Machine Learning Assignment - 2 Final Report

Ritesh Manchikanti vmanchik

Anurima Vaishnavi Kumar anurimav

Report for Part I:

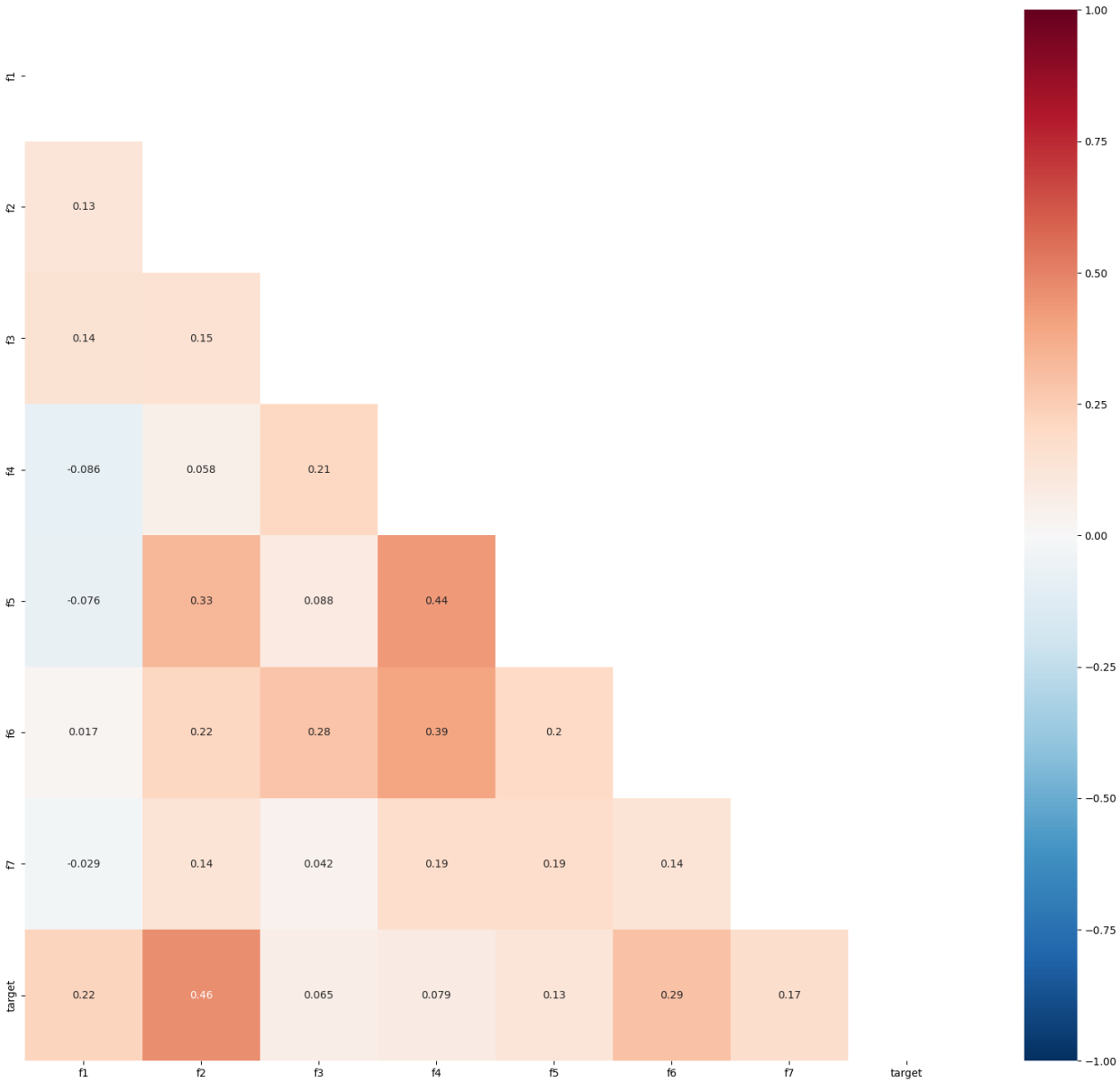
1. Provide brief details about the nature of your dataset. What type of data are we encountering? How many entries and variables does the dataset comprise? Provide the main statistics about the entries of the dataset.

Nature of the dataset:

1. It is a Comma Separated Values file(CSV File).
2. There are 8 Columns and 767 Rows
3. There are 7 Features and 1 Target
4. There are some Non numerical values in rows that we have to filter.
5. These are the main statistics of the Dataset.

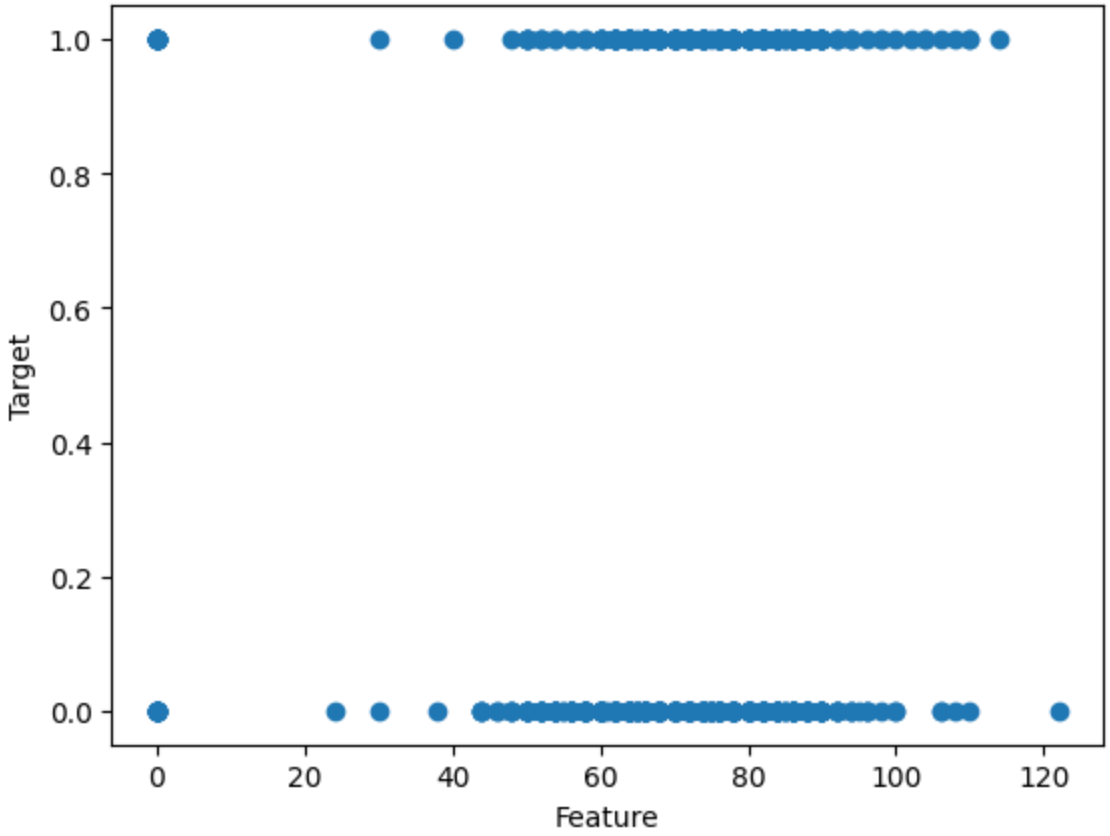
	f1	f2	f3	f4	f5	f6	f7	target
count	7.600000e+02	7.600000e+02	7.600000e+02	7.600000e+02	7.600000e+02	7.600000e+02	7.600000e+02	760.000000
mean	3.739699e-17	9.349247e-18	-2.898266e-16	9.582978e-17	2.337312e-18	-1.986715e-16	1.519253e-16	0.350000
std	1.000659e+00	1.000659e+00	1.000659e+00	1.000659e+00	1.000659e+00	1.000659e+00	1.000659e+00	0.477284
min	-1.140270e+00	-3.780041e+00	-3.556770e+00	-1.285961e+00	-6.946361e-01	-4.053275e+00	-1.190303e+00	0.000000
25%	-8.428760e-01	-6.865065e-01	-2.891809e-01	-1.285961e+00	-6.946361e-01	-5.951826e-01	-6.911439e-01	0.000000
50%	-2.480889e-01	-1.240456e-01	1.482128e-01	1.562691e-01	-3.829623e-01	1.666711e-04	-2.943761e-01	0.000000
75%	6.440919e-01	6.259022e-01	5.598776e-01	7.206199e-01	4.157018e-01	5.828490e-01	4.645270e-01	1.000000
max	3.915421e+00	2.438276e+00	2.721117e+00	4.921898e+00	6.629698e+00	4.446286e+00	5.862677e+00	1.000000

2. Provide at least 3 visualization graphs with short descriptions for each graph.



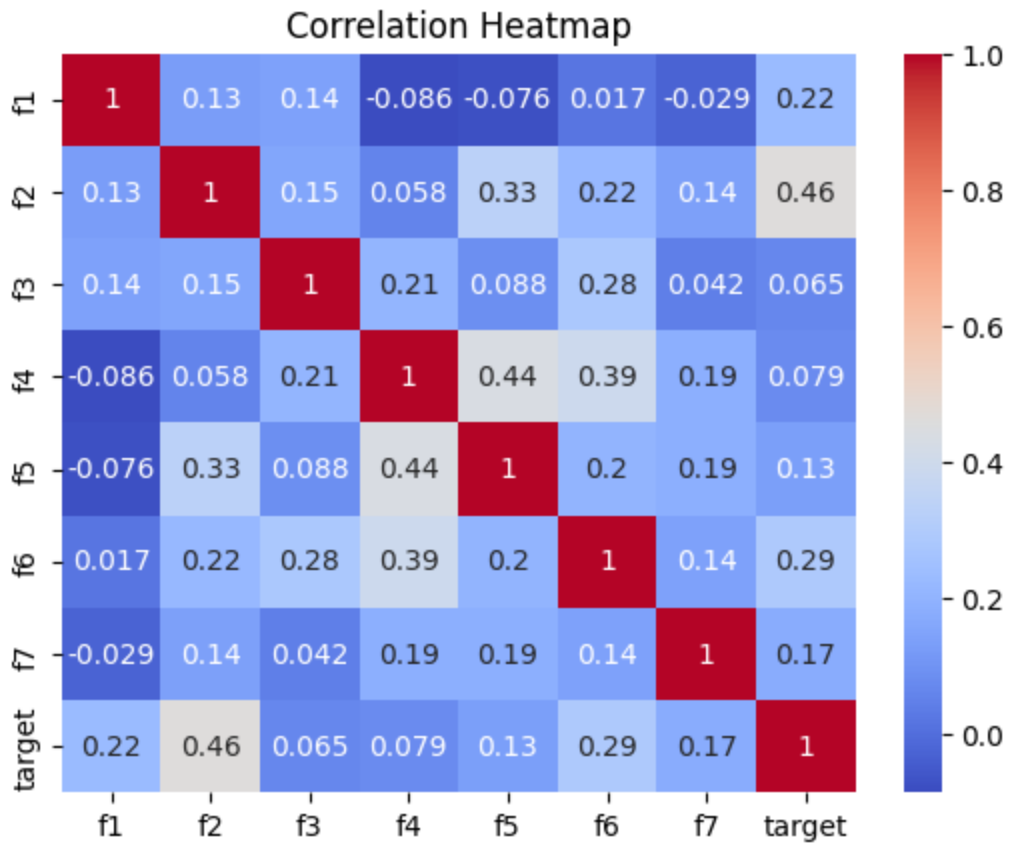
a)

This is the corelation matrix of the dataset.



b)

This above graph represents the relation between feature no 3 and target Variable.



c)

This is the correlation heatmap for all the features and the target variable.

3. For the preprocessing part, discuss if you use any preprocessing tools, that helps to increase the accuracy of your model.

```
df = df[pd.to_numeric(df['f1'], errors='coerce').notnull()]
df = df[pd.to_numeric(df['f4'], errors='coerce').notnull()]
df = df[pd.to_numeric(df['f5'], errors='coerce').notnull()]
df = df[pd.to_numeric(df['f6'], errors='coerce').notnull()]
df = df[pd.to_numeric(df['f7'], errors='coerce').notnull()]
df = df[pd.to_numeric(df['target'], errors='coerce').notnull()]
df_numeric = df.apply(pd.to_numeric, errors='coerce')
df_numeric = df_numeric.dropna()
scaler = StandardScaler()
df_numeric[['f1', 'f2', 'f3', 'f4', 'f5', 'f6', 'f7']] = scaler.fit_transform(df_numeric[['f1', 'f2', 'f3', 'f4', 'f5', 'f6', 'f7']])
```

Rows with missing or invalid values in columns 'f1', 'f4', 'f5', 'f6', 'f7', and 'target' are filtered out using the pd.to_numeric() function and the notnull() method. This is a data cleaning step that ensures that the input data contains only valid and complete rows.

The remaining rows are converted to numeric values using the pd.to_numeric() function. Any invalid or missing values are converted to NaN.

Rows containing NaN values are removed using the dropna() method. This is another data cleaning step that ensures that the input data contains only valid and complete rows.

The remaining columns 'f1', 'f2', 'f3', 'f4', 'f5', 'f6', and 'f7' are standardized using the StandardScaler() function. This is a preprocessing step that scales each column to have zero mean and unit variance. Standardization improves the performance of many machine learning algorithms.

4. Provide the architecture structure of your NN.

```
model = tf.keras.models.Sequential([
   tf.keras.layers.Dense(64, activation='relu', input_shape=(7,)),
   tf.keras.layers.Dense(1, activation='sigmoid')
])
```

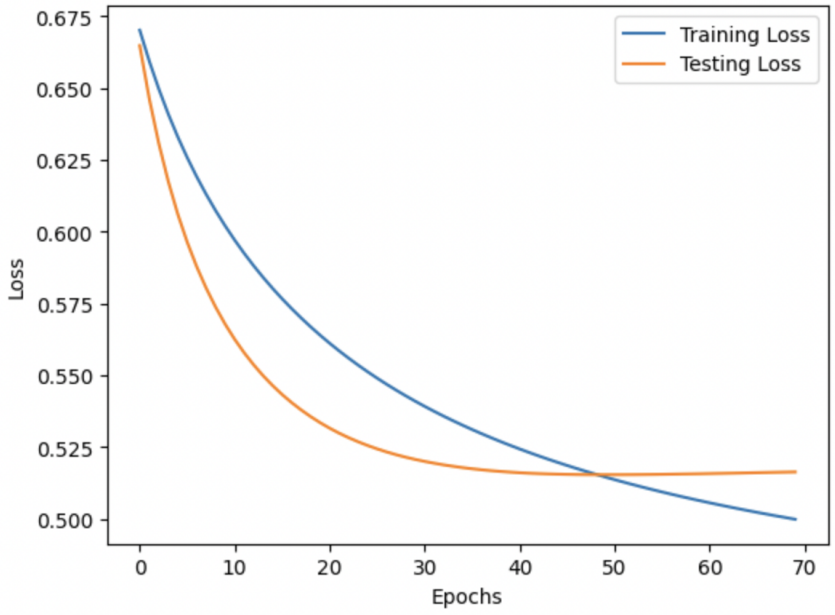
Input layer: This layer has 7 neurons, which corresponds to the number of features in the input data.

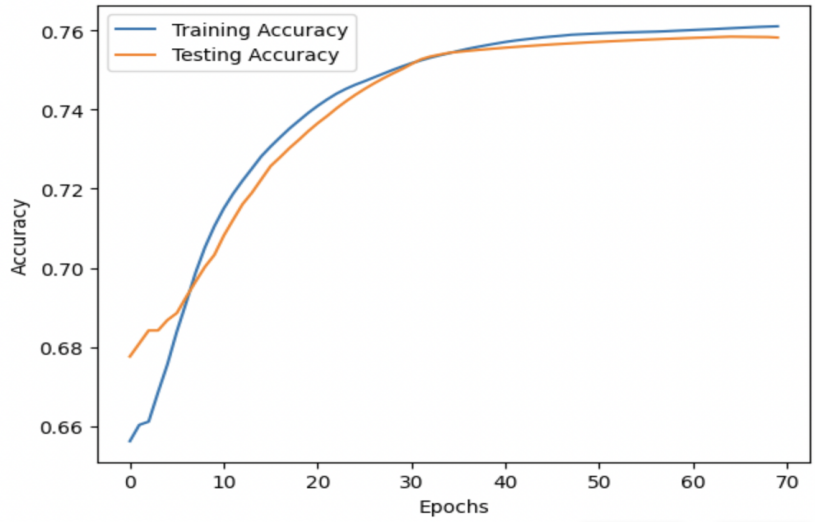
Hidden layer: This layer has 64 neurons and uses the ReLU activation function. The ReLU activation function is a non-linear activation function that is commonly used in deep learning models. It is defined as f(x) = max(0,x), which means that the output is the maximum of the input and 0.

Output layer: This layer has a single neuron and uses the sigmoid activation function. The sigmoid activation function is commonly used for binary classification problems. It is defined as f(x) = 1 / (1 + e^-x), which produces a

value between 0 and 1, representing the probability of the input belonging to the positive class.

5. Provide graphs that compare test and training accuracy on the same plot, test and training loss on the same plot. Thus in total two graphs with a clear labeling





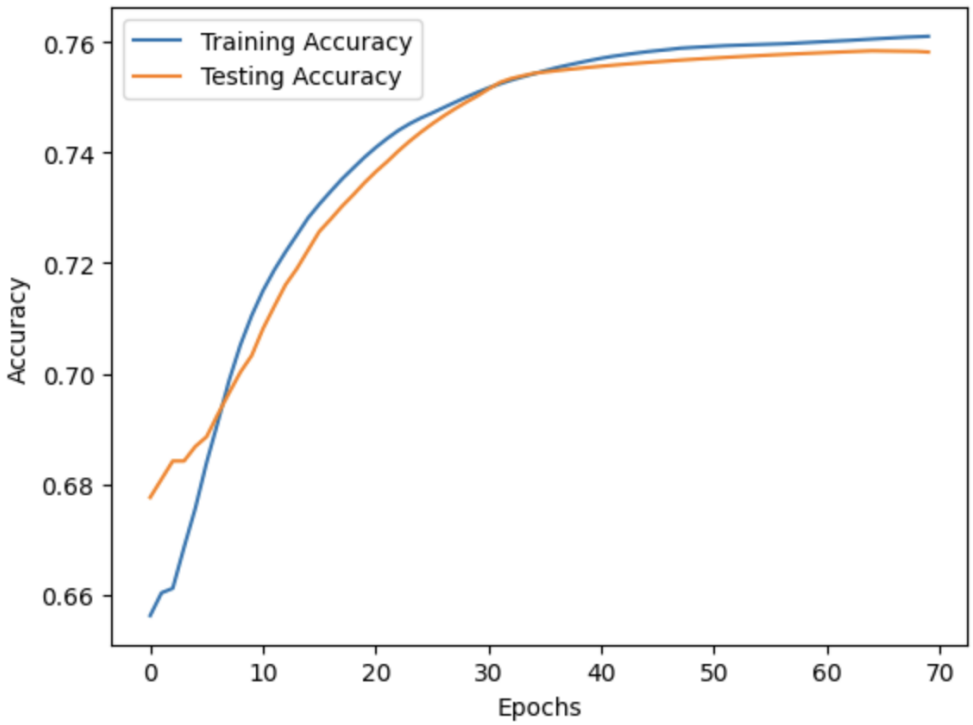
Report for Part II

1. Include all 3 tables with different NN setups.

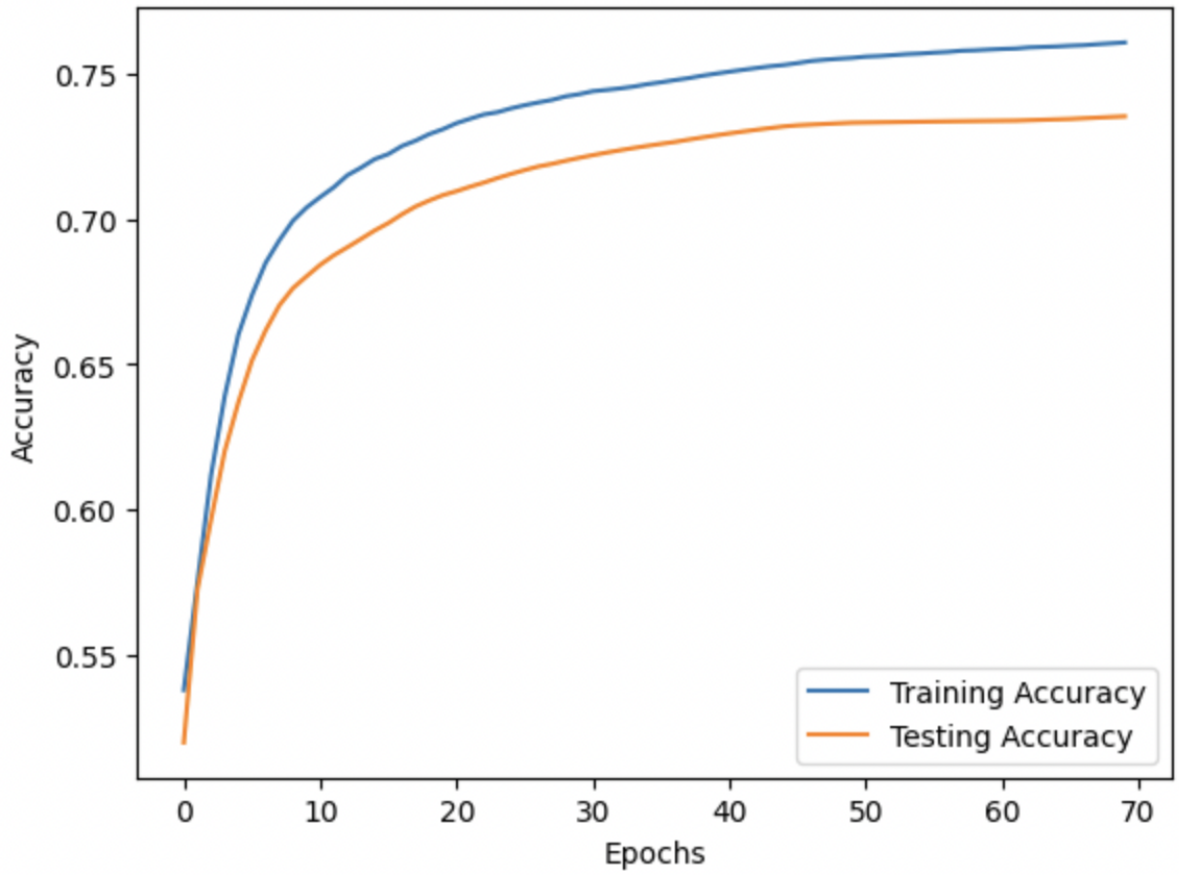
"NAME OF THE HYPERPARAMETER" Tuning

	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Setup 3	Test Accuracy
Dropout	0.1	0.7353	0.5	0.7488	0.8	0.7567
Optimizer	adam	0.7511	RMSProp	0.7467	Nadam	0.7528
Activation Function	Tanh	0.7441	LeakyReLU	0.7469	linear	0.7384
Initializer	uniform	0.7554	Random	0.7494	orthogonal	0.7658

2. Provide graphs that compares test and training accuracy on the same plot for all your setups and add a short description for each graph.

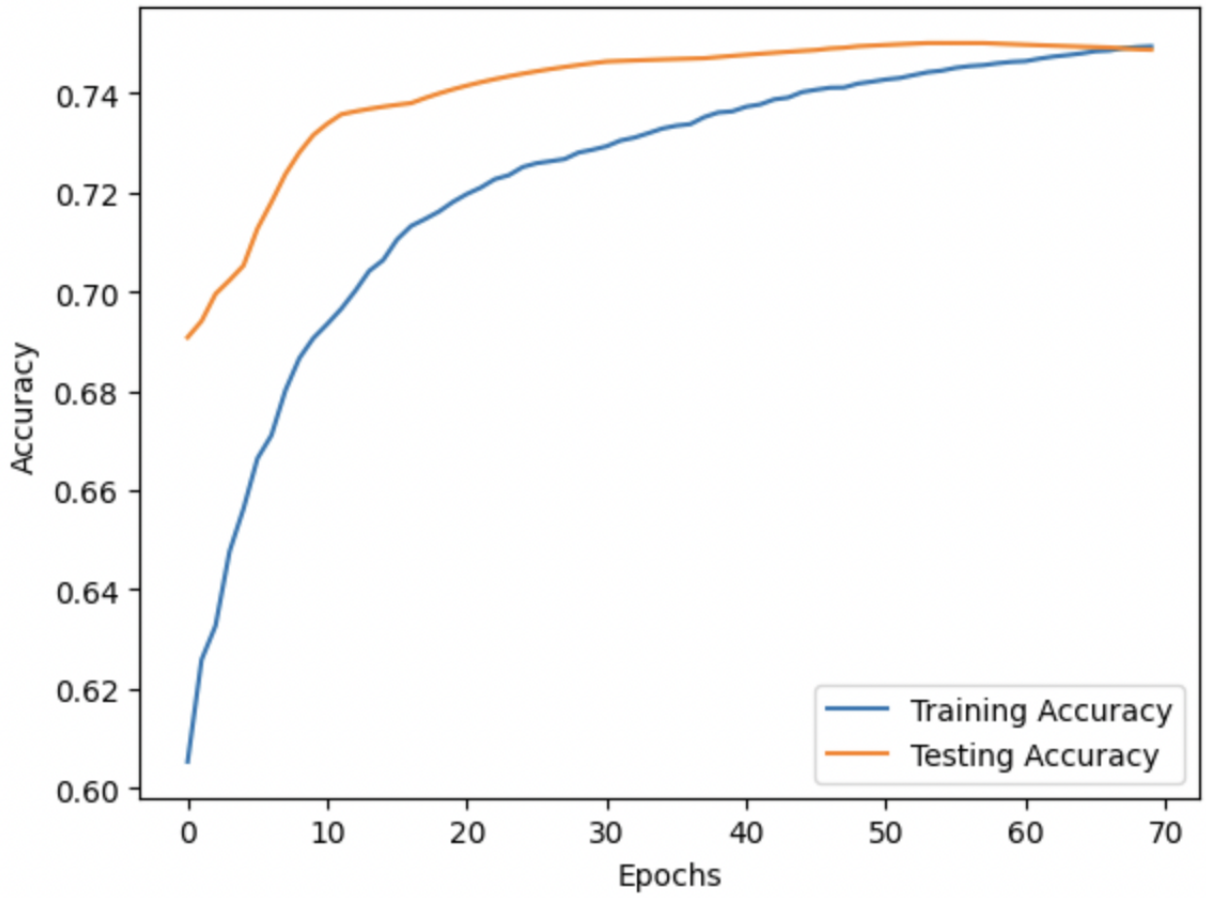


1) It has 2 layers. 64 neurons in the first hidden layer
2) 1 neuron in the output layer
3) Activation function : relu
4) Optimizer : Stochastic Gradient Descent
5) Accuracy - 75.8%

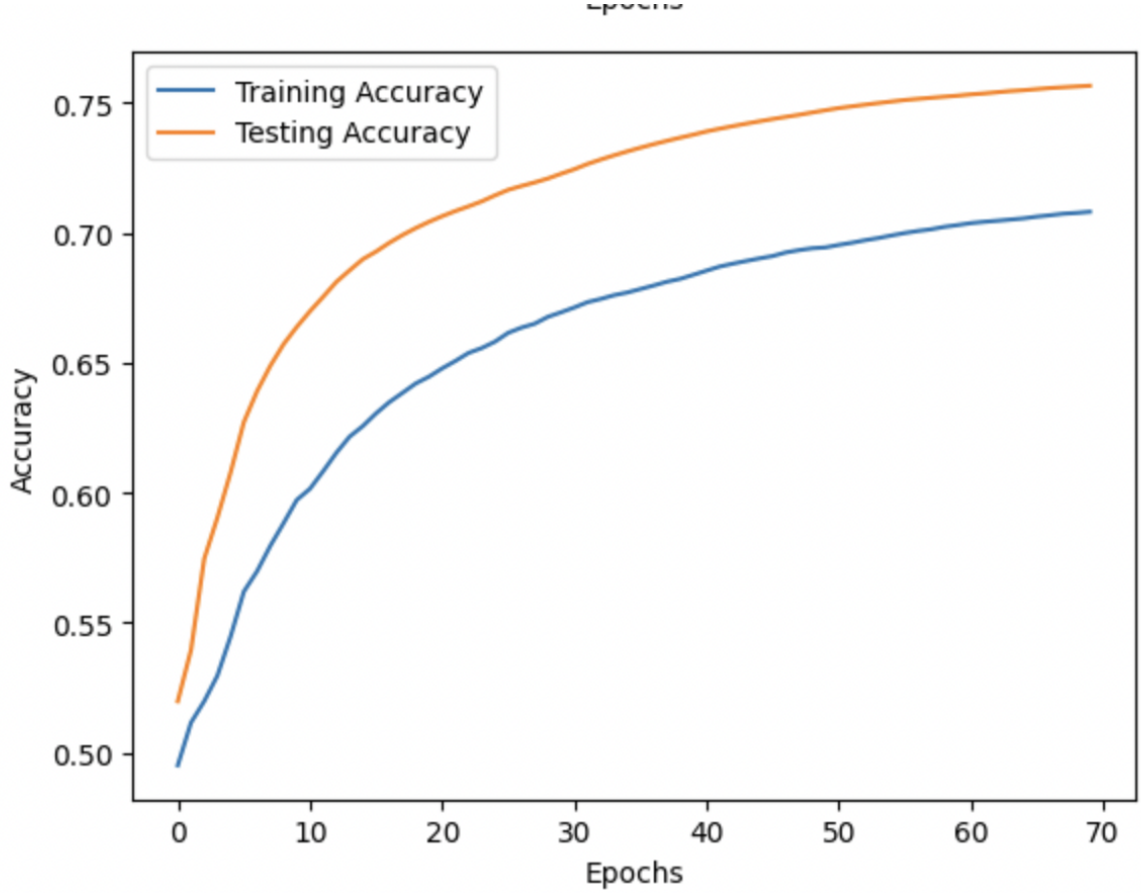


1)Neural network setup stays the same with only dropout layer added after the first layer with the dropout value being 0.1 that means 10% of neurons will randomly drop off in the first layer.

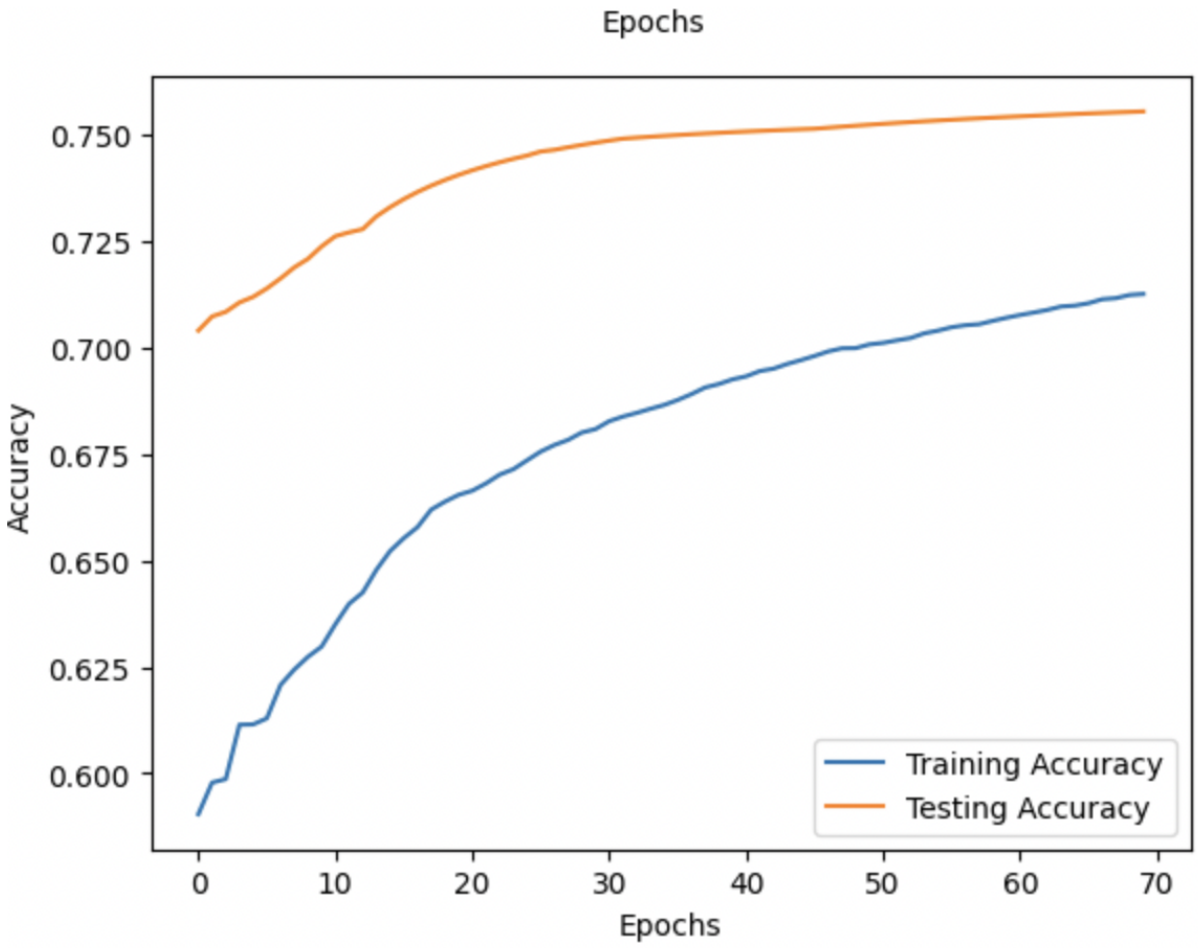
2) On learning the accuracy to be 73.5% we inferred that we need to change the parameters.



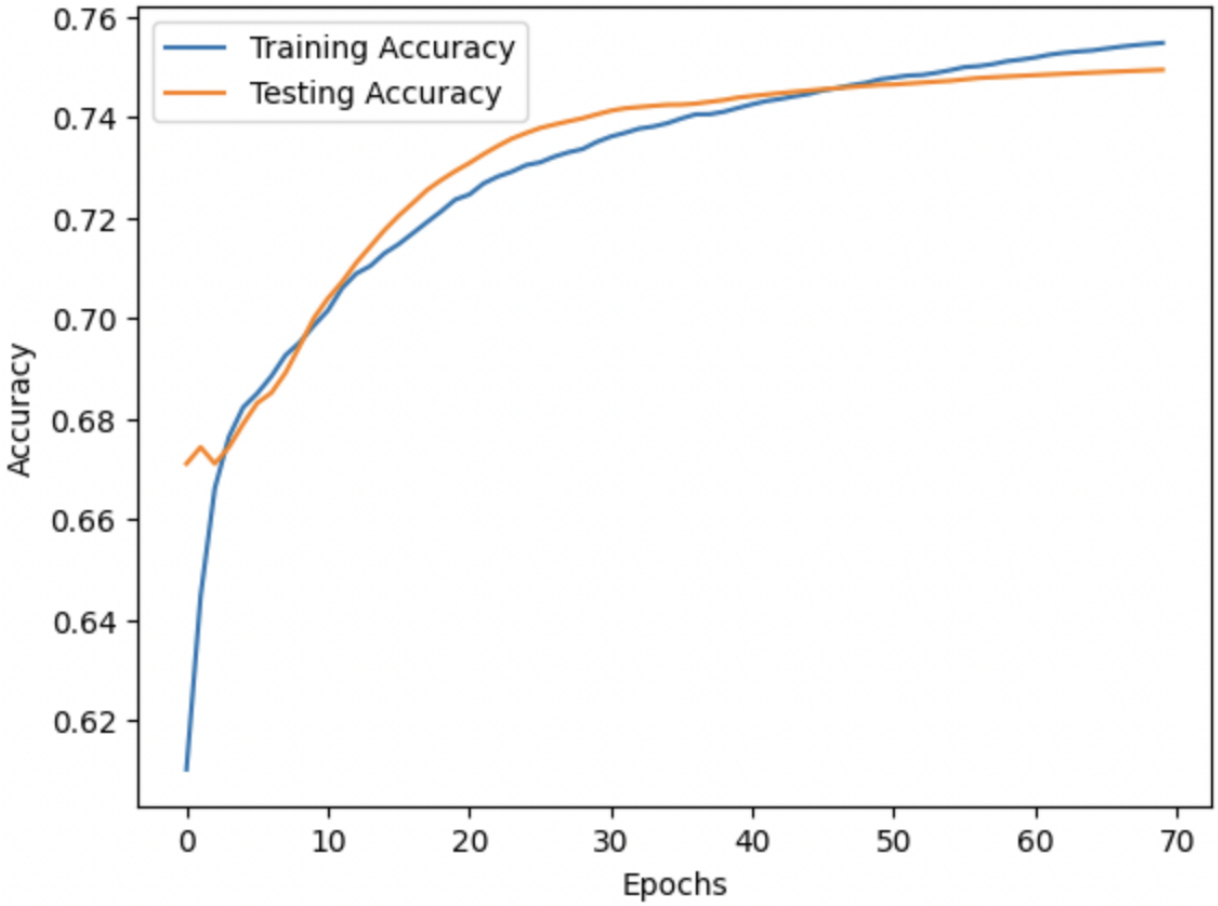
Then we increase the dropout percentage of the first hidden layer to 50% and the acuracy was 74.8%



On increasing the dropout percentage to 80%, the accuracy increased to 75.8% Inferring that this was the best accuracy achieved thus far, we fix the dropout percentage to 80% or 0.8.

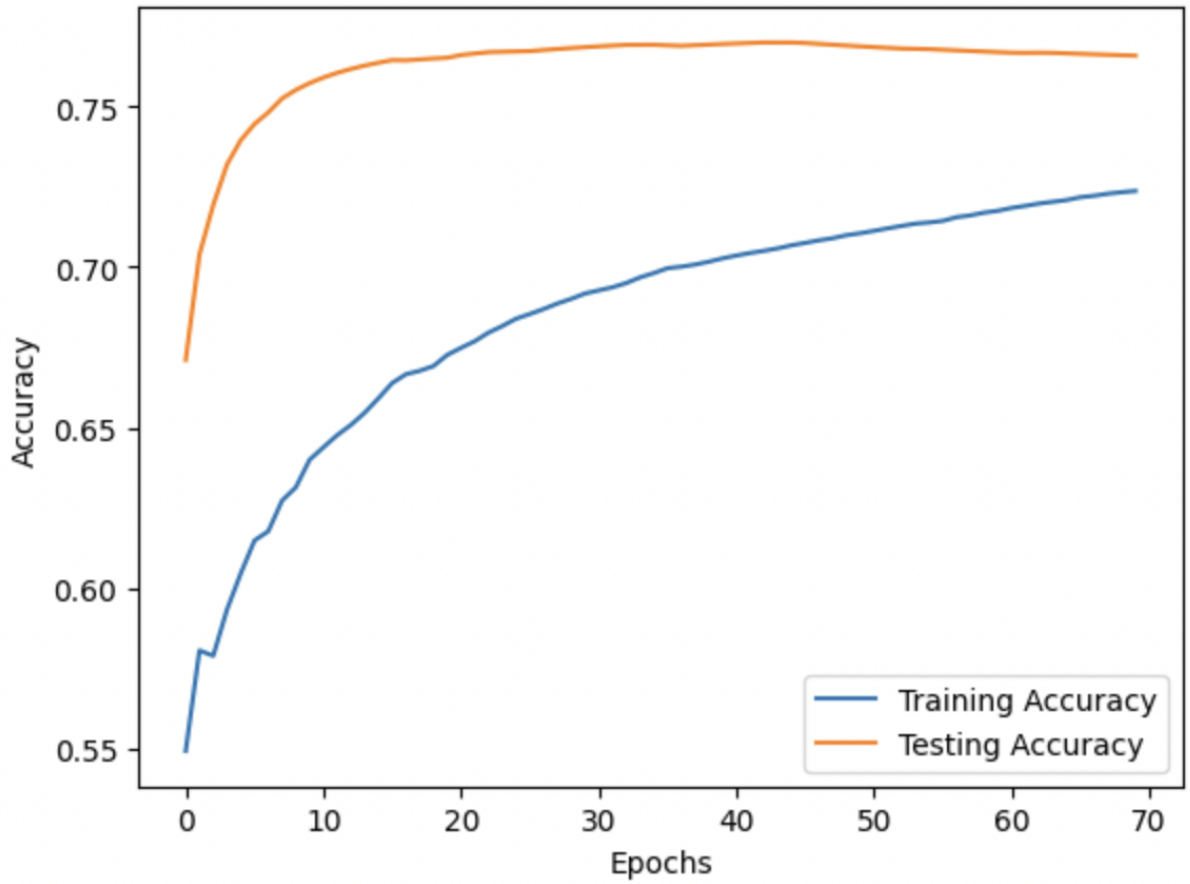


Dropout = 80%
We have used glorot_uniform to initialize the Kernel of the First hidden layer.
The accuracy thus obtained was 75.54%



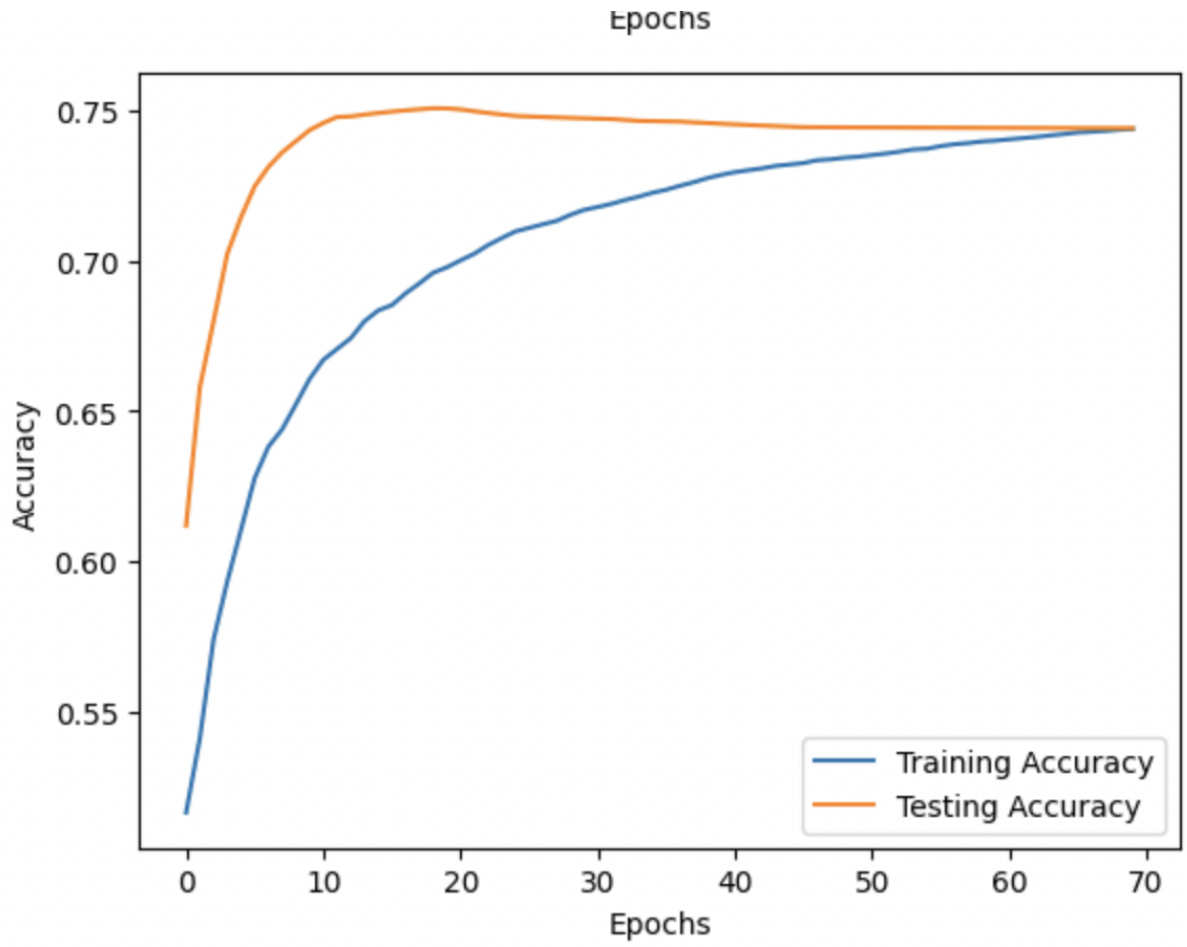
kernel_initializer=initializers.RandomNormal(stddev=0.01)

We use a random initializer here and accuracy fell down to 74.98

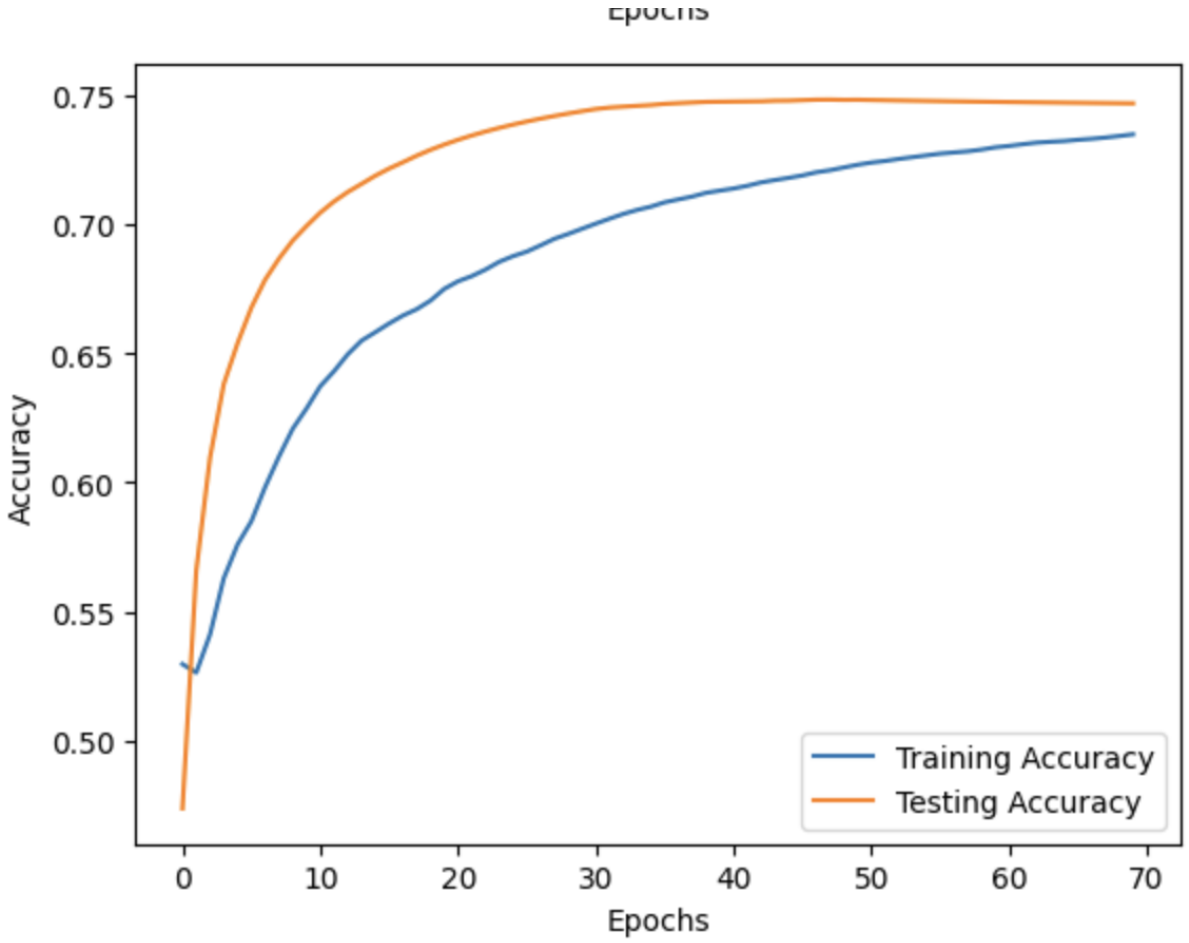


kernel_initializer=initializers.Orthogonal(gain=1.0)

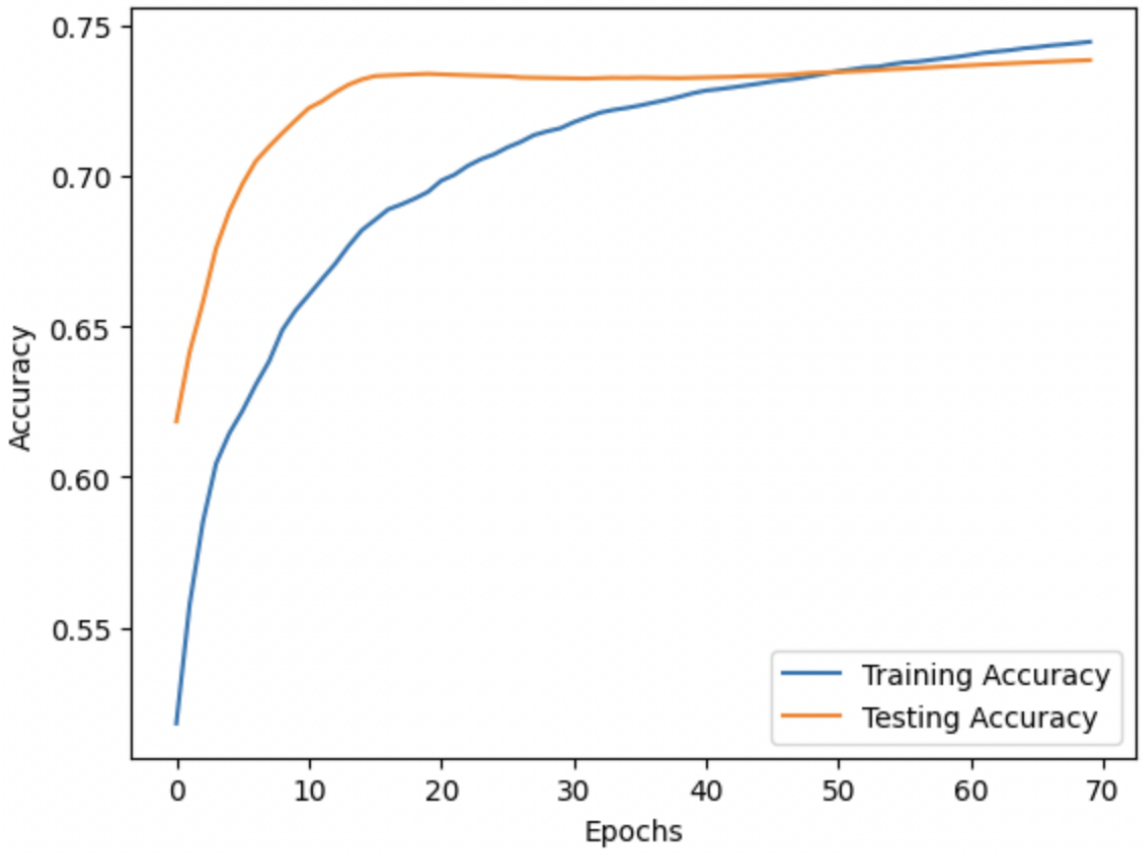
On using the Orthogonal kernel initializer the accuracy went up to 76.58. This is the best accuracy our model achieve until this point



Dropout = 0.8
Kernel = Orthogonal
Activation function = tanh
Accuracy -74.4%
This means it wasn't the best activation function for this setup

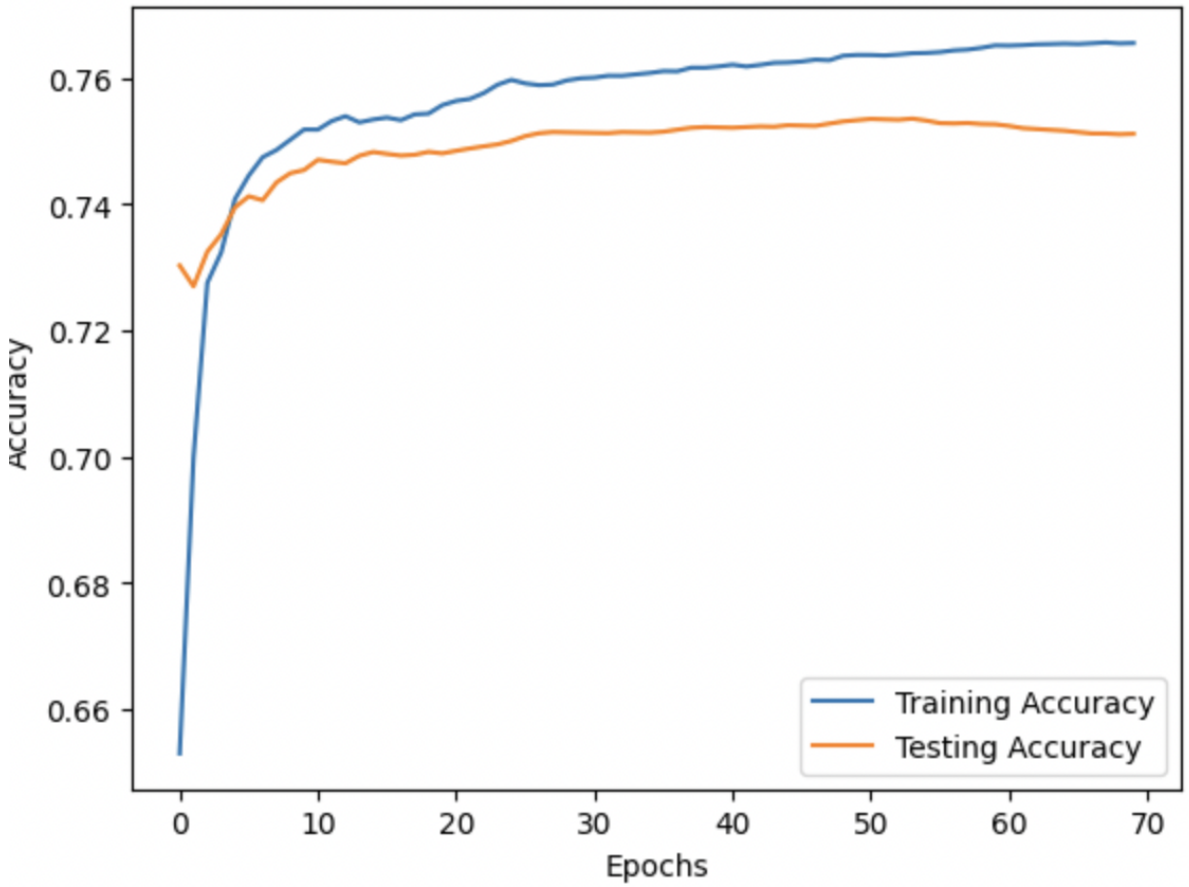


Using the exact setup as before but changing the activation function to Leaky Relu we obtained an accuracy of 74.69% indicating we need a better activation function

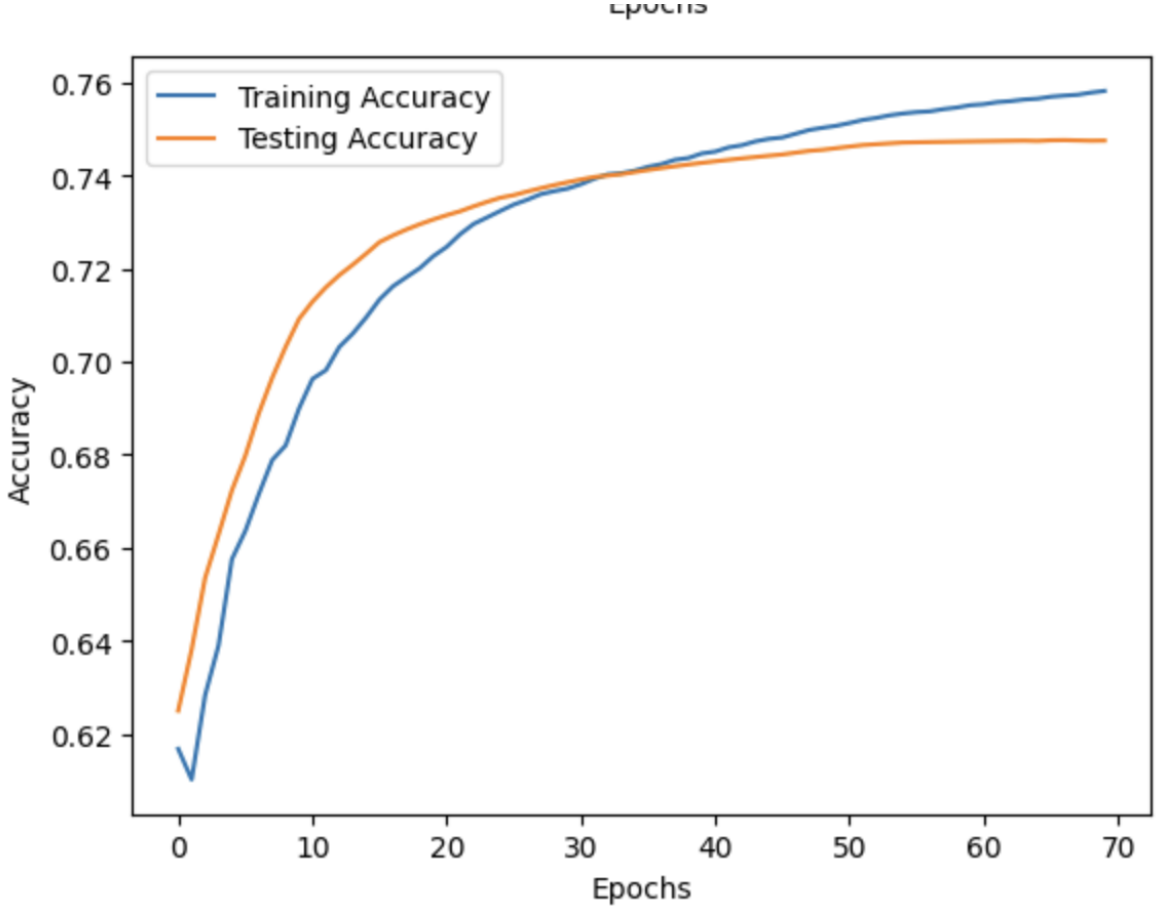


Activation function - linear

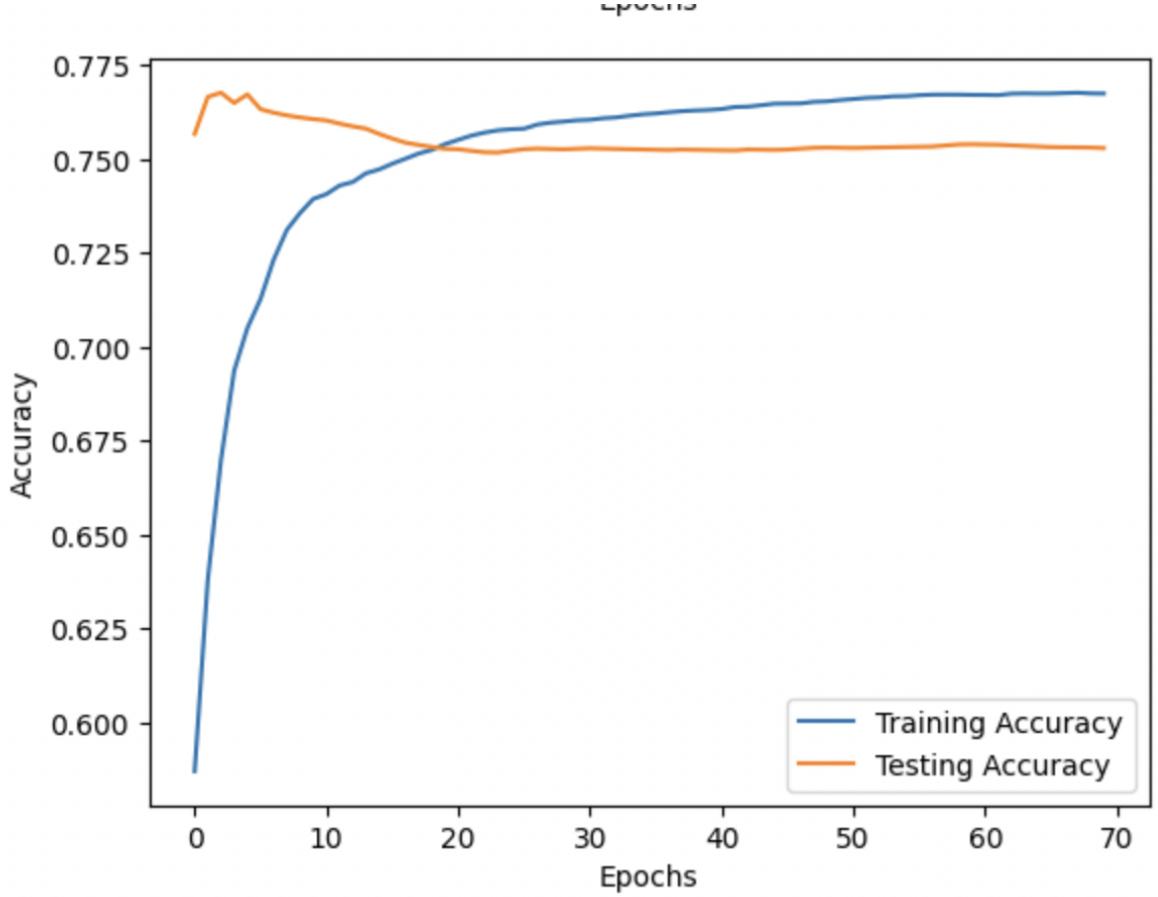
Accuracy fell down 73.84, that means initial activation function ie., relu was the best. Hence, we are going forward with ReLu activation function for our neural network setups from here on.



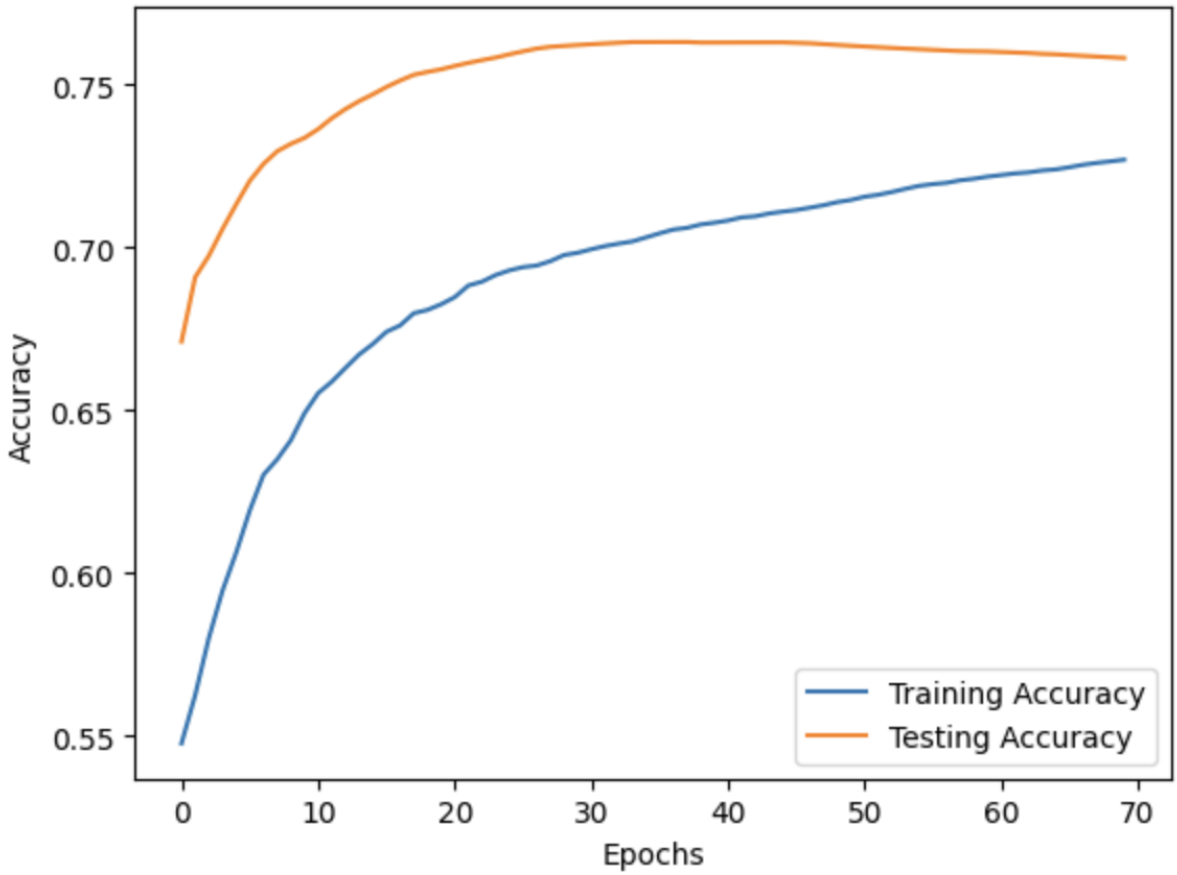
Dropout - 80%
Activation func - relu
Kernel - orthogonal
The accuracy -> 75.11%
Optimizer - Adam



Dropout - 80%
Activation func - relu
Kernel - orthogonal
Optimizer - RMSProp
The accuracy -> 74.76%



Dropout - 80%
Activation func - relu
Kernel - orthogonal
Optimizer - Adamax
The accuracy -> 74.28%



Base model:
After tuning hyperparameters as per the table given above, the best accuracy was achieved with the following setup:
The first Dense layer has 64 neurons with a rectified linear unit (ReLU) activation function and takes an input of shape (7,). The weight matrix of this layer is initialized using the orthogonal initialization method with a gain of 1.0. The purpose of this initialization is to prevent the exploding and vanishing gradients problem during training.

The Dropout layer with a rate of 0.8 is added to regularize the model and prevent overfitting.

The second Dense layer has a single neuron with a sigmoid activation function, which outputs the binary classification probability of the model.

The model is trained using stochastic gradient descent (SGD) optimizer with a default learning rate. The number of training epochs is set to 70, and the batch size is 32.

The NeuralNetwork class is a custom class that encapsulates the model training process. It takes the training and validation data, the number of epochs, batch size, and arrays to store the training and validation loss and accuracy.

3. Provide detailed analysis and reasoning about the NN setups that you tried.

Phase 1: Dropout
The first phase of your hyperparameter tuning involved trying out different values of dropout, namely 0.1, 0.5, and 0.8. Dropout is a regularization technique used in neural networks to prevent overfitting. The idea is to randomly drop out (i.e., set to zero) some neurons during training to force the network to learn more robust representations. Your experiments showed that 0.8 dropout performed the best on your dataset. This suggests that your network was prone to overfitting and that applying dropout helped to improve its generalization performance.

Phase 2: Kernel initializer
The second phase of your hyperparameter tuning involved trying out different kernel initialization methods, namely uniform, random, and orthogonal. The kernel initializer is responsible for setting the initial values of the weights in the network. The idea is to start the network with a good set of weights that allow it to converge faster and avoid getting stuck in local minima. Your experiments showed that orthogonal initialization performed the best on your dataset. This suggests that the network benefited from having well-initialized weights.

Phase 3: Activation functions
The third phase of your hyperparameter tuning involved trying out different activation functions, namely tanh, relu, linear, and lekyRelu. The activation function is responsible for introducing non-linearity into the network and allowing it to learn complex patterns. Your experiments showed that relu performed the best

on your dataset. This is not surprising as relu has been shown to be very effective in deep neural networks due to its ability to address the vanishing gradient problem.

Phase 4: Optimizers
The fourth phase of your hyperparameter tuning involved trying out different optimizers, namely adam, sgd, rmsprop, and adamax. The optimizer is responsible for updating the weights in the network during training. The idea is to find the optimal set of weights that minimize the loss function. Your experiments showed that sgd performed the best on your dataset. This suggests that the network benefited from a simple optimization algorithm that updates the weights based on the gradients of the loss function.

Overall, your hyperparameter tuning process has led to a set of optimal hyperparameters for your neural network model: 0.8 dropout, sgd optimizer, relu activation function, and orthogonal kernel initializer. These hyperparameters suggest that your network needed regularization to prevent overfitting, well-initialized weights to allow it to converge faster, an effective non-linearity to learn complex patterns, and a simple optimization algorithm to update the weights efficiently.

It is important to note that hyperparameter tuning is an iterative process that requires careful experimentation and analysis. It is also important to cross-validate the hyperparameters on a separate validation set to ensure that the results are consistent and not due to chance.

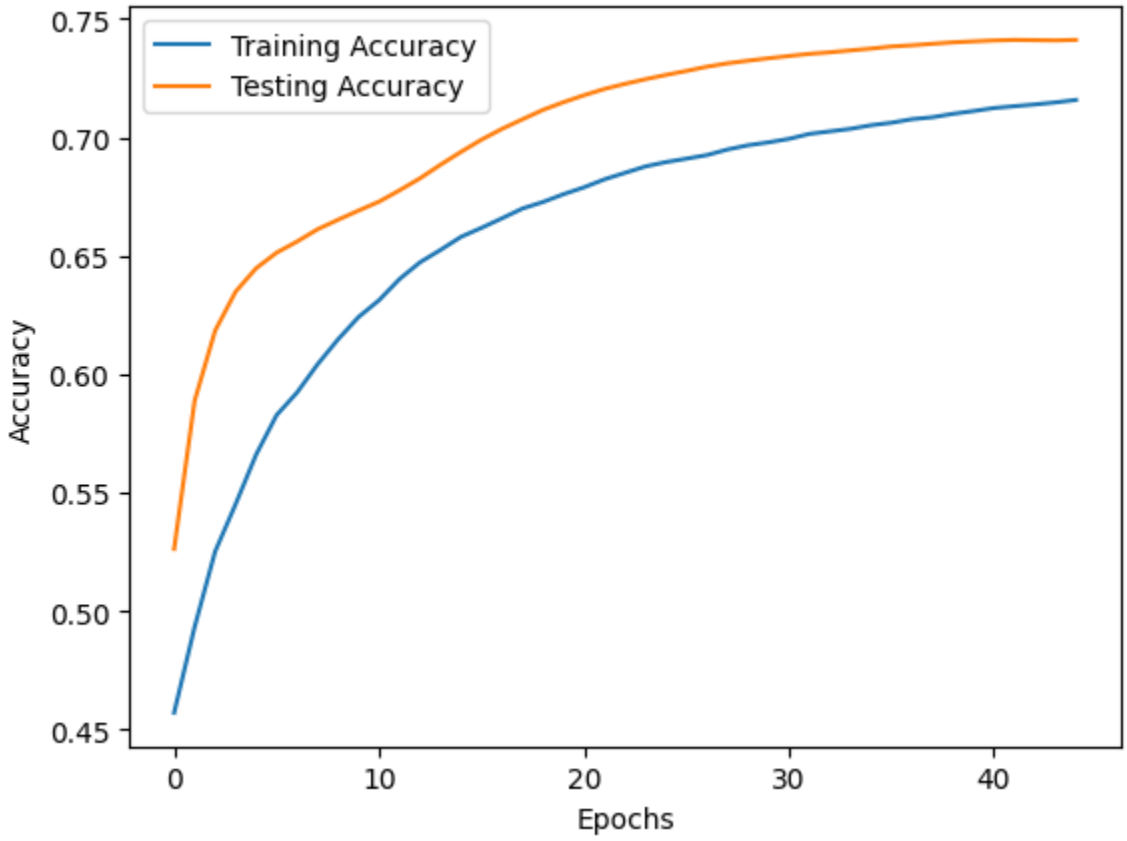
4. Briefly discuss all the methods you used that help to improve the accuracy or training time. Provide accuracy graphs and your short descriptions.

Different machine learning methods for increasing accuracy:

1) **Early stopping**

Early stopping is a common technique used in machine learning to prevent overfitting of models and to improve their ability to generalize to new data. This approach involves monitoring the performance of the model on a validation dataset during training and stopping the training process if there is no improvement in the model's performance or if the performance starts to decline.

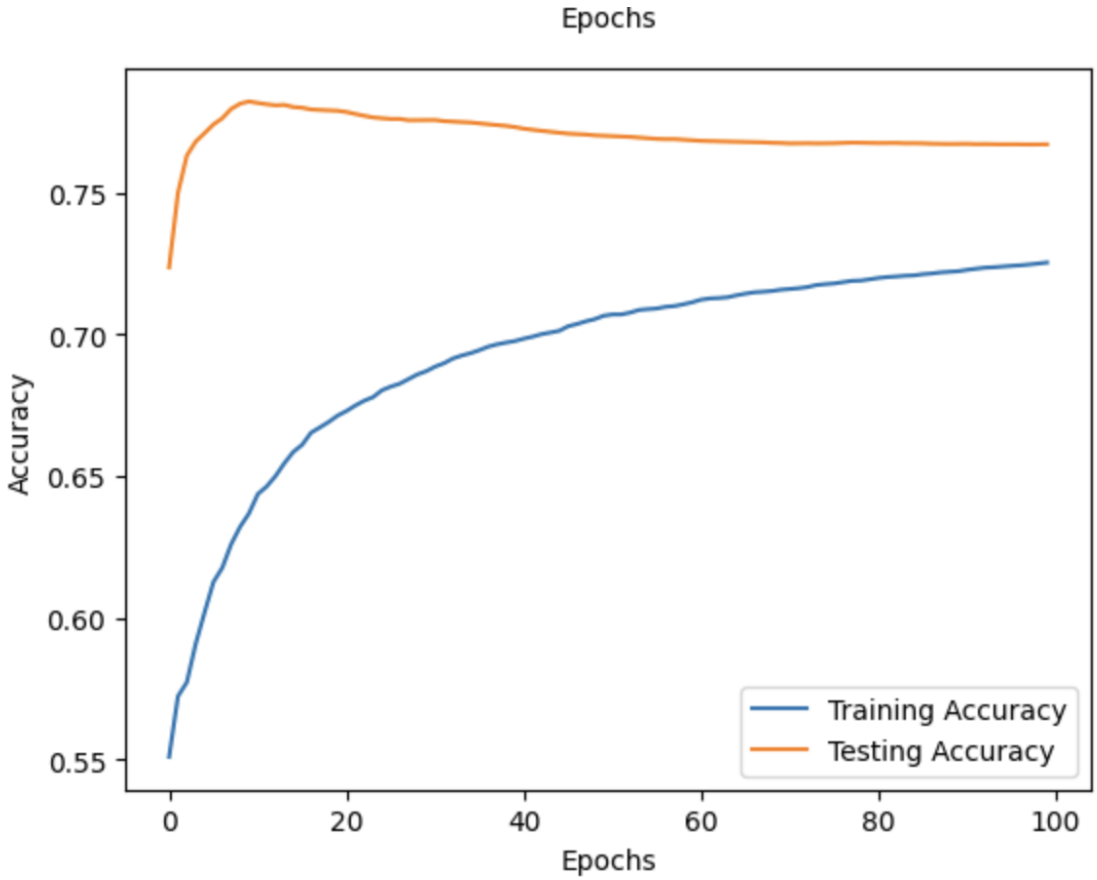
In our case, we have implemented early stopping with a patience of 5. This meant that if the model's performance did not improve for 5 consecutive epochs, the training process would be stopped. After training for 45 epochs, the model stopped with an accuracy of 74%, which did not decrease below this threshold. This suggests that the model was able to generalize well on the validation dataset, and further training would not have improved its performance.



2) BatchNormalization()

Batch normalization is a technique used in machine learning to improve the training process of neural networks. It works by normalizing the inputs of each layer to have zero mean and unit variance, which helps to reduce the effects of covariate shift and improve the stability and convergence of the training process.

In our case, we have added batch normalization after the first hidden layer in your neural network model. This helped to normalize the inputs to the layer and reduce the effects of covariate shift, which can occur when the input distributions to each layer change during training. As a result, the training process became more stable and converged faster, which led to an increase in the accuracy of the model to 76.71%.

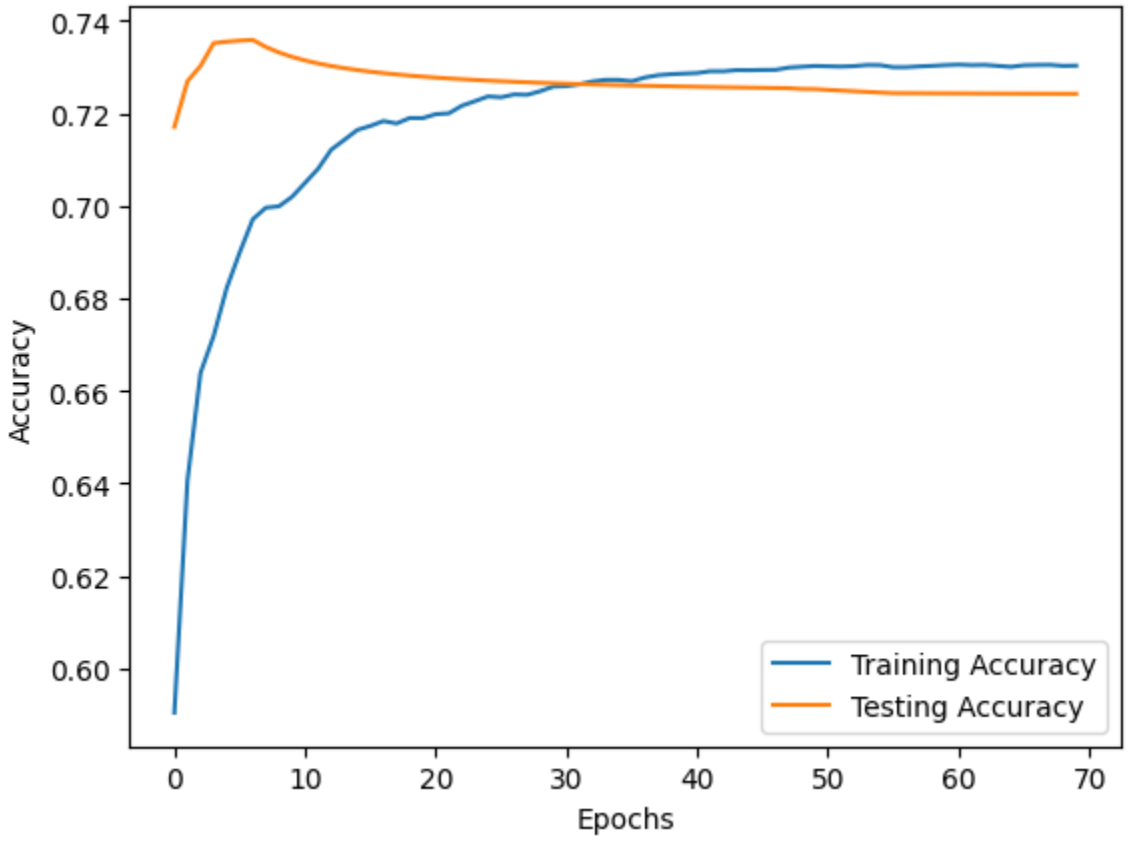


3) Learning Rate Decay()

Learning rate decay is a technique used in machine learning to reduce the learning rate of a model during training, often to prevent overfitting or improve the model's convergence speed. The learning rate controls how

much the weights of a model are updated in response to the error gradient during training, and reducing it over time can help to avoid overshooting the optimal solution and improve the generalization performance of the model.

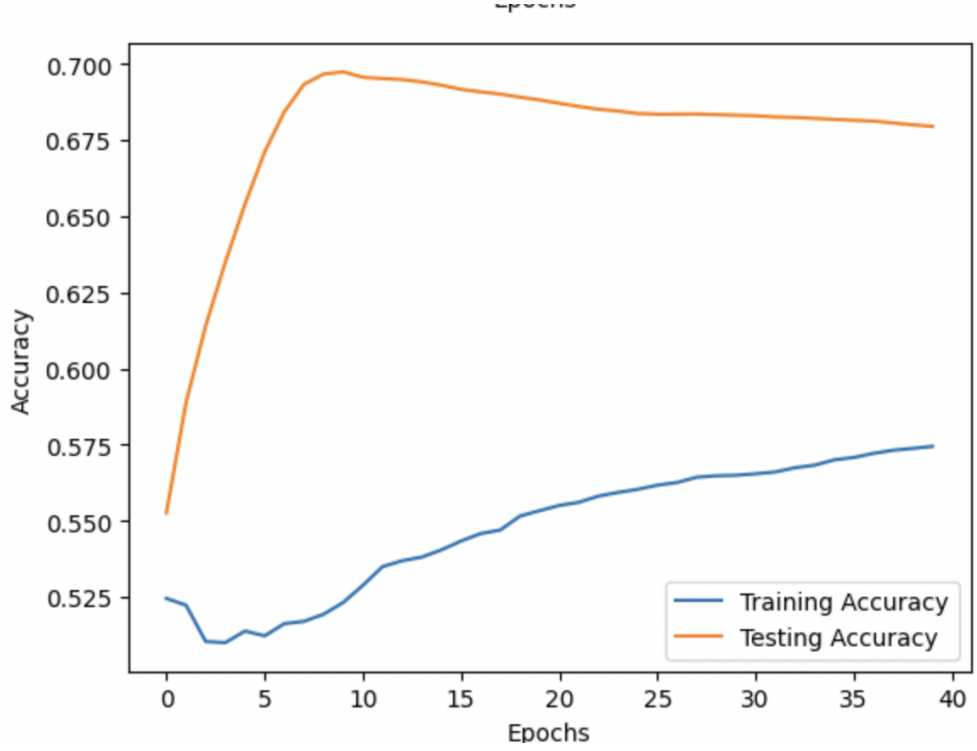
In our implementation, we have initialized the learning rate to 0.1 and set the decay rate to 0.1. During training, the learning rate was updated after each epoch using the code snippet you provided. The learning rate was reduced by a factor of 10 every 5 epochs, which means that the learning rate decayed exponentially over time.



4) K fold

K-fold cross-validation is a technique used in machine learning to evaluate the performance of a model on a dataset. It involves splitting the dataset into k subsets or folds, training the model on k-1 of the folds, and testing the model on the remaining fold. This process is repeated k times, with each fold serving as the test set exactly once.

In our implementation, we have used k-fold cross-validation with k=4. This means that you divided your dataset into 4 equal parts, and trained the model on 3 of the folds while using the remaining fold for testing. This process was repeated 4 times, with each fold being used as the test set exactly once.



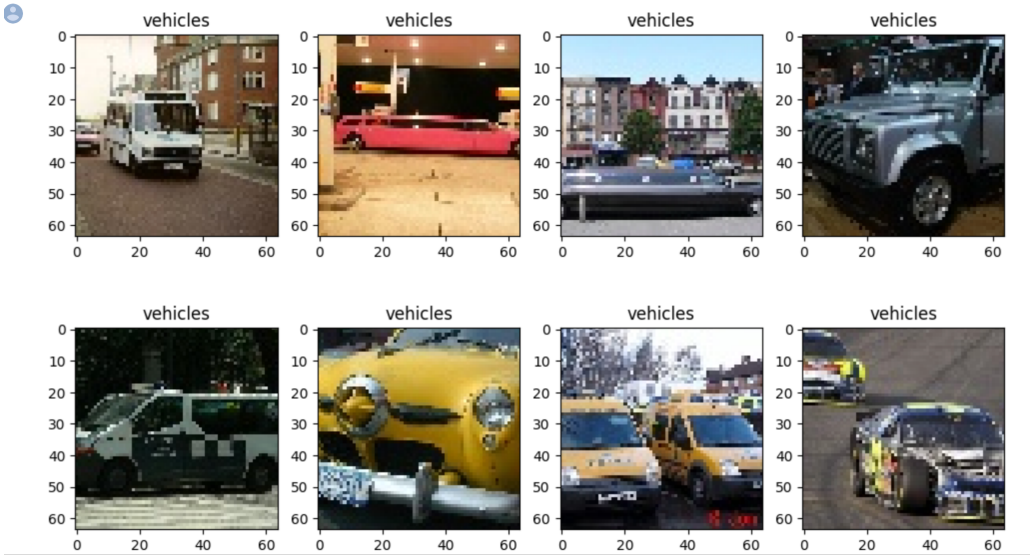
Report for Part III:

1. Provide brief details about the nature of your dataset. What is it about? What type of data are we encountering? How many entries and variables does the dataset comprise? Provide the main statistics about the entries of the dataset. Provide at least 3 visualization graphs with short descriptions for each graph.

Nature of the dataset: Images
Entries - 30,000
Labels - 3 - dogs, food, vehicles.
We used the below code to preprocess the data.

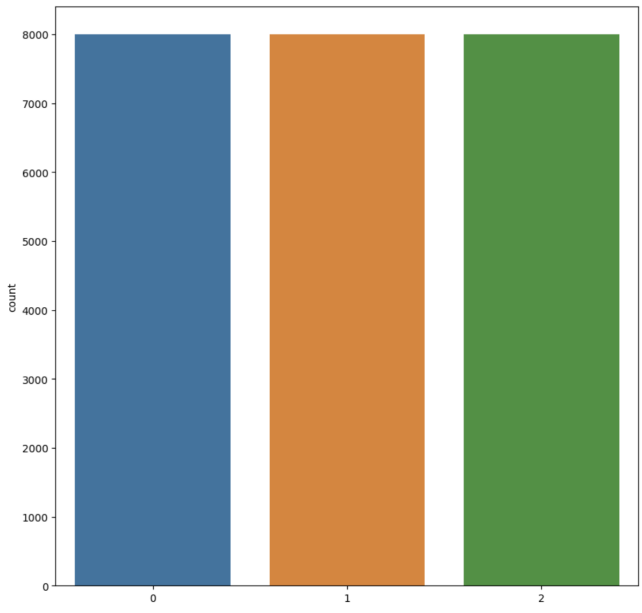
```
labels = ['dogs', 'food','vehicles']
def load_images_and_labels(data_path, cates):
  X = []
  y = []
  i = 0
  for index, cate in enumerate(cates):
    print(len(os.listdir(data_path + '/' + cate)))
    # for img_name in os.listdir(data_path + '/' + cate):
    #   i = i +1
    #   img = cv2.imread(data_path + '/' + cate + '/' + img_name)
    #   if img is not None:
    #     img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    #     img_array = Image.fromarray(img, 'RGB')
    #     img_rs = img_array.resize((64,64))
    #     img_rs = np.array(img_rs)
    #     X.append(img_rs)
    #     y.append(index)
  return X, y

X_train, y_train = load_images_and_labels(train_path, labels)
X_test, y_test = load_images_and_labels(test_path, labels)
```

```
plt.figure(figsize = (12,7))
for i in range(8):
  ind = random.randint(0, len(X_train))
  plt.subplot(240 + 1 + i)
  plt.imshow(X_train[ind])
  plt.title(cate[np.argmax(y_train[ind])])
```



The above graph is an Image grid graph, it is displaying a grid of images from the dataset and has labels to it.

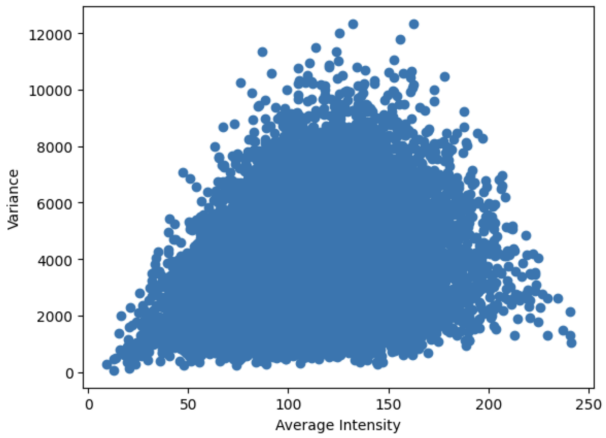
```
plt.figure(figsize=(10,10))
sns.countplot(x = y_train)
plt.show()
```



The above graph represents a distribution of the images into three classes i.e., dogs, vehicles, and food. It tells us that there is an equal distribution of all three classes in the training dataset.

The below graph is a scatter plot of the average intensity and variance of each image in the dataset:

```
plt.scatter(intensities, variances)
plt.xlabel('Average Intensity')
plt.ylabel('Variance')
plt.show()
```

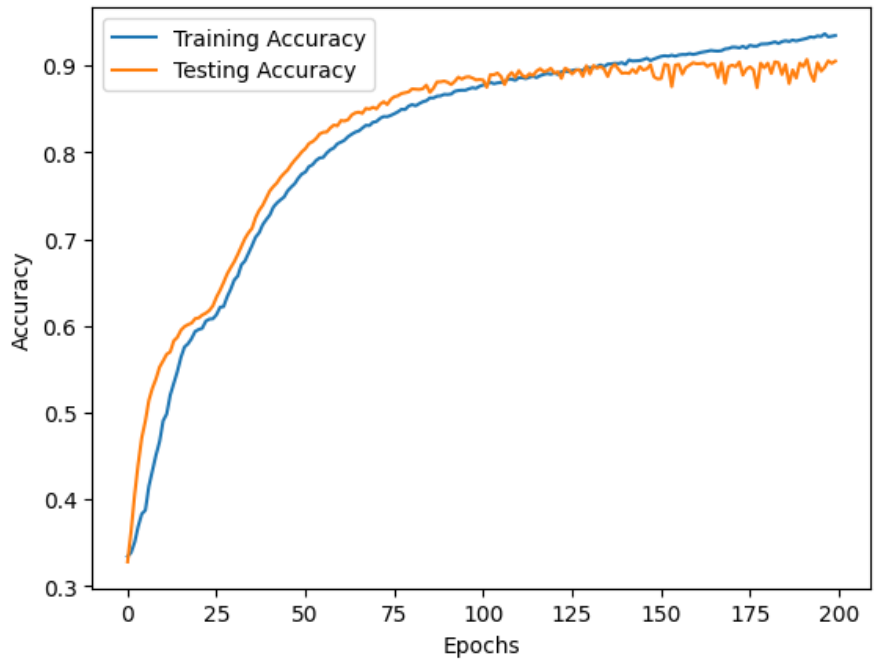


2. Provide details about your implemented model (AlexNet). Discuss your results. Provide graphs that compare test and training accuracy on the same plot.

```
model = tf.keras.models.Sequential([
 tf.keras.layers.Conv2D(96, (11,11), strides=(4,4), activation='relu',
input_shape=(64,64,3)),
 tf.keras.layers.MaxPooling2D((3,3), strides=(2,2)),
 tf.keras.layers.BatchNormalization(),
 tf.keras.layers.Conv2D(256, (5,5), strides=(1,1),  padding='same',
activation='relu'),
 tf.keras.layers.MaxPooling2D((3,3), strides=(2,2)),
 tf.keras.layers.BatchNormalization(),
 tf.keras.layers.Conv2D(384, (3,3), strides=(1,1),  padding='same',
activation='relu'),
 tf.keras.layers.Conv2D(384, (3,3), strides=(1,1),  padding='same',
activation='relu'),
 tf.keras.layers.Conv2D(256, (3,3), strides=(1,1),  padding='same',
activation='relu'),
 tf.keras.layers.MaxPooling2D((3,3), strides=(2,2), padding='same'),
 tf.keras.layers.Flatten(),
 tf.keras.layers.Dense(4096, activation='relu'),
 tf.keras.layers.Dropout(0.5),
 tf.keras.layers.Dense(4096, activation='relu'),
 tf.keras.layers.Dropout(0.5),
 tf.keras.layers.Dense(units=3, activation='softmax')
])
```

1. Input layer: Conv2D layer with 96 filters of size (11,11) with a stride of (4,4) and ReLU activation function. The input shape of the image is (64,64,3).
2. MaxPooling2D layer with pool size of (3,3) and stride of (2,2).
3. BatchNormalization layer to normalize the activations of the previous layer.
4. Conv2D layer with 256 filters of size (5,5) with a stride of (1,1), padding of 'same' and ReLU activation function.
5. MaxPooling2D layer with pool size of (3,3) and stride of (2,2).
6. BatchNormalization layer to normalize the activations of the previous layer.
7. Conv2D layer with 384 filters of size (3,3) with a stride of (1,1), padding of 'same' and ReLU activation function.

8. Conv2D layer with 384 filters of size (3,3) with a stride of (1,1), padding of 'same' and ReLU activation function.
9. Conv2D layer with 256 filters of size (3,3) with a stride of (1,1), padding of 'same' and ReLU activation function.
10. MaxPooling2D layer with pool size of (3,3) and stride of (2,2).
11. Flatten layer to flatten the output of the previous layer.
12. Dense layer with 4096 units and ReLU activation function.
13. Dense layer with 4096 units and ReLU activation function.
14. Dropout layer with a rate of 0.5 to prevent overfitting.
15. Dense layer with 3 units and softmax activation function for multiclass classification.



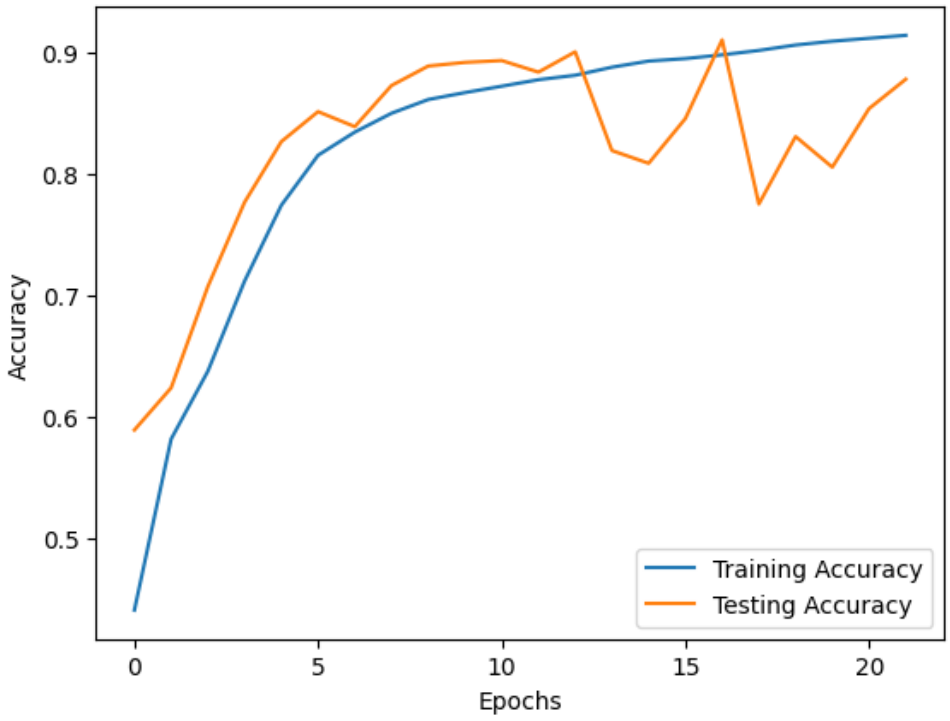
Both training and testing accuracy start from 0.3 and grow exponentially. Training keeps growing but you can see slight fluctuations in the testing graph.

As there is no early stopping the model is fluctuating in the end.

3. Discuss how you improve AlexNet, what methods and tools you have tried, and how that helped to improve the training accuracy and training time. Provide graphs that compare test and training accuracy on the same plot.

We used early stopping to prevent overfitting of the model. Early stopping works by preventing the model from continuing to learn and adapt to the training data beyond the point where it becomes too specialized and starts to overfit. By stopping the training process early, the model can achieve a better balance between fitting the training data and generalizing it to new data.

The model had achieved 90.08% but then it was going down the next epochs so after the patience of 5 the training was stopped preventing the model from overfitting.



We gave 30 epochs but the model stopped before that since we enabled early stopping to prevent overfitting.

Report for Part IV

1. Provide brief details about the nature of your dataset. What is it about? What type of data are we encountering? How many entries and variables does the dataset comprise? Provide the main statistics about the entries of the dataset. Provide at least 3 visualization graphs with short descriptions for each graph.

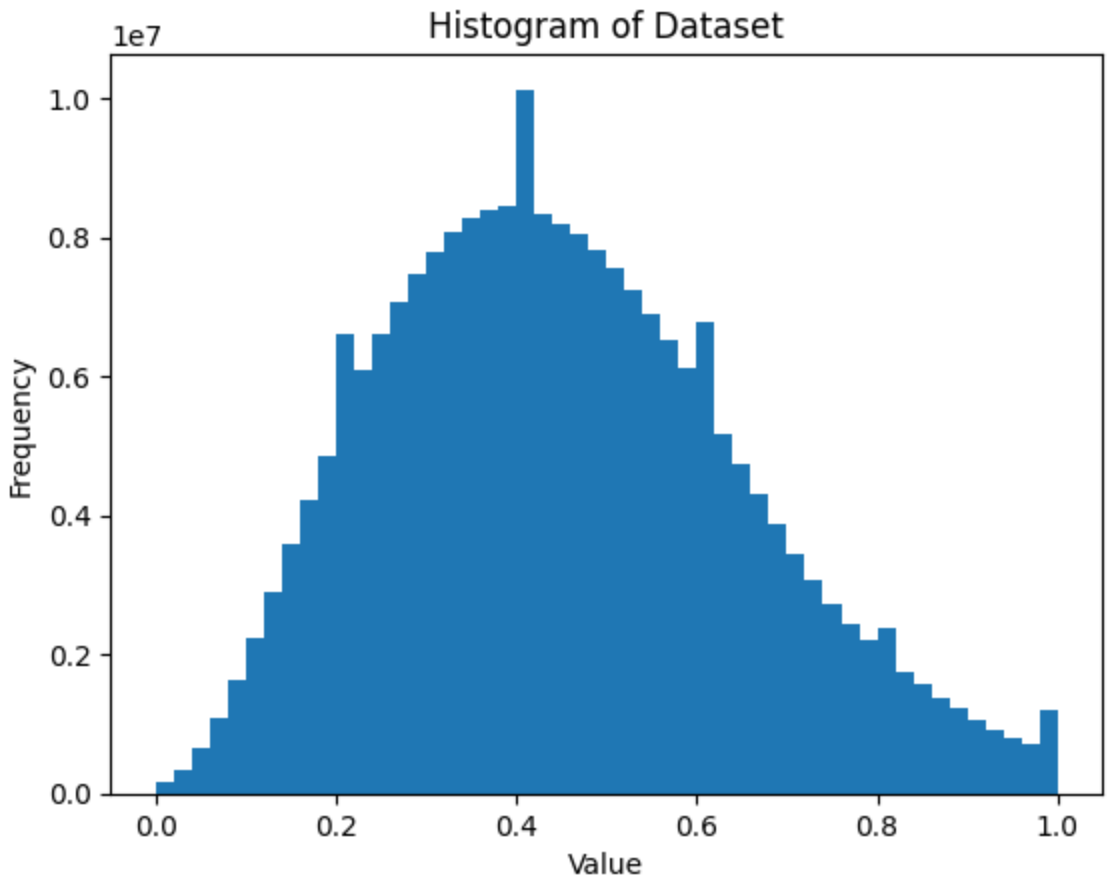
Statistics of the data set:

```
predicted_classes = np.argmax(train_labels, axis=1)
class_counts = np.bincount(predicted_classes)
print(class_counts)
```

```
[13861 10585  8497  7458  6882  5727  5595  5045  4659  4948]
```

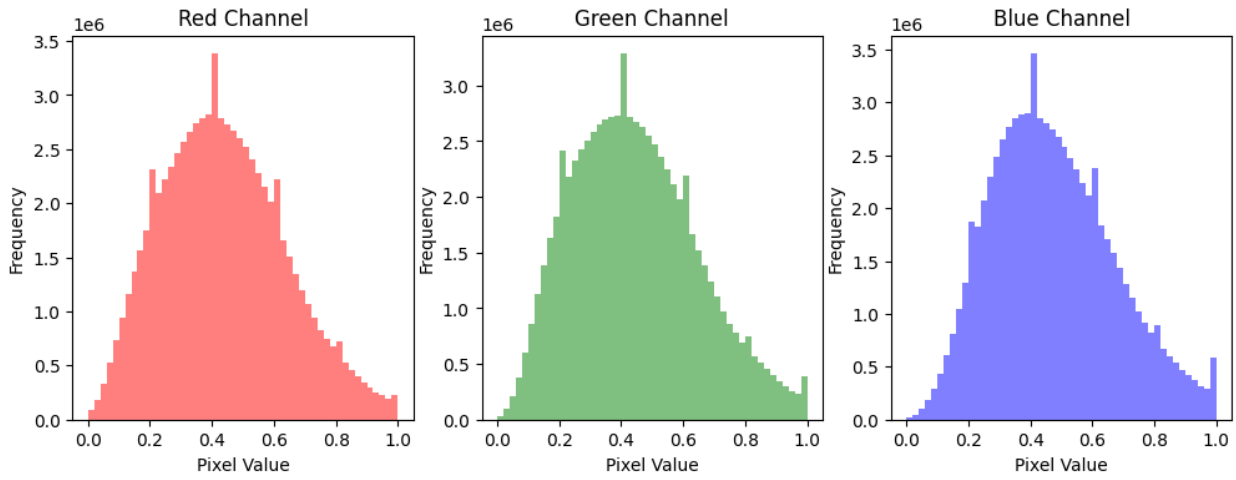
These are the no of images that belong to each class.

1) **Histogram of the pixel value of the images**



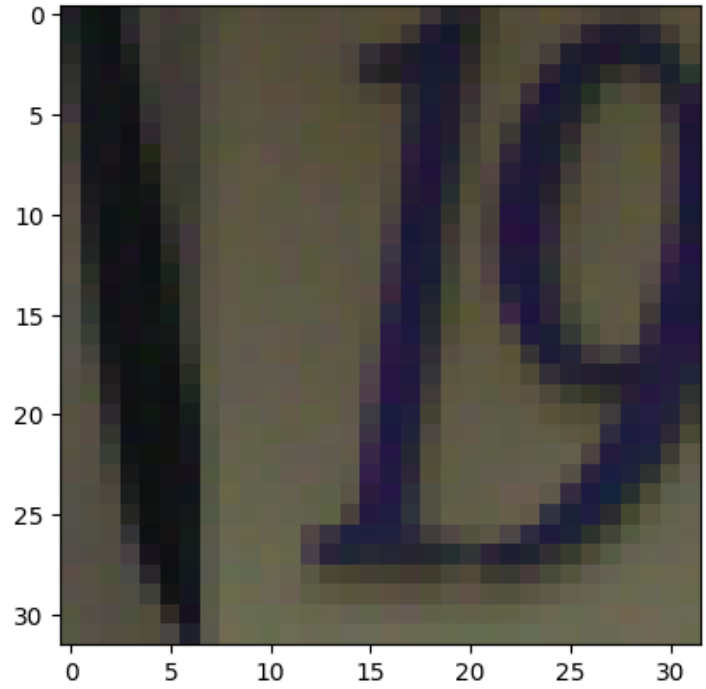
This shows the pixel intensity histogram of the images after flattening the train images.

2) Color Distribution of the images



This shows the pixel colur intensity.

3) Sample of images



This is a sample image of the training data images.

2. **Discuss briefly how you adjusted your model from Part III for this task.**

```
model = tf.keras.models.Sequential([
 tf.keras.layers.Conv2D(32, (3,3), strides=(1,1), activation='relu',
input_shape=(32,32,3)),
 tf.keras.layers.MaxPooling2D((3,3), strides=(2,2)),
```

```
 tf.keras.layers.BatchNormalization(),
 tf.keras.layers.Conv2D(256, (5,5), strides=(1,1),  padding='same',
activation='relu'),
 tf.keras.layers.MaxPooling2D((3,3), strides=(2,2)),
 tf.keras.layers.BatchNormalization(),
 tf.keras.layers.Conv2D(64, (3,3), strides=(1,1),  padding='same',
activation='relu'),
 tf.keras.layers.Conv2D(64, (3,3), strides=(1,1),  padding='same',
activation='relu'),
 tf.keras.layers.Conv2D(64, (3,3), strides=(1,1),  padding='same',
activation='relu'),
 tf.keras.layers.MaxPooling2D((3,3), strides=(2,2), padding='same'),
 tf.keras.layers.Flatten(),
 tf.keras.layers.Dense(128, activation='relu'),
 tf.keras.layers.Dropout(0.5),
 tf.keras.layers.Dense(128, activation='relu'),
 tf.keras.layers.Dropout(0.5),
 tf.keras.layers.Dense(units=10, activation='softmax')
])
```

This model has 5 convolutional layers, 2 max pooling layers, and 3 fully connected layers. The input shape of the model is (32, 32, 3), and the model produces an output of shape (10,) with a softmax activation function.

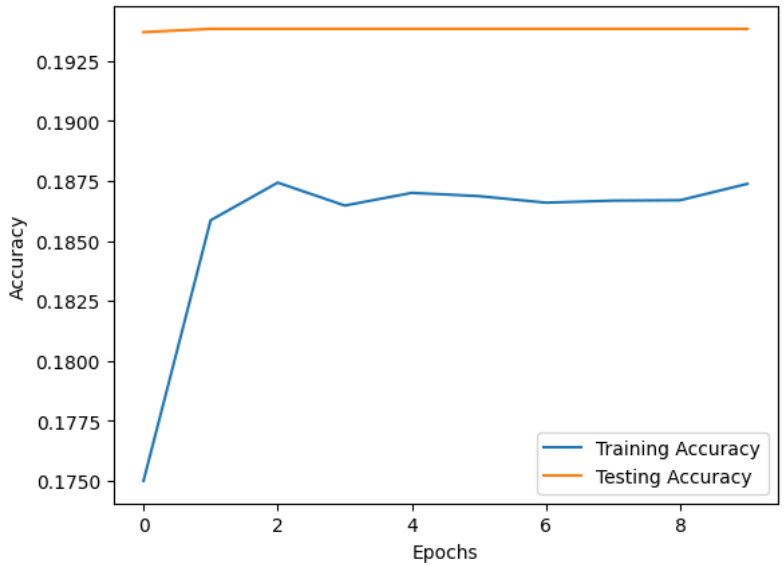
The model in part 3 has 6 convolutional layers, 2 max pooling layers, and 3 fully connected layers. The input shape of the model is (64, 64, 3), and the model produces an output of shape (3,) with a softmax activation function.

We realized that this dataset requires a model with lesser complexity and hence we reduced the number of convolutional layers to 5.

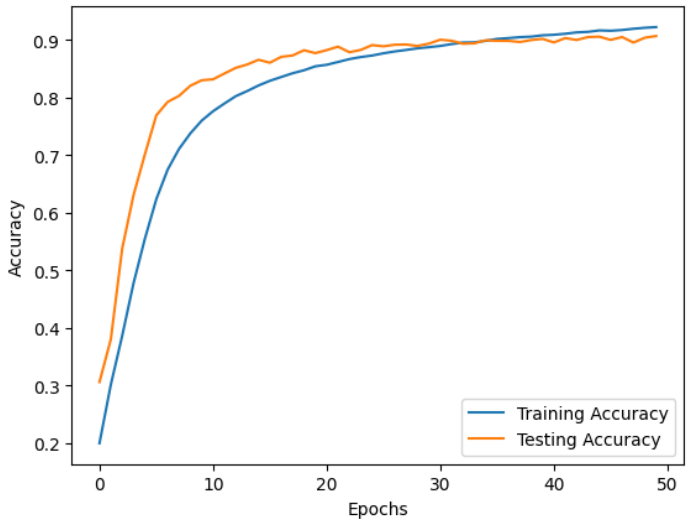
3. **Discuss data augmentation methods you tried and reason how they helped to increase the accuracy of the model.**

```python
import numpy as np
from PIL import Image
horizontalflip=True
rescale=1./255
width_shift_range=0.2
height_shift_range=0.2
zoom_range = 0.2
def apply_augmentation(images,labels):
  augmented_images = []
  augmented_labels = []
  for img,label in zip(images,labels):
    augmented_images.append(np.array(img))
    augmented_labels.append(label)
    rotated_img = np.fliplr(img)
    augmented_images.append(rotated_img)
    augmented_labels.append(label)
  return np.array(augmented_images),np.array(augmented_labels)
```

The training process without augmentation was not great as the accuracy was fluctuating near 0.192



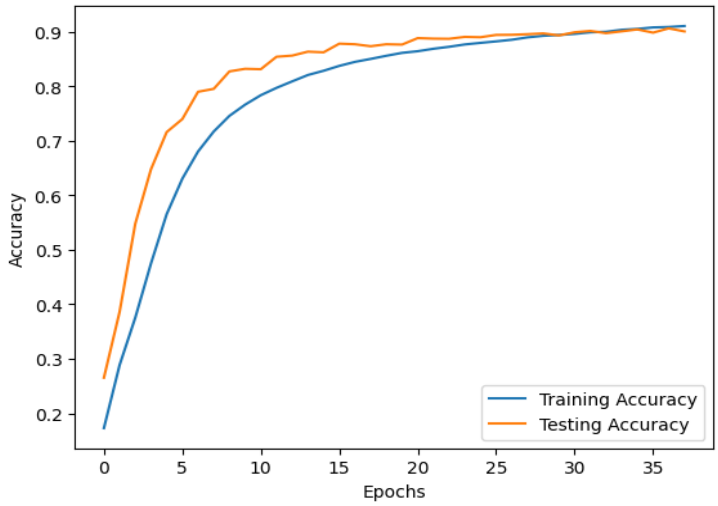
We are applying the data augmentation using the above logic. We are simply flipping the images horizontally to double the number of images.



We obtain an accuracy greater than 0.964 after augmenting the images.

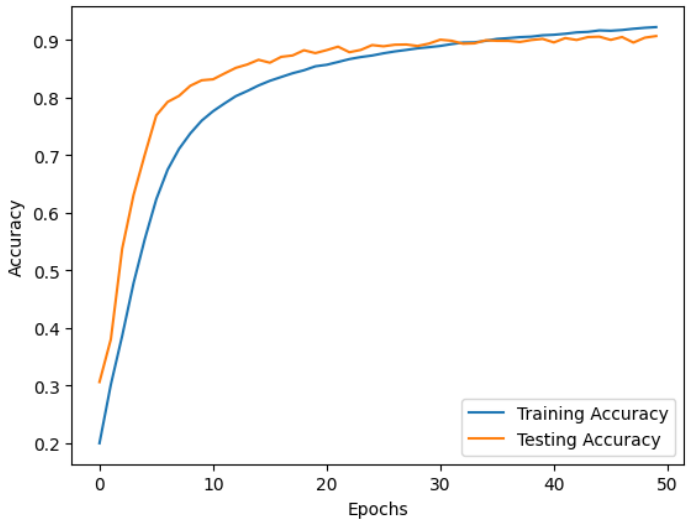
4. **Provide graphs that compare test and training accuracy on the same plot for all your setups and add a short description for each graph.**

Below is the graph when we enabled early stopping:



The model stopped around epoch 35 even though we gave 50 epochs. The maximum reached is around 0.9

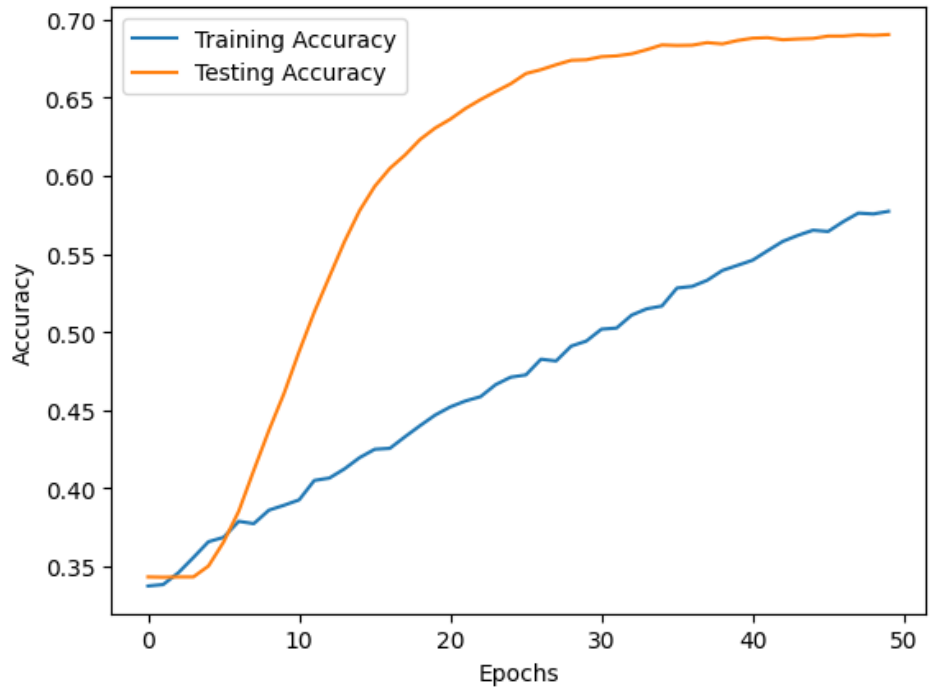
Without early stopping:



Since early stopping was disabled here, the model ran for all 50 epochs. The maximum accuracy achieved here was also around 0.964 but it was fluctuating in the end

BONUS TASK

In this, we have completely implemented the VGG - 13 Architecture and trained it on the Data set provided and achieved an accuracy of 69% on the test data set.



Contributions:

Team Member	Assignment Part	Contribution (%)
Ritesh Manchikanti	Part 1, 2, 3, 4 and bonus	50% and 50%
Anurima Vaishnavi Kumar	Part 1,2,3,4 and bonus	50% and 50%