Assignment 2

Part 1: Building a Basic Neural Network

Dataset:

We have totally 8 columns in the given data set,7 input and 1 output column with total of 766 entries.

```
: df=pd.read_csv("dataset.csv")
  df.shape

: (766, 8)
```

f1, f2, f3, f4, f5, f6, f7 are the input columns and target is the output column.

```
df.columns
```

```
Index(['f1', 'f2', 'f3', 'f4', 'f5', 'f6', 'f7', 'target'], dtype='object')
```

Initially we have some categorical data in the dataset, we will remove the rows with categorical data after removing we have 760 entries in the dataset now.

```
cols = df.select_dtypes(include=['object']).columns
values=['c','f','a','b','d','e']
i=0
for col in cols:
    df = df.loc[~df[col].str.contains(values[i])]
    i=i+1
print(df)
```

```
: df.shape

: (760, 8)
```

The output column target data is binary output 0's and 1's.

Data Visualization

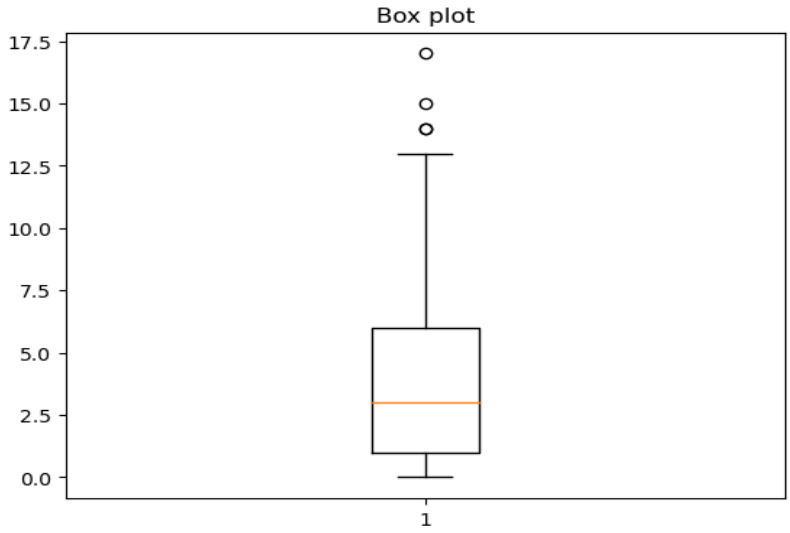
Correlation matrix:

The matrix which describes the correlation between all the features in the dataset. By using this correlation matrix, we can find how much each feature is correlated with another feature. Heatmap is the one visualization which is used for plotting the correlation matrix to visualize in the best way.

From the above heat map, we can visualize that f2 is most correlated with the target followed by the f1 compared to all other features. f3 feature is best correlated with other features but least with the target.

Box Plot:

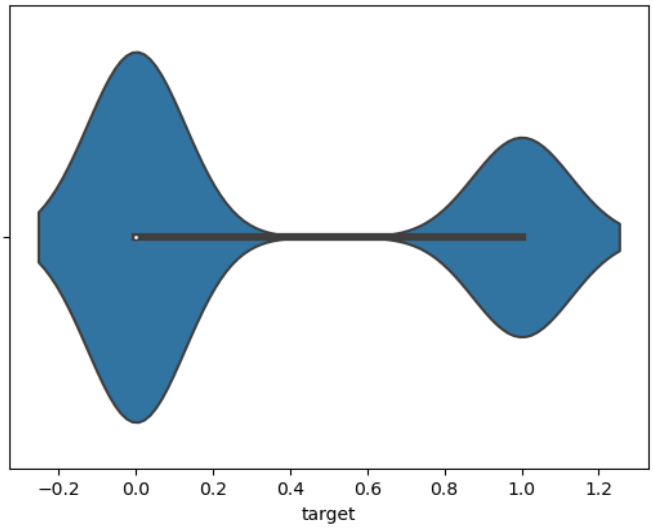
Box plot is the plot which will be used to find the outliers. As F1 is mostly correlated with the target we are plotting the box plot for f1 feature to check the outliers.



From the box plot there are less outliers for the f1 feature.

Violin Plot

Violin plots are used to visualize the distribution of data and its probability density.



As the target values are 1’s and 0’s by this violin plot we can visualize the values 0’s is more compared to 1’s.

NN Architecture Structure

```
class NeuralNetwork(nn.Module):
    def __init__(self,dropout):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
        nn.Linear(7, 64),
        nn.ReLU(),
        nn.Dropout(p=dropout),
        nn.Linear(64, 32),
        nn.ReLU(),
        nn.Dropout(p=dropout),
        nn.Linear(32, 1),
        nn.Sigmoid()
        )
    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

The model consists of three fully connected layers with ReLU activation functions, where the first two layers also have dropout regularization.

The forward method of the class is used to define the forward pass of the model, which is the computation that transforms the input tensor into an output tensor. The input tensor is first flattened into a 1D tensor using the flatten layer. The flattened tensor is then passed through the linear_ReLU_stack layer, which consists of the fully connected layers with ReLU activation functions and dropout regularization. The resulting tensor, called logits, is returned as the output of the forward pass.

Accuracy and Loss Graphs:

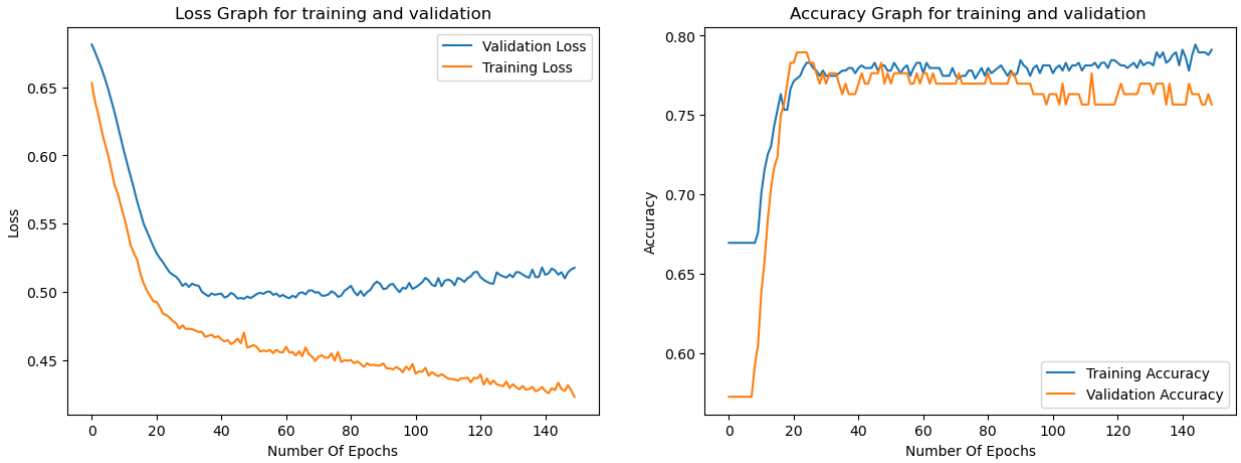


Fig: Comparison of loss and Accuracy plot for base model

The above plot is the loss and accuracy graph for training and testing done on the base model. The accuracy of the graph increased gradually for both the training and validation for the initial epochs. The training accuracy increased from 67% to 79% with less increment after 20 epochs. The validation accuracy reaches the peak of 78% and there is the fluctuation after 20 epochs within the range of 75% to 76%.

Loss and accuracy are inversely proportional to each other, with the epochs getting increased the model started fitting and loss decreases.

From the initial base model, we got the test accuracy 75.66%.

Part 2: Optimizing Neural Network

In this part we have optimized the neural network with different hyperparameters. The hyperparameters includes adding dropouts to the CNN Architecture, optimizer tuning, checking with different Activation function and finally initializer tuning.

Dropout Tuning:

Dropout is a regularization technique that is commonly used in deep learning models to prevent overfitting. In this technique, randomly selected neurons are ignored or "dropped out" during training.

Input, hidden, and output layers of a neural network can all be affected by dropout. Each neuron has a probability of p either being kept or being dropped out during training. The predicted value of each kept neuron is maintained throughout training and testing by multiplying the retained neurons by 1/p.

By adding dropout in the Architecture with three different probabilities p(p=0.1,0.2,0.3) we have checked the model.

	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Setup 3	Test Accuracy
Dropout	0.1		0.2		0.3	
Optimizer	SGD		SGD		SGD	
Activation function	ReLU		ReLU		ReLU	
Initializer	None	74.34	None	75.66	None	75.66

Table1: Comparison of results with different dropout probability

With dropout probability **p=0.2** and **p=0.3** we got the best accuracy of **75.66**% which is same as the base accuracy.

Dropout Probability P = 0.1

We have tuned the model with probability p=0.1 and we got the accuracy 74.34% which is less than the base accuracy without any dropouts.

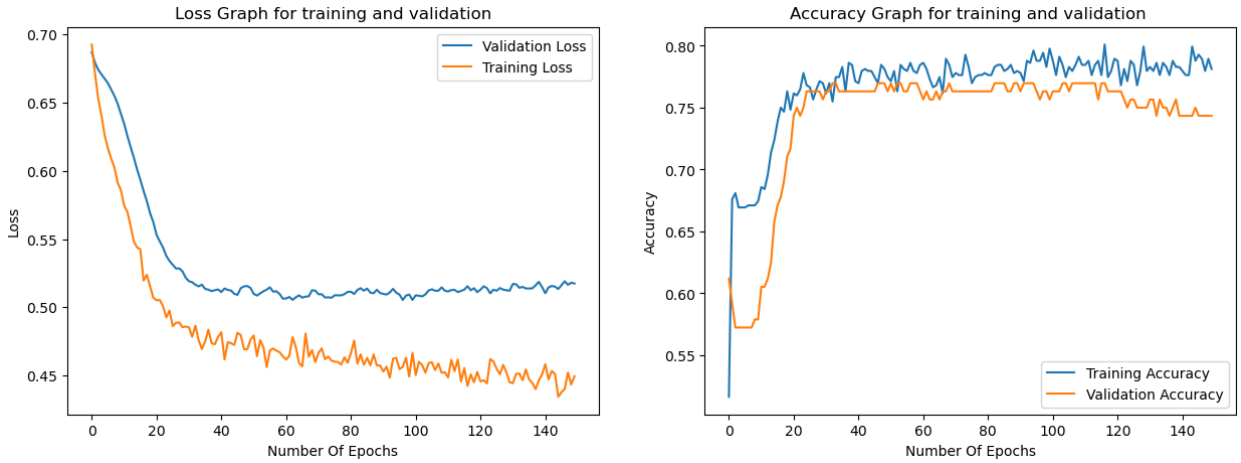


Fig: Comparison of loss and Accuracy plot with dropout probability 0.1

From the above plot the training accuracy started at 67% and decreased after the initial epoch and decreased and increased gradually after 20 epochs. The validation accuracy started with the less and increased to peak in 20 epochs and fluctuated after in the range of 75% to 76% and settled at 74% with 150 epochs.

Initially the train loss is 0.58 and validation loss is 0.64 after model started fitting the train loss decreased to 0.44 and validation loss decreased to 0.51.

	Initial Epoch	Final Epoch
Training Accuracy	67%	78%
Validation Accuracy	57%	74%
Training Loss	0.58	0.44
Validation Loss	0.64	0.51

Table 2: Comparison of loss and Accuracy of initial and final epochs with dropout probability p=0.1

Dropout Probability P = 0.2

We have tuned the model with probability p=0.2 and we got the accuracy 75.66% which is same as the base accuracy.

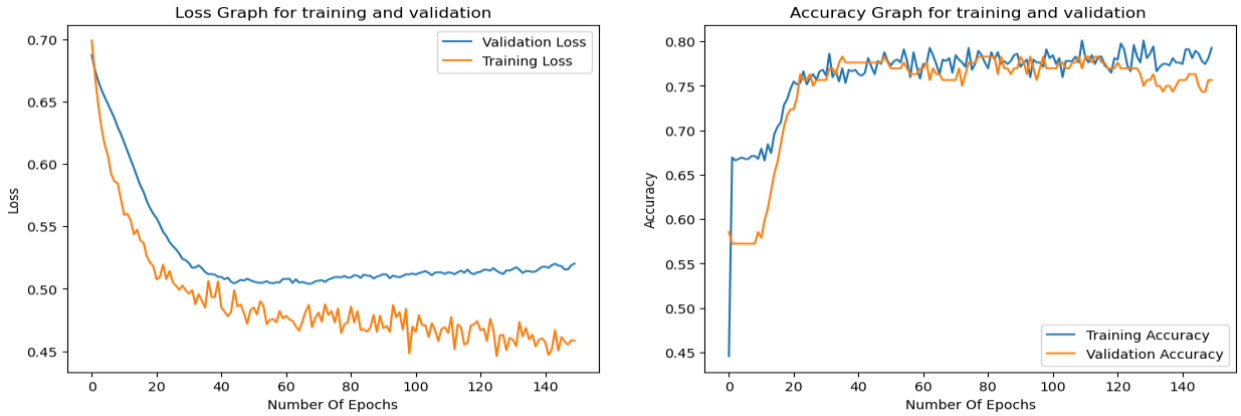


Fig: Comparison of loss and Accuracy plot with dropout probability 0.2

With dropout probability p=0.2 the training accuracy started at 66% and validation accuracy 58% which gradually increased after the initial 20 epochs and settled. We got the maximum accuracy at the 100 epochs and decreased a little after that.

The initial train loss is 0.57 and the validation loss is 0.62, with the model fitting with p=0.2 the loss decreased.

	Initial Epoch	Final Epoch
Training Accuracy	66%	79%
Validation Accuracy	58%	75%
Training Loss	0.57	0.45
Validation Loss	0.62	0.52

Table 3: Comparison of loss and Accuracy of initial and final epochs with dropout probability p=0.2

Dropout Probability P = 0.3

We have tuned the model with probability p=0.3 and we got the accuracy 75.66% which is same as the base accuracy.

Fig: Comparison of loss and Accuracy plot with dropout probability 0.3

With probability p=0.3 the train accuracy started at 70% which is higher compared to other drop out probabilities and the validation accuracy started at 63%. The accuracies increased and settled at 75%.

From the plot we can see there is a lot of fluctuations in the training loss after 20 epochs but there are less fluctuations in the validation loss.

	Initial Epoch	Final Epoch
Training Accuracy	70%	76%
Validation Accuracy	63%	75%
Training Loss	0.55	0.46
Validation Loss	0.60	0.51

Table 4: Comparison of loss and Accuracy of initial and final epochs with dropout probability p=0.3

Optimizer Tuning:

Training a deep neural network is an extremely time-consuming task especially with complex problems. Using a faster optimizer for the network is an efficient way to speed up the training speed.

One risk of employing the wrong optimizers is that the model will take a long time to reach a global minimum or will become trapped at a local minimum, producing a worse model. Therefore, knowing which Optimizer suits mostly on the problem will save you tons of training hours. The optimizer is responsible to change the learning rate and weights of neurons in the neural network

to reach the minimum loss function. Optimizer is very important to achieve the possible highest accuracy or minimum loss.

There are several optimizers we can use, in the initial training model we have used SGD optimizer, now we are changing the optimizer in the initial model and checking how the model works. The optimizer we are trying are Adagrad, RMSProp and Adam

	Setup 1	**Test Accuracy**	**Setup 2**	**Test Accuracy**	**Setup 3**	**Test Accuracy**
Dropout	0		0		0	
Optimizer	Adam		RMSprop		Adagrad	
Activation function	ReLU		ReLU		ReLU	
Initializer	None	66.45	None	67.76	None	73.68

Table 5: Comparison of results with different optimizer

Adam:

Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first order and second-order moments.

Adam optimization method is "computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters".

The accuracy we got using the Adam optimization is 66.45 which is less than the base accuracy in which we have used SGD optimizer.

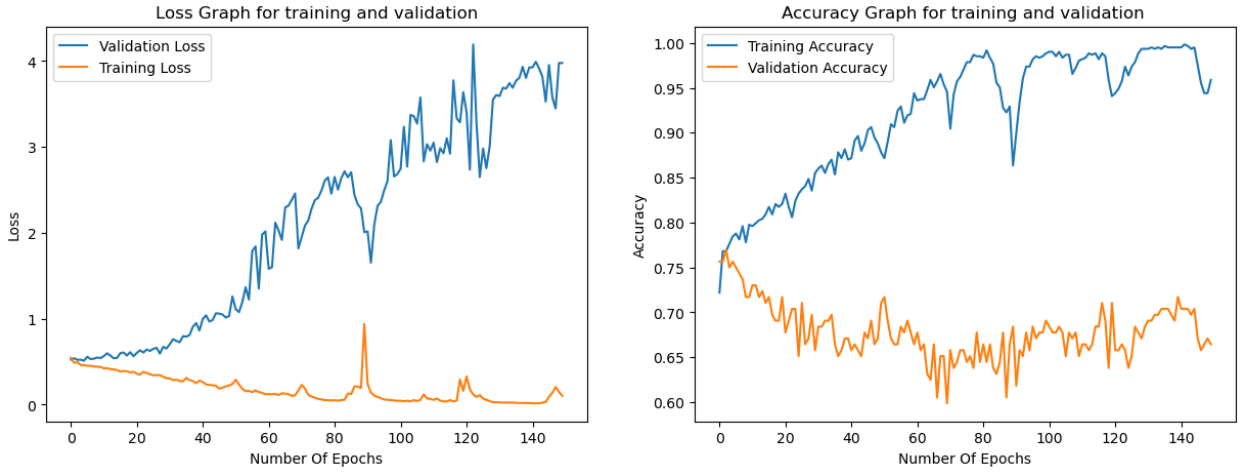


Fig: Comparison of loss and Accuracy plot with Adam optimizer

From the above plot we can describe that the training accuracy increased from 79% to 95% but the model does not fit on the validation data as the accuracy started decreased from 71% to 66%.

Train loss almost reached to 0 but the validation loss increased with epochs getting increased, the validation loss increased from 0.54 to almost 4 which tells that Adam optimizer does not fit well on our dataset.

	Initial Epoch	Final Epoch
Training Accuracy	79%	95%
Validation Accuracy	71%	66%
Training Loss	0.44	0.10
Validation Loss	0.54	3.97

Table 6: Comparison of loss and Accuracy of initial and final epochs with Adam Optimizer

RMSProp:

The RMSprop optimizer is like the gradient descent algorithm with momentum. The RMSprop optimizer limits oscillations that move vertically. As a result, we can speed up learning and our algorithm will be able to make bigger horizontal strides and converge more quickly.

The accuracy we got using the Adam optimization is 67.76 which is less than the base accuracy in which we have used SGD optimizer but little more than Adam optimizer.

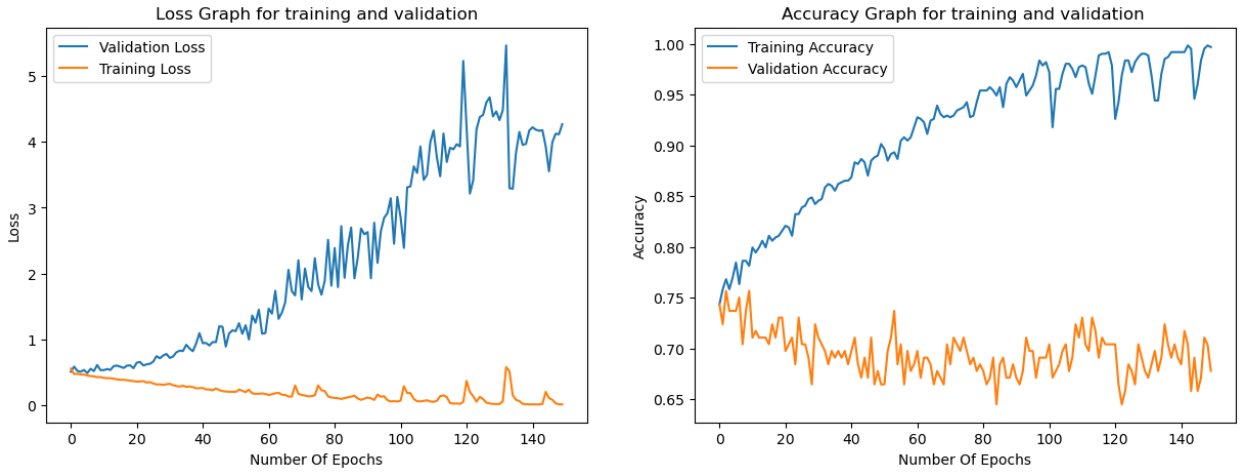


Fig: Comparison of loss and Accuracy plot with RMSProp optimizer

From the above plot we can describe that the training and validation loss and accuracy are both in the opposite direction where the training accuracy increased from 78% to 99% whereas the validation accuracy decreased from 75% to 67%.

The training accuracy almost decreased to 0 with the number of epochs increasing but the validation accuracy increased up to 4.5.

	Initial Epoch	Final Epoch
Training Accuracy	78%	99%
Validation Accuracy	75%	67%
Training Loss	0.42	0.01
Validation Loss	0.53	4.26

Table 7: Comparison of loss and Accuracy of initial and final epochs with RMSProp Optimizer

Adagrad:

Like RMSprop, Adagrad is an adaptive learning rate algorithm. Based on the historical sum of squares in each dimension, Adagrad scales the gradient element by element. Consequently, we maintain a running sum of squared gradients. One of the Adaptive learning rate methods, in which the algorithm goes faster down the steep slopes than the gentle slopes.

The accuracy we got using the Adam optimization is 73.68 which is little less than the base accuracy in which we have used SGD optimizer.

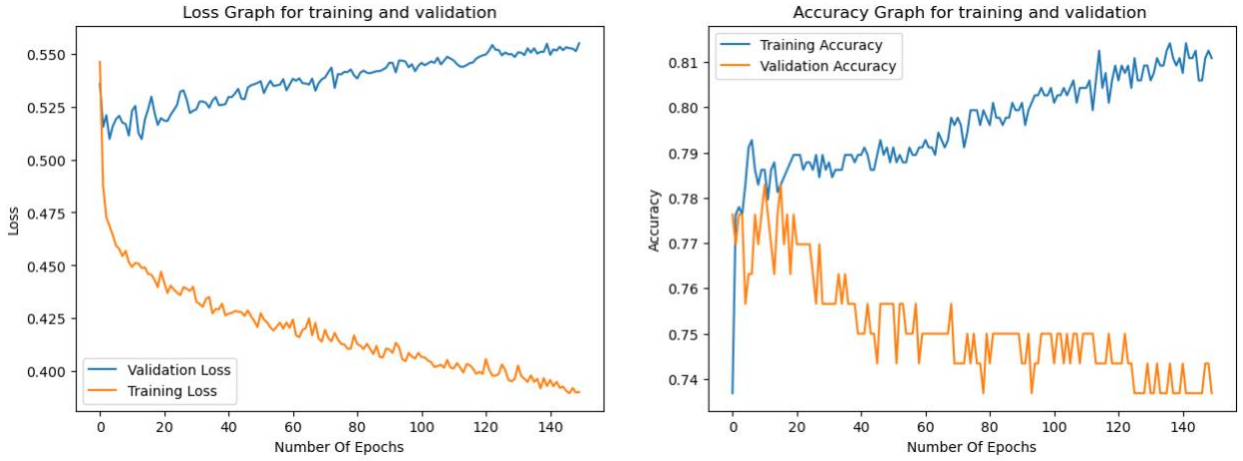


Fig: Comparison of loss and Accuracy plot with Adagrad optimizer

Like Adam and RMSProp optimizer Adagrad optimizer plots are same that the validation and training plots are in the opposite direction. The training accuracy increased from 78% to 81% but the validation accuracy decreased from 77% to 73%.

In the above two optimizers the loss is almost 0 for the training data but here using Adagrad optimizer the training loss decreased from 0.45 to 0.39 where the validation loss increased from 0.51 to 0.55.

From all the above the SGD optimizer which we have used in the base model gives the best result followed by the Adagrad optimizer.

	Initial Epoch	Final Epoch
Training Accuracy	78%	81%
Validation Accuracy	77%	73%
Training Loss	0.45	0.39
Validation Loss	0.51	0.55

Table 8: Comparison of loss and Accuracy of initial and final epochs with Adagrad Optimizer

Activation Function Tuning:

The activation function is the non-linear function that we apply over the output data coming out of a particular layer of neurons before it propagates as the input to the next layer.

In the initial Architecture we have used ReLU as the activation function in the hidden layers and sigmoid at the output layer.

ReLU activation will give the output as the input if the value is positive, if the input is negative the output will be zero.

The different activation functions we have tried in our models are Tanh, LeakyReLU, RReLU.

	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Setup 3	Test Accuracy
Dropout	0		0		0	
Optimizer	SGD		SGD		SGD	
Activation function	LeakyReLU		Tanh		RReLU	
Initializer	None	72.37	None	76.97	None	73.68

Table 9: Comparison of results with different Activation function

LeakyReLU:

Leaky ReLU activation function is like ReLU activation where the output is same as the input when the value is positive, output is 0.01 times the input in LeakyReLU where it is 0 using ReLU Activation function.

The accuracy we got using the LeakyReLU activation function is 72.37 which is less than the base accuracy.

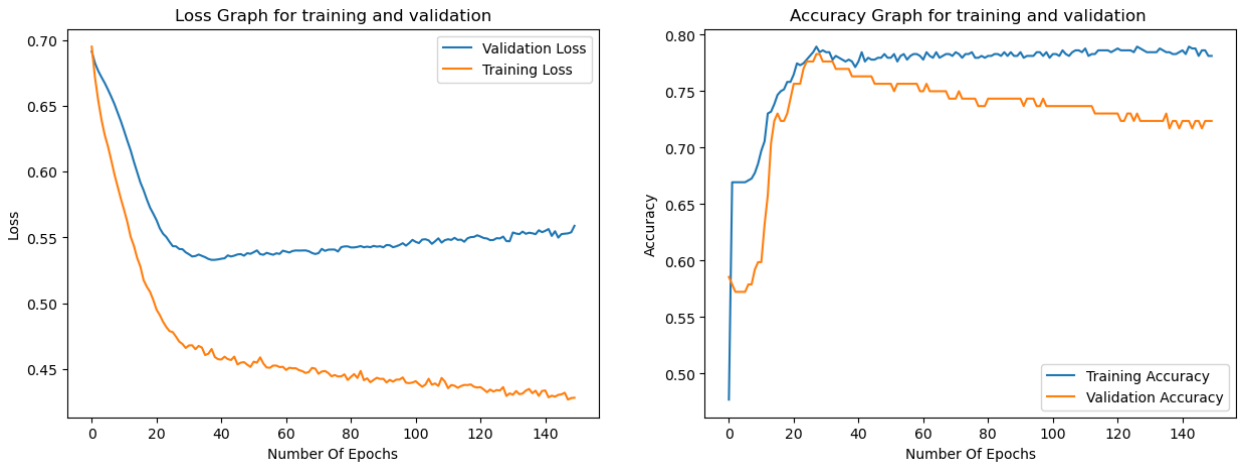


Fig: Comparison of loss and Accuracy plot with LeakyReLU Activation function

The above plot describes that accuracy increased up to 30 epochs and reaches peak with training accuracy 78% and validation accuracy 77%, after that there is no major decrease in the train accuracy, but the validation accuracy decreased to 72%.

Similarly, the initial training loss is 0.57 and the validation loss is 0.63 but the training loss dropped to 0.42 whereas the validation loss is 0.55.

	Initial Epoch	**Final Epoch**
Training Accuracy	68%	78%
Validation Accuracy	59%	72%
Training Loss	0.57	0.42
Validation Loss	0.63	0.55

Table 10: Comparison of loss and Accuracy of initial and final epochs with LeakyReLU Activation function

Tanh:

The function takes any real value as input and outputs values in the range -1 to 1. The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.

The accuracy we got using the Tanh activation function is 76.97 which is higher than the base accuracy.

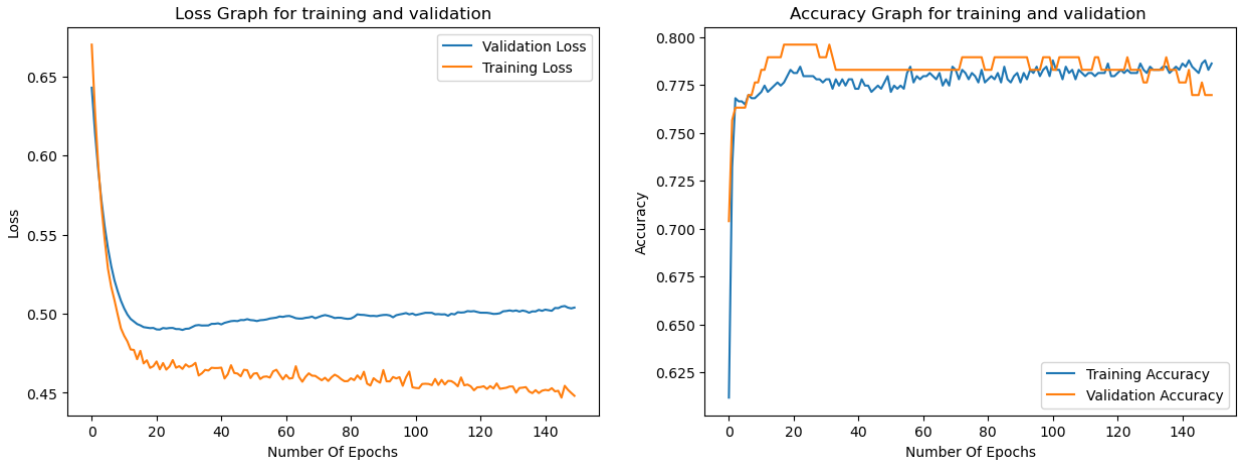


Fig: Comparison of loss and Accuracy plot with Tanh Activation function

The above plot depicts that the validation accuracy 77% is higher than the training accuracy 76% at the initial stage. At some point the validation accuracy reaches to 80% and it got decreased a little bit after certain epochs and settled at 77% where the train accuracy settles at 78%.

The training and validation loss got decreased and reaches to 0.44 and 0.50 respectively after 150 epochs.

	Initial Epoch	**Final Epoch**
Training Accuracy	76%	78%
Validation Accuracy	77%	76%
Training Loss	0.49	0.44
Validation Loss	0.5082	0.5039

Table 11: Comparison of loss and Accuracy of initial and final epochs with Tanh Activation function

RReLU:

Randomized rectified linear unit also known as the parametric rectified linear unit activation function is like LeakyReLU where the output when the input value is negative gives the alpha times of input where the alpha is the random value.

The accuracy we got using the RReLU activation function is 73.68.

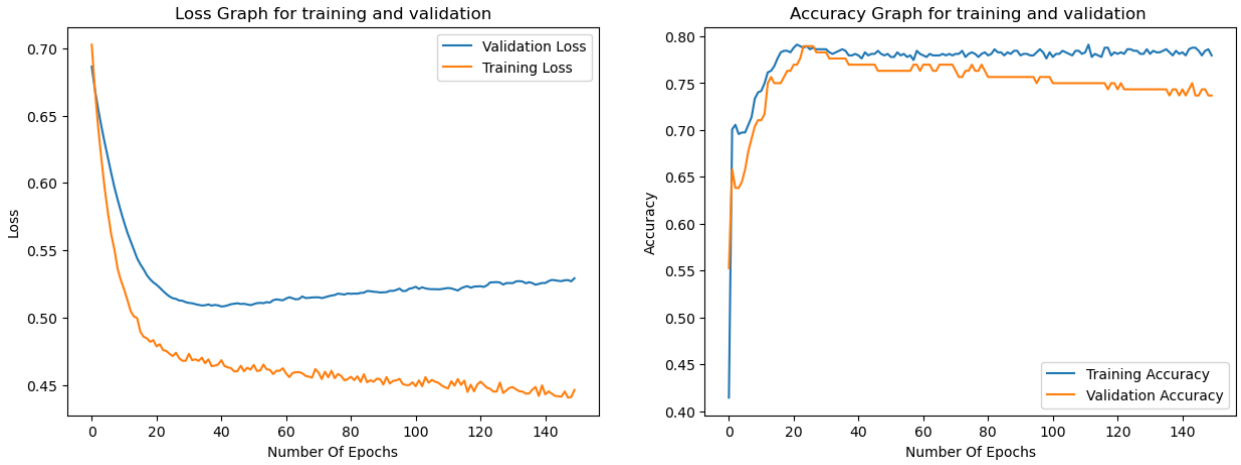


Fig: Comparison of loss and Accuracy plot with RReLU Activation function

The above plot describes that the training accuracy started at 74% and validation accuracy started at 71% and the training accuracy reached to 77% and validation accuracy increased to 73%.

Training loss decreased from 0.52 to 0.44 and the validation loss decreased from 0.57 to 0.52.

	Initial Epoch	**Final Epoch**
Training Accuracy	74%	77%
Validation Accuracy	71%	73%
Training Loss	0.52	0.44
Validation Loss	0.57	0.52

Table 12: Comparison of loss and Accuracy of initial and final epochs with RReLU Activation function

Initializer Tuning:

weight initialization refers to the process of setting initial weights for neural network layers before training. Proper weight initialization can help neural networks converge faster and achieve better performance. The initialization step can be critical to the model's ultimate performance, and it requires the right method.

Here we have tried with three different initializers Xavier, Kaiming and Xavier_Normal.

	Setup 1	**Test Accuracy**	**Setup 2**	**Test Accuracy**	**Setup 3**	**Test Accuracy**
Dropout	0		0		0	
Optimizer	SGD		SGD		SGD	
Activation function	ReLU		ReLU		ReLU	
Initializer	xavier	73.03	Kaiming	71.71	xavier_Normal	75.00

Table 13: Comparison of results with different Initializer

Xavier Initializer:

With Xavier Initialization we initialize the weights such that the variance of the activations is the same across every layer. This constant variance helps prevent the gradient from exploding or vanishing.

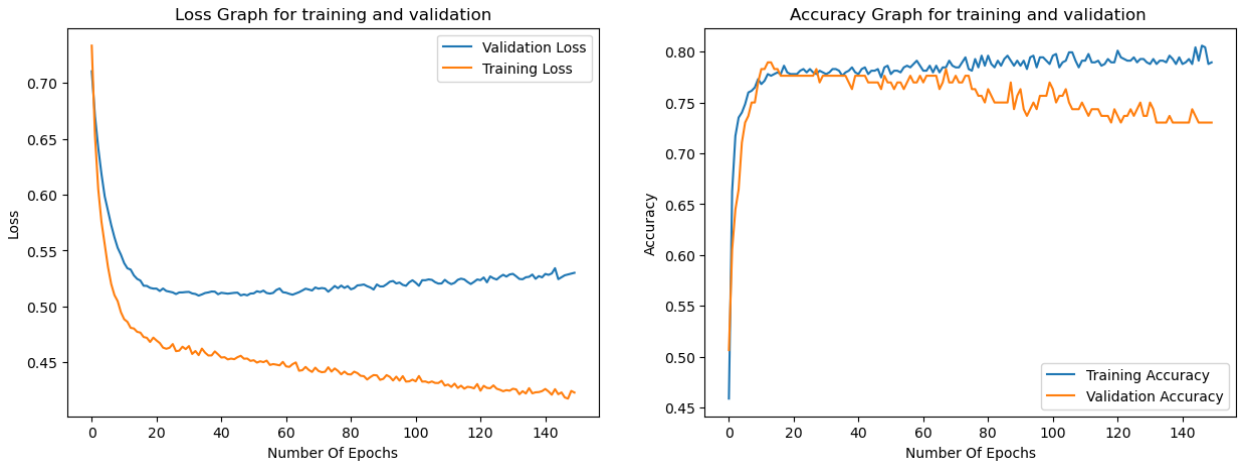


Fig: Comparison of loss and Accuracy plot with Xavier Initializer

From the above plot we can describe the training accuracy started at 50% and it reached peak in the initial 10 epochs to 77% and after that there is slight increment in the accuracy and increased to 79%. The validation accuracy changes from 77% to 73% where it started decreasing after reaching 80 epochs.

The loss graphs are inverse to the accuracy plots the training loss decreased from 0.49 to 0.42 while the validation loss decreased from 0.54 to 0.53 which is a very little less even if the validation loss decreased the accuracy also decreased when it reaches to 150 epochs.

	Initial Epoch	**Final Epoch**
Training Accuracy	77%	79%
Validation Accuracy	77%	73%
Training Loss	0.49	0.42
Validation Loss	0.54	0.53

Table 14: Comparison of loss and Accuracy of initial and final epochs with Xavier initializer

Kaiming Initializer:

The Kaiming initializer is designed to overcome the problems with the popular Xavier initialization method. The kaimimg initializer is like Xavier initializer with a little difference where Kaiming takes into account the activation function, whereas Xavier does not use any activation function.

The accuracy we got using the kaiming initializer is 71.71%.

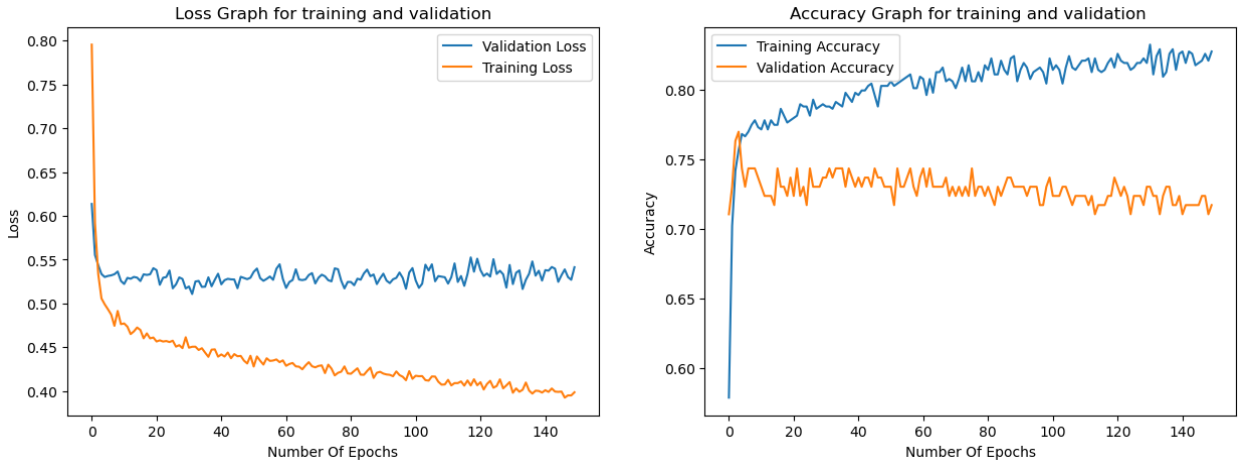


Fig: Comparison of loss and Accuracy plot with Kaiming Initializer

The above plot shows there is a sharp increase in the training accuracy initially from 55% to 77% in the initial 10 epochs and after that with a little increment it recaches to 82% with 150 epochs, the validation accuracy reached to 78% and dropped to 73% in initial 10 epochs and after that it decreased to 71%.

The training and testing loss started splitting at 0.54 in the initial stage, the validation loss increases from 0.52 to 0.54 while the training loss decreased from 0.47 to 0.39.

	Initial Epoch	**Final Epoch**
Training Accuracy	77%	82%
Validation Accuracy	73%	71%
Training Loss	0.47	0.39
Validation Loss	0.52	0.54

Table 15: Comparison of loss and Accuracy of initial and final epochs with Kaiming initializer

Xavier_Normal Initializer:

For the normal Xavier initialization, we draw each weight w from a normal distribution with a mean of 0, and a standard deviation equal to 2, divided by the number of inputs, plus the number of outputs for the transformation.

$$\text{STD}=\text{gain}*\sqrt{\frac{2}{fan_{in}+fan_{out}}}$$

Fig: Comparison of loss and Accuracy plot with Xavier_Normal Initializer

The above plot depicts that there is a sharp increase in the validation and training accuracy initially and after that the training accuracy is steady with lot of fluctuations while the validation accuracy decreased a little after 40 epochs and the accuracy decreased from 77% to 75%.

With the increase of the epochs the training loss decreased from 0.48 to 0.42, the validation loss is almost constant with very minor increment from 0.5136 to 0.5169.

	Initial Epoch	Final Epoch
Training Accuracy	78.12%	78.95%
Validation Accuracy	77.63%	75%
Training Loss	0.48	0.42
Validation Loss	0.5136	0.5169

Table 16: Comparison of loss and Accuracy of initial and final epochs with Xavier_normal initializer

The below are the final accuracies we got using the different NN setups.

```
: accuracies
```

```
: {'BaseModel1_acc': 0.756578947368421,
   'Dropout1_acc': 0.743421052631579,
   'Dropout2_acc': 0.756578947368421,
   'Dropout3_acc': 0.756578947368421,
   'Optimizer1_acc': 0.6644736842105263,
   'Optimizer2_acc': 0.6776315789473685,
   'Optimizer3_acc': 0.7368421052631579,
   'ActivationFunction1_acc': 0.7236842105263158,
   'ActivationFunction2_acc': 0.7697368421052632,
   'ActivationFunction3_acc': 0.7368421052631579,
   'Initializer1_acc': 0.7302631578947368,
   'Initializer2_acc': 0.7171052631578947,
   'Initializer3_acc': 0.75}
```

We got the base accuracy as 75.65 and by using different NN setups with using of Tanh activation function we got the accuracy of 76.97 which is greater than base accuracy. With the use of Adagrad optimizer we got the accuracy of 73.68% which is little less than the base model accuracy.

Early Stopping:

Early stopping is an optimization technique used to reduce overfitting without compromising on model accuracy. The main idea behind early stopping is to stop training before a model starts too overfit.

We store and update the current best parameters during training, and when parameter updates no longer yield an improvement (after a set number of iterations) we stop training and use the last best parameters.

As we have seen in the above NN setups where the accuracy of model is getting decreased after certain epochs so the early stopping will help here where the model will stop training if the values are getting to decrease continuously, here we will pass the parameter called patience based on this value the early stopping will be stopped. If the accuracy is not getting increased for the patience number of iterations, then the model will stop training.

The accuracy we got using the early stopping is 78.29%.

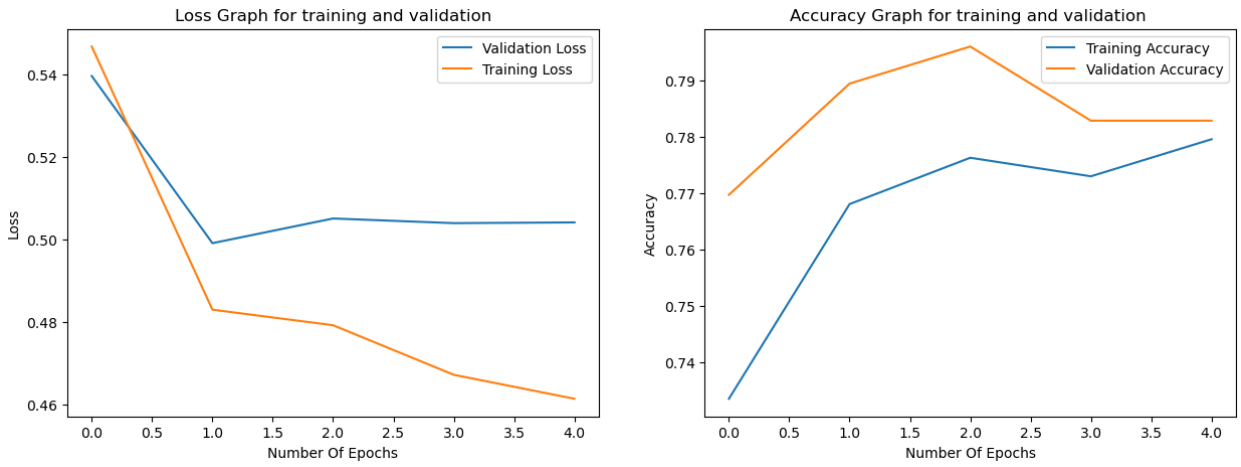


Fig: Comparison of loss and Accuracy plot with Early Stopping

From the above plot we can see as we have given the 150 epochs but with early stopping it stopped at 5 epochs as the accuracy of the validation model stops for 3 continues epochs.

The initial training and validation accuracy are 73% and 76% respectively, at 3 epoch we got the validation accuracy of 79% after that it dropped to 78% and did not increase. As we have given the patience value as 3 and for 3 continuous values the accuracy did not increase so the training the model stopped.

The training loss decreased from 0.54 to 0.46 while the validation loss decreased from .53 to 0.50.

	Initial Epoch	Final Epoch
Training Accuracy	73%	77%
Validation Accuracy	76%	78%
Training Loss	0.54	0.46
Validation Loss	0.53	0.50

Table 17: Comparison of loss and Accuracy of initial and final epochs with Early stopping

Learning Rate Scheduler

Learning rate schedules seek to adjust the learning rate during training by reducing the learning rate according to a pre-defined schedule.

In our code we have used the learning rate scheduler as below.

scheduler = lr_scheduler. StepLR (optimizer, step_size=50, gamma=0.1)

Here we have used the step size scheduler where the learning rate of the optimizer will decay by a certain factor after a certain number of epochs. In this case, the learning rate will be multiplied by gamma=0.1 every step_size=50 epochs.

In our model the initial learning rate of the optimizer is 0.01, then after 50 epochs, the learning rate will be reduced to 0.001. After another 50 epochs, the learning rate will be further reduced to 0.0001, and so on.

The accuracy we got using the learning rate scheduler optimization technique is 74.34 which is little less than the base accuracy.

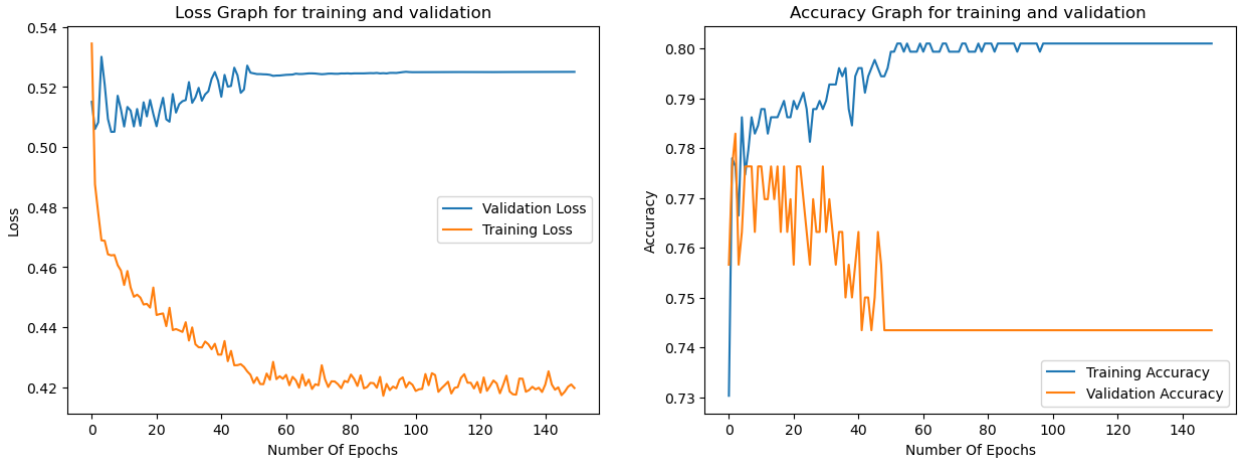


Fig: Comparison of loss and Accuracy plot with Learning Rate Scheduler

From the above plot we can describe that the training and validation both accuracy and loss are in opposite direction. As the model starts fitting the training accuracy is increasing while the validation accuracy is getting decreased after certain epochs.

We can depict that the validation loss and accuracy become constant after 60 epochs with the values 0.52 loss and 74% accuracy which means that there is no impact with changing the learning rate further.

	Initial Epoch	Final Epoch
Training Accuracy	78%	80%
Validation Accuracy	77%	74%
Training Loss	0.45	0.41
Validation Loss	0.51	0.52

Table 18: Comparison of loss and Accuracy of initial and final epochs with Learning rate scheduler

Batch Normalization:

The process of batch normalization is used to enhance the performance, speed, and stability of neural networks. The goal is to standardize the inputs of each layer such that they have a mean activation output of zero and a unit standard deviation.

It may be challenging for a neural network to converge to a solution when the distribution of inputs to a layer change during training. By normalizing the inputs in each batch of data, batch normalization solves this problem and aids in maintaining a steady distribution of inputs to the layer.

The accuracy we got using the batch normalization technique is 68% which is far less than the base accuracy.

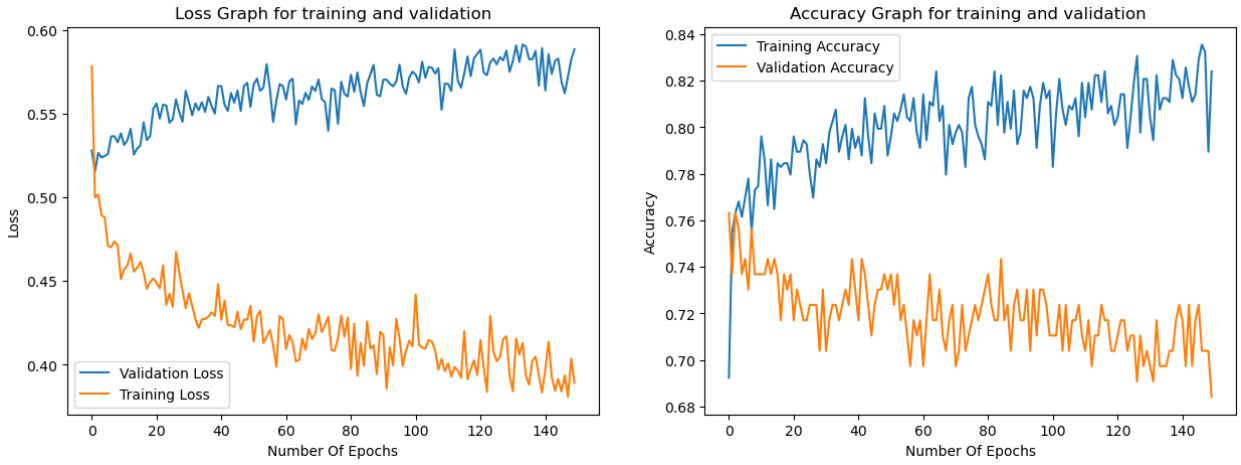


Fig: Comparison of loss and Accuracy plot with Batch Normalization

From the above plot we can describe with the increase of epochs the training accuracy increased from 77% to 82% while the validation accuracy decreased from 73% to 68%.

At initial the training loss decreased suddenly and after that it was decreasing steadily with number of epochs getting increased while the validation loss increased from 0.53 to 0.58.

	Initial Epoch	Final Epoch
Training Accuracy	77%	82%
Validation Accuracy	73%	68%
Training Loss	0.45	0.38
Validation Loss	0.53	0.58

Table 19: Comparison of loss and Accuracy of initial and final epochs with Batch Normalization

Gradient Clipping:

Gradient Clipping is a technique that modifies or clips the error derivative to a threshold during backward propagation through the network and updates the weights using the clipped gradients.

It forces the gradient values to a specific minimum or maximum value if the gradient exceeded an expected range. We set a threshold value and if the gradient is more than that then it is clipped.

if max_norm is not None:

torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm)

clip_grad_norm_ function is used to clip the gradient norms of the parameters of a neural network model. The max_norm parameter specifies the maximum norm value allowed for the gradients. If any gradient's norm exceeds max_norm, then all gradients are rescaled to have a norm of max_norm.

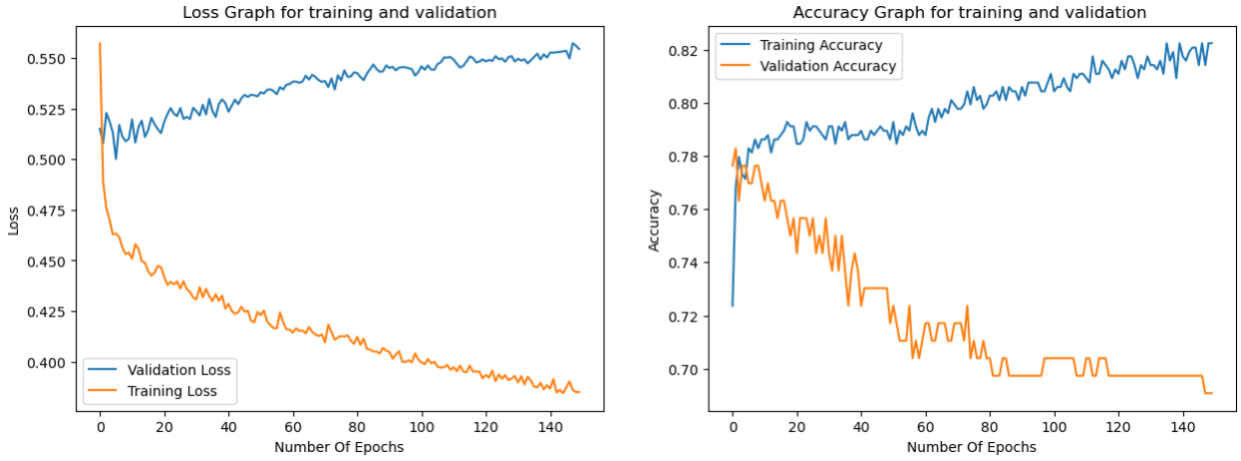


Fig: Comparison of loss and Accuracy plot with Gradient Clipping

From the above plot we can depict that the initial training and validation accuracy is at 78% and 76% respectively, then from there the accuracy started splitting in the opposite direction while the training accuracy increased to 82% but the validation decreased to 69%.

The training loss decreased from 0.45 to 0.38 while model started fitting but the model does not fit on the validation data, the validation loss increases from 0.51 to 0.55.

	Initial Epoch	Final Epoch
Training Accuracy	78%	82%
Validation Accuracy	76%	69%
Training Loss	0.45	0.38
Validation Loss	0.51	0.55

Table 20: Comparison of loss and Accuracy of initial and final epochs with Gradient Clipping

Part III: Implementing & Improving AlexNet

Introduction:

The dataset includes 30,000 examples of 64x64 photos from three different classes: dogs, vehicles, and food. The goal is to categorize photos into one of the three groups. Convolutional Neural Network (CNN)-based image recognition model is used for training and testing the dataset.

We are dealing with image data, specifically a dataset of 64x64 pixel photos of dogs, automobiles, and food. 10,000 samples apiece for each category make up the dataset's total 30,000 samples. There is only the image itself connected with each entry in the dataset, which means that each image in the dataset represents one entry. Consequently, the dataset consists of 30,000 items and 1 variable, which is the image data.

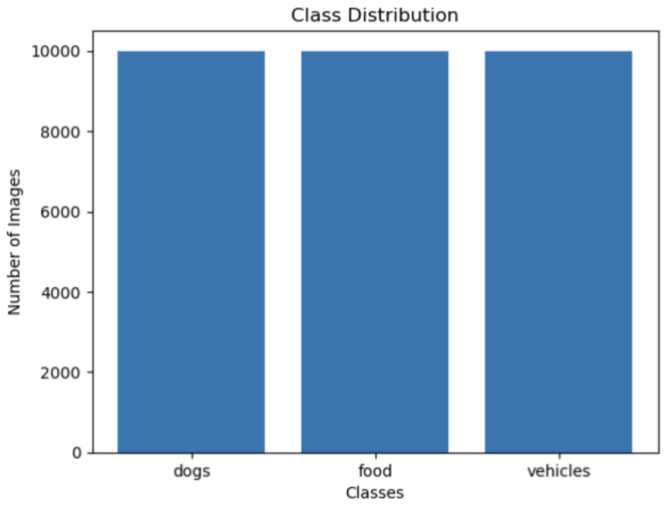
The main statistics about the entries of the dataset is shown below.

```
print("Number of images in the dataset:", len(dataset))
print("Number of classes in the dataset:", len(dataset.classes))
```

```
Number of images in the dataset: 30000
Number of classes in the dataset: 3
```

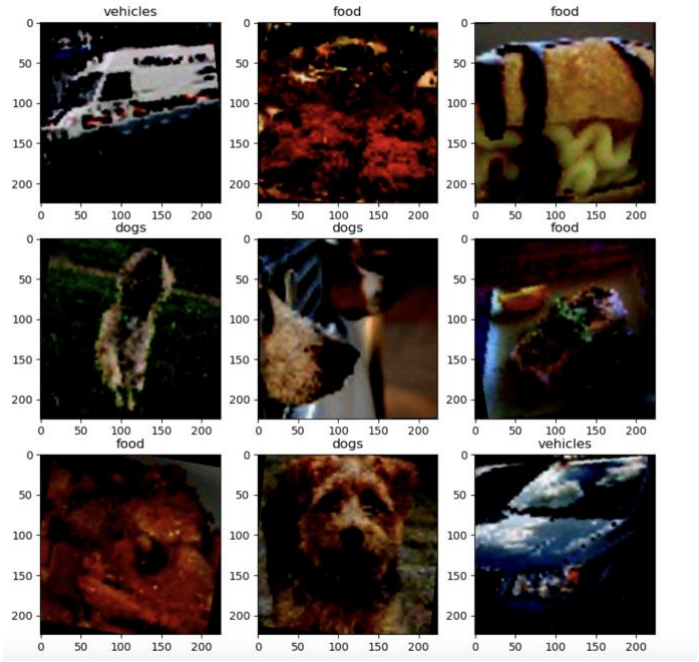
Visualizations:

1. Bar chart of class distribution



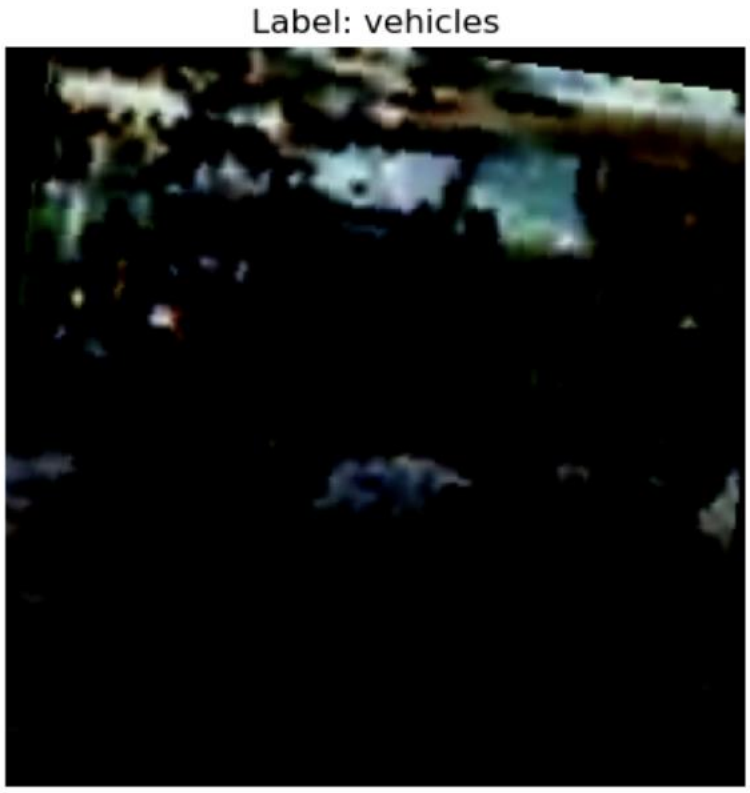
This graph shows the number of images in each class. Each of the three classes, which include dogs, food, and vehicles, contains 10,000 images. This totals 30,000 images in all. we have three class present and there are 10,000 images in each class i.e., dogs, food, vehicles. This adds up to a total of 30,000 images.

2. Sample images



This graph shows a few sample images from the dataset. Each row of a 3x3 grid of images randomly selected from the dataset represents a different class (dogs, food, and vehicles). It is a visual representation of the dataset that shows a sample of images from each class, which can help illustrate the visual qualities of each class and the overall complexity of the dataset.

3. Random image with its label:



This graph shows a single image from the dataset along with its label. It is a visual representation of one image and its label from the dataset that can be helpful for studying the qualities of certain images and confirming the accuracy of the labels in the dataset.

AlexNet Model:

The AlexNet Model implemented here consists of five convolutional layers, three fully connected layers, and a SoftMax layer for classification.

The first layer of the model is a convolutional layer with 96 filters, each of size 11x11 and stride 4, followed by a batch normalization layer to normalize the outputs. The output of the first layer is passed through a rectified linear unit (ReLU) activation function and then through a max-pooling layer of size 3x3 and stride 2, which down samples the output of the first layer.

The second layer of the model is also a convolutional layer with 256 filters, each of size 5x5 and stride 1, followed by batch normalization and ReLU activation. The output of the second layer is then passed through another max-pooling layer of size 3x3 and stride 2.

The third, fourth, and fifth layers of the model are all convolutional layers with 384, 384, and 256 filters, respectively. All these layers use a kernel size of 3x3 and stride 1 and are followed by batch normalization and ReLU activation. Unlike the first two layers, these layers do not use any pooling.

After the last convolutional layer, the output of the model is flattened and passed through two fully connected layers with 4096 neurons each. Both these layers use ReLU activation and dropout regularization with a probability of 0.5 to prevent overfitting.

The final layer of the model is another fully connected layer with the number of neurons equal to the number of classes in the dataset. This layer uses a SoftMax activation function to output a vector of probabilities for each class, and the class with the highest probability is selected as the predicted class for the input image.

Testing and Training Accuracy:

```
model = AlexNet().to(device)
validation_loss,training_loss,train_acc,valid_acc = training(model,train_loader, val_loader, epochs = 10)
```

```
run started at 2023-04-13 19:43:25.210843
2023-04-13 19:46:44.895508
Epoch [1/10], Train Loss: 0.4332, Train Acc: 0.8343, Val Loss: 0.5665, Val Acc: 0.7913
2023-04-13 19:48:50.197694
Epoch [2/10], Train Loss: 0.3187, Train Acc: 0.8843, Val Loss: 0.4448, Val Acc: 0.8473
2023-04-13 19:50:54.398155
Epoch [3/10], Train Loss: 0.2731, Train Acc: 0.8978, Val Loss: 0.2614, Val Acc: 0.9057
2023-04-13 19:52:57.986297
Epoch [4/10], Train Loss: 0.2560, Train Acc: 0.9035, Val Loss: 0.2662, Val Acc: 0.8867
2023-04-13 19:55:01.522496
Epoch [5/10], Train Loss: 0.2367, Train Acc: 0.9123, Val Loss: 0.2536, Val Acc: 0.9040
2023-04-13 19:57:04.097238
Epoch [6/10], Train Loss: 0.2121, Train Acc: 0.9226, Val Loss: 0.1791, Val Acc: 0.9338
2023-04-13 19:59:06.870059
Epoch [7/10], Train Loss: 0.1985, Train Acc: 0.9265, Val Loss: 0.1797, Val Acc: 0.9362
2023-04-13 20:01:09.129631
Epoch [8/10], Train Loss: 0.1955, Train Acc: 0.9297, Val Loss: 0.2226, Val Acc: 0.9138
2023-04-13 20:03:11.658065
Epoch [9/10], Train Loss: 0.1808, Train Acc: 0.9344, Val Loss: 0.2200, Val Acc: 0.9150
2023-04-13 20:05:14.893914
Epoch [10/10], Train Loss: 0.1756, Train Acc: 0.9369, Val Loss: 0.1815, Val Acc: 0.9312
run ended at 2023-04-13 20:05:14.894065
```

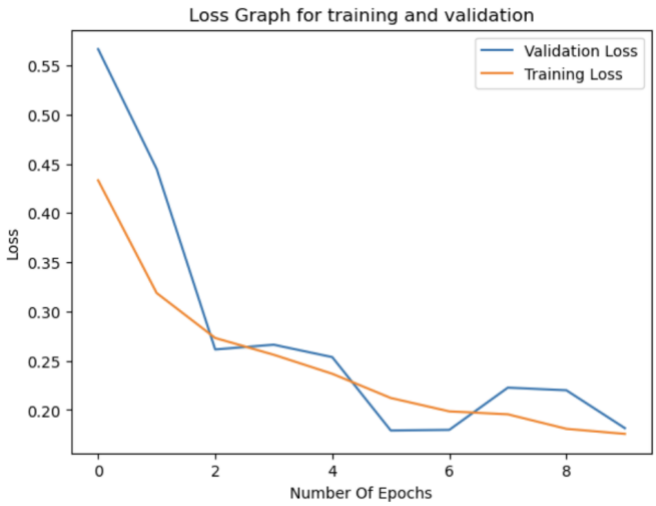
The Alex Net Model mentioned above had a final testing accuracy of 93.50% and a test loss of 0.1793.

Loss and Accuracy Graphs for Training and Validation:

The below is code to find the accuracy and loss for both the training and validation data.

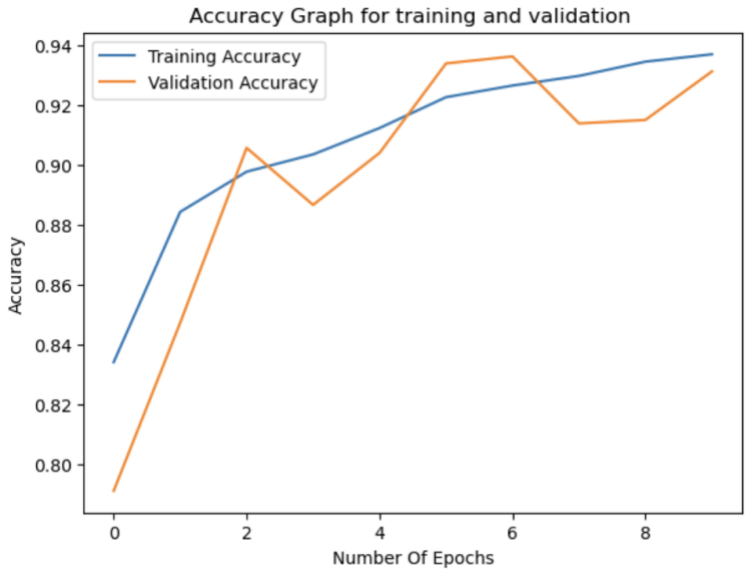
```
loss_acc_graphs(validation_loss,training_loss,train_acc,valid_acc)
```

Loss Graph for training and validation data:



The above graph shows the Loss Graph for training and validation data. The Loss is found for different number of epochs. Both the validation and training loss are decreasing when the number of epochs increases. The Validation Loss starts from 0.5665 and decreases to 0.1815. The Training Loss starts from 0.4332 and decreases gradually to 0.1756.

Accuracy Graph for training and validation data:



The above graph shows the Accuracy Graph for training and validation data. The Accuracy is found for different number of epochs. Both the validation and training accuracy are increasing when the number of epochs increases. The Validation Accuracy starts from 79.13% and gradually increases to 93.12%. The Training Accuracy starts from 83.43% and increases to 93.69%.

Early Stopping:

Here, we are using Early stopping method on the AlexNet model to improve the training accuracy and training time.

```
# training_es(model, train_loader, val_loader)
validation_loss,training_loss,train_acc,valid_acc = training_es(model, train_loader, val_loader, epochs=10)
```

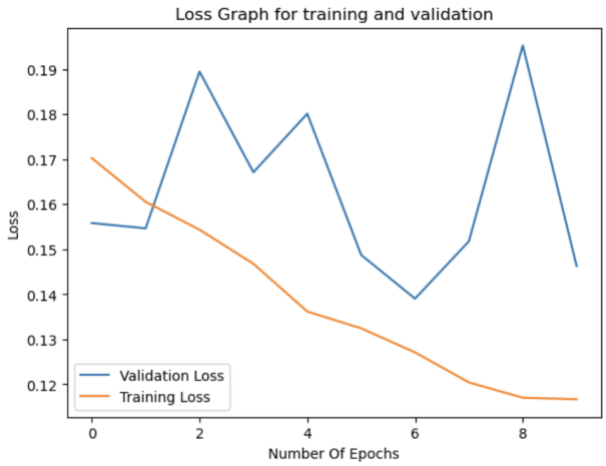
```
run started at 2023-04-13 20:05:35.742798
2023-04-13 20:07:37.152766
Epoch [1/10], Train Loss: 0.1703, Train Acc: 0.9393, Val Loss: 0.1558, Val Acc: 0.9423
2023-04-13 20:09:39.119457
Epoch [2/10], Train Loss: 0.1605, Train Acc: 0.9410, Val Loss: 0.1547, Val Acc: 0.9402
2023-04-13 20:11:41.389696
Epoch [3/10], Train Loss: 0.1544, Train Acc: 0.9437, Val Loss: 0.1895, Val Acc: 0.9305
2023-04-13 20:13:44.076299
Epoch [4/10], Train Loss: 0.1468, Train Acc: 0.9465, Val Loss: 0.1671, Val Acc: 0.9335
2023-04-13 20:15:46.676418
Epoch [5/10], Train Loss: 0.1362, Train Acc: 0.9497, Val Loss: 0.1801, Val Acc: 0.9390
2023-04-13 20:17:48.709430
Epoch [6/10], Train Loss: 0.1325, Train Acc: 0.9514, Val Loss: 0.1488, Val Acc: 0.9450
2023-04-13 20:19:50.329181
Epoch [7/10], Train Loss: 0.1271, Train Acc: 0.9534, Val Loss: 0.1390, Val Acc: 0.9535
2023-04-13 20:21:51.338687
Epoch [8/10], Train Loss: 0.1204, Train Acc: 0.9555, Val Loss: 0.1518, Val Acc: 0.9433
2023-04-13 20:23:52.802509
Epoch [9/10], Train Loss: 0.1170, Train Acc: 0.9572, Val Loss: 0.1953, Val Acc: 0.9315
2023-04-13 20:25:53.693105
Epoch [10/10], Train Loss: 0.1167, Train Acc: 0.9568, Val Loss: 0.1463, Val Acc: 0.9492
run ended at 2023-04-13 20:25:53.693324
```

Despite implementing early stopping, the model was required to complete 10 epochs as there was minimal impact on the validation loss. Nevertheless, the accuracy of the model improved significantly to 96.68% as compared to the original model.

Overall, The Early Stopping Method used here helps to decrease the training time.

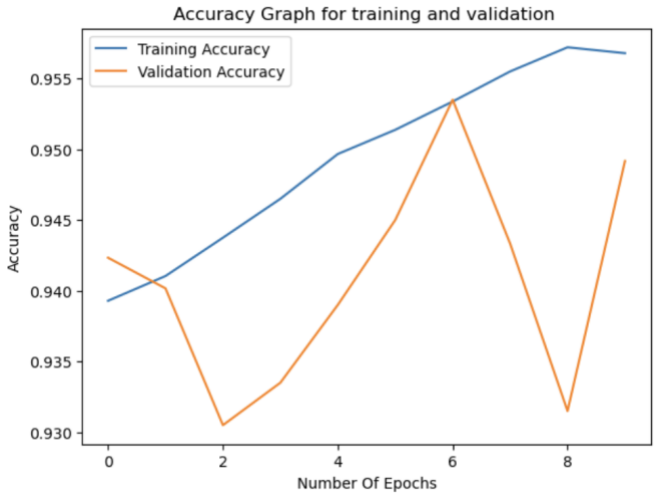
Loss and Accuracy Graphs for Training and Validation After Early Stopping:

Loss Graph for training and validation data:



The above graph shows the Loss Graph for training and validation data. The Loss is found for different number of epochs. The Validation Loss is increasing at beginning and reached the peak and then decreased to be approximately constant along the remaining epochs. The Training Loss is slighting decreasing as the number of epochs is increasing.

Accuracy Graph for training and validation data:



The above graph shows the Accuracy Graph for training and validation data. The Accuracy is found for different number of epochs. The Validation Accuracy decreased and fell to least and increased and reached a constant value along the remaining epochs. The Training Accuracy slightly increased as the number of the epochs increased.

Part IV: Optimizing CNN + Data Argumentation

The dataset used is the SVHN (Street View House Numbers) dataset, which is a collection of digit images extracted from Google Street View. The dataset consists of 32x32 RGB images of house numbers and their corresponding labels. Recognizing the digit that the image represents is the dataset's objective. The dataset also includes labels for each image, indicating the digit that the image represents.

The data is in the form of images, with each image representing a single digit from 0 to 9. Each image has three color channels (RGB) and is of size 32x32 pixels. The dataset consists of 26,032 images for testing and 73,257 images for training. Each image in the dataset has labels that specify the digit it represents. There are 10 classes (0-9).

Consequently, the dataset consists of 73257 items and 2 variables, which is the image data.

```
svhn_transform = transforms.Compose([ transforms.Resize(255),
                transforms.CenterCrop(224),transforms.ToTensor()])
svhn_dataset=torchvision.datasets.SVHN(root='./data', download=True, transform=svhn_transform)
len(svhn_dataset)
```

```
Using downloaded and verified file: ./data/train_32x32.mat
```

```
73257
```

```
print("Number of entries (images) in SVHN dataset: ", len(svhn_dataset))
print("Number of variables in SVHN dataset: ", len(svhn_dataset[0]))
```

```
Number of entries (images) in SVHN dataset:  73257
Number of variables in SVHN dataset:  2
```

```
svhn_train_data=torchvision.datasets.SVHN(root='./data',split='train', download=True, transform=svhn_transform)
len(svhn_train_data)
```

```
Using downloaded and verified file: ./data/train_32x32.mat
```

```
73257
```

```
svhn_test_data = torchvision.datasets.SVHN(root='./data',split='test', download=True, transform=svhn_transform)
len(svhn_test_data)
```

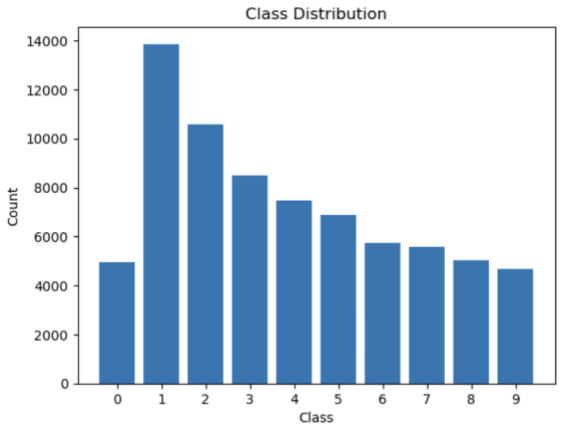
```
Using downloaded and verified file: ./data/test_32x32.mat
```

```
26032
```

The first model, SVHN_AlexNet, is an adaptation of the original AlexNet architecture that was created specifically to be compatible with the SVHN dataset. Its overall architecture is the same as Alex Net's, with five convolutional layers followed by three fully connected layers, but the final fully connected layer's number of neurons and final convolutional layer's output channels have been changed to correspond to the number of classes in the SVHN dataset.

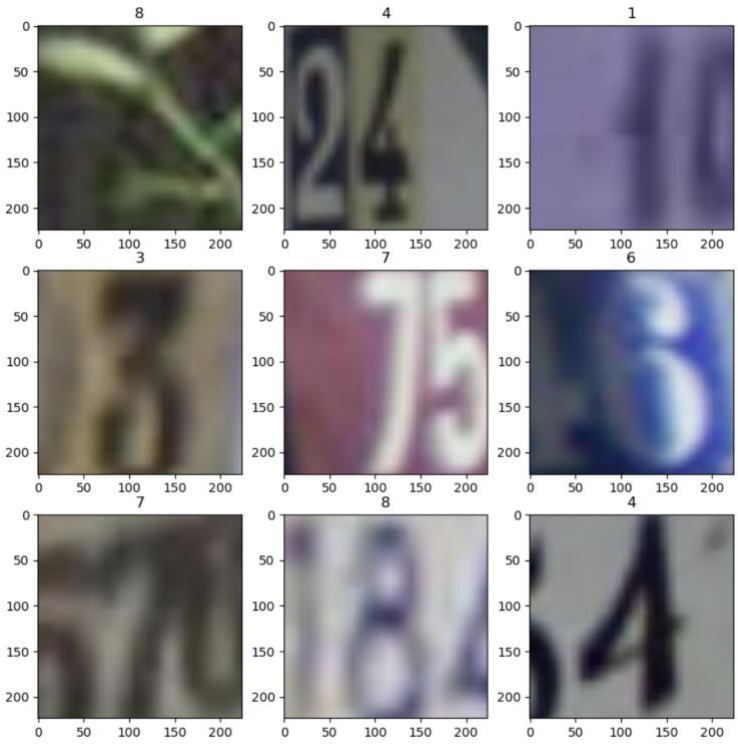
Visualizations:

Bar chart of class distribution:



This graph shows the number of images in each class. Each bar shows the count of images in that class.

Sample images:



This graph shows a few sample images from the dataset. A 3X3 grid image is given as an output which has different images of different house number from the dataset.

Random image with its label:

This graph shows a single image from the dataset along with its label. The Output gives the house number from the image given i.e., it identifies and labels the house number in the image.

Data Augmentation:

The following transforms (Data Augmentation technique) are performed to double the data.

Method 1:

```
aug_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.RandomCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

```
aug_train_data=torchvision.datasets.SVHN(root='./data',split='train', download=True, transform=aug_transform)
len(aug_train_data)
```

```
Using downloaded and verified file: ./data/train_32x32.mat
```

```
73257
```

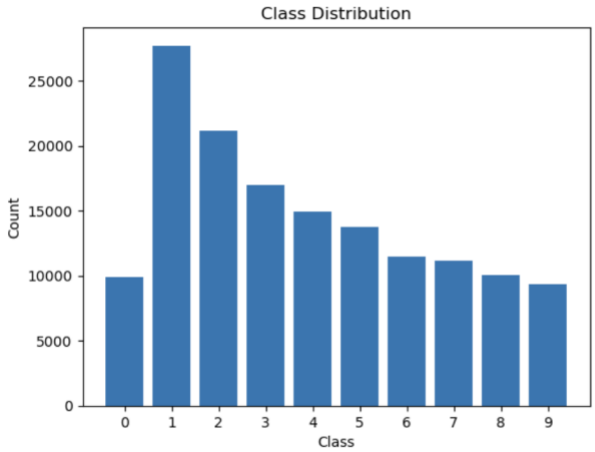
```
dataset_aug=torch.utils.data.ConcatDataset([svhn_train_data,aug_train_data])
```

- transforms.Resize(256): Resizes the image to 256 pixels on the shorter side while maintaining the aspect ratio.
- transforms.RandomCrop(224): Randomly crops a square patch of size 224 x 224 pixels from the resized image.
- transforms.RandomHorizontalFlip(): Randomly flips the image horizontally with a given probability.

- transforms.RandomRotation(10): Randomly rotates the image by a maximum of 10 degrees.
- transforms.ToTensor(): Converts the image from PIL Image format to PyTorch Tensor format.
- transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)): Normalizes the tensor by subtracting the mean (0.5, 0.5, 0.5) and dividing by the standard deviation (0.5, 0.5, 0.5) for each channel. This helps to reduce the effect of illumination variations in the images.

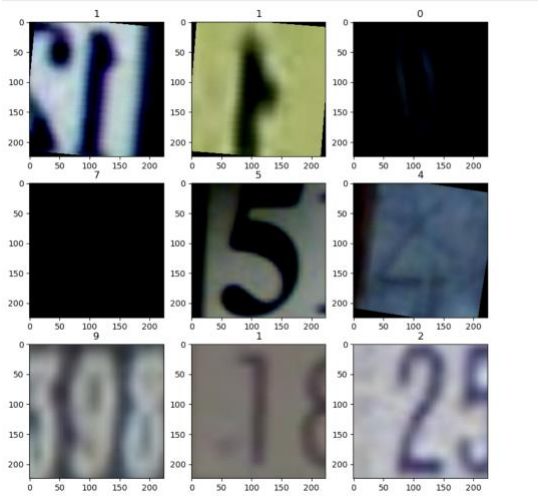
Visualizations for Data Augmentation (Method 1):

Bar chart of class distribution:



The graph provides an easy way to visualize the distribution of the different classes in the dataset. Each bar in the graph represents the count of each class in the dataset, and the height of the bar represents the count of the corresponding class.

Sample images:



This graph shows a few sample images from the dataset. A 3x3 grid of images chosen at random from the augmented dataset is produced. Every image is shown alongside its appropriate label, which appears as the image's title. This can be helpful for seeing the images in the dataset after data augmentation and for determining whether the method can be used to produce realistic and varied images.

Random image with its label:

This graph shows a single image from the dataset along with its label. It is a visual representation of one image and its label from the dataset that can be helpful for studying the qualities of certain images and confirming the accuracy of the labels in the dataset.

Testing and Training Accuracy for Data Augmentation (Method 1):

```
svhn_model = SVHN_AlexNet().to(device)
validation_loss,training_loss,train_acc,valid_acc =
training(svhn_model,svhn_train_loader, svhn_val_loader, epochs = 10)
```

```
run started at 2023-04-13 17:44:35.504767
2023-04-13 17:51:40.731481
Epoch [1/10], Train Loss: 0.9566, Train Acc: 0.6770, Val Loss: 0.4646, Val Acc: 0.8580
2023-04-13 17:58:50.074727
Epoch [2/10], Train Loss: 0.4175, Train Acc: 0.8707, Val Loss: 0.3406, Val Acc: 0.8966
2023-04-13 18:05:56.123845
Epoch [3/10], Train Loss: 0.3421, Train Acc: 0.8959, Val Loss: 0.3048, Val Acc: 0.9079
2023-04-13 18:13:01.713300
Epoch [4/10], Train Loss: 0.3026, Train Acc: 0.9098, Val Loss: 0.2760, Val Acc: 0.9213
2023-04-13 18:20:13.452767
Epoch [5/10], Train Loss: 0.2720, Train Acc: 0.9194, Val Loss: 0.2464, Val Acc: 0.9276
2023-04-13 18:27:23.184836
Epoch [6/10], Train Loss: 0.2527, Train Acc: 0.9254, Val Loss: 0.2300, Val Acc: 0.9330
2023-04-13 18:34:34.164006
Epoch [7/10], Train Loss: 0.2340, Train Acc: 0.9305, Val Loss: 0.2232, Val Acc: 0.9375
2023-04-13 18:41:42.862729
Epoch [8/10], Train Loss: 0.2156, Train Acc: 0.9359, Val Loss: 0.2108, Val Acc: 0.9397
2023-04-13 18:48:56.239458
Epoch [9/10], Train Loss: 0.2028, Train Acc: 0.9411, Val Loss: 0.2083, Val Acc: 0.9395
2023-04-13 18:56:03.643682
Epoch [10/10], Train Loss: 0.1921, Train Acc: 0.9434, Val Loss: 0.1979, Val Acc: 0.9423
run ended at 2023-04-13 18:56:03.643878
```

```
final_test_loss, final_test_acc = testing(svhn_model, svhn_test_loader)
Base_loss, Base_acc = final_test_loss, final_test_acc

# if Base_acc >= 0.9:
#     torch.save(model.state_dict(), 'weights.pt')
```

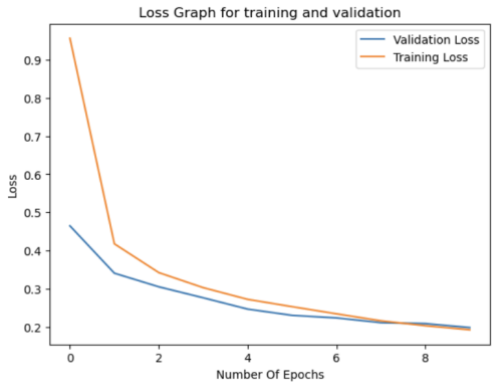
```
Test Loss: 0.1958, Test Acc: 0.9438
```

The Model mentioned above had a final testing accuracy of 94.38% and a test loss of 0.1958.

Loss and Accuracy Graphs for Training and Validation for Data Augmentation (Method 1):

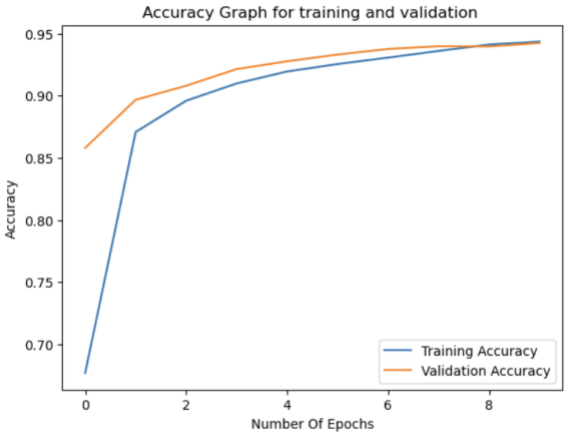
```
loss_acc_graphs(validation_loss,training_loss,train_acc,valid_acc)
```

Loss Graph for training and validation data:



The above graph shows the Loss Graph for training and validation data. The Loss is found for different number of epochs. Both the validation and training loss are decreasing when the number of epochs increases. The Validation Loss starts from 0.4646 and decreases to 0.1979. The Training Loss starts from 0.9566 and decreases gradually to 0.1921.

Accuracy Graph for training and validation data:



The above graph shows the Accuracy Graph for training and validation data. The Accuracy is found for different number of epochs. Both the validation and training accuracy are increasing when the number of epochs increases. The Validation Accuracy starts from 85.80% and gradually increases to 94.23%. The Training Accuracy starts from 67.70% and increases to 94.34%.

Data Augmentation:

Method 2:

```
aug_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.RandomAffine(degrees=15, translate=(0.1, 0.1), scale=(0.9, 1.1), shear=10),
    transforms.RandomResizedCrop(size=224, scale=(0.8, 1.0), ratio=(3.0/4.0, 4.0/3.0)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

```
aug_train_data=torchvision.datasets.SVHN(root='./data',split='train', download=True, transform=aug_transform)
len(aug_train_data)
```

```
Using downloaded and verified file: ./data/train_32x32.mat
```

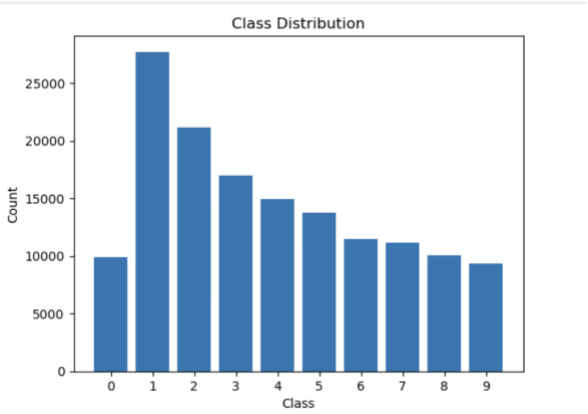
```
73257
```

```
dataset_aug=torch.utils.data.ConcatDataset([svhn_train_data,aug_train_data])
```

- transforms.Resize(256): Resizes the image to have a height and width of 256 pixels.
- transforms.CenterCrop(224): Crops the center of the image to have a height and width of 224 pixels.
- transforms.RandomHorizontalFlip(): Flips the image horizontally with a given probability.
- transforms.RandomRotation(10): Rotates the image by a random angle between -10 and 10 degrees.
- transforms.ColorJitter(): Adjusts the brightness, contrast, saturation, and hue of the image with random values within a certain range.
- transforms.RandomAffine(): Applies a random affine transformation to the image, which includes translation, scaling, and shearing.
- transforms.RandomResizedCrop(): Resizes and crops the image to a random size and aspect ratio within a certain range.
- transforms.ToTensor(): Converts the image to a PyTorch tensor.
- transforms.Normalize(): Normalizes the pixel values of the tensor to have a mean and standard deviation in each channel.

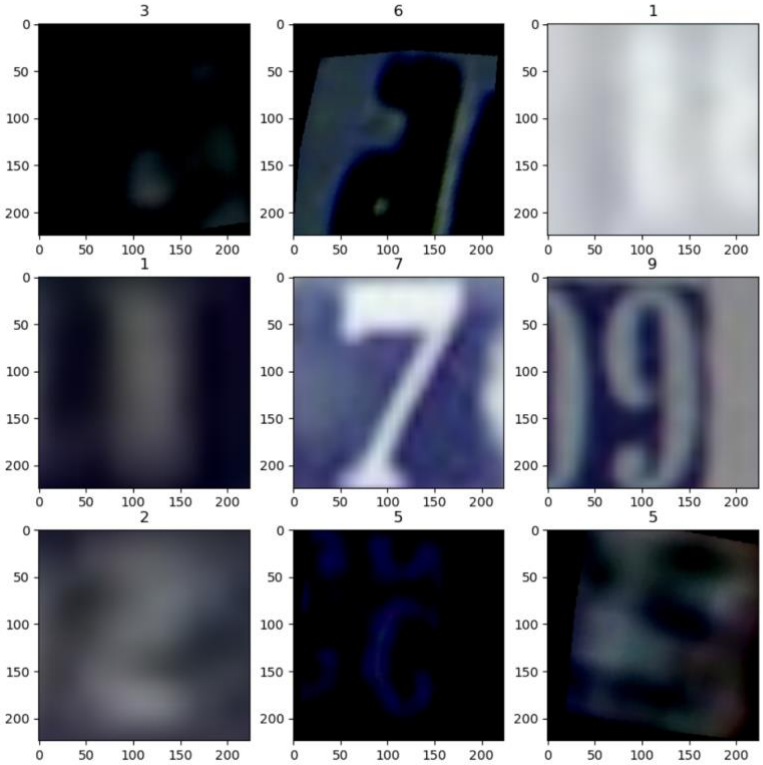
Visualizations for Data Augmentation (Method 2):

Bar chart of class distribution:



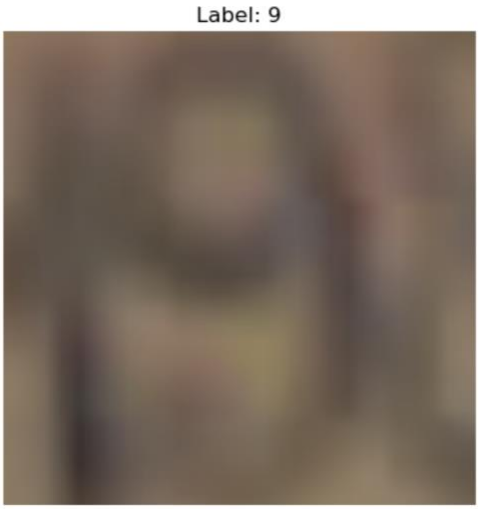
An accessible approach to see how the various classes in the dataset are distributed is through the graph. The height of each bar in the graph indicates the number of instances of each class in the dataset, and vice versa.

Sample images:



This graph displays a few sample images from the dataset. The dataset is used to generate a 3x3 grid of randomly selected images. Each image has a label next to it that corresponds to the image's title. This can be useful for seeing the images in the dataset after data augmentation and for figuring out whether the technique can be utilized to create realistic and varied images.

Random image with its label:



A single image from the dataset is displayed in this graph along with its label. It is a graphic depiction of a single image and its associated label from the dataset that can be useful for examining the characteristics of images and verifying the validity of the labels in the dataset.

Testing and Training Accuracy for Data Augmentation (Method 2):

```
svhn_model = SVHN_AlexNet().to(device)
validation_loss,training_loss,train_acc,valid_acc =
training(svhn_model,svhn_train_loader, svhn_val_loader, epochs = 5)
```

```
run started at 2023-04-13 21:02:54.889350
2023-04-13 21:17:38.090998
Epoch [1/5], Train Loss: 1.0376, Train Acc: 0.6480, Val Loss: 0.5036, Val Acc: 0.8410
2023-04-13 21:32:17.866497
Epoch [2/5], Train Loss: 0.4810, Train Acc: 0.8491, Val Loss: 0.3914, Val Acc: 0.8828
2023-04-13 21:46:53.218070
Epoch [3/5], Train Loss: 0.3957, Train Acc: 0.8786, Val Loss: 0.3352, Val Acc: 0.8990
2023-04-13 22:01:27.624936
Epoch [4/5], Train Loss: 0.3476, Train Acc: 0.8941, Val Loss: 0.3138, Val Acc: 0.9078
2023-04-13 22:16:04.262070
Epoch [5/5], Train Loss: 0.3177, Train Acc: 0.9039, Val Loss: 0.2879, Val Acc: 0.9148
run ended at 2023-04-13 22:16:04.262244
```

```
final_test_loss, final_test_acc = testing(svhn_model, svhn_test_loader)
Base_loss, Base_acc = final_test_loss, final_test_acc

# if Base_acc >= 0.9:
#     torch.save(model.state_dict(), 'weights.pt')
```

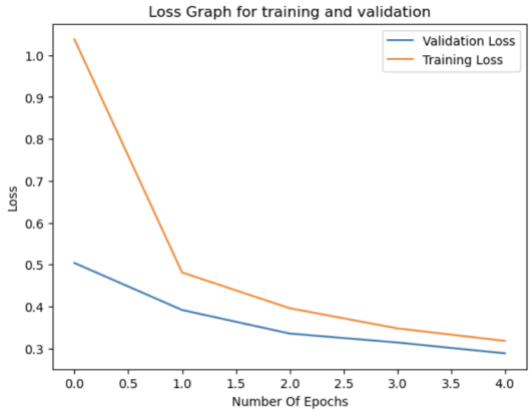
```
Test Loss: 0.2920, Test Acc: 0.9136
```

The Model mentioned above had a final testing accuracy of 91.36% and a test loss of 0.2920.

Loss and Accuracy Graphs for Training and Validation for Data Augmentation (Method 2):

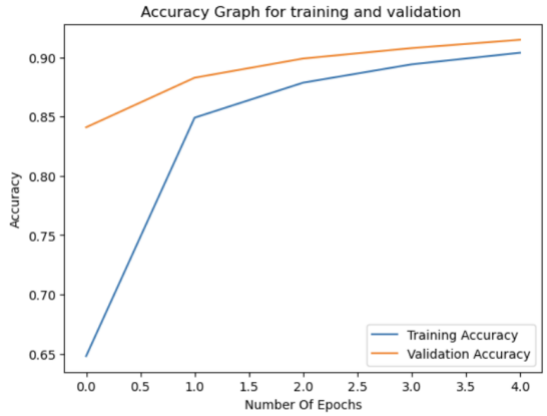
```
loss_acc_graphs(validation_loss,training_loss,train_acc,valid_acc)
```

Loss Graph for training and validation data:



The above graph shows the Loss Graph for training and validation data. The Loss is found for different number of epochs. Both the validation and training loss are decreasing when the number of epochs increases. The Validation Loss starts from 0.5036 and decreases to 0.2879. The Training Loss starts from 1.0376 and decreases gradually to 0.3177.

Accuracy Graph for training and validation data:



The above graph shows the Accuracy Graph for training and validation data. The Accuracy is found for different number of epochs. Both the validation and training accuracy are increasing when the number of epochs increases. The Validation Accuracy starts from 84.10% and gradually increases to 91.48%. The Training Accuracy starts from 64.80% and increases to 90.39%.

Contribution:

Team Member	Assignment Part	Contribution
Preethi Abhilasha Vaddi	Part-1	50%
Vinitha Vudhayagiri	Part-1	50%
Preethi Abhilasha Vaddi	Part-2	50%
Vinitha Vudhayagiri	Part-2	50%
Preethi Abhilasha Vaddi	Part-3	50%
Vinitha Vudhayagiri	Part-3	50%
Preethi Abhilasha Vaddi	Part-4	50%
Vinitha Vudhayagiri	Part-4	50%
Preethi Abhilasha Vaddi	Bonus	50%
Vinitha Vudhayagiri	Bonus	50%

References:

https://keras.io/api/optimizers/adam/

https://towardsdatascience.com/a-look-at-gradient-descent-and-rmsprop-optimizers-f77d483ef08b

https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1

https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/

https://pouannes.github.io/blog/initialization/

https://365datascience.com/tutorials/machine-learning-tutorials/what-is-xavier-initialization/

https://www.programcreek.com/python/example/105105/torchvision.datasets.SVHN

https://debuggercafe.com/pytorch-imagefolder-for-training-cnn-models/

https://pytorch.org/tutorials/beginner/data_loading_tutorial.html#visualize-the-data

https://www.kaggle.com/code/saksham219/plotting-random-satellite-images

https://arxiv.org/pdf/1409.1556.pdf

https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

https://www.kaggle.com/code/leifuer/intro-to-pytorch-loading-image-data

https://blog.paperspace.com/alexnet-pytorch/

https://blog.paperspace.com/vgg-from-scratch-pytorch/

https://pytorch.org/vision/stable/transforms.html

https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html

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

https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html

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

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

https://pytorch.org/ignite/generated/ignite.handlers.early_stopping.EarlyStopping.html

https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html

https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

https://pytorch.org/docs/stable/generated/torch.nn.functional.normalize.html

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

https://www.tensorflow.org/guide/keras/preprocessing_layers#:~:text=The%20Keras%20preprocessing%20layers%20API,part%20of%20a%20Keras%20SavedModel

https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html#torch.optim.lr_scheduler.ReduceLROnPlateau