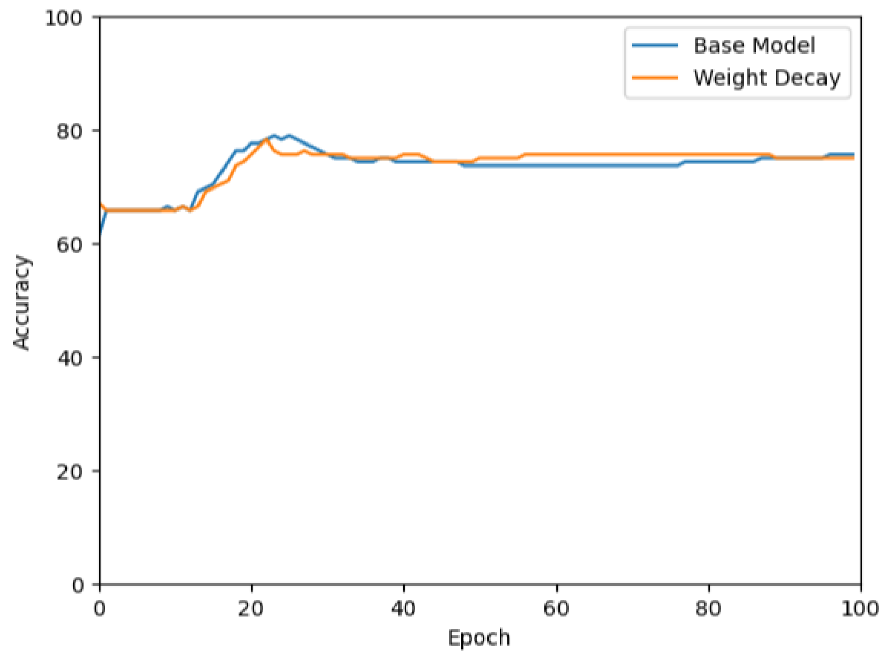


Fig: Testing accuracy comparison of Base model and the base model with the Weight Decay Technique.

Observation: The model achieved a testing accuracy of 75.0% on applying this technique on our base model. Unlike other techniques, this model did not reduce the accuracy rather had almost the same accuracy as the base model.

4) LEARNING RATE SCHEDULER

This is the technique in which the learning rate of the model is decreased over time to avoid overfitting the data to the model.

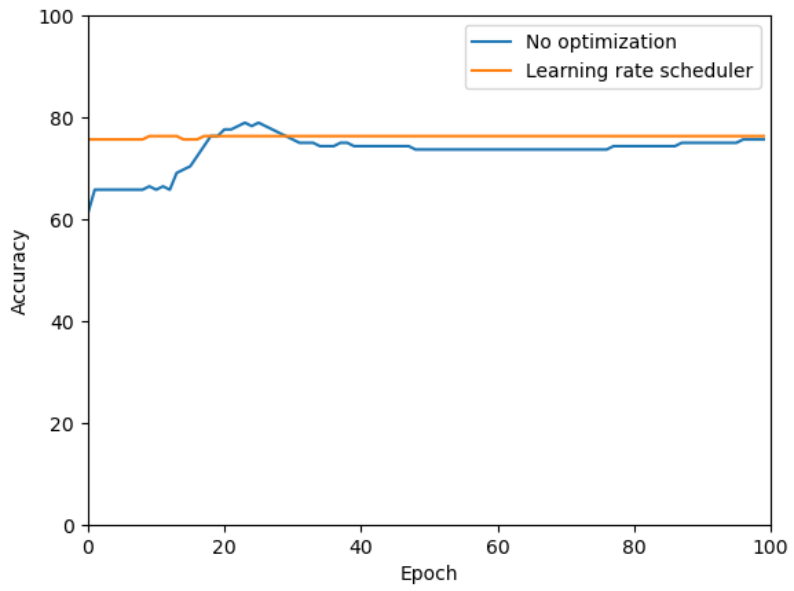


Fig: Testing accuracy comparison of Base model and the base model with the Learning rate scheduler Technique.

Observation: Out of all the techniques, for our dataset, this technique achieved the highest accuracy. By implementing this technique on our base model, we achieved an accuracy of 76.3%.

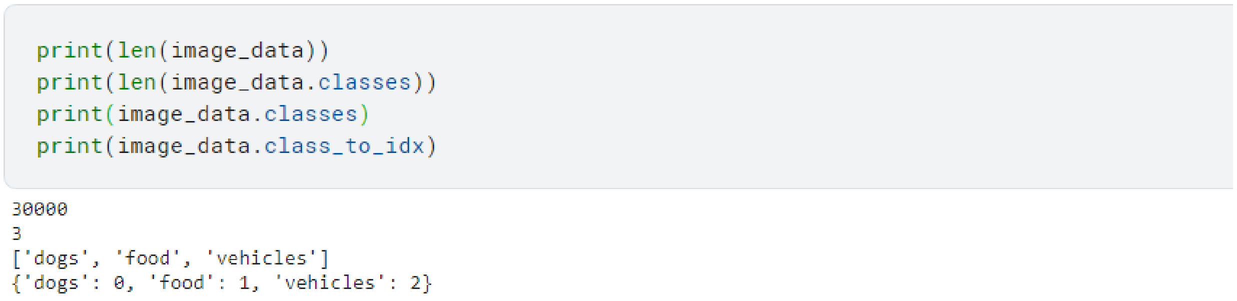
Part III

Implementing and Improving AlexNet

(1)

Nature of the Dataset

The given CNN dataset comprises a number of images stored in various file directories based on their class. We are encountering an image type of data where image is the input and we get a label as the output for that image. The following is the code to look at the main statistics of the data.



```
print(len(image_data))
print(len(image_data.classes))
print(image_data.classes)
print(image_data.class_to_idx)
```

```
30000
3
['dogs', 'food', 'vehicles']
{'dogs': 0, 'food': 1, 'vehicles': 2}
```

‘len(image_data)’ gives the number of entries in the complete dataset. There are **30000 images** in the provided CNN dataset.

‘len(image_data.classes)’ gives the number of labels or classes in the dataset. There are **3 labels** under which all the 30000 images are categorized.

‘Image_data.classes’ gives the names for the 3 existing labels or classes. **‘dogs’, ‘food’, ‘vehicles’** are the label names assigned for the image entries in the dataset.

‘image_data.class_to_idx’ gives the **indices** for all the existing labels.

There are 10000 images in each category. In total, there are 30000 images.

Visualizations

1) Images in the dataset:

Fig: Random images in the dataset

To get an idea about the different classes of images in the dataset, we did the above plot to visually see the data and proceed further with the preprocessing steps, training and building the model.

2) Dataset Images after normalizing:

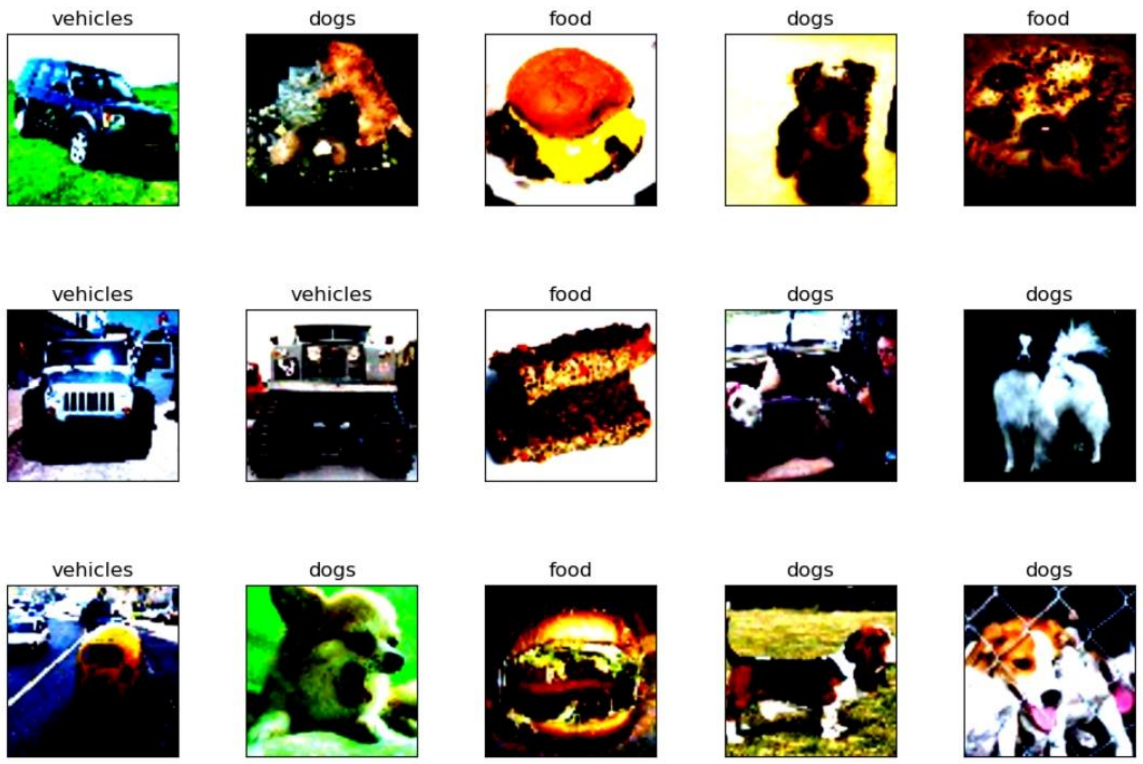


Fig: Random images in the dataset after performing normalization

Before sending the images to the AlexNet, we have performed few preprocessing steps and one such step is normalizing the images. Here's a figure that shows random images in the dataset after performing normalization

3) Histogram

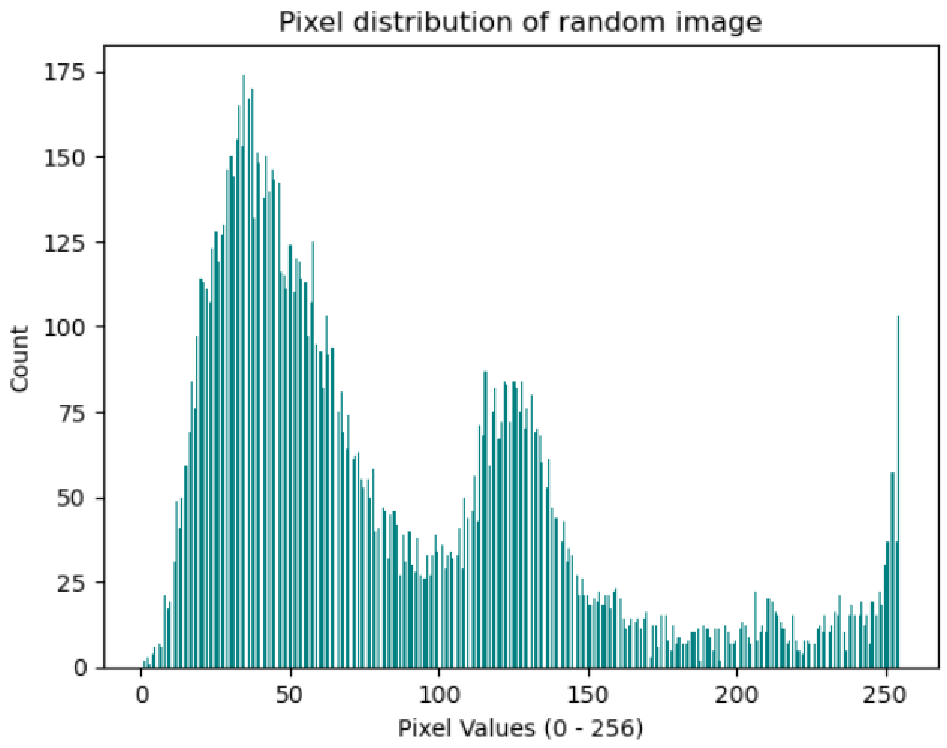


Fig: Histogram plot for a random image in the dataset

Histogram plot of a image helps us in understanding various features of the image like the overall contrast and brightness of the image. The above histogram is a plot of a random image from the dog dataset and we can infer that in the particular image, most have the pixel value in between 0-50.

4) PIE chart

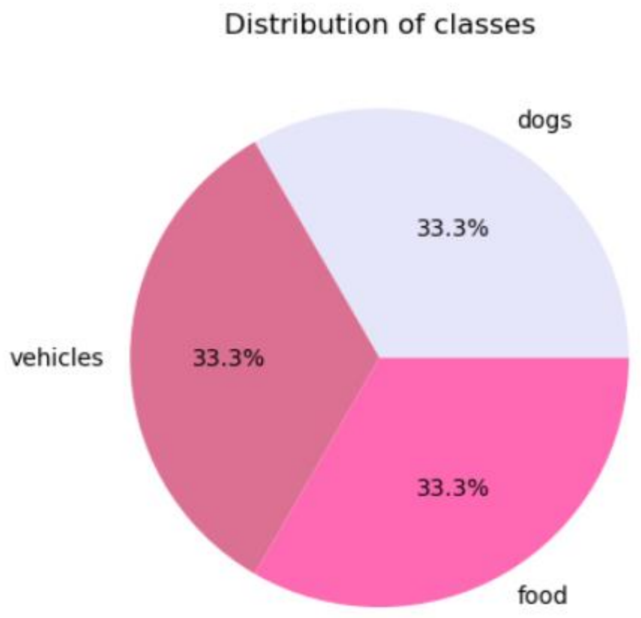


Fig: Pie chart depicting the distribution of each class in the given dataset

From the pie chart above, we can infer that there are 3 classes in the dataset namely: dogs, vehicles, and food. Each of class has equal share (each class forms 33.3% of the entire dataset). The classes are well balanced in this case and during the preprocessing stage we need perform class balancing to achieve a higher accuracy as the classes are already balance.

(2)

Initial AlexNet CNN architecture

```
: model = Sequential([
      Conv2D(96, (11, 11), strides=(4, 4), activation='relu', input_shape=(224, 224, 3)),
      MaxPooling2D(pool_size=(3, 3), strides=(2, 2)),
      Conv2D(256, (5, 5), strides=(1, 1), activation='relu', padding='same'),
      MaxPooling2D(pool_size=(3, 3), strides=(2, 2)),
      Conv2D(384, (3, 3), strides=(1, 1), activation='relu', padding='same'),
      Conv2D(384, (3, 3), strides=(1, 1), activation='relu', padding='same'),
      Conv2D(256, (3, 3), strides=(1, 1), activation='relu', padding='same'),
      MaxPooling2D(pool_size=(3, 3), strides=(2, 2)),
      Flatten(),
      Dense(4096, activation='relu'),
      Dropout(0.5),
      Dense(4096, activation='relu'),
      Dropout(0.5),
      Dense(3, activation='softmax')
  ])

  # Compile the model
  model.compile(loss='sparse_categorical_crossentropy',
                optimizer='adam',
                metrics=['accuracy'])

  # Train the model
  model.fit(x_train, y_train, epochs=10, batch_size=64, validation_data=(x_test, y_test))
  test_loss, test_acc = model.evaluate(x_test,y_test)
  print('Test accuracy:', test_acc)
```

```
375/375 [==============================] - 20s 24ms/step - loss: 1.0972 - accuracy: 0.3912 - val_loss: 1.0937 - val_accuracy: 0.4560
Epoch 2/25
375/375 [==============================] - 7s 19ms/step - loss: 1.0763 - accuracy: 0.5955 - val_loss: 1.0066 - val_accuracy: 0.5875
Epoch 3/25
375/375 [==============================] - 7s 19ms/step - loss: 0.8650 - accuracy: 0.6158 - val_loss: 0.8005 - val_accuracy: 0.6345
Epoch 4/25
375/375 [==============================] - 7s 19ms/step - loss: 0.7286 - accuracy: 0.6849 - val_loss: 0.6595 - val_accuracy: 0.7223
Epoch 5/25
375/375 [==============================] - 8s 20ms/step - loss: 0.6373 - accuracy: 0.7319 - val_loss: 0.5564 - val_accuracy: 0.7752
Epoch 6/25
375/375 [==============================] - 7s 20ms/step - loss: 0.5719 - accuracy: 0.7645 - val_loss: 0.5575 - val_accuracy: 0.7622
Epoch 7/25
375/375 [==============================] - 7s 20ms/step - loss: 0.5250 - accuracy: 0.7908 - val_loss: 0.5958 - val_accuracy: 0.7580
Epoch 8/25
375/375 [==============================] - 7s 20ms/step - loss: 0.4841 - accuracy: 0.8076 - val_loss: 0.4610 - val_accuracy: 0.8165
Epoch 9/25
375/375 [==============================] - 7s 20ms/step - loss: 0.4524 - accuracy: 0.8225 - val_loss: 0.4493 - val_accuracy: 0.8252
Epoch 10/25
375/375 [==============================] - 8s 20ms/step - loss: 0.4348 - accuracy: 0.8320 - val_loss: 0.4176 - val_accuracy: 0.8395
Epoch 11/25
375/375 [==============================] - 8s 20ms/step - loss: 0.4125 - accuracy: 0.8423 - val_loss: 0.4045 - val_accuracy: 0.8420
Epoch 12/25
375/375 [==============================] - 8s 20ms/step - loss: 0.3964 - accuracy: 0.8465 - val_loss: 0.3936 - val_accuracy: 0.8492
Epoch 13/25
375/375 [==============================] - 8s 21ms/step - loss: 0.3784 - accuracy: 0.8534 - val_loss: 0.3950 - val_accuracy: 0.8510
Epoch 14/25
375/375 [==============================] - 8s 21ms/step - loss: 0.3655 - accuracy: 0.8597 - val_loss: 0.3581 - val_accuracy: 0.8600
Epoch 15/25
375/375 [==============================] - 8s 21ms/step - loss: 0.3441 - accuracy: 0.8695 - val_loss: 0.4148 - val_accuracy: 0.8442
Epoch 16/25
375/375 [==============================] - 8s 21ms/step - loss: 0.3290 - accuracy: 0.8738 - val_loss: 0.4262 - val_accuracy: 0.8323
Epoch 17/25
375/375 [==============================] - 8s 21ms/step - loss: 0.3112 - accuracy: 0.8827 - val_loss: 0.4106 - val_accuracy: 0.8410
Epoch 18/25
375/375 [==============================] - 8s 21ms/step - loss: 0.2933 - accuracy: 0.8897 - val_loss: 0.3461 - val_accuracy: 0.8697
Epoch 19/25
375/375 [==============================] - 8s 20ms/step - loss: 0.2719 - accuracy: 0.8976 - val_loss: 0.3923 - val_accuracy: 0.8560
Epoch 20/25
375/375 [==============================] - 8s 21ms/step - loss: 0.2559 - accuracy: 0.9029 - val_loss: 0.5552 - val_accuracy: 0.8182
Epoch 21/25
375/375 [==============================] - 8s 21ms/step - loss: 0.2337 - accuracy: 0.9121 - val_loss: 0.3412 - val_accuracy: 0.8748
Epoch 22/25
375/375 [==============================] - 8s 21ms/step - loss: 0.2122 - accuracy: 0.9225 - val_loss: 0.3659 - val_accuracy: 0.8747
Epoch 23/25
375/375 [==============================] - 8s 20ms/step - loss: 0.1885 - accuracy: 0.9296 - val_loss: 0.3995 - val_accuracy: 0.8643
Epoch 24/25
375/375 [==============================] - 8s 20ms/step - loss: 0.1595 - accuracy: 0.9422 - val_loss: 0.3909 - val_accuracy: 0.8635
Epoch 25/25
375/375 [==============================] - 8s 20ms/step - loss: 0.1417 - accuracy: 0.9496 - val_loss: 0.5167 - val_accuracy: 0.8487
188/188 [==============================] - 2s 7ms/step - loss: 0.5167 - accuracy: 0.8487
Test Accuracy: 0.8486666679382324
```

Training and Testing plots:

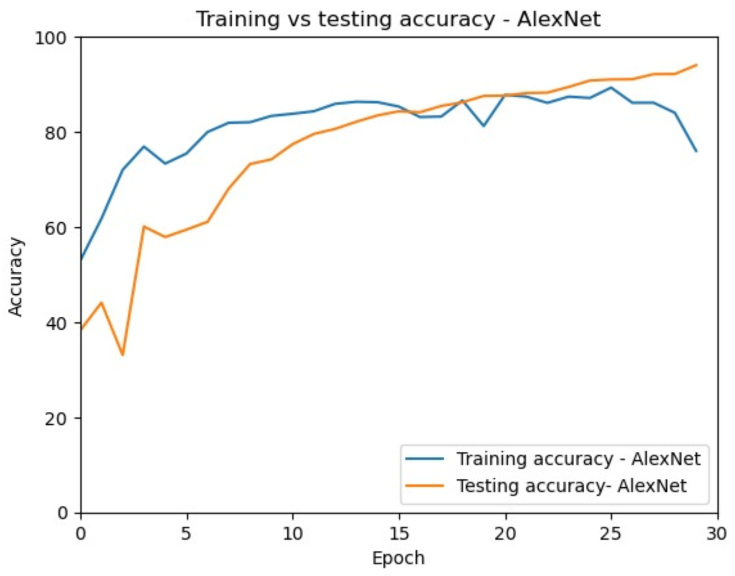


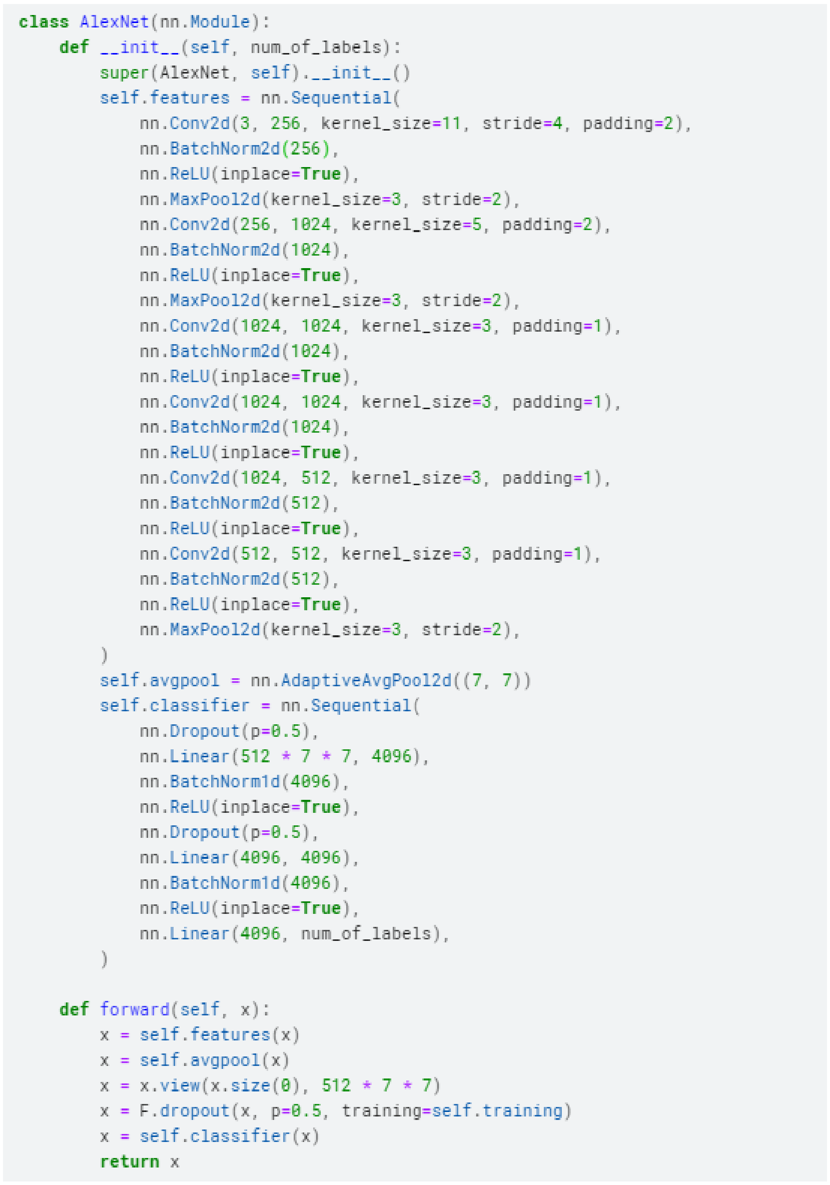
Fig: Training and Testing accuracy of AlexNet (initial)

This basic version of the CNN architecture yields an accuracy of not more than 85%. The third part in this report has the improvised version of the architecture which gives an accuracy of more than 90%. Below is the snippet of output consisting of training, test accuracies and loss values for epoch.

(3)

AlexNet Architecture Implementation

Below is the implementation of Improvised AlexNet CNN configuration. This model yielded better results with testing accuracies greater than 90%. And for the basic AlexNet CNN model, the testing accuracies ranged from 75% - 80%.



```
class AlexNet(nn.Module):
    def __init__(self, num_of_labels):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 256, kernel_size=11, stride=4, padding=2),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(256, 1024, kernel_size=5, padding=2),
            nn.BatchNorm2d(1024),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(1024, 1024, kernel_size=3, padding=1),
            nn.BatchNorm2d(1024),
            nn.ReLU(inplace=True),
            nn.Conv2d(1024, 1024, kernel_size=3, padding=1),
            nn.BatchNorm2d(1024),
            nn.ReLU(inplace=True),
            nn.Conv2d(1024, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
        self.classifier = nn.Sequential(
            nn.Dropout(p=0.5),
            nn.Linear(512 * 7 * 7, 4096),
            nn.BatchNorm1d(4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(4096, 4096),
            nn.BatchNorm1d(4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_of_labels),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), 512 * 7 * 7)
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.classifier(x)
        return x
```

This improvised version of AlexNet CNN architecture has the following layers:

- There are 6 **2D convolution layers**. The first convolution layer takes in 3 input channels, gives out 256 output channels with a kernel length of 11, stride of 4 and padding of 2. These are followed by batch normalization and ReLU layers. The second convolution layer takes in 256 input channels, gives out 1024 output channels with a kernel length of 5, and padding of 2. The next three convolution layers take in 1024 input channels, and give out 512 output channels in the fifth convolution layer with a kernel length of 3, and padding of 1. The last convolution layer takes in 512 input channels and gives out 512 output channels with a kernel size of 3 and stride 2.
- This version has **max pooling layers** after the first and second 2D convolutional layers with a kernel length of 3 and strides of 2. And there is one more max pool layer after the last convolution layer which has a kernel size of 3 and stride ‘2’.
- This implementation makes use of adaptive average pooling layers. These layers help in changing the dimensions that are spatial to the fixed dimensions. In our case, the fixed size is 7 * 7. This is to aid in serving inputs to the fully connected layers.
- There are three **fully connected layers** out of which the first two have 4096 output channels each. These two layers are followed by batch normalization layers and ReLU activation functions. In addition, we added a dropout layer with a dropout probability of 0.5. The last layer would have output units equal to the number of labels or classes that are present in the dataset.
- In addition, we used **dropout layers** with a dropout probability of 0.5 and **batch normalization layers**.

Testing and Training Accuracies with Graphs (after improving):

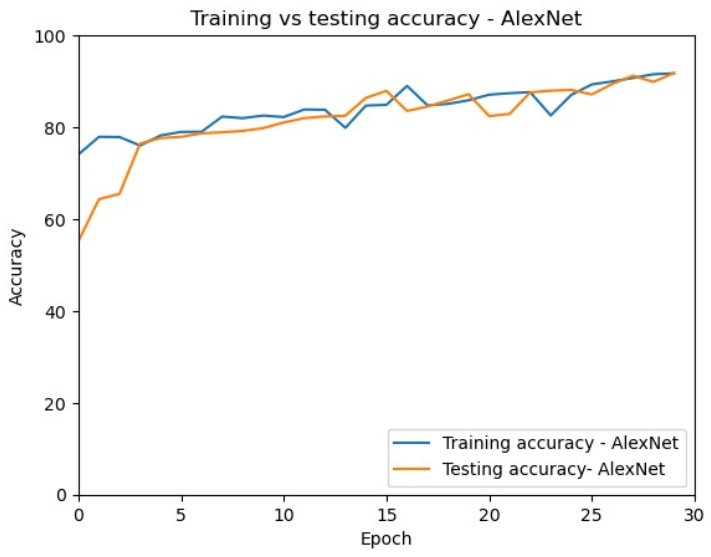


Fig: Training and Testing accuracy of AlexNet

Part IV

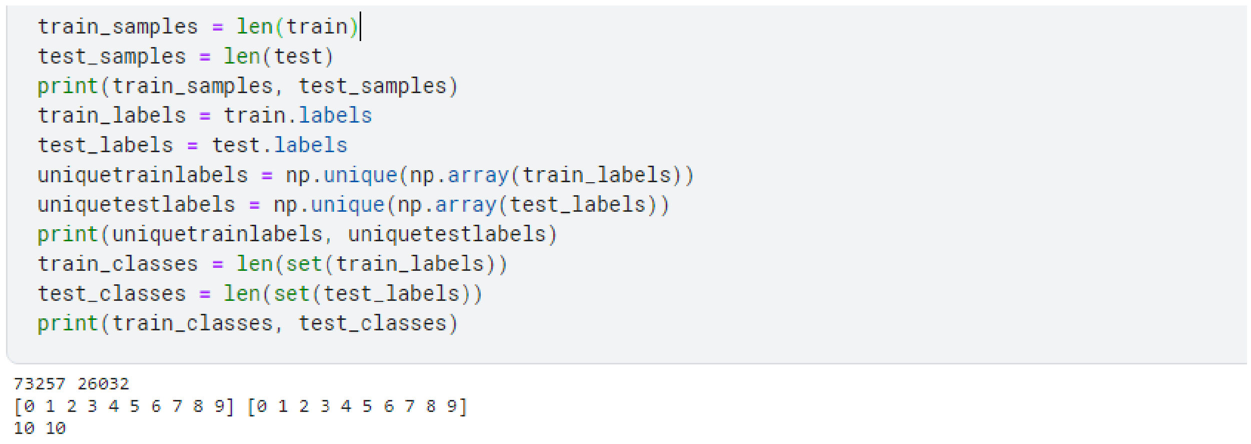
Optimizing CNN and Data Augmentation

(1)

Nature of the Dataset

The SVHN dataset can be imported directly from python libraries or from resources on the internet. We have directly imported from the **torchvision.datasets** package. Following are the statistics applied on the dataset to know about its images and labels.

SVHN is an image dataset consisting of number images and labels assigned to each image.



```
train_samples = len(train)
test_samples = len(test)
print(train_samples, test_samples)
train_labels = train.labels
test_labels = test.labels
uniquetrainlabels = np.unique(np.array(train_labels))
uniquetestlabels = np.unique(np.array(test_labels))
print(uniquetrainlabels, uniquetestlabels)
train_classes = len(set(train_labels))
test_classes = len(set(test_labels))
print(train_classes, test_classes)
```

```
73257 26032
[0 1 2 3 4 5 6 7 8 9] [0 1 2 3 4 5 6 7 8 9]
10 10
```

‘len(train)’, ‘len(test)’ gives the number of entries in the train and test datasets respectively. There are **73257 and 26032 images** in the provided train and test datasets.

uniquetrainlabels, uniquetestlabels gives the number of labels or classes in the train and test datasets. There are 10 labels under which all these images are categorized. These labels are numbered from **0 to 9**.

train_classes, test_classes gives the number of the existing labels or classes. There are **10 labels** in train and test datasets.

Visualizations

1) Images in the dataset:

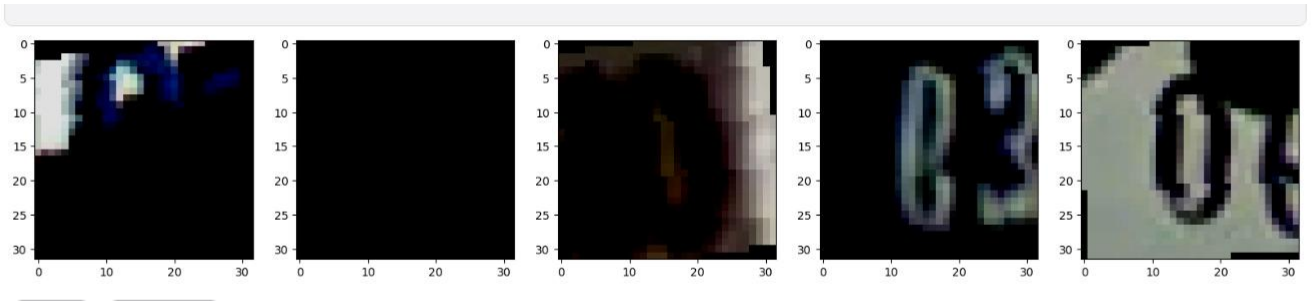


Fig: Random images in the dataset

The above plot helps us in understanding the way the images are present in the dataset. By visualizing the images we can decide on the preprocessing steps to train the model to get a better accuracy.

2) Histogram

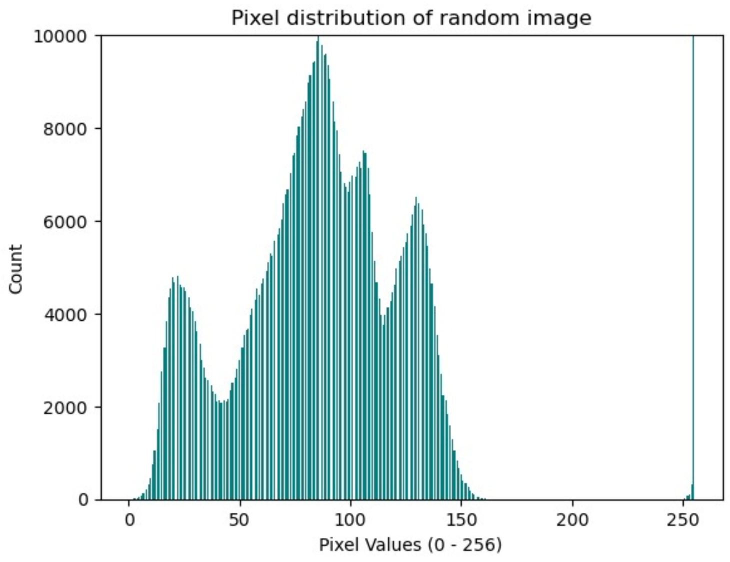


Fig: Histogram plot for a random image in the SVHN dataset

Histograms are plotted to get a better understanding of the pixel value distribution in the image. By this we can get an idea about the various elements of the image like brightness, intensity etc.

From the above histogram we can conclude that most of them have the pixel values between 50-100.

3) PIE chart

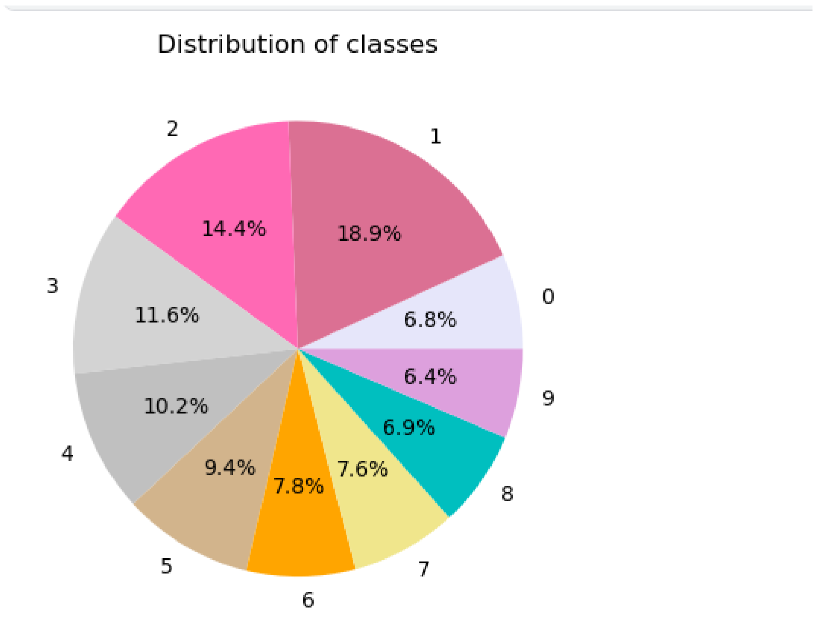
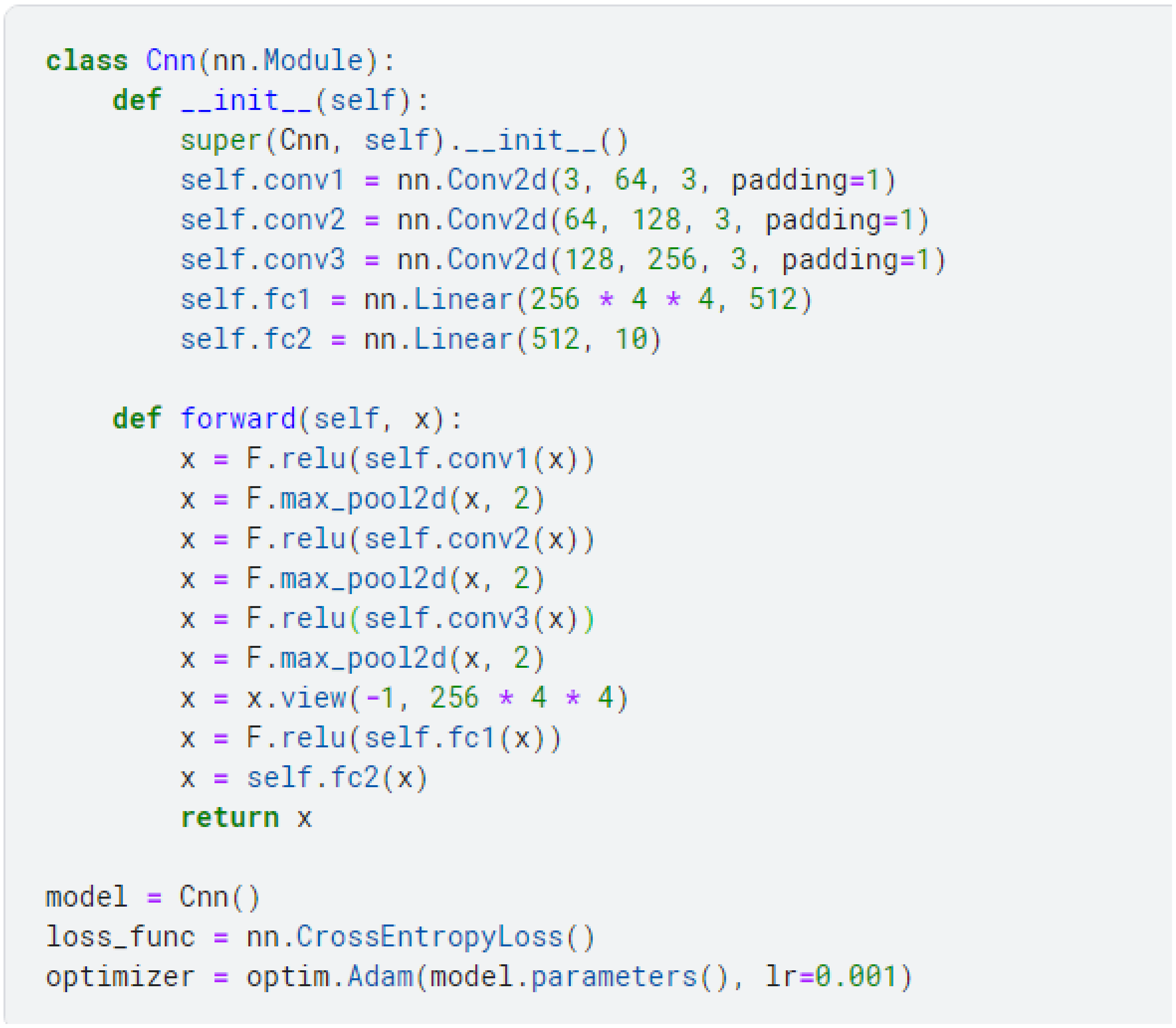


Fig: Pie chart depicting the distribution of each class in the given dataset

From the above pie chart we can conclude that there are 10 classes and the highest percentage of images belong to the **class 1** while the lowest number of images are of the **class 9**.

(2)

CNN Architecture



```python
class Cnn(nn.Module):
    def __init__(self):
        super(Cnn, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
        self.fc1 = nn.Linear(256 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.conv3(x))
        x = F.max_pool2d(x, 2)
        x = x.view(-1, 256 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = Cnn()
loss_func = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

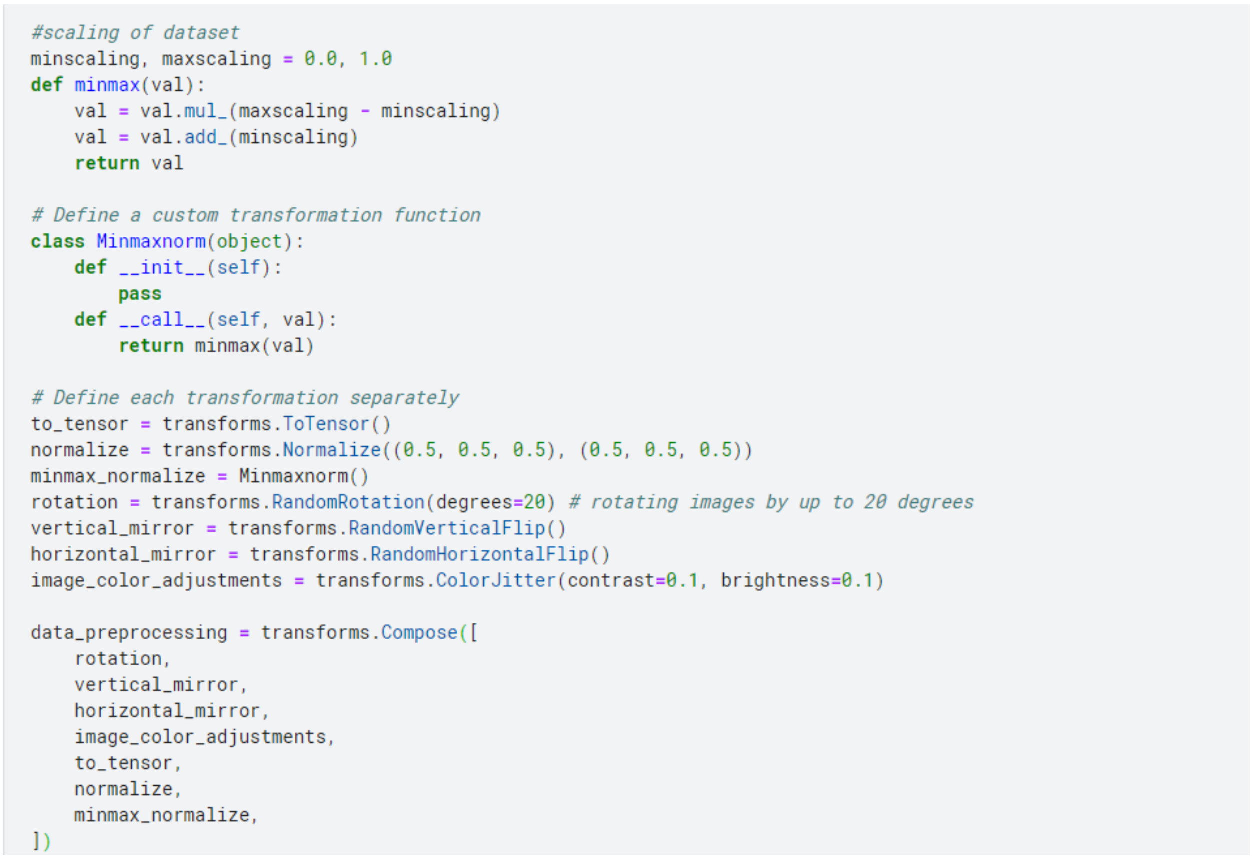
This CNN architecture is an improvised version based on the SVHN dataset. Following is the detailed description for the architecture:

- This CNN version has **three convolutional layers {conv1, conv2, conv3}** with ReLU activation functions followed by max pooling layers to reduce the spatial dimensions of input. The first convolutional layer takes input images with three channels with a kernel size of 3x3 and padding of 1. The second convolutional layer takes the 64 feature maps with the same kernel size and padding giving 128 feature maps. The third convolutional layer has 128 feature maps from the previous layer and with the same kernel size and padding, producing 256 feature maps.

- We make use of **two fully connected layers** ‘fc1’ and ‘fc2’ with ReLU activation functions. ‘fc1’ has 512 output units, and the second fully connected layer has 10 output units.
- We use **CrossEntropyLoss** as the loss function. The Adam optimizer is used along with a learning rate of 0.001.

(3)

Augmentation techniques



```
#scaling of dataset
minscaling, maxscaling = 0.0, 1.0
def minmax(val):
    val = val.mul_(maxscaling - minscaling)
    val = val.add_(minscaling)
    return val

# Define a custom transformation function
class Minmaxnorm(object):
    def __init__(self):
        pass
    def __call__(self, val):
        return minmax(val)

# Define each transformation separately
to_tensor = transforms.ToTensor()
normalize = transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
minmax_normalize = Minmaxnorm()
rotation = transforms.RandomRotation(degrees=20) # rotating images by up to 20 degrees
vertical_mirror = transforms.RandomVerticalFlip()
horizontal_mirror = transforms.RandomHorizontalFlip()
image_color_adjustments = transforms.ColorJitter(contrast=0.1, brightness=0.1)

data_preprocessing = transforms.Compose([
    rotation,
    vertical_mirror,
    horizontal_mirror,
    image_color_adjustments,
    to_tensor,
    normalize,
    minmax_normalize,
])
```

data_preprocessing is a user defined image augmentation method to apply various techniques on image data to improve the results. **‘rotation’** randomly rotates the image by 20 degrees.

‘vertical_mirror’ flips the image vertically. In the same way ‘horizontal_mirror’ flips the image horizontally. ‘Image_color_adjustments’ changes the image effects. We make use of a pre-defined ‘to_tensor’ to convert data to tensor format. ‘minmax_normalize’ scales image making use of user defined minmax function. ‘nomalize’ is used to normalize images using mean and standard deviations.

These augmentation techniques helps to increase the accuracy by correctly predicting the output.

(4) Training and Testing Accuracy Graphs

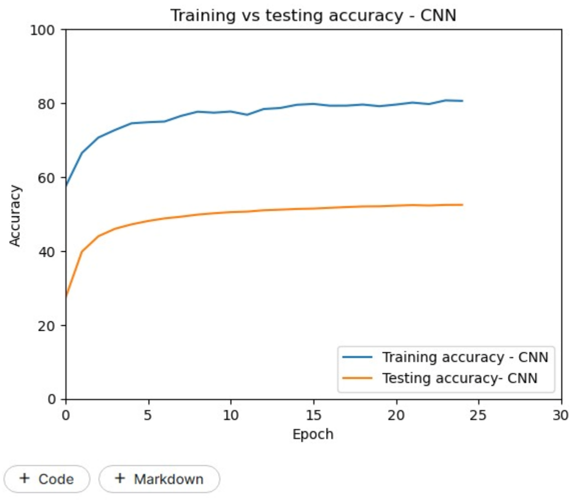


Fig: Training and testing accuracy for SVHN dataset

From the above graph, we can infer that the training accuracy achieved is 52.46% while the testing accuracy achieved for the model is 80.59% for the SVHN dataset.

Contribution:

Team Member	Assignment Part	Contribution (%)
Shravani Soma	**Part - 1,3** **Report – Part 2,4**	**50%**
Sriinitha Chinnapatlola	**Part - 2,4** **Report – Part 1,3**	**50%**

REFERENCES

1) https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

2) https://pytorch.org/tutorials/beginner/saving_loading_models.html

3) https://scikitlearn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

4) https://pytorch.org/docs/stable/nn.html#loss-functions

5) https://pytorch.org/docs/stable/optim.html

6) torch.nn.init — PyTorch 2.0 documentation

7) StepLR — PyTorch 2.0 documentation

8) https://www.tensorflow.org/tutorials/images/data_augmentation