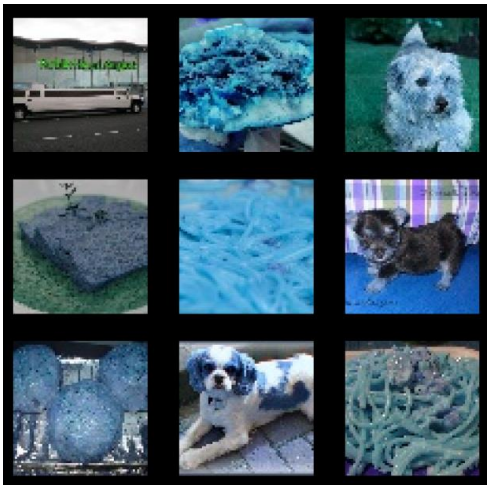
Part 3

1.

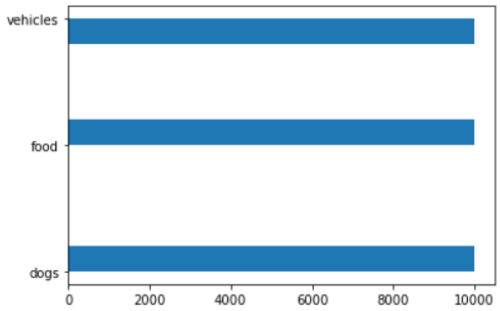
The given dataset (cnn_datset) is a zip file which upon extraction has 3 folders containing 10,000 images in each folder. It is a Image dataset consisting of 10000 food images, 10000 dogs images and 10000 vehicles images. Each image size lies around 4kb on disk. Each image is of 64 X 64 pixels. We have 30000 entries in total and have 3 variables (food, dogs, vehicles)

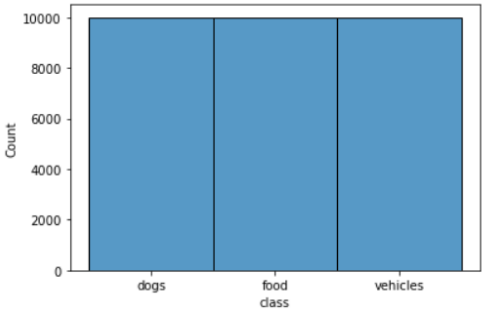
```
import cv2
random_pick = list(map(lambda x: df.iloc[x] , [k for k in random.sample(range(len(df)),10)]))
```

```
def plot_random_images(iterator):   #iterator has image paths

    plt.figure(figsize=(20,20))
    plt.subplot(331)
    plt.title(iterator[0][0])
    plt.imshow(cv2.imread(iterator[0][1]))
    plt.subplot(332)
    plt.title(iterator[1][0])
    plt.imshow(cv2.imread(iterator[1][1]))
    plt.subplot(333)
    plt.title(iterator[2][0])
    plt.imshow(cv2.imread(iterator[2][1]))
    plt.subplot(334)
    plt.title(iterator[3][0])
    plt.imshow(cv2.imread(iterator[3][1]))
    plt.subplot(335)
    plt.title(iterator[4][0])
    plt.imshow(cv2.imread(iterator[4][1]))
    plt.subplot(336)
    plt.title(iterator[5][0])
    plt.imshow(cv2.imread(iterator[5][1]))
    plt.subplot(337)
    plt.title(iterator[6][0])
    plt.imshow(cv2.imread(iterator[6][1]))
```



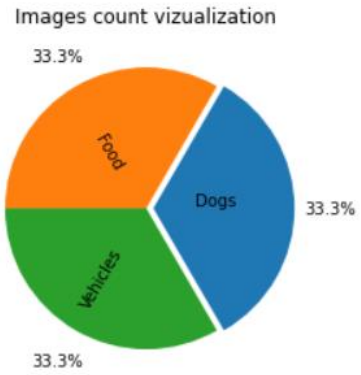
The above code randomly picks 9 images from the datset and prints them using plotly's imshow and open cv is used to open the path of image and convert it into image Tensor.





The above graphs are barplots of the entries we have. We got 10,000 images of each variable(food, dogs, vehicles)

```
lables_class = ["Dogs","Food","Vehicles"]

figure, axis = plt.subplots()
axis.pie(df["class"].value_counts(),labels=lables_class,autopct='%1.1f%%',
       pctdistance=1.25, labeldistance=0.3,rotatelabels=20,explode=(0.05,0,0),startangle=300)
plt.title("Images count vizualization")
plt.show()
```



This is pie graph/chart which shows data visualization in a circle with the percentage of each distribution.

2.

AlexNet is a deep convolutional neural network architecture designed for image classification. It has Eight different layers. The last three layers are fully connected layers, while the first five layers are convolutional layers. Layers for ReLU activation and max-pooling are placed after the convolutional layers. The adaptive average pooling layer comes before the fully connected layers as well.

Below code is of the base model without any code part for improvements. The forward method defines the network's forward pass by iterating through the convolutional and fully connected layers and returning the output tensor.

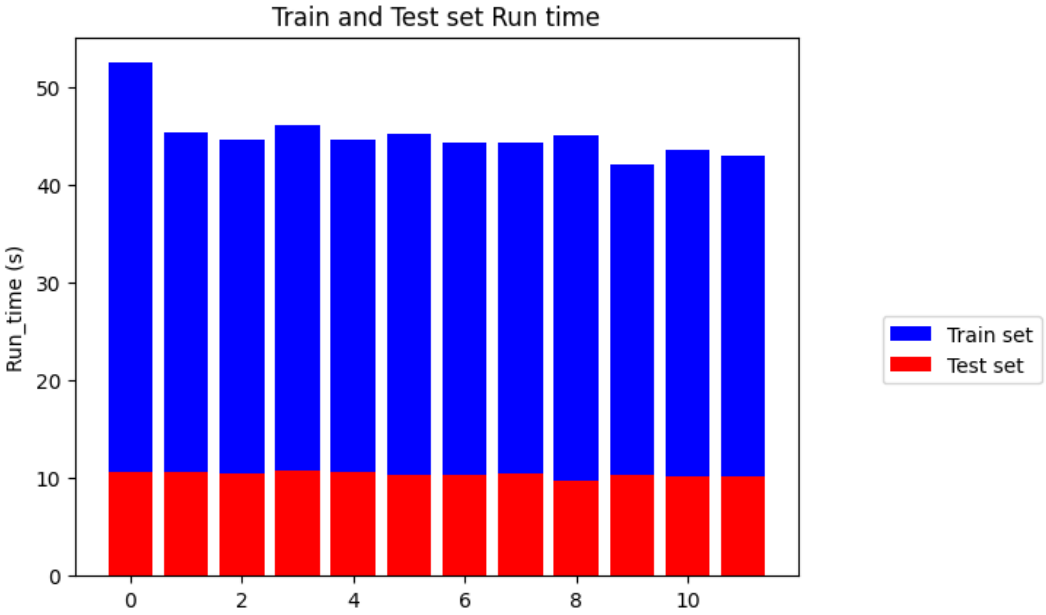
The self.features module defines the convolutional layers, which are a sequential container of convolutional, activation (ReLU), and max pooling layers. The first convolutional layer receives a (RGB) tensor and applies a filter of 96 11x11 kernels with a stride of four and padding of two. The next convolutional layers are defined in the same

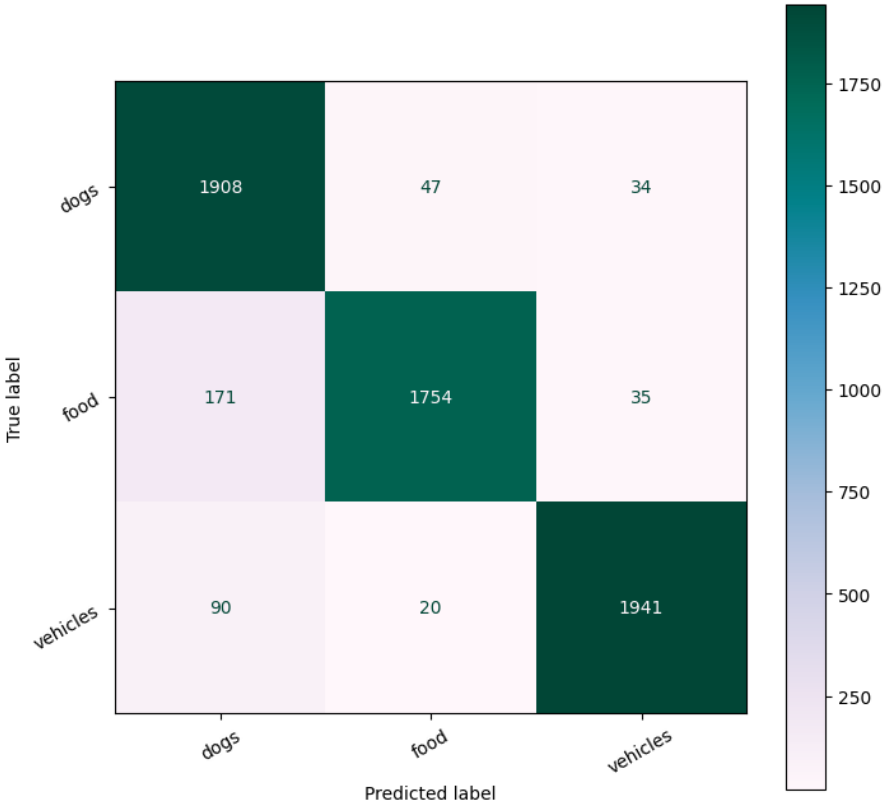
way, with decreasing kernel size and increasing number of output channels to match input of next layerds. ReLU activation is used after each convolutional layer, followed by max pooling with a kernel size of 3 and stride of 2. The self.avgpool module applies adaptive average pooling to the final convolutional layer's output. It resizes the input tensor to 6x6.

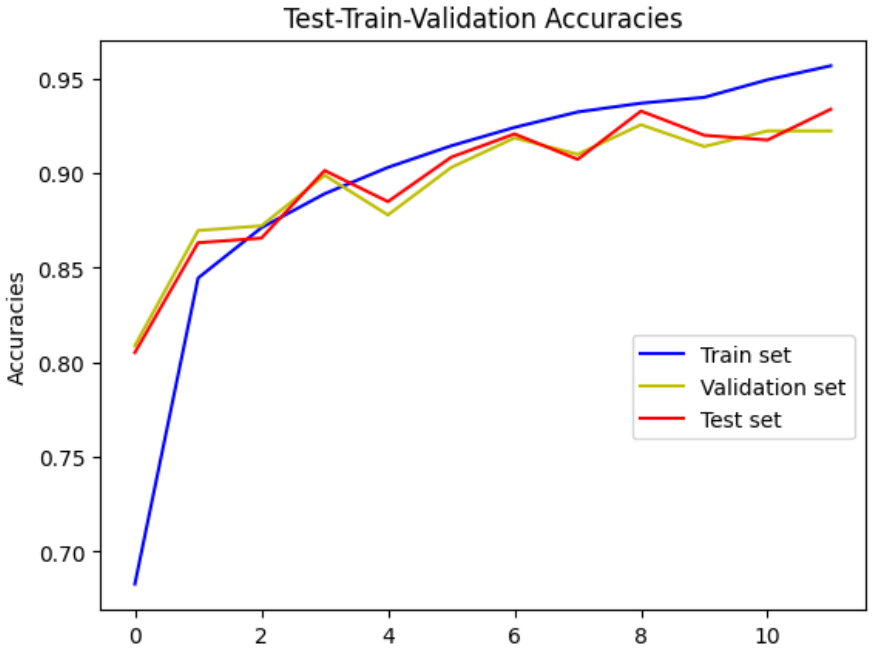
```
# Alexnet code implementation!

class AlexNet(nn.Module):
  def __init__(self, output_class = class_count):
      super().__init__()
      self.features = nn.Sequential(
          nn.Conv2d(in_channels=3, out_channels=96,kernel_size=11,stride=4,padding=2,bias=False),
          nn.ReLU(inplace=True),
          nn.MaxPool2d(kernel_size=3,stride=2,padding=0),

          nn.Conv2d(in_channels=96, out_channels=192,kernel_size=5,stride=1,padding=2,bias=False),
          nn.ReLU(inplace=True),
          nn.MaxPool2d(kernel_size=3,stride=2,padding=0),

          nn.Conv2d(in_channels=192, out_channels=384,kernel_size=3,stride=1,padding=1,bias=False),
          nn.ReLU(inplace=True),

          nn.Conv2d(in_channels=384, out_channels=256,kernel_size=3,stride=1,padding=1,bias=False),
          nn.ReLU(inplace=True),

          nn.Conv2d(in_channels=256, out_channels=256,kernel_size=3,stride=1,padding=1,bias=False),
          nn.ReLU(inplace=True),
          nn.MaxPool2d(kernel_size=3,stride=2,padding=0),
          )
```

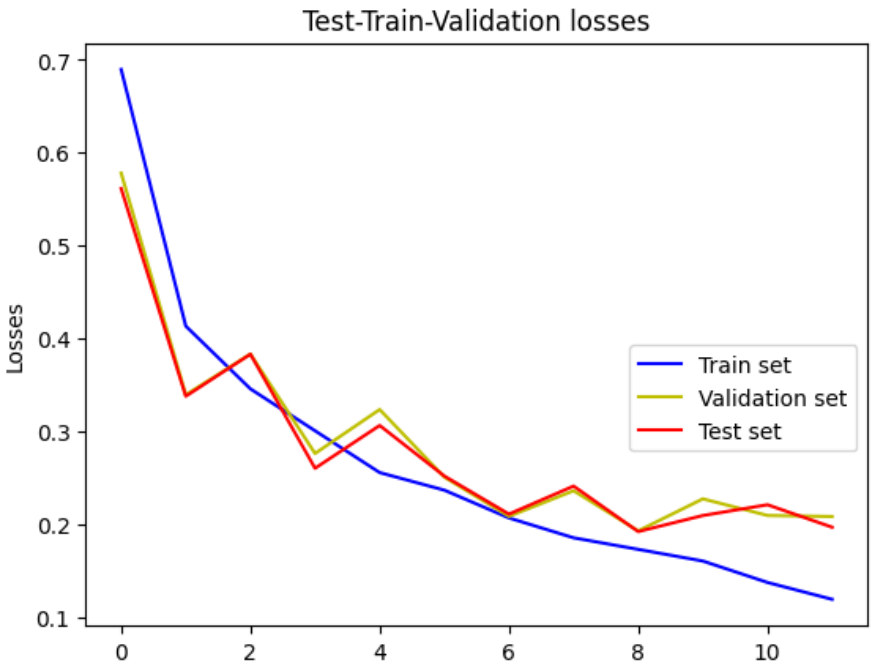
```
      self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
      self.classifier = nn.Sequential(
          nn.Dropout(),
          nn.Linear(in_features=256*6*6,out_features=4096),
          nn.ReLU(inplace=True),
          nn.Dropout(),
          nn.Linear(in_features=4096,out_features=4096),
          nn.ReLU(inplace=True),
          nn.Linear(in_features=4096,out_features=output_class),
        )

  def forward(self,x):
    x = self.features(x)
    x = self.avgpool(x)
#     print(f"Shape of input after 5 Convolution layers : {x.shape}")
    x = x.view(x.size(0),256*6*6)
#     print(f"Shape of x after viewing : {x.shape}")
    x = self.classifier(x)
    return x
```

Due to computational restrictions we ran the model for 12 epochs. The following results are for the Base model(Model with no modification) follows AlexNet architecture. Best Test Accuracy is achieved at 12^{th} epoch which is 93%.


Train and Test set Run time
Run_time (s)
0
10
20
30
40
50
0
2
4
6
8
10
Train set
Test set


True label
dogs
food
vehicles
1908
47
34
171
1754
35
90
20
1941
dogs
food
vehicles
Predicted label
1750
1500
1250
1000
750
500
250


Test-Train-Validation Accuracies
Accuracies
0.95
0.90
0.85
0.80
0.75
0.70
0
2
4
6
8
10
Train set
Validation set
Test set


Test-Train-Validation losses
Losses
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
2
4
6
8
10
Train set
Validation set
Test set

3.

Five methods are implemented for improving Alex net model.

1) Weights Initializer using kaiming and xavier normal dist.

```
model_alex_1 = AlexNet(output_class=3).to(device)

# Function to initialize parameters
def params_initialization(layer):
    if isinstance(layer,nn.Conv2d):
        print(type(layer),"True")
        nn.init.kaiming_normal_(layer.weight.data , nonlinearity="relu")
        nn.init.constant_(layer.bias.data,val=0)
    elif isinstance(layer, nn.Linear):
        nn.init.xavier_normal_(layer.weight.data,gain=nn.init.calculate_gain("relu"))
        nn.init.constant_(layer.bias.data,val=0)

params_initialization(model_alex_1)
```

Kaiming and Xavier are two commonly used initialization methods for deep neural networks. Convolutional layers are initiallized with Kaiming initialization (alias He) And Linear layers are initiallized with xavier initialization. Kaiming uses mean : 0 and standard deviation : sqrt(2/n), n is no.of inputs to the layer. While xavier uses mean :0 and standard deviation : sqrt(1/n). It is used to avoid vanishing gradient problems

2) Early stopping based on validation loss :

The concept of early stopping is to stop at best parameters rather than running all epochs and overfitting the model. Validation loss quantifies the model's error using a different validation dataset that isn't used for training. In our case it is 10% of the train dataset. The validation loss is determined at the end of each epoch when the model is trained. If the validation loss begins to grow after a given number of epochs (like 5 epochs or so consecutively), the model is likely to overfit.

We can implement early stopping in various ways:

```
if validation_loss < loss_best :  # first iteration always holds True
    loss_best = validation_loss        # we don't know by how much margin the cha

elif (loss_best - validation_loss) <= delta:
    trigger+= 1
    # checking trigger >= patience every time trigger variable is triggered
    if trigger >= patience :
        print()
        print(f"Early stopping implemented! Loop stopped at Epoch {epoch}!")
        print()
        torch.save(model_alex_2.state_dict(),"model_best_state_2.pt")
        break

elif (loss_best - validation_loss) > delta:
    loss_best = validation_loss
    trigger = 0  # trigger reset! It should happen 3 times in a row!
    torch.save(model_alex_2.state_dict(),"model_best_state_2.pt")
```

```
patience = 3 ################   3 times (since we only have 12 epochs) the validation loss
delta = 0.1 ################  here delta is minium change of loss to stop the epoch loop!
trigger = 0 ################  counter variable
```

3) Gradient clipping

Gradients can become very big or very tiny during training, causing training instability or slowing down the learning process. Gradient clipping is a technique used to prevent these concerns by limiting the size of the gradients.

We are using inbuilt function to implement the gradient clipping. It is implemented with model training.

```
def trainModel_GC(model,iterator,optimizer,criterion,device):
    epoch_loss = 0
    epoch_acc = 0
    model.train()

    for x,y in iterator:
        x = x.to(device)   # images
        y = y.to(device)   # class - labels
        optimizer.zero_grad()

        y_pred = model(x)

        loss = criterion(y_pred,y)
        acc = model_accuracy(y_true=y,y_pred=y_pred)

        loss.backward()
        nn.utils.clip_grad_norm_(model_alex_3.parameters(), max_norm=1.0)  ######### gradient clipping
        optimizer.step()
        epoch_loss += loss.item()
        epoch_acc += acc.item()
```

4) Dropouts

Dropout is a machine learning regularization technique intended to minimize overfitting. It works by randomly removing a particular number of neurons in a neural network during training. This reduces overfitting by forcing the network to acquire more robust and generalizable features rather than depending too much on a single set of characteristics rather say remembering the pattern . Each neuron has a chance/probability ‘p’ of being dropped out during training, which is commonly set at 0.5.

```
self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
self.classifier = nn.Sequential(
    nn.Dropout(p=dropout_prob),
    nn.Linear(in_features=256 * 6 * 6, out_features=4096),
    nn.ReLU(inplace=True),
    nn.Dropout(p=dropout_prob),
    nn.Linear(in_features=4096, out_features=4096),
    nn.ReLU(inplace=True),
    nn.Linear(in_features=4096, out_features=output_class),
)
```

Dropout probability set to 0.5.

5) Batch Normalization

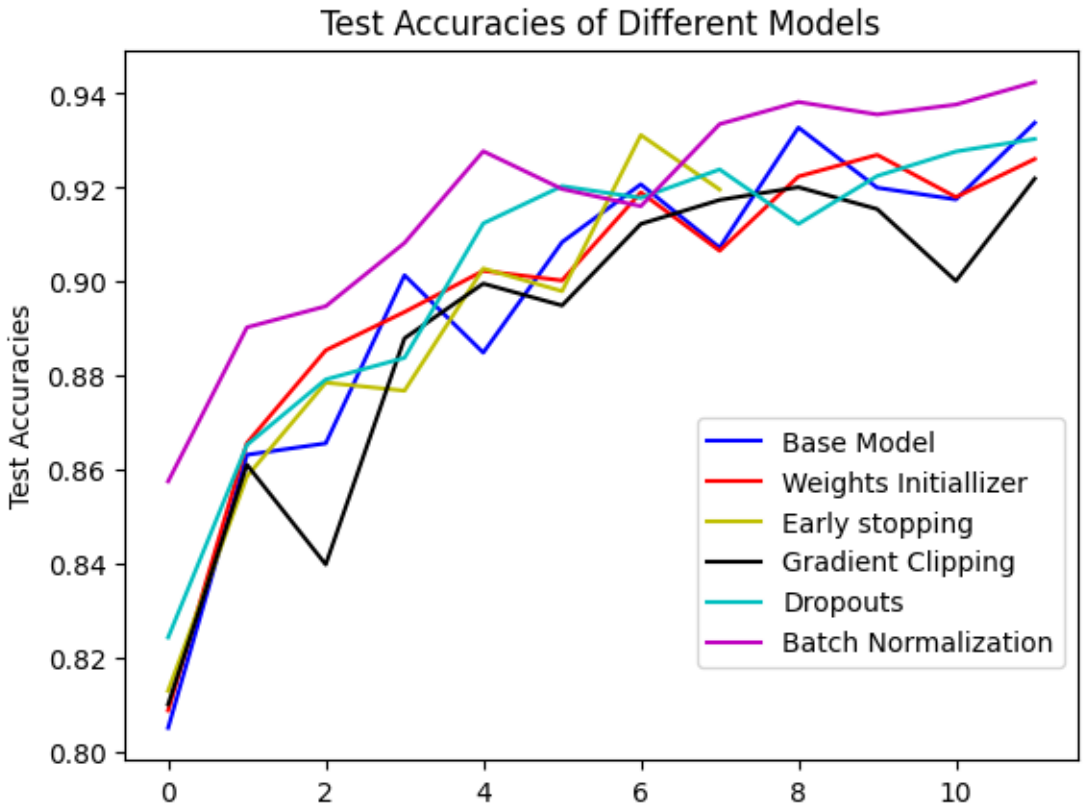
Batch normalization is a deep learning approach that improves neural network performance and stability by normalizing the input layer by modifying and scaling the activations. During training, each mini-batch is normalized, with the mean and variance of the inputs determined and the inputs standardized using these values. Batch normalization reduces the network's reliance on weight initialization by normalizing the input, resulting in faster convergence and greater generalization. It also prevents overfitting.

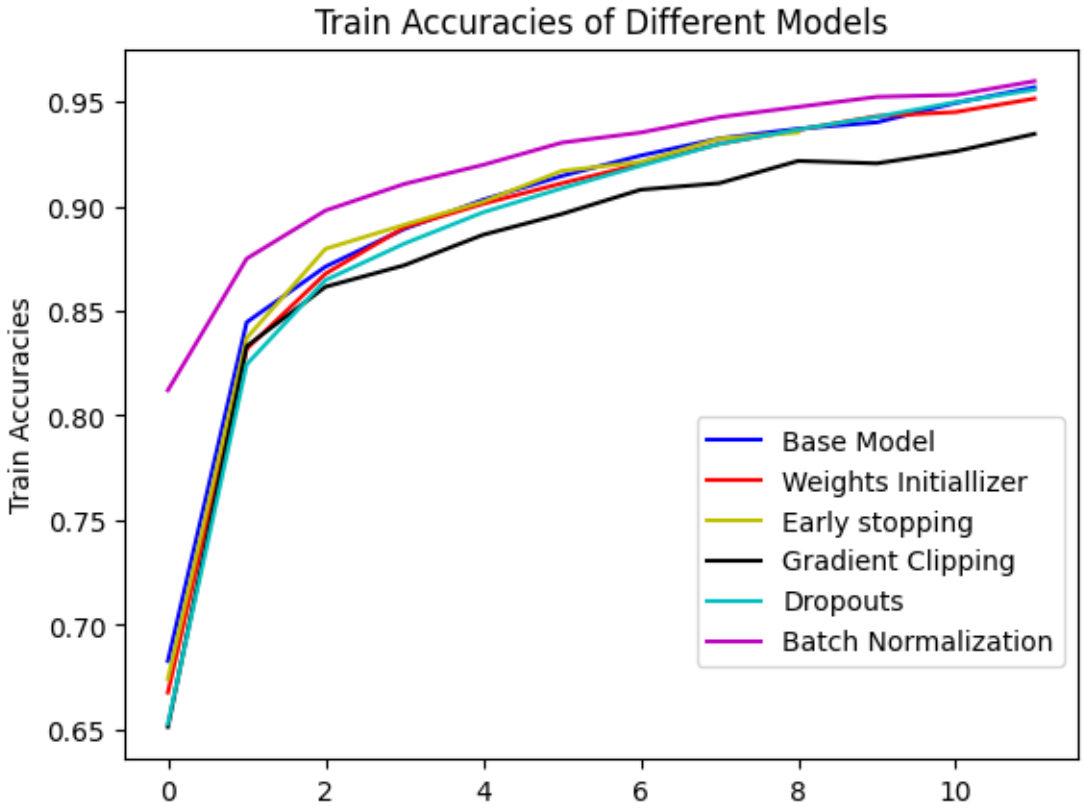
```
class AlexNet_BN(nn.Module):
    def __init__(self, output_class = class_count):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=96,kernel_size=11,stride=4,padding=2,bias=False),
            nn.BatchNorm2d(num_features=96),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3,stride=2,padding=0),

            nn.Conv2d(in_channels=96, out_channels=192,kernel_size=5,stride=1,padding=2,bias=False),
            nn.BatchNorm2d(num_features=192),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3,stride=2,padding=0),

            nn.Conv2d(in_channels=192, out_channels=384,kernel_size=3,stride=1,padding=1,bias=False),
            nn.BatchNorm2d(num_features=384),
            nn.ReLU(inplace=True),

            nn.Conv2d(in_channels=384, out_channels=256,kernel_size=3,stride=1,padding=1,bias=False),
            nn.BatchNorm2d(num_features=256),
            nn.ReLU(inplace=True),

            nn.Conv2d(in_channels=256, out_channels=256,kernel_size=3,stride=1,padding=1,bias=False),
            nn.BatchNorm2d(num_features=256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3,stride=2,padding=0),
        )
```

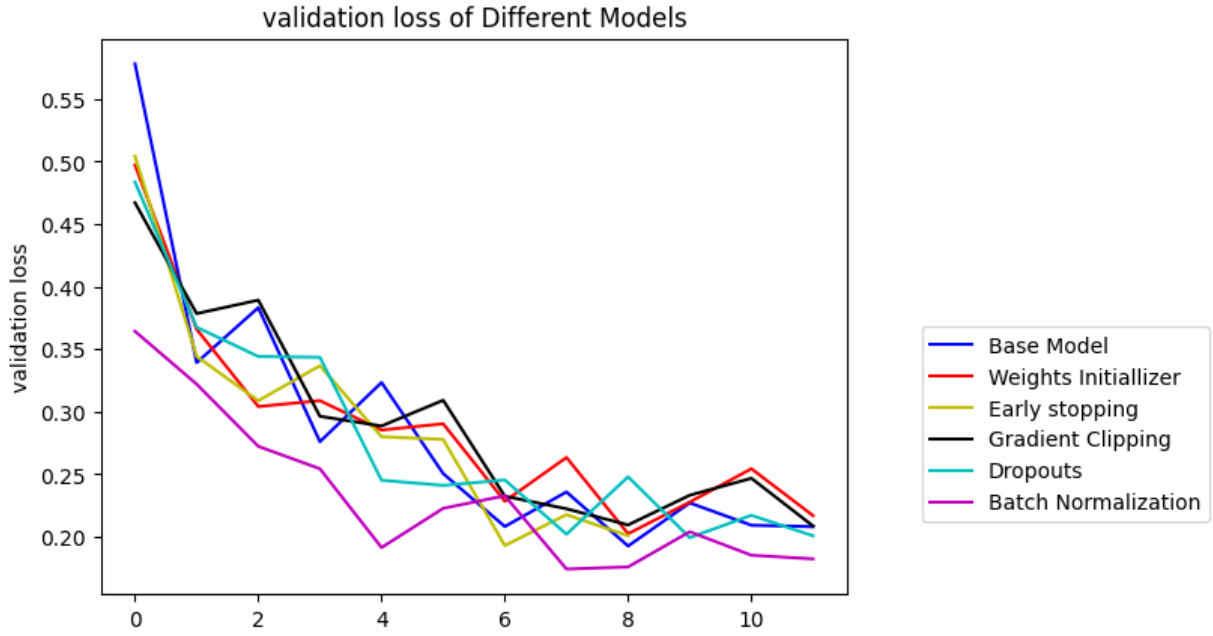
```
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(in_features=256*6*6,out_features=4096),
            nn.BatchNorm1d(num_features=4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(in_features=4096,out_features=4096),
            nn.BatchNorm1d(num_features=4096),
            nn.ReLU(inplace=True),
            nn.Linear(in_features=4096,out_features=output_class),
        )
```

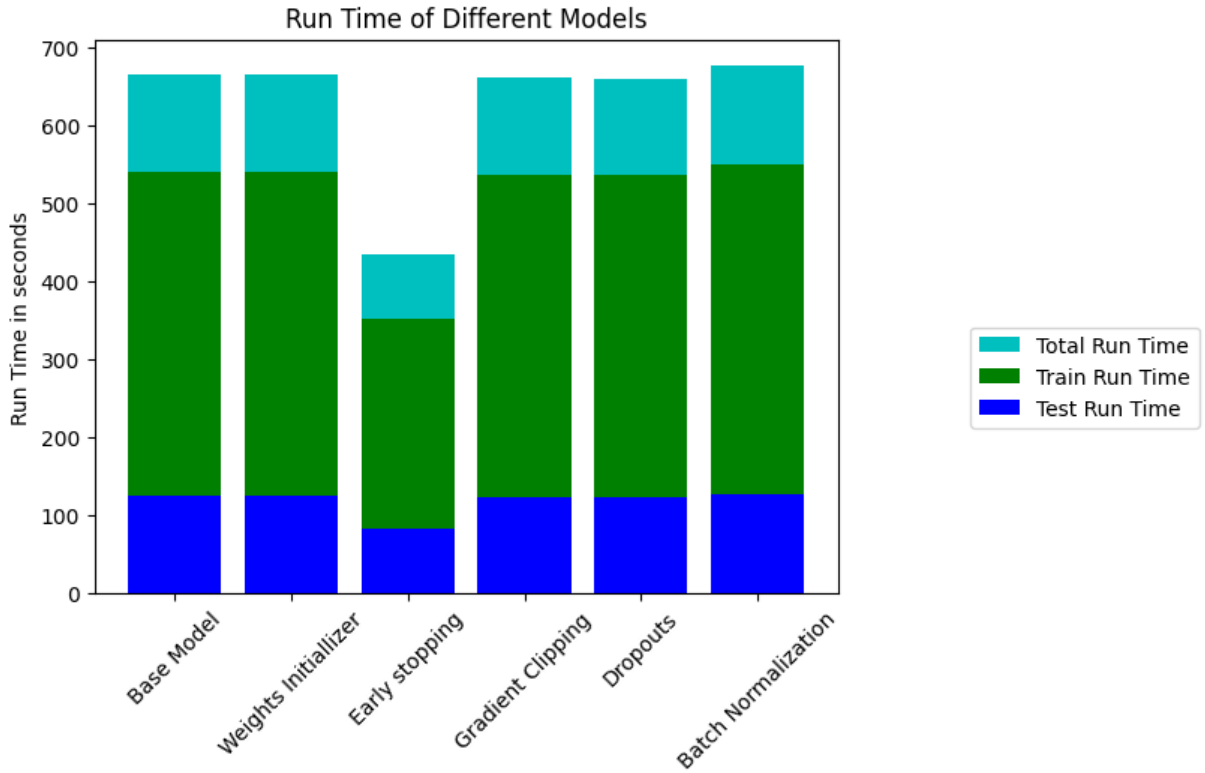
Overall there is an improvement in accuarcy but the training time kinda remained same.

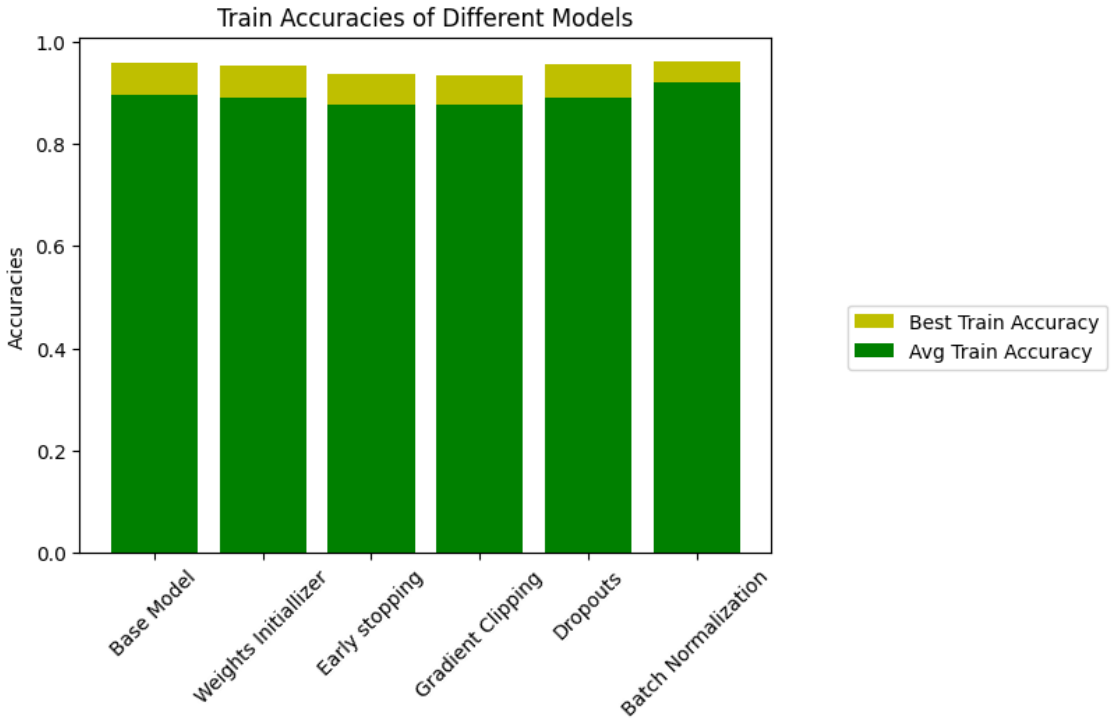
Below are different graphs on train/validation/test data about accuracy/loss/run time.

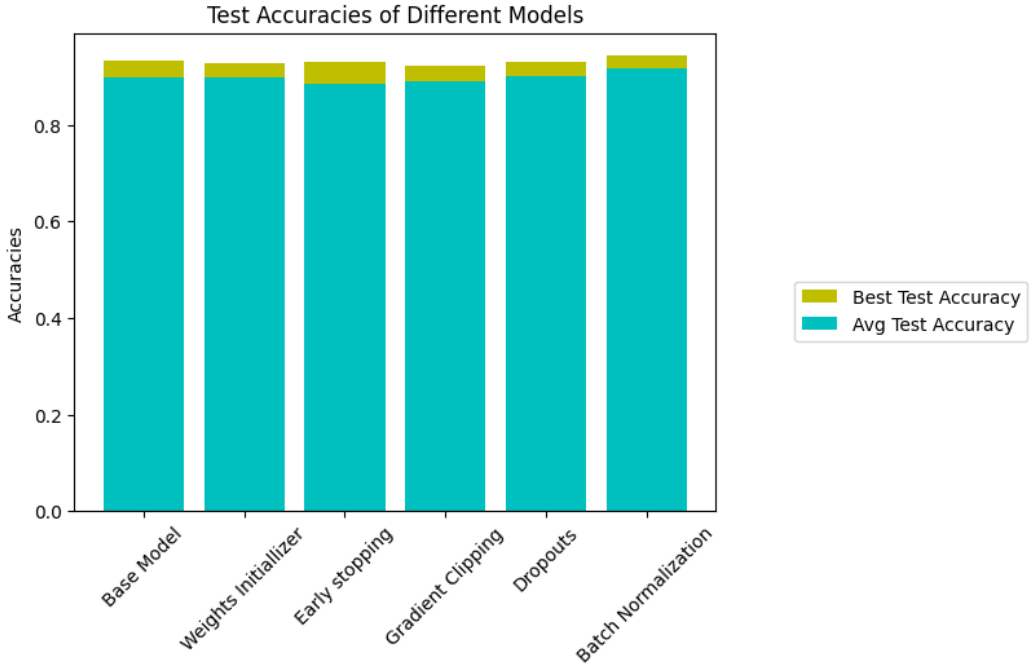

Test Accuracies of Different Models
Test Accuracies
0.94
0.92
0.90
0.88
0.86
0.84
0.82
0.80
0
2
4
6
8
10
Base Model
Weights Initiallizer
Early stopping
Gradient Clipping
Dropouts
Batch Normalization


Train Accuracies of Different Models
Train Accuracies
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0
2
4
6
8
10
Base Model
Weights Initiallizer
Early stopping
Gradient Clipping
Dropouts
Batch Normalization


validation loss of Different Models
validation loss
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0
2
4
6
8
10
Base Model
Weights Initiallizer
Early stopping
Gradient Clipping
Dropouts
Batch Normalization


Run Time of Different Models
Run Time in seconds
700
600
500
400
300
200
100
0
Base Model
Weights Initiallizer
Early stopping
Gradient Clipping
Dropouts
Batch Normalization
Total Run Time
Train Run Time
Test Run Time

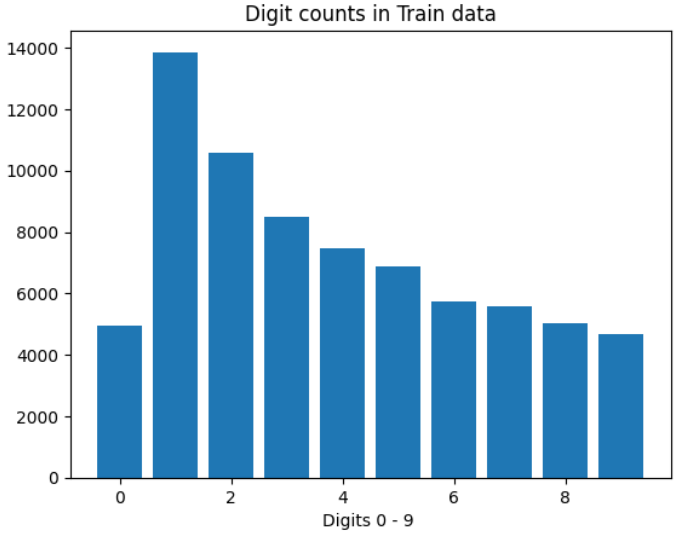




Best Test Accuracy is achieved by Model 5(Batch Normalization) which is 94%.

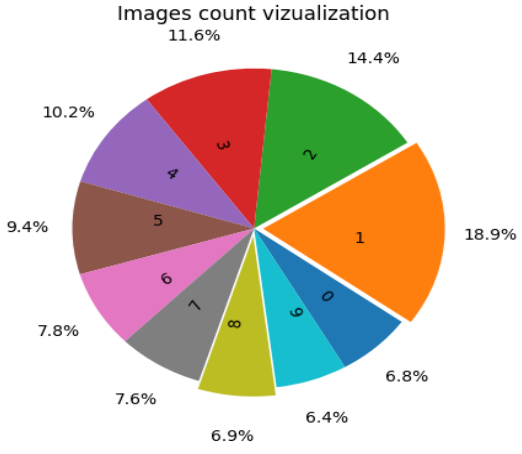
Part 4

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 description for each graph.



This graph shows the distribution of the number of images per class in the training set of the SVHN dataset. It is a pie chart with each slice representing a different digit class.

This graph shows a 3x3 grid of sample images from the SVHN dataset. Each image is labeled with the digit that it represents.



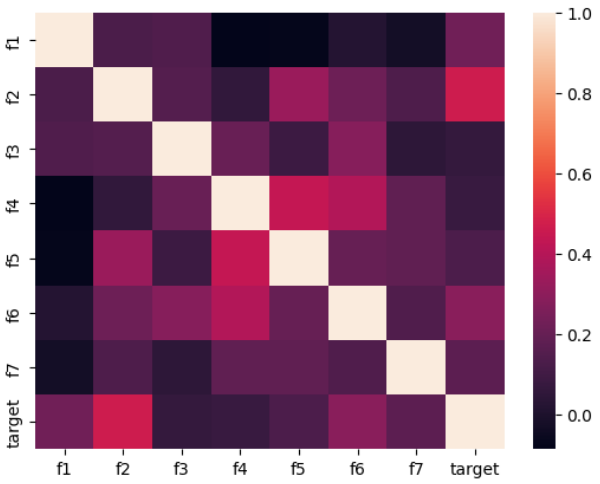
This graph shows the color distribution of the SVHN dataset. It is a histogram with 256 bins, one for each possible value of a pixel. The histogram shows the frequency of each pixel value across all images in the dataset.

Assignment 2

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.

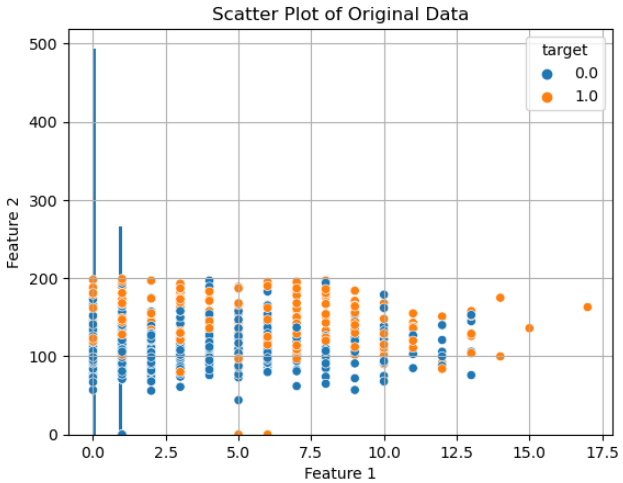
```
1 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 766 entries, 0 to 765
Data columns (total 8 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   f1      766 non-null    object
 1   f2      766 non-null    object
 2   f3      766 non-null    int64
 3   f4      766 non-null    object
 4   f5      766 non-null    object
 5   f6      766 non-null    object
 6   f7      766 non-null    object
 7   target  766 non-null    int64
dtypes: int64(2), object(6)
memory usage: 48.0+ KB
```

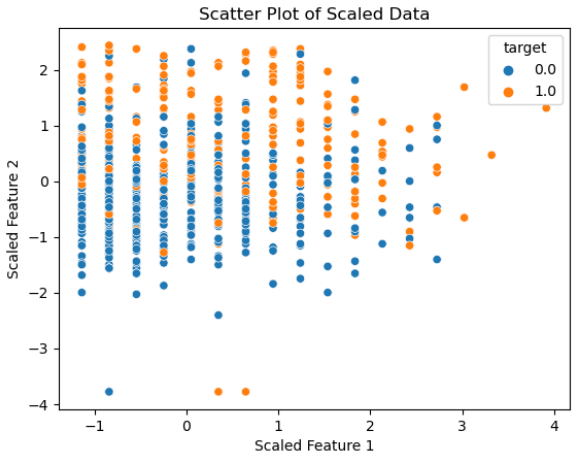


The dataset contains a mix of data types, including 2 columns of integer data (f3 and target) and 6 columns of object data (which typically represent string or categorical data). Some basic preprocessing steps to ensure that the dataset is properly formatted and ready for use in machine learning models. Specifically, it is removing any rows with non-numeric values, converting the data types of certain columns to float and removing columns with missing values.

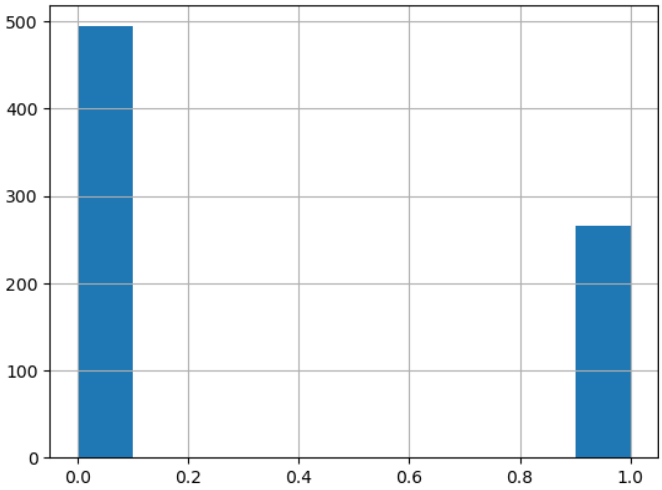
2. Provide at least 3 visualization graphs with short description for each graph.



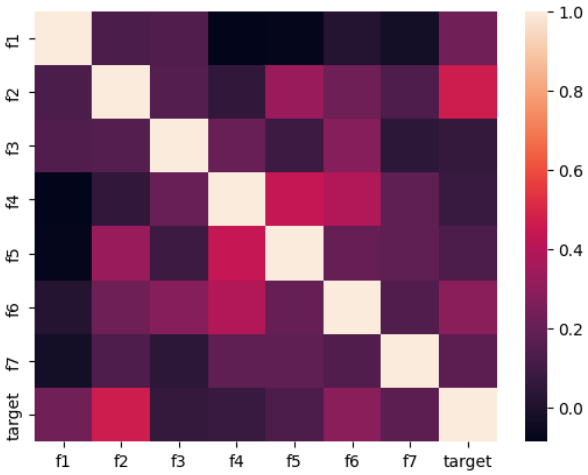
In the first plot, you'll see the scatter plot of the original data, where the two features are plotted against each other, and the color of each point represents the target value.



the scatter plot of the scaled data, where the two scaled features are plotted against each other, and the color of each point represents the target value.



Shows the distribution of target variable values in the dataset.



This is a heat map plotted using seaborn library. It is a visual representation of corelation coefficients, how good are columns related to each other and the target varaible

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

the numerical features in X are first scaled using the StandardScaler, which scales the data to have zero mean and unit variance. The scaled data is then converted to PyTorch tensors and split into training and test sets. The training and test sets are loaded into data loaders, and a neural network model is defined using SimpleBinaryClassifier. Overall, the preprocessing step of scaling the input features can help to improve the accuracy of the model by ensuring that all features are treated equally and reducing the impact of outliers.

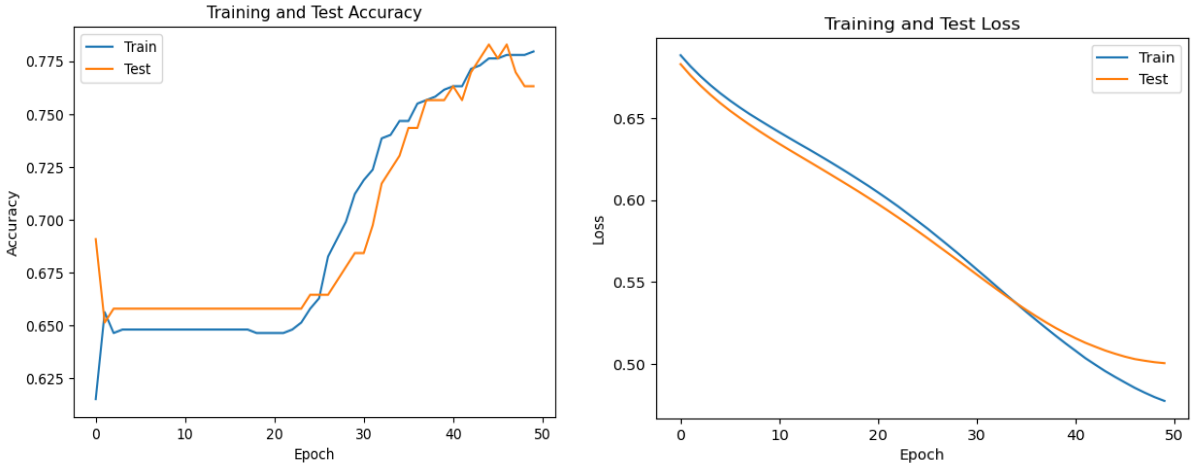
```
5
6  # Scale numerical features using StandardScaler
7  scaler = StandardScaler()
8  X = scaler.fit_transform(X)
9  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
10 # print(type(X_train),type(X_test),type(y_train),type(y_test))
```

4. Provide the architecture structure of your NN.

The input layer of the network has 7 nodes, which corresponds to the number of input features in the dataset. The network consists of four hidden layers with 128, 128, 64, and 1 nodes, respectively. The first three hidden layers use the Rectified Linear Unit (ReLU) activation function, which is a common choice for hidden layers in neural networks. The final layer of the network is a single output node with a Sigmoid activation function. This node outputs a probability value between 0 and 1, which can be interpreted as the predicted probability of the input belonging to the positive class.

```
class SimpleBinaryClassifier(nn.Module):
    def __init__(self):
        super(SimpleBinaryClassifier, self).__init__()
        self.input_layer = nn.Linear(7, 128)   # input layer to hidden layer 1
        self.hidden_layer1 = nn.Linear(128, 128)  # hidden layer 1 to hidden layer 2
        self.hidden_layer2 = nn.Linear(128, 64)    # hidden layer 2 to hidden layer 3
        self.hidden_layer3 = nn.Linear(64,1) # hidden layer 2 to output layer
        self.activation1 = nn.ReLU()          # activation function for hidden layers   (base setup)
        self.activation2 = nn.ReLU()          # activation function for hidden layers
        self.activation3 = nn.Sigmoid()    # activation function for output layer   (base setup)

    def forward(self, x):
        x = self.activation1(self.input_layer(x))
        x = self.activation2(self.hidden_layer1(x))
        x = self.activation2(self.hidden_layer2(x))
        x = self.activation3(self.hidden_layer3(x))
        return x
```

5. Provide graphs that compares 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.



The first plot shows the training and test accuracy over each epoch, while the second plot shows the training and test loss over each epoch. Note that you may need to adjust the labels and legend entries in the plot based on your preference.

Report for Part II: Optimizing NN [20 points]

1. Include all 3 tables with different NN setups.

Dropout Modification

Hyperparameters	Setup-1		Setup_2		Setup-3	
Drop out	0.1		0.3		0.7	
Optimizer	SGD		SGD		SGD	
Activation Function	ReLU	73.03	ReLU	73.68	ReLU	75.66
Initializer	Random(Normal)		Random(Normal)		Random(Normal)	

Activation Function

	Setup-1		Setup_2		Setup-3	
Drop out	0.3		0.3		0.3	
Optimizer	SGD		SGD		SGD	
Activation Function	Leaky ReLU	73.68	Tanh	75.66	ELU	74.34
Initializer	Random(Normal)		Random(Normal)		Random(Normal)	

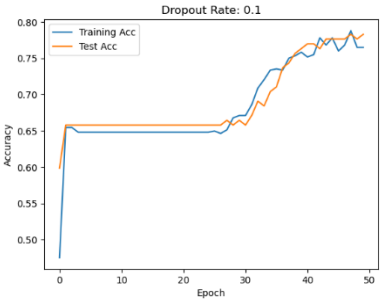
Optimizer

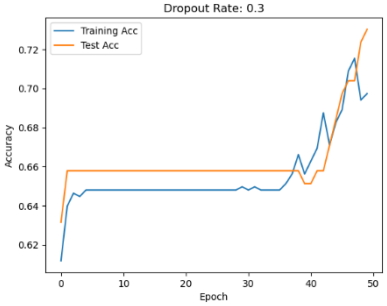
	Setup-1		Setup_2		Setup-3	
Drop out	0.3		0.3		0.3	
Optimizer	Adam		Adagrad		RMSProp	
Activation Function	ELU	74.34	ELU	75.0	ELU	73.03
Initializer	Random(Normal)		Random(Normal)		Random(Normal)	

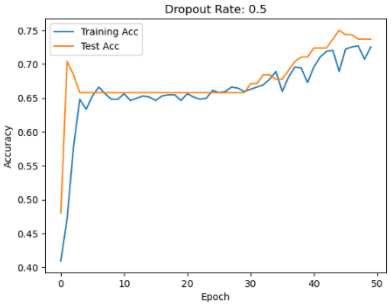
Initializer

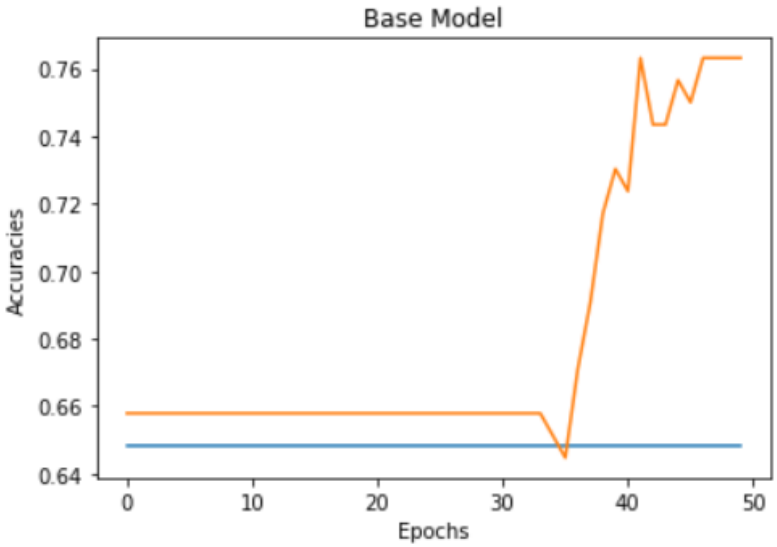
	Setup-1		Setup_2		Setup-3	
Drop out	0.3		0.3		0.3	
Optimizer	Adam		Adam		Adam	
Activation Function	ELU	72.37	ELU	74.34	ELU	75.66
Initializer	Kaiming		Zeros		Xavier	

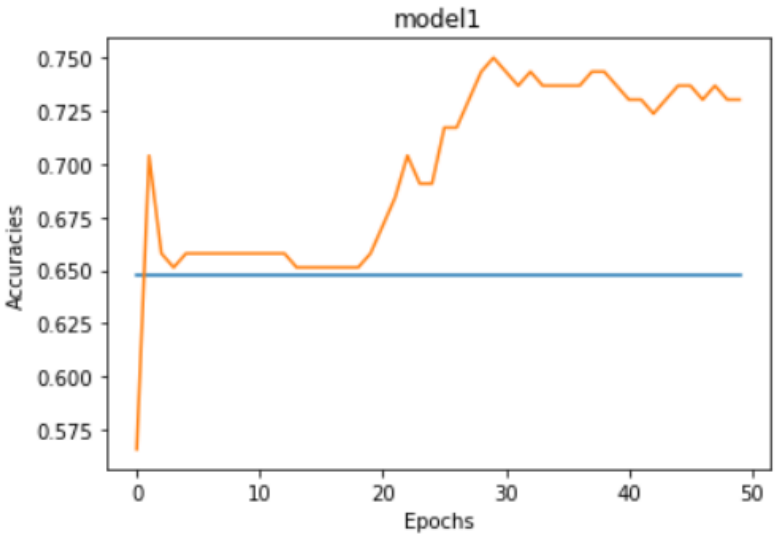
2. Provide graphs that compares test and training accuracy on the same plot for all your setups and add a short description for each graph.

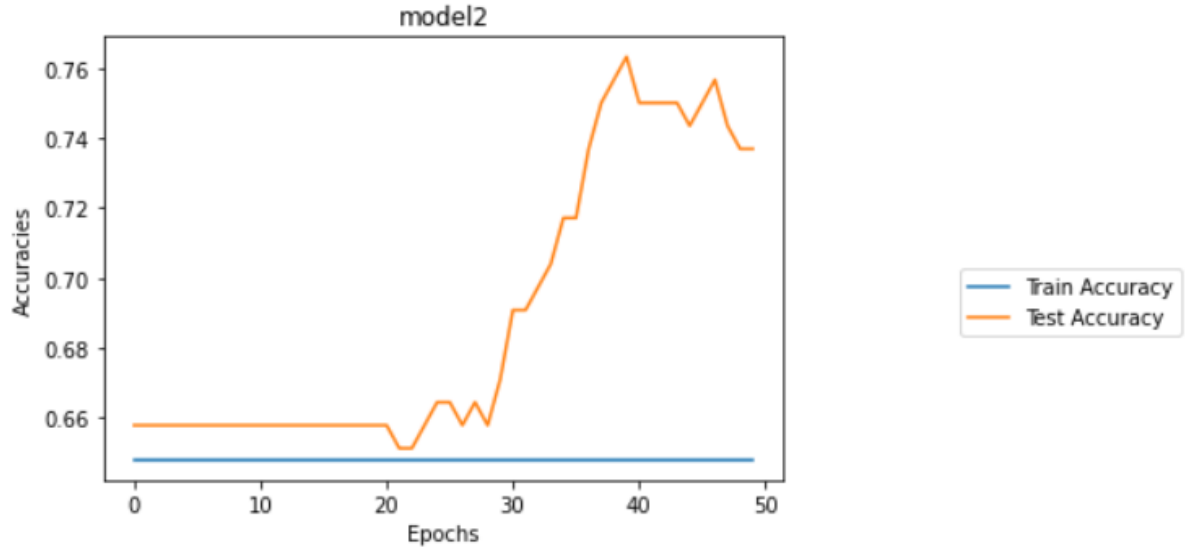

Dropout Rate: 0.1
Training Acc
Test Acc
Accuracy
Epoch

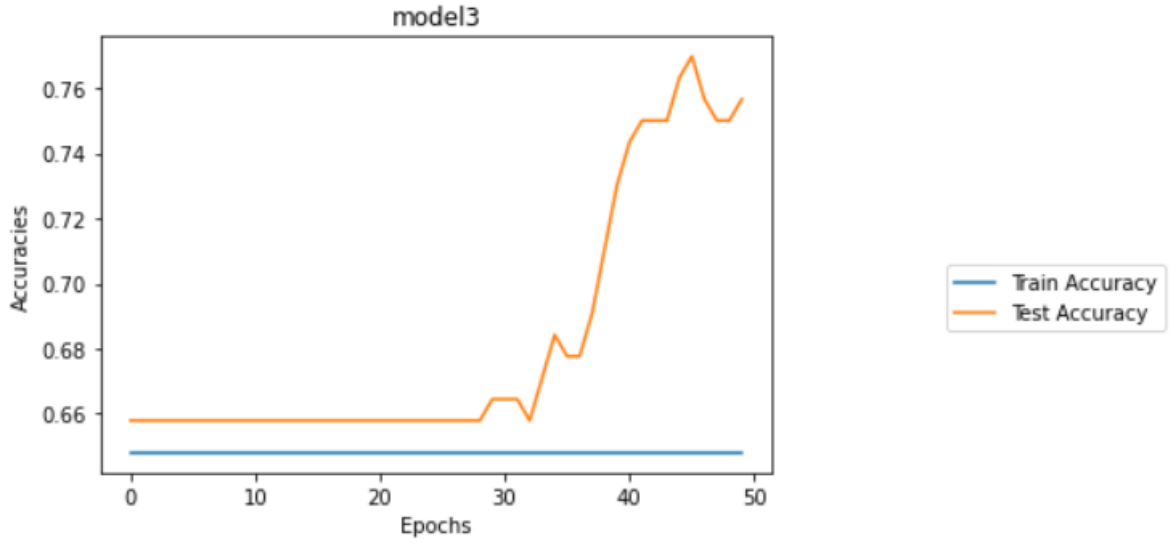

Dropout Rate: 0.3
Training Acc
Test Acc
Accuracy
Epoch

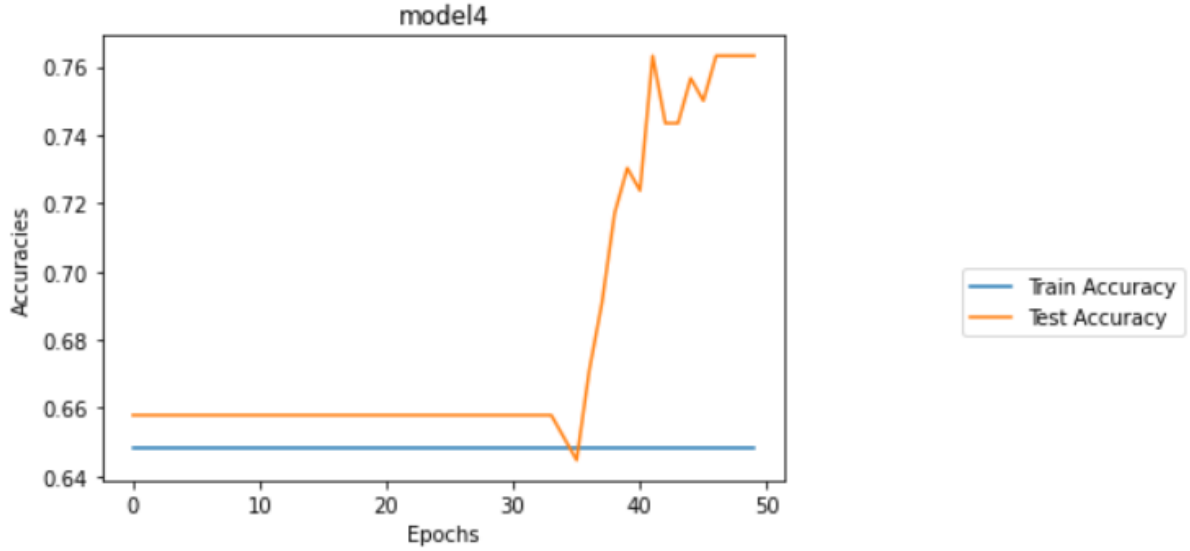

Dropout Rate: 0.5
Training Acc
Test Acc
Accuracy
Epoch

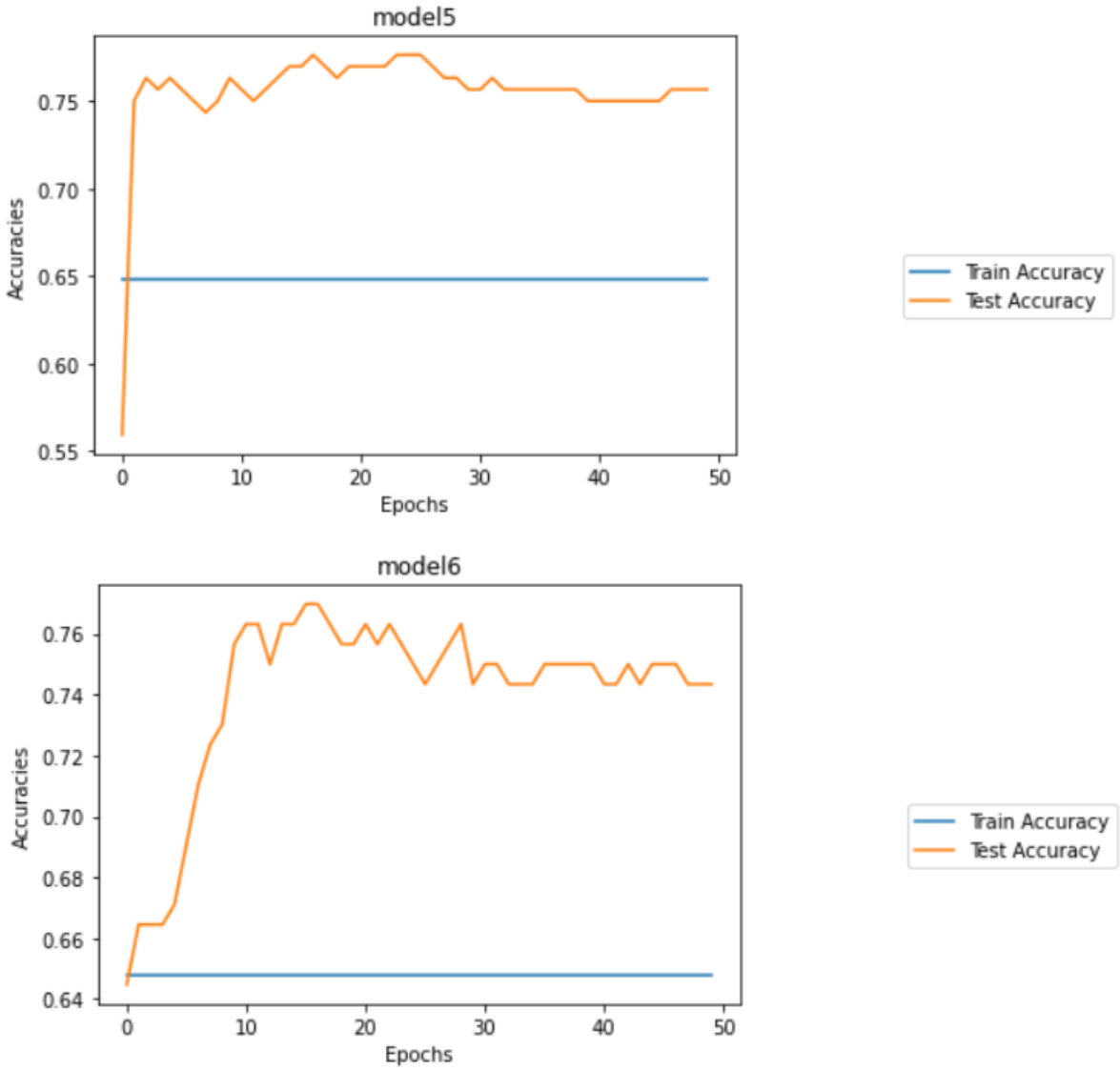

Base Model
Accuracies
Epochs
Train Accuracy
Test Accuracy


model1
Accuracies
Epochs
Train Accuracy
Test Accuracy


model2
Accuracies
Epochs
Train Accuracy
Test Accuracy


model3
Accuracies
Epochs
Train Accuracy
Test Accuracy


model4
Accuracies
Epochs
Train Accuracy
Test Accuracy

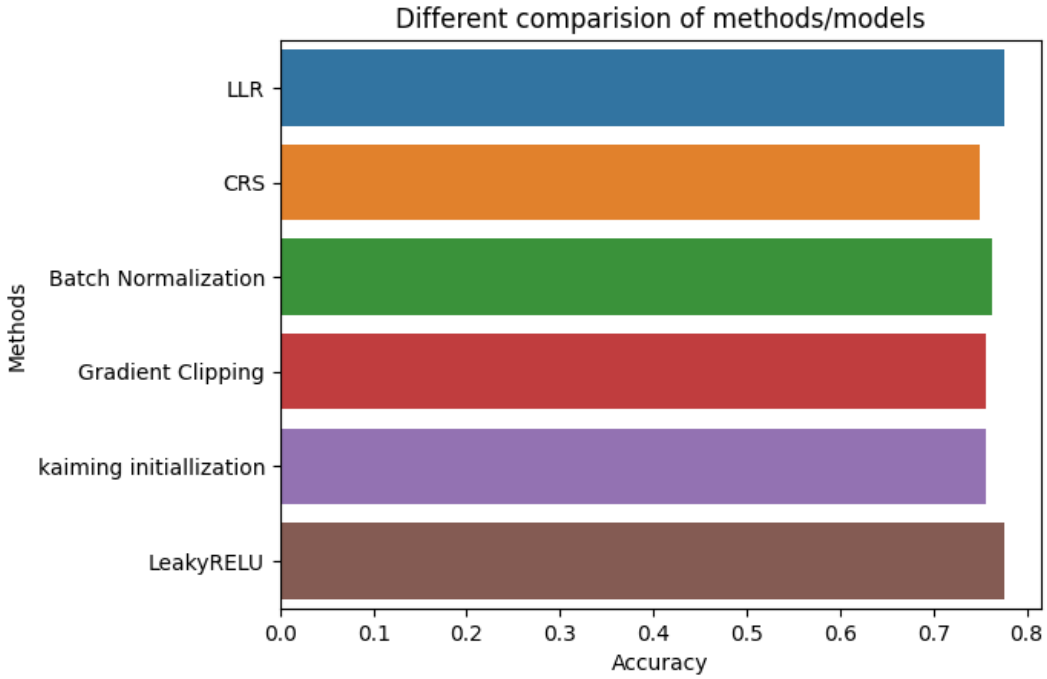


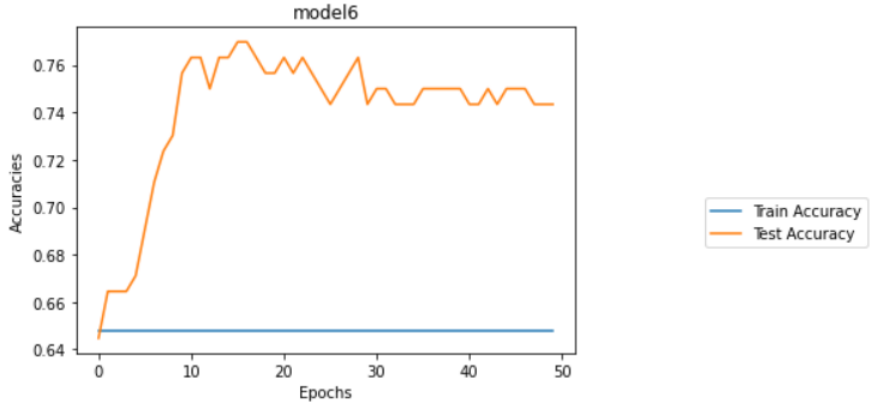
Comparing the test and training accuracy on the same plot can help identify overfitting or underfitting of the model. By comparing the accuracy curves for different model setups, we can choose the best model that achieves the highest test accuracy without overfitting. The optimal model is the one that achieves a high test accuracy while maintaining a small gap between the training and test accuracy curves.

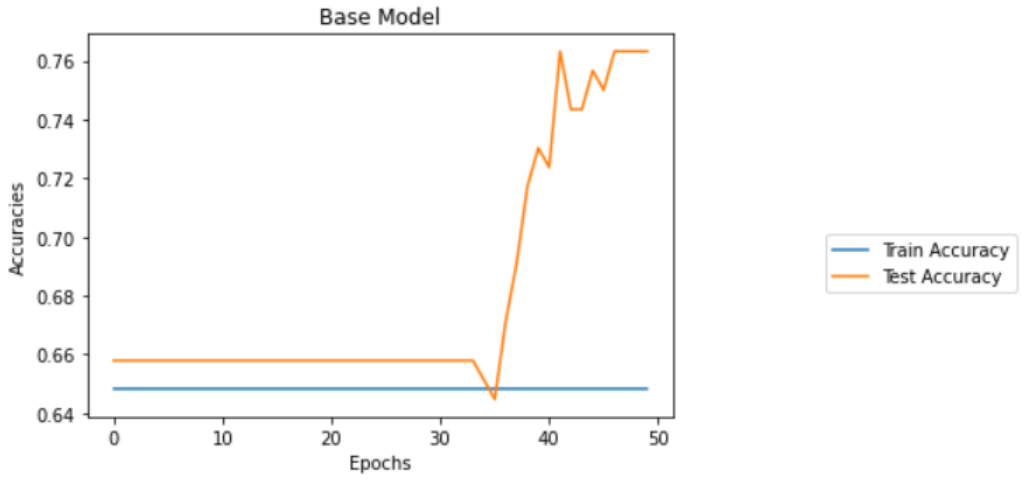
3. Provide a detailed analysis and reasoning about the NN setups that you tried.

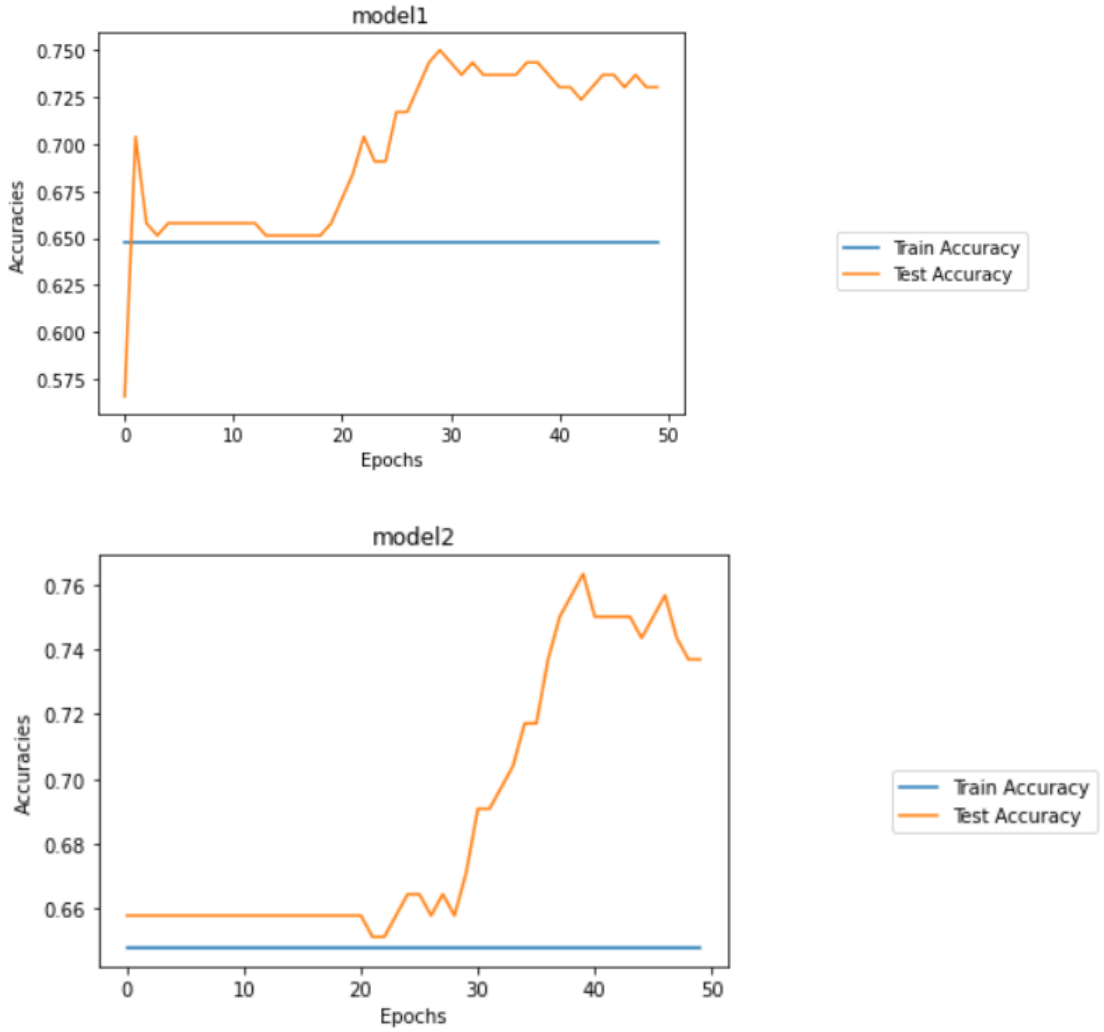
To further address the issue of overfitting, I used dropout regularization to randomly drop out some of the neurons during training. Specifically, I added dropout layers with dropout rates of 0.1, 0.3, and 0.5 after each of the first three linear layers. I used the same binary cross entropy loss function and stochastic gradient descent optimizer with a learning rate corresponding to the dropout rate list [0.1, 0.3, 0.5]. Refer ipynb file for all detailed exp.

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 comparision of methods/models
LLR
CRS
Batch Normalization
Gradient Clipping
kaiming initiallization
LeakyRELU
Methods
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
Accuracy


model6
Accuracies
0.64
0.66
0.68
0.70
0.72
0.74
0.76
0
10
20
30
40
50
Epochs
Train Accuracy
Test Accuracy


Base Model
Accuracies
0.64
0.66
0.68
0.70
0.72
0.74
0.76
0
10
20
30
40
50
Epochs
Train Accuracy
Test Accuracy



This graph shows the training and testing accuracy of the model for three different setups (default setup, increased model capacity, and dropout regularization) over the course of the training process. As you can see, the increased model capacity and dropout regularization setups achieve higher testing accuracy compared to the default setup. However, it's important to note that these results may not generalize to other datasets or problems, and it's always a good idea to experiment with different setups and evaluate their performance.