ASSIGNMENT-2

Members:

1. Tharunsa
2. Saikorup

PART-1

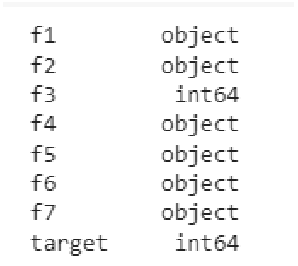
Problem Statement:

To predict the Target variable which is a binary component based on training rest of the independent features of the dataset using the neural network algorithm. Pytorch environment is used to build the Neural Network Algorithm.

Dataset Description:

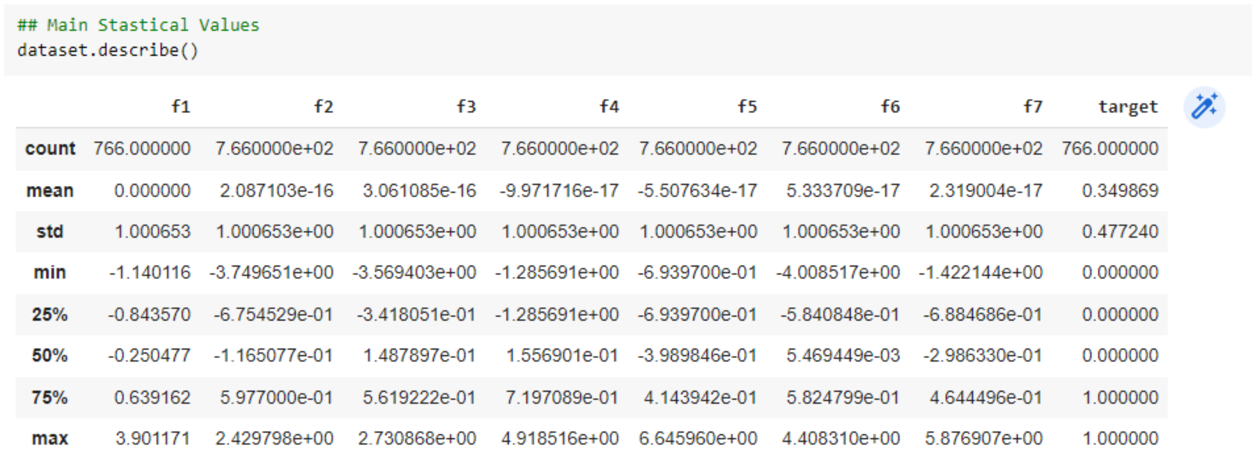
The Dataset consists of 8 attributes and 766 instances or rows in which the target variable is dependent on the rest of the independent variables. The Target variable is a binary component, that is it is of category type we must predict 1/0. The Raw dataset must be preprocessed or cleaned before fitting dataset to the algorithm. The attributes are of different datatypes, some are of object datatype. Therefore, these object type attributes corrected to numeric type. There are no categorical datatype attributes, if there are any categorical type attributes, then those must be changed to object or to any numeric type as our algorithms cannot work for the character type attributes. The Algorithms are built assuming data to be normally distributed, so we must normalize the numerical data before feeding the data to the algorithm.

Column Names: ‘f1’,’f2’,’f3’,’f4’,’f5’,’f6’,’f7’ and ‘target’.



```
f1        object
f2        object
f3         int64
f4        object
f5        object
f6        object
f7        object
target     int64
```

After cleaning, preprocessing and normalizing the dataset, we can now feed the refined data to the model we have built. One of the points in attaining accuracy of the model is how statistically significant the data is, the statistical values of the data can be found by using the describe () command.



```
## Main Stastical Values
dataset.describe()
```

	f1	f2	f3	f4	f5	f6	f7	target
count	766.000000	7.660000e+02	7.660000e+02	7.660000e+02	7.660000e+02	7.660000e+02	7.660000e+02	766.000000
mean	0.000000	2.087103e-16	3.061085e-16	-9.971716e-17	-5.507634e-17	5.333709e-17	2.319004e-17	0.349869
std	1.000653	1.000653e+00	1.000653e+00	1.000653e+00	1.000653e+00	1.000653e+00	1.000653e+00	0.477240
min	-1.140116	-3.749651e+00	-3.569403e+00	-1.285691e+00	-6.939700e-01	-4.008517e+00	-1.422144e+00	0.000000
25%	-0.843570	-6.754529e-01	-3.418051e-01	-1.285691e+00	-6.939700e-01	-5.840848e-01	-6.884686e-01	0.000000
50%	-0.250477	-1.165077e-01	1.487897e-01	1.556901e-01	-3.989846e-01	5.469449e-03	-2.986330e-01	0.000000
75%	0.639162	5.977000e-01	5.619222e-01	7.197089e-01	4.143942e-01	5.824799e-01	4.644496e-01	1.000000
max	3.901171	2.429798e+00	2.730868e+00	4.918516e+00	6.645960e+00	4.408310e+00	5.876907e+00	1.000000

Data Preprocessing:

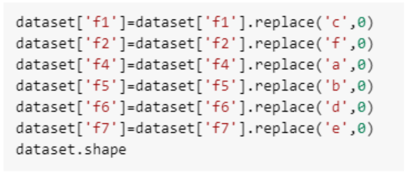
Data preprocessing or data cleaning is done after loading or reading the dataset into python.

After loading the dataset, check for the null values using 'is.na ()'. There are no null values in the columns. Check whether the entire data is of uniform datatype are not, if there is any unusual data in the dataset, the algorithm cannot give us the accurate results, therefore other few steps we have carried out as done below to obtain the best accuracy from the algorithm.

- Dealing String values:

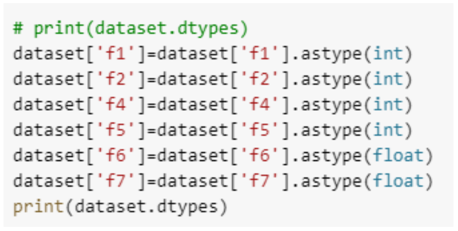
  Some attributes consist of string values, we replaced those string values with the "zero".

  We have replaced them '0,'so need not remove those rows from the dataset. Therefore, we are getting sufficient to train our model.

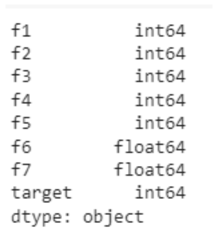


```
dataset['f1']=dataset['f1'].replace('c',0)
dataset['f2']=dataset['f2'].replace('f',0)
dataset['f4']=dataset['f4'].replace('a',0)
dataset['f5']=dataset['f5'].replace('b',0)
dataset['f6']=dataset['f6'].replace('d',0)
dataset['f7']=dataset['f7'].replace('e',0)
dataset.shape
```

- Converting DataTypes:
  Some attributes are in object datatype, even those must be in numeric datatype. Therefore, we have changed object data type to numeric datatype.

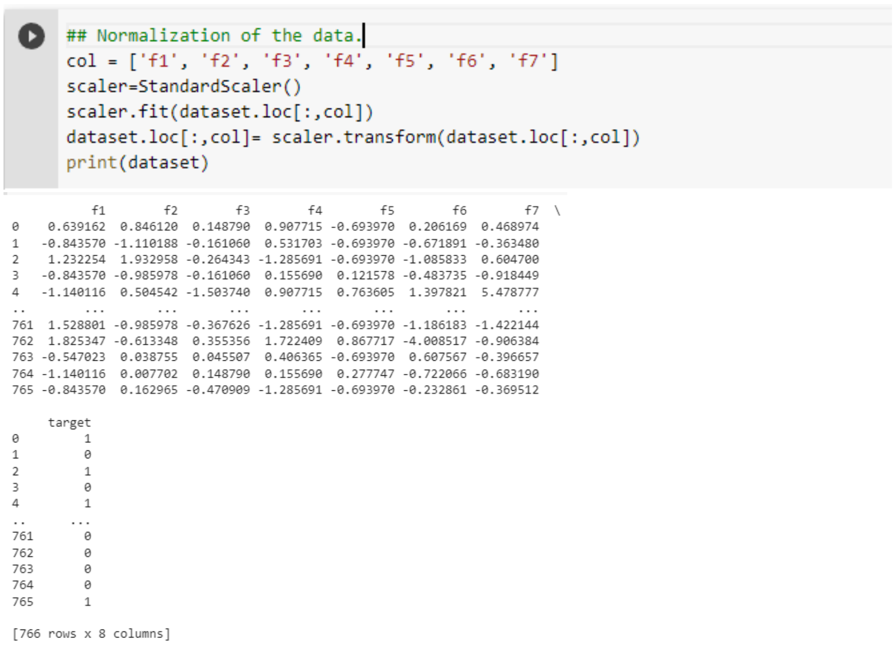


```
# print(dataset.dtypes)
dataset['f1']=dataset['f1'].astype(int)
dataset['f2']=dataset['f2'].astype(int)
dataset['f4']=dataset['f4'].astype(int)
dataset['f5']=dataset['f5'].astype(int)
dataset['f6']=dataset['f6'].astype(float)
dataset['f7']=dataset['f7'].astype(float)
print(dataset.dtypes)
```



```
f1          int64
f2          int64
f3          int64
f4          int64
f5          int64
f6        float64
f7        float64
target      int64
dtype: object
```

- Normalization:
  Most of the algorithms work best for the data with zero bias and 1 as the standard deviation. Normalization helps in faster converging of the data during training the model, as in the normalization the values are scaled to be in the range between 0-1. It also helps in preventing the saturation of the model. Here we are using the standard scaler command to make the data normalized.

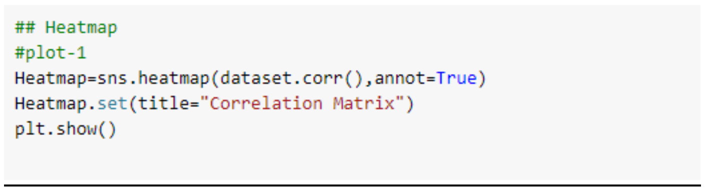


```
## Normalization of the data.
col = ['f1', 'f2', 'f3', 'f4', 'f5', 'f6', 'f7']
scaler=StandardScaler()
scaler.fit(dataset.loc[:,col])
dataset.loc[:,col]= scaler.transform(dataset.loc[:,col])
print(dataset)
```

```
           f1        f2        f3        f4        f5        f6        f7  \
0    0.639162  0.846120  0.148790  0.907715 -0.693970  0.206169  0.468974
1   -0.843570 -1.110188 -0.161060  0.531703 -0.693970 -0.671891 -0.363480
2    1.232254  1.932958 -0.264343 -1.285691 -0.693970 -1.085833  0.604700
3   -0.843570 -0.985978 -0.161060  0.155690  0.121578 -0.483735 -0.918449
4   -1.140116  0.504542 -1.503740  0.907715  0.763605  1.397821  5.478777
..        ...       ...       ...       ...       ...       ...       ...
761  1.528801 -0.985978 -0.367626 -1.285691 -0.693970 -1.186183 -1.422144
762  1.825347 -0.613348  0.355356  1.722409  0.867717 -4.008517 -0.906384
763 -0.547023  0.038755  0.045507  0.406365 -0.693970  0.607567 -0.396657
764 -1.140116  0.007702  0.148790  0.155690  0.277747 -0.722066 -0.683190
765 -0.843570  0.162965 -0.470909 -1.285691 -0.693970 -0.232861 -0.369512

     target
0         1
1         0
2         1
3         0
4         1
..      ...
761       0
762       0
763       0
764       0
765       1

[766 rows x 8 columns]
```

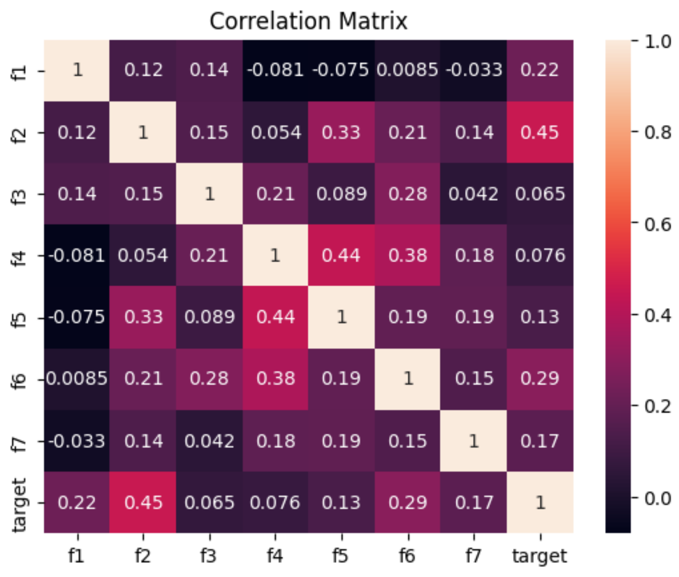
3. Visualization Graphs:

Plot:1



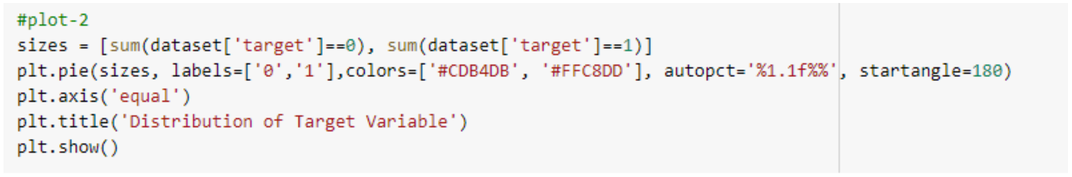
```
## Heatmap
#plot-1
Heatmap=sns.heatmap(dataset.corr(),annot=True)
Heatmap.set(title="Correlation Matrix")
plt.show()
```

HeatMap: HeatMap describes the correlation between the variables. Correlation is the Statistical measure to know how two or more variables are related to each other. We can select the variables depending on our requirement, if we require independent variables then we need to select the Negatively Correlated terms and if we need to draw relations between the variables for any analysis, that is predicting one variable depending on the other variable then we need to select the positively correlated variables.



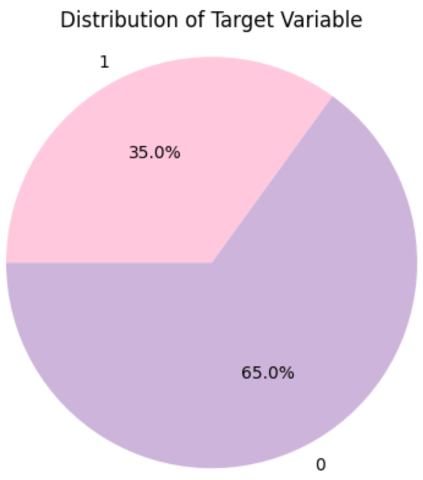
The above graph shows how one variable is correlated to another using the color codes. The diagonal pass gives us the individual variable autocorrelation. so, it is 1.
F2 and target variable are highly correlated with each other whereas F4 and target variable are least correlated with each other.

Plot:2



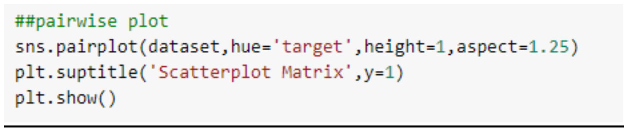
```
#plot-2
sizes = [sum(dataset['target']==0), sum(dataset['target']==1)]
plt.pie(sizes, labels=['0','1'],colors=['#CDB4DB', '#FFC8DD'], autopct='%1.1f%%', startangle=180)
plt.axis('equal')
plt.title('Distribution of Target Variable')
plt.show()
```

Pie Chart:
Here a pie chart is used to describe the spread of the target variable along the data.



From the above graph we can see that thee target variable with ‘0’ is spread over 65% of the data, whereas target variable with ‘1’ is spread over 35% of the data.

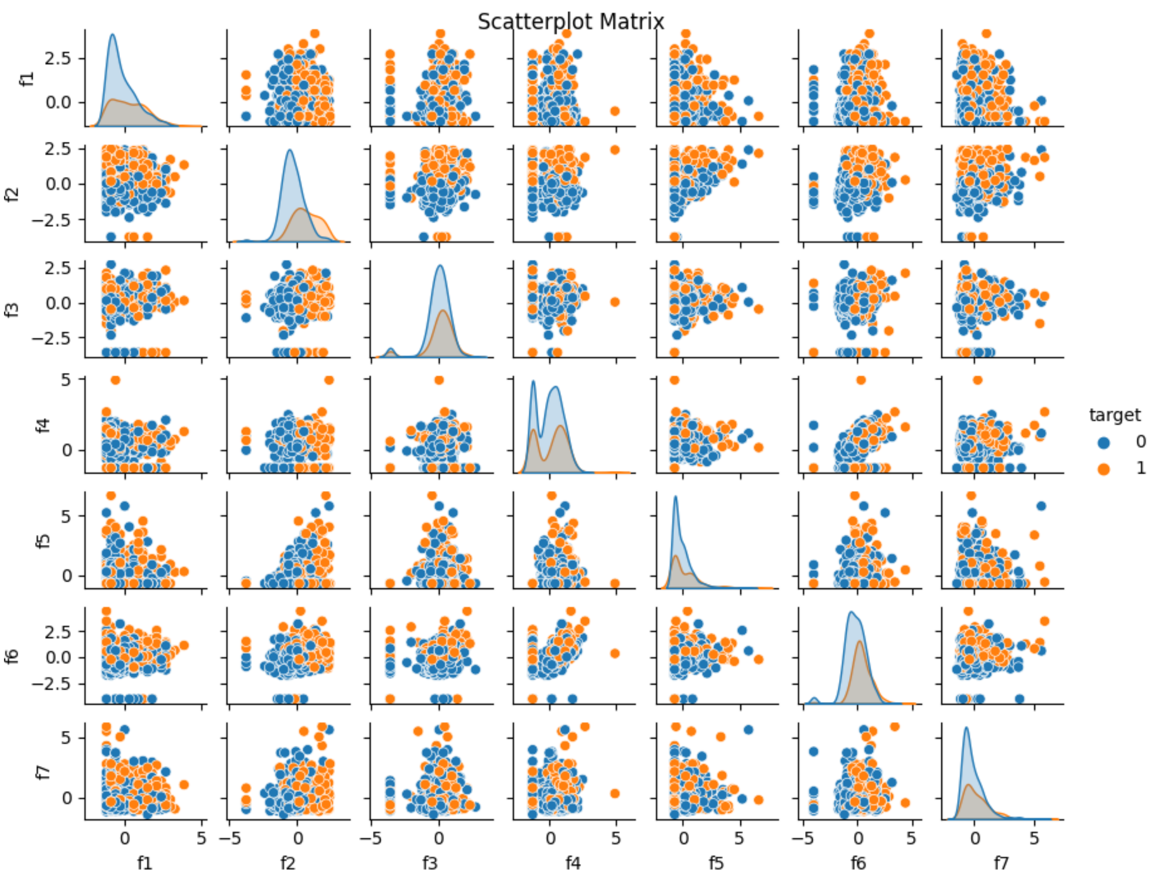
<u>Plot:3</u>



```
##pairwise plot
sns.pairplot(dataset,hue='target',height=1,aspect=1.25)
plt.suptitle('Scatterplot Matrix',y=1)
plt.show()
```

<u>Pair plot:</u>

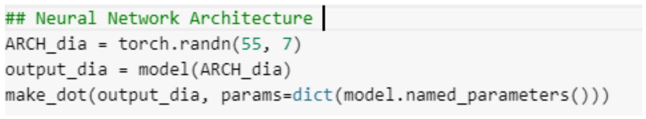
This scatter plot matrix shows the relationship between one variable with another variable in the individual plots. The diagonal plots describe the individual distribution of each feature compared to target variable.



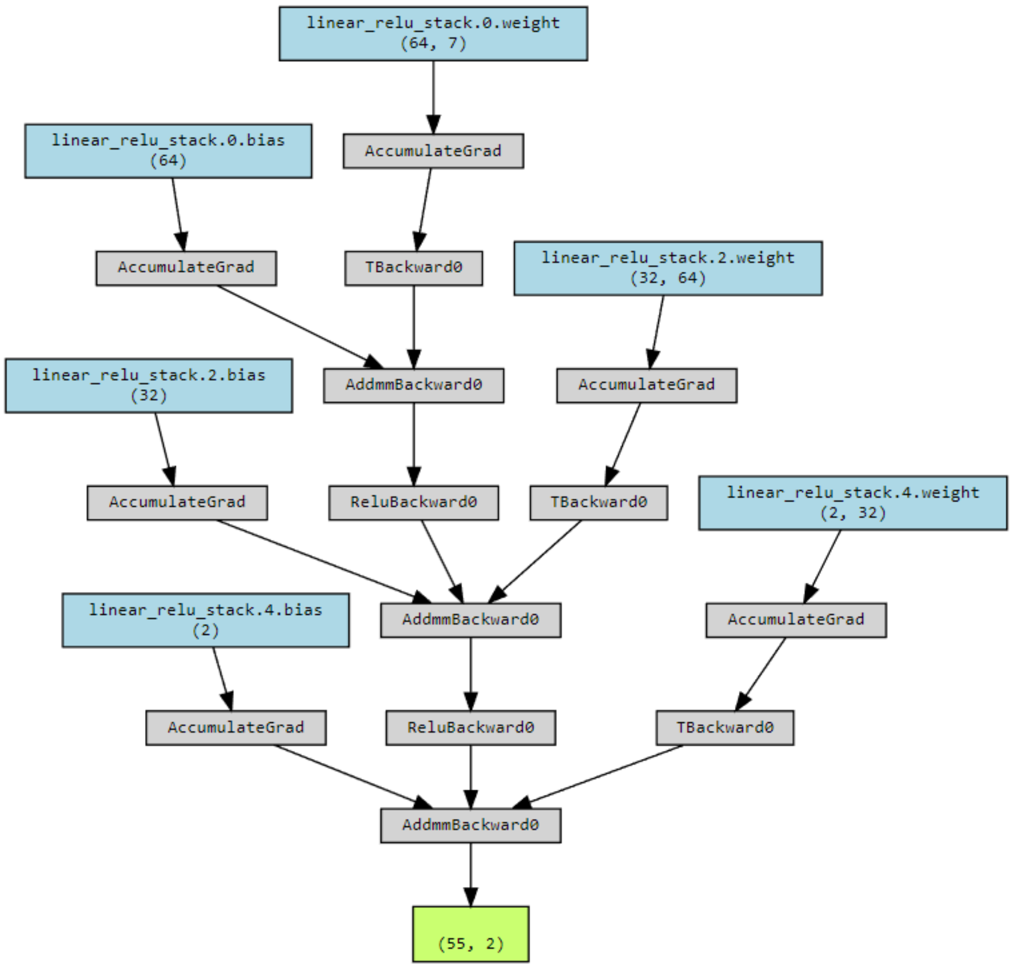
After performing the preprocessing of the data, we built the algorithm right from scratch. And trained the data using the algorithm.

4.Architecture of the Neural Network algorithm:

To get the Neural Network Architecture in pytorch we use the following code.



```
## Neural Network Architecture
ARCH_dia = torch.randn(55, 7)
output_dia = model(ARCH_dia)
make_dot(output_dia, params=dict(model.named_parameters()))
```



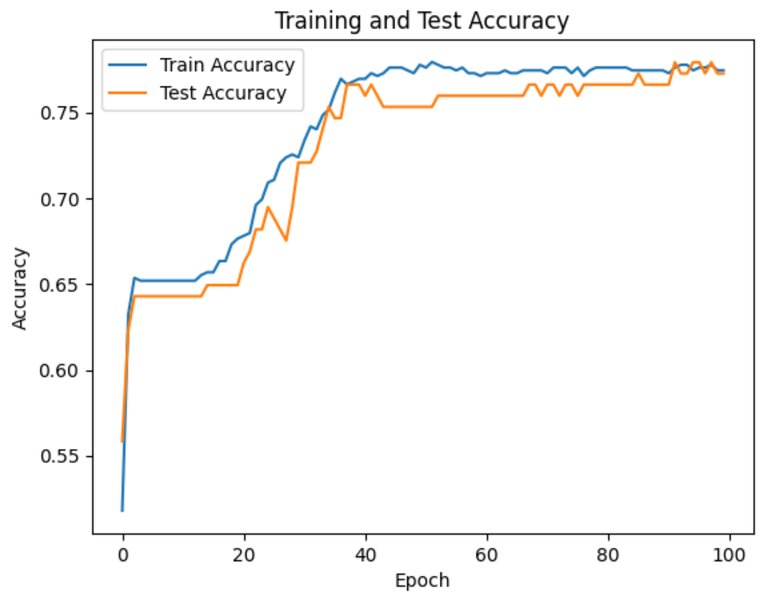
linear_relu_stack.0.weight (64, 7)
linear_relu_stack.0.bias (64)
AccumulateGrad
AccumulateGrad
TBackward0
linear_relu_stack.2.weight (32, 64)
linear_relu_stack.2.bias (32)
AddmmBackward0
AccumulateGrad
AccumulateGrad
ReluBackward0
TBackward0
linear_relu_stack.4.weight (2, 32)
linear_relu_stack.4.bias (2)
AddmmBackward0
AccumulateGrad
AccumulateGrad
ReluBackward0
TBackward0
AddmmBackward0
(55, 2)

5. (a) Plots to compare the Train and Test Accuracy:

   Accuracy:
   Accuracy is the number of correct predictions from all predictions made. It is calculated by dividing the number of correct predictions by the total number of predictions made.

   Accuracy = (Number of correct predictions)/ (total number of predictions)

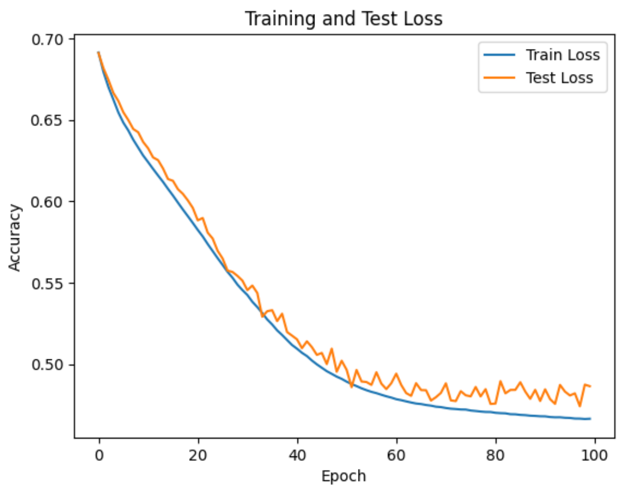
   Training Accuracy conveys to us how good the model is performing, and it give us the confidence how good that model will work on the new or unseen data.



The above graph Conveys us how the Test and Train Accuracies are Varying along the Epoch's. In the initial epochs the accuracy of both test and train data is increasing but as the model learned the data, the train accuracy increases but the test accuracy slightly decreases, and both continues to increase for rest of the epochs. As in general we can say that, with the increase in the Epoch, the Test and Train Accuracy is increases. We can see from the above graph that the model is well fitted, that is there is not much variation in the test and the train accuracies of the model. Therefore, we can say that the model is not overfitted and model works good for the new and unseen data.

(b) Test and Train Losses:

losses are defined as how much the predicted values deviate from the actual values. Losses can be decreased by using the perfect optimizer and updating the weights through backpropagation technique.



The test and train losses associated with the neural network which we have developed are close to each other over the y-axis. This is a clear indication that our model has learned all the patterns accurately. As shown in the above diagram with decrease in train loss the test loss has also been decreasing with the same variable difference between the predicted values and the actual values. The loss function which we have used is the standard cross entropy loss function which is the log loss of the predicted and the actual values. Our loss function diagram clearly shows that the loss started over 0.7 in the first epoch and has steadily decreased to less than 0.25. This is true in the case of both train and test losses, which is a clear indication of no overfitting or underfitting having taken place, since the train and test losses are pretty like each other over the epochs constantly. We have trained over 100 epochs, and we can clearly observe that the loss has decreased linearly with a variable difference of 0.05 for over 20 epochs. This pattern is observed for 80 epochs and then loss has gradually stopped decreasing since our loss function or the criterion function has achieved the local minima. One of the approaches we could follow is to decrease the learning rate according to the stability of the loss values i.e., varying learning rate over the epochs to achieve the global minima.

PART-2

We have obtained an Accuracy of 76% nearly, in order to increase the accuracy of the model further, we can tune the hypermeters like learning rate, dropout etc. And accuracy can also be increased by choosing an accurate optimizer, activation function, and initialization function.

Step1 of the process, choosing the perfect dropout probability and fixing it. Later, choosing the optimizer, activation and initialization functions.

Dropout Regularization:

Training a single deep neutral network with many parameters on a dataset may result in overfitting. However, using a dropout technique enables training multiple neural networks with varying configurations on the same dataset, and then averaging their predictions. Dropout works by randomly deactivating some neurons during training step, thus training the data on a network with dropped out nodes. In subsequent iterations, different neurons are deactivated due to the probabilistic nature of dropout. During testing, the original neural network with all activations present is used.

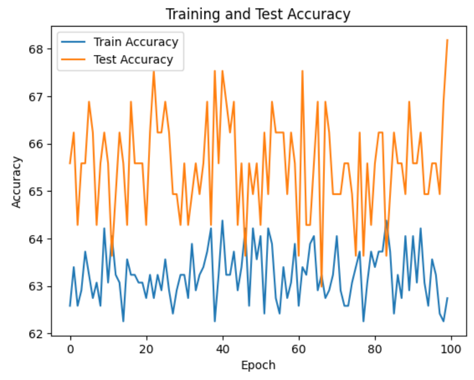
	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Setup 3	Test Accuracy
Dropout	0.3	66%	0.5	56%	0.7	60%
optimizer	SGD		SGD		SGD	
Activation Function	Sigmoid		Sigmoid		Sigmoid	
Initializer	-		-		-	

Here the accuracy obtained for the dropout probability of 0.3 is 66%, which is the highest accuracy among all the dropout probabilities. But the model gives us the highest accuracy without using the dropouts. So, we have taken the base model without dropouts for our further steps.

Plots:

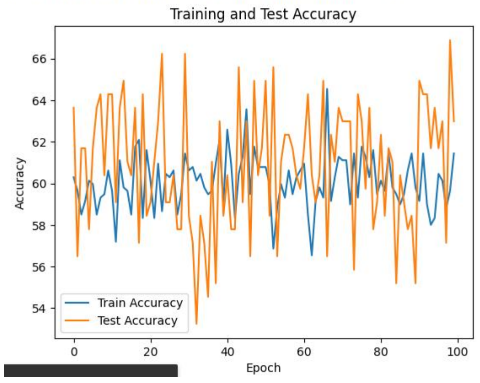
Dropout probability:0.3

```
for SGD optimizer accuracy is 68% for dropout probability 0.3
```



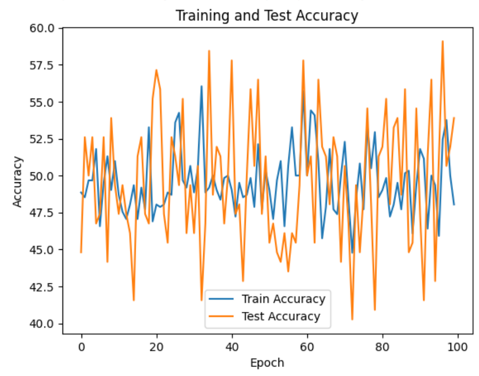
Dropout probability: 0.5

```
for SGD optimizer accuracy is 63% for dropout probability 0.5
```



Dropuout probability: 0.7

```
for SGD optimizer accuracy is 54% for dropout probability 0.7
```

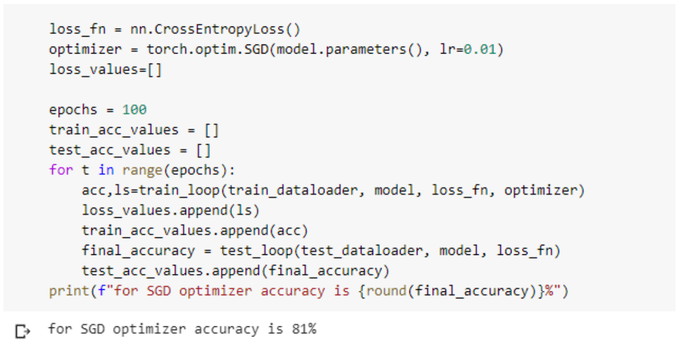


Optimizer:

Optimizers are tools that help reduce the loss in a neural network by adjusting its weights and learning rates. They determine how the weights and learning rates should be changed to minimize the loss function during training. The selection of an optimizer is critical since different optimizers use different techniques to update the parameters and choosing the appropriate one can affect the speed and effectiveness of the training process. The Selection of an optimizer should be based on the specific neural network and training data being data.

The Optimization algorithms used here are:

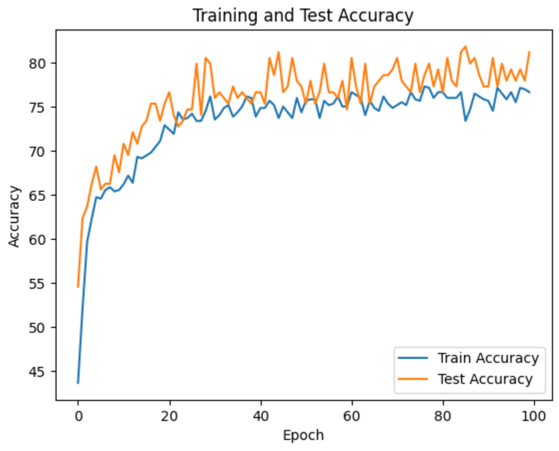
(a) SGD (Stochastic Gradient Descent):
Stochastic Gradient Descent (SGD) is a type of algorithm used for optimizing machine learning models. It differs from the standard Gradient Descent algorithm in that it randomly selects one training example at each iteration to calculate the gradient and update the parameters.



```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
loss_values=[]

epochs = 100
train_acc_values = []
test_acc_values = []
for t in range(epochs):
    acc,ls=train_loop(train_dataloader, model, loss_fn, optimizer)
    loss_values.append(ls)
    train_acc_values.append(acc)
    final_accuracy = test_loop(test_dataloader, model, loss_fn)
    test_acc_values.append(final_accuracy)
print(f"for SGD optimizer accuracy is {round(final_accuracy)}%")
```

```
for SGD optimizer accuracy is 81%
```

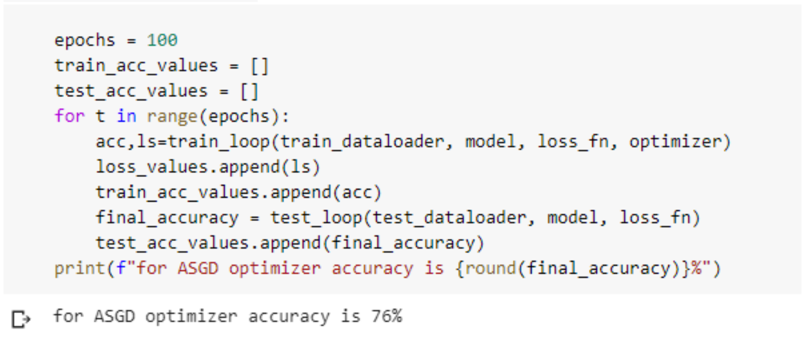
Plots:



(b) <u>Averaged Stochastic Gradient descent (ASGD):</u>

Averaged Stochastic Gradient Descent (ASGD) is a machine learning optimization algorithm that is based on Stochastic Gradient Descent (SGD). SGD updates the model parameters for each training example, which can be noisy and slow down convergence. ASGD addresses this issue by taking an average of the model parameters over time to reduce the noise and improve convergence. At each iteration of ASGD, a random subset of training data is used to estimate the gradient, which is then used to update the model parameters. An exponential moving average of the model parameters is taken over time, and the final parameters are obtained by averaging the parameter values seen during training. ASGD is particularly useful for large datasets where computing the full gradient for each iteration can be expensive. By using a subset of the data, ASGD can converge faster than batch gradient descent while still avoiding the noise of SGD.

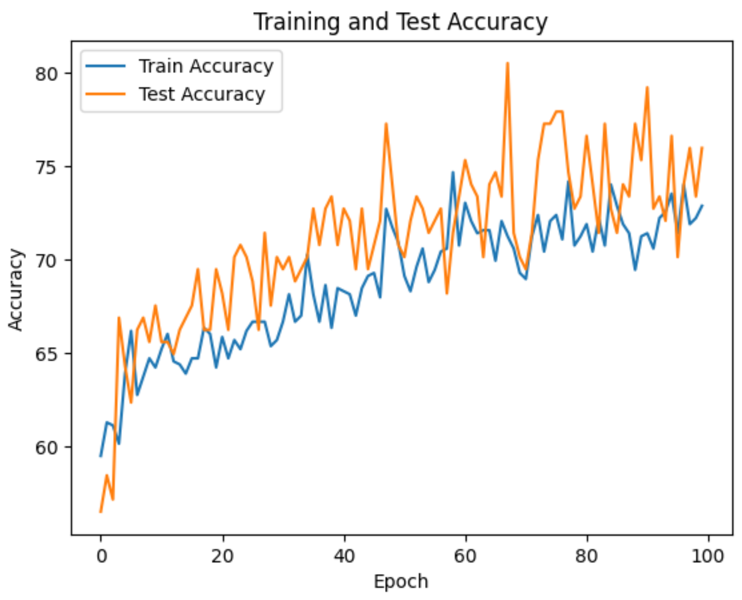
Accuracy obtained is:



```
epochs = 100
train_acc_values = []
test_acc_values = []
for t in range(epochs):
    acc,ls=train_loop(train_dataloader, model, loss_fn, optimizer)
    loss_values.append(ls)
    train_acc_values.append(acc)
    final_accuracy = test_loop(test_dataloader, model, loss_fn)
    test_acc_values.append(final_accuracy)
print(f"for ASGD optimizer accuracy is {round(final_accuracy)}%")
```

```
for ASGD optimizer accuracy is 76%
```

Plots :



(c) Adam Algorithm:

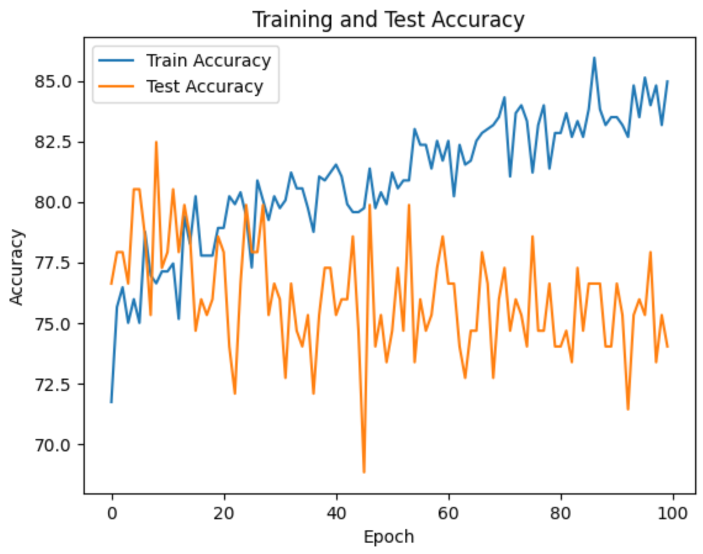
Adaptive Moment Estimation (Adam) is a powerful optimization algorithm that is particularly effective for handling large-scale problems involving extensive data or parameters. Compared to other techniques, it uses less memory and is highly efficient. Adam combines the strengths of two popular optimization methods: gradient descent with momentum and RMSP. This combination allows Adam to efficiently navigate complex optimization landscapes and achieve superior results.

Accuracy obtained:

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
loss_values=[]

epochs = 100
train_acc_values = []
test_acc_values = []
for t in range(epochs):
    acc,ls=train_loop(train_dataloader, model, loss_fn, optimizer)
    loss_values.append(ls)
    train_acc_values.append(acc)
    final_accuracy = test_loop(test_dataloader, model, loss_fn)
    test_acc_values.append(final_accuracy)
print(f"for Adam optimizer accuracy is {round(final_accuracy)}%")
```

```
for Adam optimizer accuracy is 74%
```

Plots:



(d) <u>Adagrad Algorithm:</u>

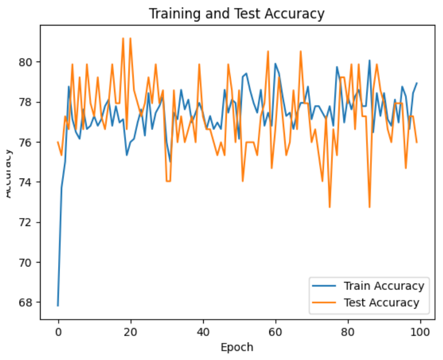
AdaGrad is a stochastic gradient optimization algorithm that adjusts the learning rate for each parameter. Unlike other methods that use a single learning rate, AdaGrad uses a unique learning rate for every parameter. This technique is highly effective for problems with sparse gradients, like those found in natural language or computer vision applications, and significantly enhances performance.

Accuracy:

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adagrad(model.parameters(), lr=0.01)
loss_values=[]

epochs = 100
train_acc_values = []
test_acc_values = []
for t in range(epochs):
    acc,ls=train_loop(train_dataloader, model, loss_fn, optimizer)
    loss_values.append(ls)
    train_acc_values.append(acc)
    final_accuracy = test_loop(test_dataloader, model, loss_fn)
    test_acc_values.append(final_accuracy)
print(f"for Adagrad optimizer accuracy is {round(final_accuracy)}%")
```

```
for Adagrad optimizer accuracy is 76%
```

Plots:



	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Setup 3	Test Accuracy	Setup 4	Test Accuracy
Dropout	0.2	75%	0.2	73%	0.2	71%	0.2	73%
optimizer	SGD		ASGD		Adam		AdaGrad	
Activation Function	Sigmoid		Sigmoid		Sigmoid		Sigmoid	
Initializer	He		He		He			

With the no dropout probability and with SGD optimizer tool we got the highest test accuracy of 75%. Therefore, we have fixed the optimizer algorithm as SGD and started changing other parameters like Activation and initializer.

Activation Function:

The activation function is a mathematical operation that takes the sum of the inputs multiplied by their respective weights and adds a bias value to determine whether a neuron should be activated or not. The purpose of the activation function is to introduce non-linear behavior to the neuron's output.

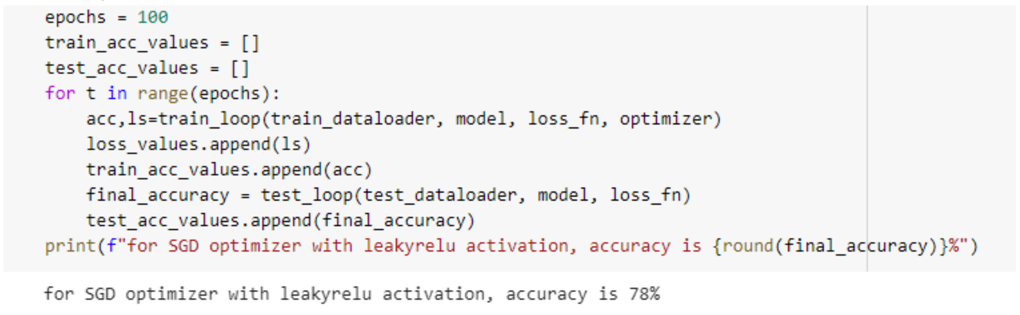
Activation functions used in this model are:

(a) Leaky Relu activation function:
The ReLU function, which stands for Rectified Linear Unit, is a commonly used activation function in neural networks. Unlike other activation functions, ReLU only activates some neurons at a time. This is because when the input is negative, the neuron is not activated, and the output is set to zero.

The Leaky ReLU function is an enhanced version of the ReLU function, where instead of setting the output to 0 for input values less than 0, we introduce a slight linear component to the output based on the input value.

F(x)=ax, x<0
F(x)=x, otherwise

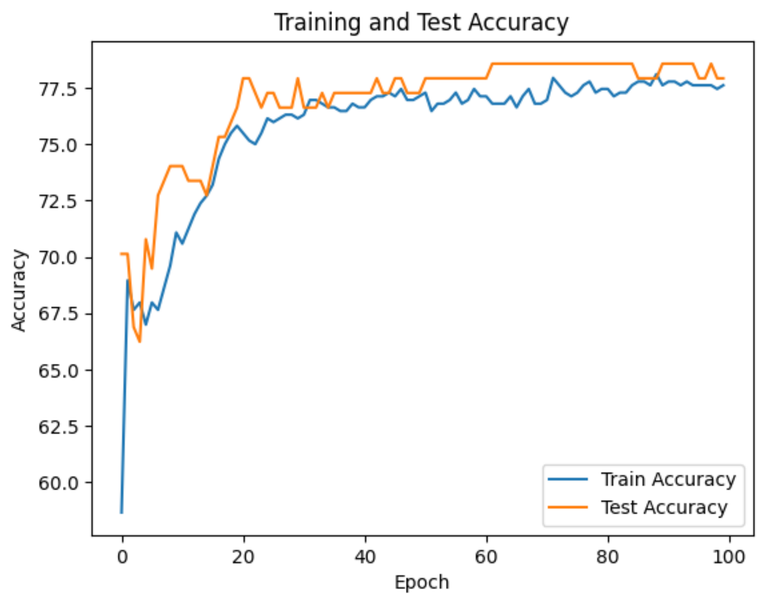
Accuracy:



```
epochs = 100
train_acc_values = []
test_acc_values = []
for t in range(epochs):
    acc,ls=train_loop(train_dataloader, model, loss_fn, optimizer)
    loss_values.append(ls)
    train_acc_values.append(acc)
    final_accuracy = test_loop(test_dataloader, model, loss_fn)
    test_acc_values.append(final_accuracy)
print(f"for SGD optimizer with leakyrelu activation, accuracy is {round(final_accuracy)}%")
```

```
for SGD optimizer with leakyrelu activation, accuracy is 78%
```

Plots:



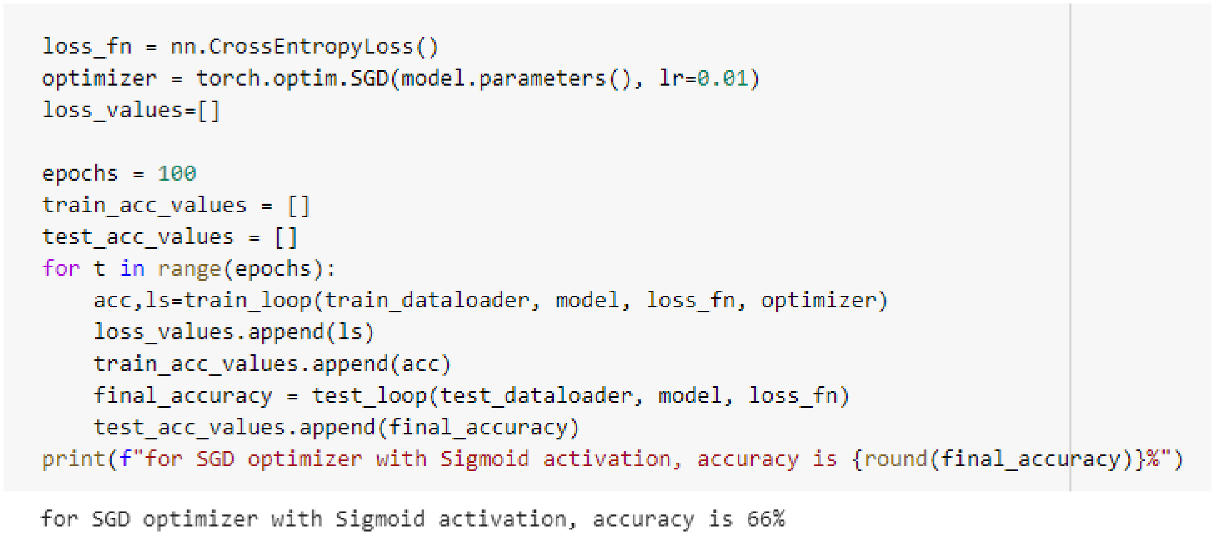
(b) <u>Sigmoid function</u>:

The sigmoid function is a commonly used activation function which is mathematically defined as:

1/(1+e^-x)

The sigmoid function is a type of smooth function that is continuously differentiable. Its main advantage over linear and step functions is that it is non-linear, which is a very interesting characteristic. This means that if multiple neurons use the sigmoid function as their activation function, the output will also be non-linear. The sigmoid function has an S shape and ranges from 0 to 1.

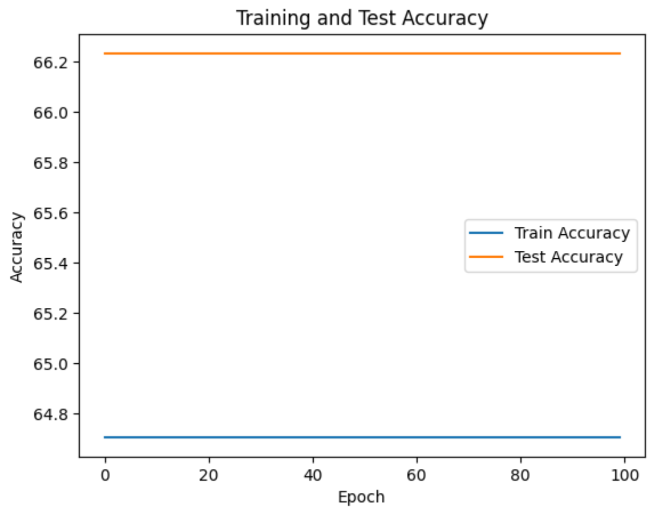
Accuracy:



```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
loss_values=[]

epochs = 100
train_acc_values = []
test_acc_values = []
for t in range(epochs):
    acc,ls=train_loop(train_dataloader, model, loss_fn, optimizer)
    loss_values.append(ls)
    train_acc_values.append(acc)
    final_accuracy = test_loop(test_dataloader, model, loss_fn)
    test_acc_values.append(final_accuracy)
print(f"for SGD optimizer with Sigmoid activation, accuracy is {round(final_accuracy)}%")
```

```
for SGD optimizer with Sigmoid activation, accuracy is 66%
```

Plots:

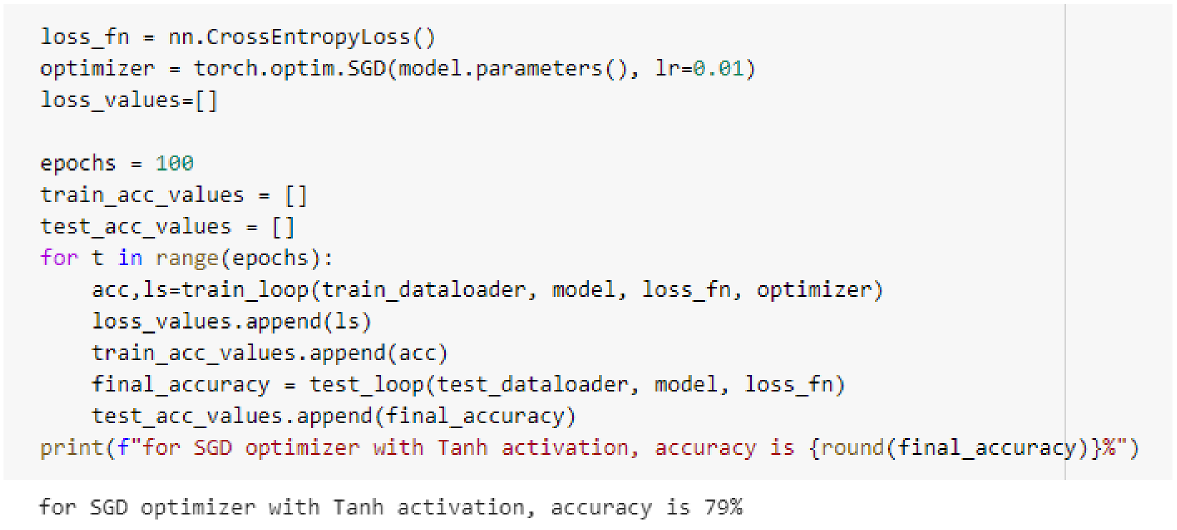


(c) <u>Tanh Activation function</u>:

The Tanh function, also known as the Tangent Hyperbolic function, is typically more effective than the sigmoid function for activation in neural networks. The Tanh function is actually a modified version of the sigmoid function and the two are closely related mathematically.

F(x) = Tanh(x)

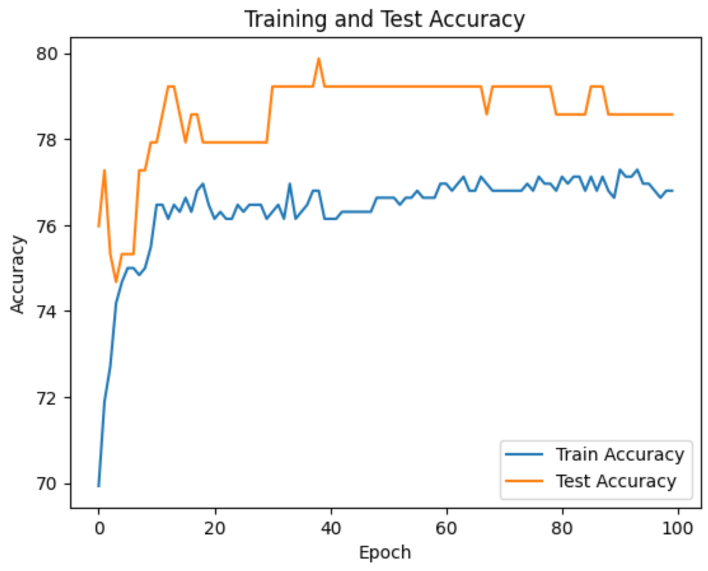
Accuracy:



```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
loss_values=[]

epochs = 100
train_acc_values = []
test_acc_values = []
for t in range(epochs):
    acc,ls=train_loop(train_dataloader, model, loss_fn, optimizer)
    loss_values.append(ls)
    train_acc_values.append(acc)
    final_accuracy = test_loop(test_dataloader, model, loss_fn)
    test_acc_values.append(final_accuracy)
print(f"for SGD optimizer with Tanh activation, accuracy is {round(final_accuracy)}%")
```

```
for SGD optimizer with Tanh activation, accuracy is 79%
```

Plots:



	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Setup 3	Test Accuracy
Dropout	-	77%	-	62%	-	76%
optimizer	SGD		SGD		SGD	
Activation Function	Leaky relu		Sigmoid		Tanh	
Initializer	-		-		-	

With no dropout probability and with optimizer as SGD we have tried different activation Functions like leaky relu, sigmoid, Tanh. With Leaky relu we have achieved the highest test Accuracy of 77%.

Therefore, we have fixed the activation function and optimizer to leaky relu and sigmoid respectively and tried changing the other hyperparameter named initialization functions to achieve the maximum accuracy.

Initialization function:

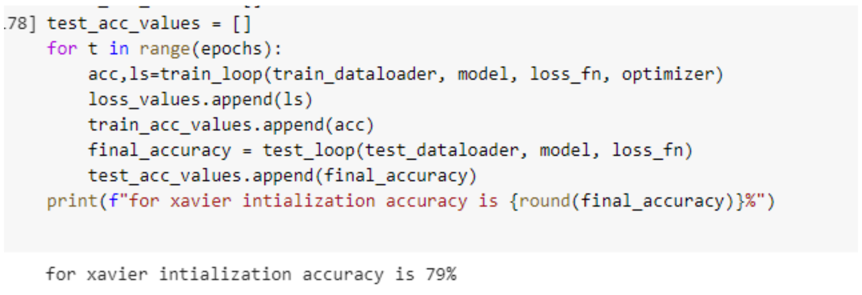
To create accurate neural networks, it is important to initialize the weights properly during their construction and training. Inadequate weight initialization can cause either the Vanishing Gradient or Exploding Gradient problem, which can reduce the model's accuracy.

Initialization function used in the model are:

1. Xavier Initialization – it is suitable for layers with sigmoid activation function.
2. Uniform initialization.
3. Ones initialization.
4. Zeroes initialization.

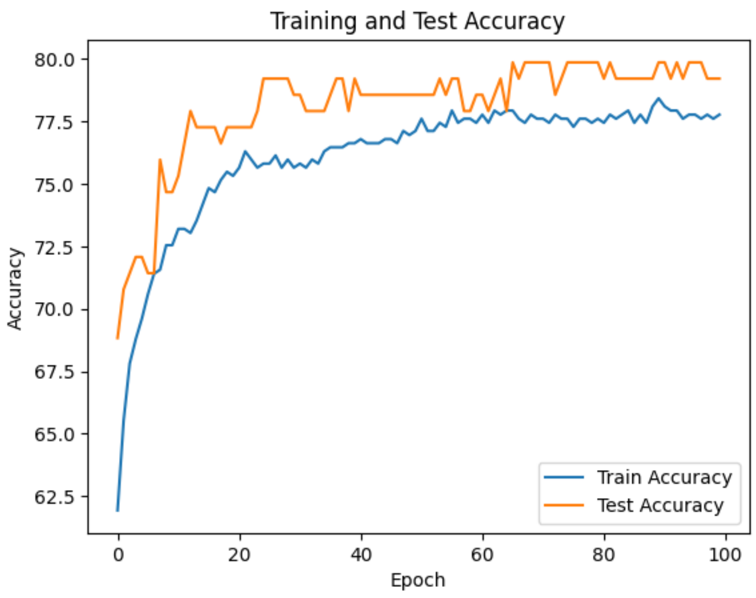
Accuracy:

Xavier:



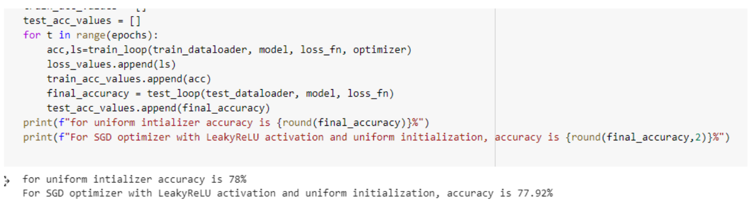
```
.78] test_acc_values = []
     for t in range(epochs):
         acc,ls=train_loop(train_dataloader, model, loss_fn, optimizer)
         loss_values.append(ls)
         train_acc_values.append(acc)
         final_accuracy = test_loop(test_dataloader, model, loss_fn)
         test_acc_values.append(final_accuracy)
     print(f"for xavier intialization accuracy is {round(final_accuracy)}%")

     for xavier intialization accuracy is 79%
```

Plot:



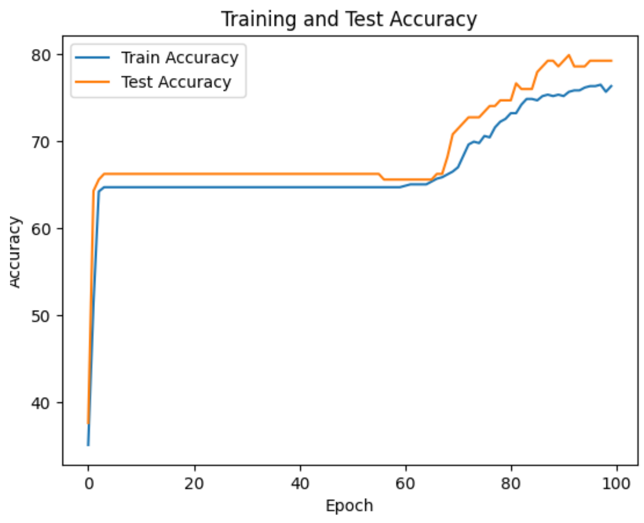
Uniform initialization:

Accuracy:



```
test_acc_values = []
for t in range(epochs):
    acc,ls=train_loop(train_dataloader, model, loss_fn, optimizer)
    loss_values.append(ls)
    train_acc_values.append(acc)
    final_accuracy = test_loop(test_dataloader, model, loss_fn)
    test_acc_values.append(final_accuracy)
print(f"for uniform intializer accuracy is {round(final_accuracy)}%")
print(f"For SGD optimizer with LeakyReLU activation and uniform initialization, accuracy is {round(final_accuracy,2)}%")

for uniform intializer accuracy is 78%
For SGD optimizer with LeakyReLU activation and uniform initialization, accuracy is 77.92%
```

Plots:

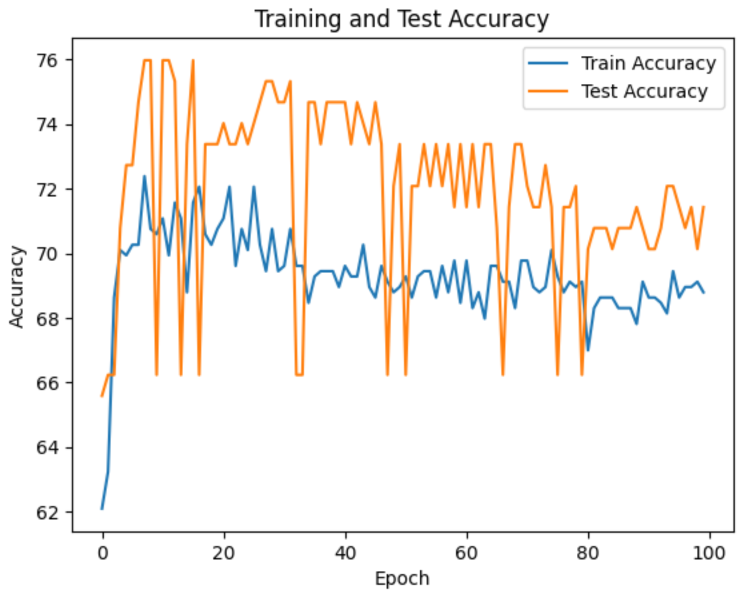


Ones Initializer:

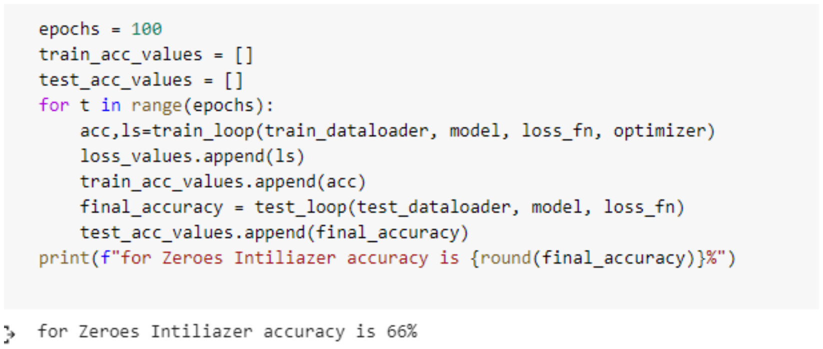
```python
epochs = 100
train_acc_values = []
test_acc_values = []
for t in range(epochs):
    acc,ls=train_loop(train_dataloader, model, loss_fn, optimizer)
    loss_values.append(ls)
    train_acc_values.append(acc)
    final_accuracy = test_loop(test_dataloader, model, loss_fn)
    test_acc_values.append(final_accuracy)
print(f"for ones intializer accuracy is {round(final_accuracy)}%")
```

```
for ones intializer accuracy is 71%
```

Plots:



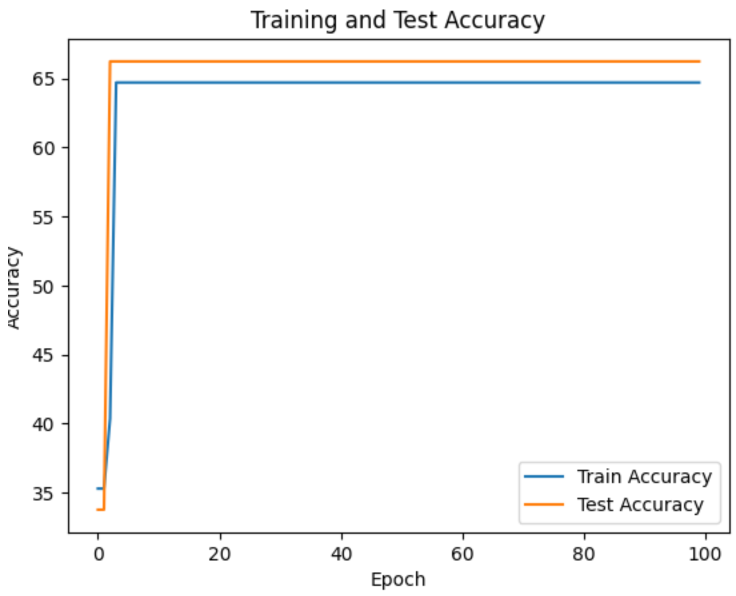
Zeroes Initializer:



```
epochs = 100
train_acc_values = []
test_acc_values = []
for t in range(epochs):
    acc,ls=train_loop(train_dataloader, model, loss_fn, optimizer)
    loss_values.append(ls)
    train_acc_values.append(acc)
    final_accuracy = test_loop(test_dataloader, model, loss_fn)
    test_acc_values.append(final_accuracy)
print(f"for Zeroes Intiliazer accuracy is {round(final_accuracy)}%")
```

```
for Zeroes Intiliazer accuracy is 66%
```

Plot:



	Setup 1	Test Accuracy	Setup 2	Test Accuracy	Setup 3	Test Accuracy	Setup 4	
Dropout	-	78.57%	-	75.32%	-	68.83%	-	67.53%
optimizer	SGD		SGD		SGD		SGD	
Activation Function	Leaky relu		Leaky relu		Leaky relu		Leaky relu	
Initializer	Xavier		Uniform		ones		zeroes	

When we have updated the model with different initializer functions by fixing the optimizer and activation function. We have achieved the highest accuracy with Xavier initializer, that is 78.57%. Therefore, the model with no dropout probability, with SGD as optimizer, leaky relu as activation function and with Xavier initializer we have achieved the highest accuracy.

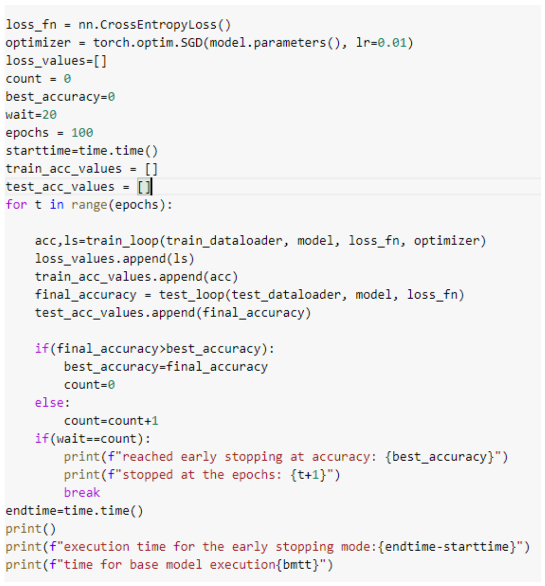
Therefore, we have set up the base model with Leaky relu, SGD as the optimizer and Xavier function as an initializer

Later, to improve the base model accuracy we have, used the Early stopping method.

(1)Early Stopping:

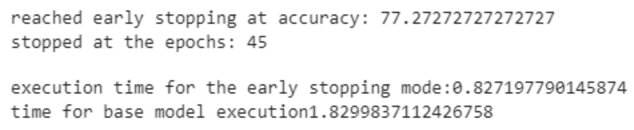
Early stopping is a method of optimizing models that aims to prevent overfitting while maintaining high accuracy. It involves stopping the training process before the model begins to overfit, thus preventing it from memorizing the training data excessively.

By applying early stopping to the model, with the given below code.



```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
loss_values=[]
count = 0
best_accuracy=0
wait=20
epochs = 100
starttime=time.time()
train_acc_values = []
test_acc_values = []
for t in range(epochs):

    acc,ls=train_loop(train_dataloader, model, loss_fn, optimizer)
    loss_values.append(ls)
    train_acc_values.append(acc)
    final_accuracy = test_loop(test_dataloader, model, loss_fn)
    test_acc_values.append(final_accuracy)

    if(final_accuracy>best_accuracy):
        best_accuracy=final_accuracy
        count=0
    else:
        count=count+1
    if(wait==count):
        print(f"reached early stopping at accuracy: {best_accuracy}")
        print(f"stopped at the epochs: {t+1}")
        break
endtime=time.time()
print()
print(f"execution time for the early stopping mode:{endtime-starttime}")
print(f"time for base model execution{bmtt}")
```

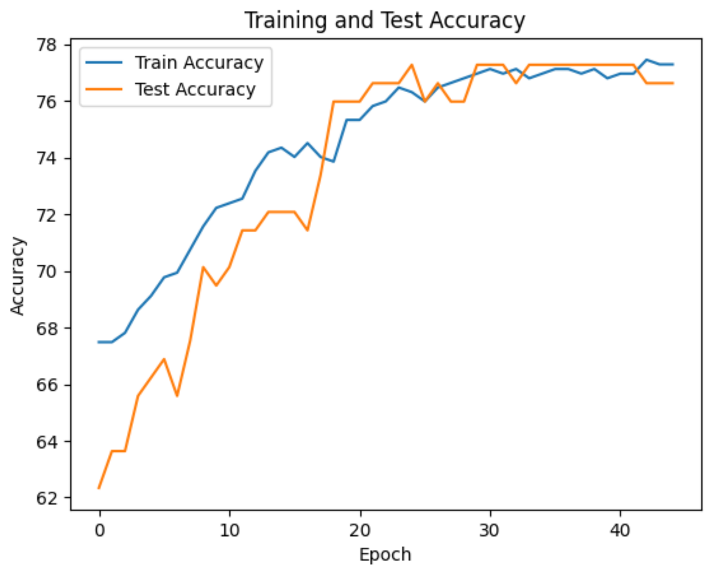
With early stopping we acquired accuracy in 45 epochs and with less execution time.



```
reached early stopping at accuracy: 77.27272727272727
stopped at the epochs: 45

execution time for the early stopping mode:0.827197790145874
time for base model execution1.8299837112426758
```

Inferences from the graph:

From the below graph we can observe that the test and train accuracies of the model started increasing the epochs increases but suddenly after 10 epochs both the test and train started decreasing. May be the model started learning about the data again in the epochs phase 10-15 then again, the train data started increasing near the 18th epoch and the test accuracy also behaved in the same manner as the train accuracy and started increasing after the 15th epoch and both achieved the almost similar accuracy after the 40th epoch.



Through early stopping time as drastically decreased from 3seconds to 1.2 seconds.

(2) Learning rate scheduler:

The learning rate is a parameter used in the backpropagation algorithm for training neural networks. It determines the size of the updates made to the model weights during training. It is typically set as a value between 0 and 1.0 and can be adjusted to control the speed and accuracy of the training process.
The learning rate is responsible for determining the size of the steps taken by an optimizer during the process of minimizing the loss function. It helps to regulate the rate at which the model's parameters are updated during training, and ultimately affects the quality of the model's predictions.

Scheduler: A learning rate schedule is a predetermined approach for modifying the learning rate during the training process, typically on a per-epoch or per-iteration basis. There are two common techniques for implementing a learning rate schedule:

- Constant learning rate, where the learning rate remains fixed throughout the training process.

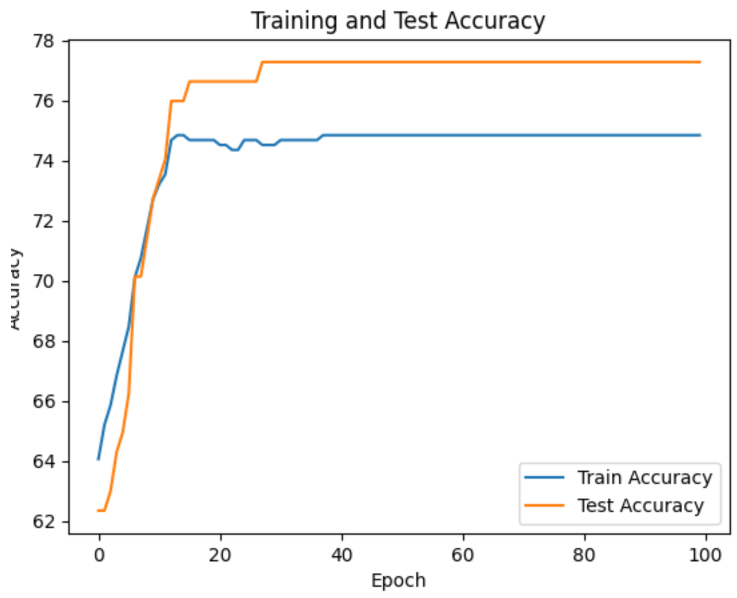
- Learning rate decay, where the learning rate is initially set at a certain value and gradually decreased over time according to a predefined schedule.

```
    acc,ls=train_loop(train_dataloader, model, loss_fn, optimizer)
    loss_values.append(ls)
    train_acc_values.append(acc)
    final_accuracy = test_loop(test_dataloader, model, loss_fn)
    test_acc_values.append(final_accuracy)

    scheduler.step(ls)
endtime=time.time()
print(f"accuracy of model:{final_accuracy}")
mtt=endtime-starttime
print(f"time for the model execution{mtt}")
```

```
accuracy of model:77.27272727272727
time for the model execution1.7418689727783203
```

The accuracy we achieved is 77.27 % in a very less time.

Plot:



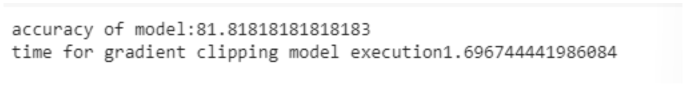
Inferences from the graph:

From the above graph we can observe that with the learning rate scheduler, the model achieved its accuracy in very less no of epochs. The test and train data almost behaved in the same manner as this learning scheduler also. The train data slightly varied from the test data in the epoch range of 15 to 25 approximately and later on test accuracy achieved a slight higher accuracy of 77 % and followed the same trend until 100 epochs but the train data got 74% and remained same until the 100 epochs.

3.<u>Gradient Clipping</u>:

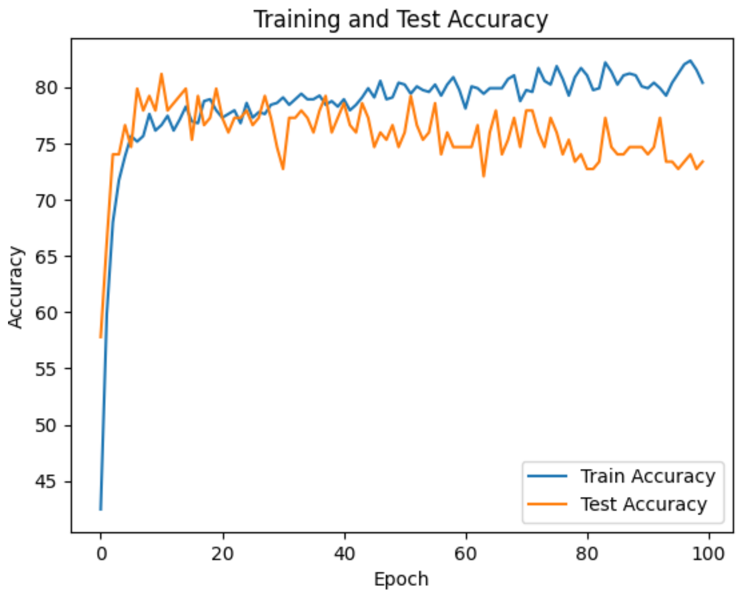
Gradient clipping is a technique used in neural networks to limit the magnitude of the gradients during backpropagation. This is done by setting a threshold and rescaling the gradients if they exceed the threshold, before using them to update the weights. The purpose of gradient clipping is to prevent the gradients from becoming too large or too small, which can lead to numerical instability or slow convergence during training.

Using the Gradient clipping we got the almost same accuracy as base model, but the run time of the model with the gradient clipping is too less.



```
accuracy of model:81.81818181818183
time for gradient clipping model execution1.696744441986084
```

Plot of the Train and Test Accuracies with the drop out probability of 0.5,optimizer Stochastic gradient descent, activation function leaky relu , loss function is Cross entropy loss.

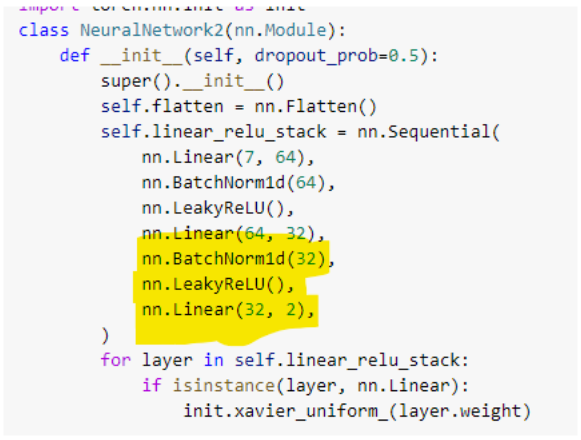


Inferences from the graph:
We observed that using the gradient clipping makes the training data convergence fast, therefore max accuracy can be achieved in a much less amount of time that is for few numbers of epochs. Here, we must be careful while setting the clipping threshold, if not the model may underfit or overfit.
Here in the above graph the train and test data obtained its max accuracy in very few epochs, that is at 18 to 20 epochs and later on the accuracy remained constant.

4. <u>Batch Normalization</u>:

Batch normalization is a technique that rescales the outputs of each layer in a deep neural network to have zero mean and unit variance based on the statistics of the current batch. This rescaling helps to accelerate the training process and mitigate issues arising from inadequate parameter initialization. In essence, batch normalization ensures that the network's inputs are normalized, leading to better convergence and improved generalization performance.

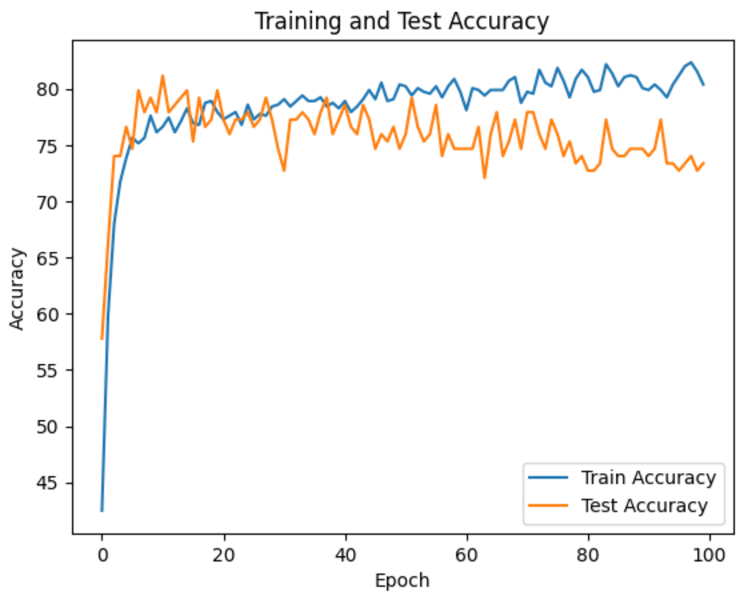
Batch normalization introduces noise to the internal layers of a deep neural network, which helps to prevent overfitting and acts as a form of regularization. As a result, when using batch normalization, it is recommended to use less dropout as it also introduces noise. The combination of batch normalization and dropout can sometimes be redundant and may even hurt the model's performance.



```
import torch.nn.init as init
class NeuralNetwork2(nn.Module):
    def __init__(self, dropout_prob=0.5):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(7, 64),
            nn.BatchNorm1d(64),
            nn.LeakyReLU(),
            nn.Linear(64, 32),
            nn.BatchNorm1d(32),
            nn.LeakyReLU(),
            nn.Linear(32, 2),
        )
        for layer in self.linear_relu_stack:
            if isinstance(layer, nn.Linear):
                init.xavier_uniform_(layer.weight)
```

We have acquired an accuracy of nearly 74% using the batch normalization technique. With the dropout probability of 0.2 , optimizer-stochastic gradient descent, activation function-Tanh , loss function cross entropy loss and initializer – Xavier initialization function with all these as the base model we applied batch normalization technique to it, to get the accuracy and to omit the overfitting of the data if any We have achieved slightly lesser accuracy than the normal model, But the run time is less than the base model but its run time is more than rest of the models with other techniques used in it.

Plot for the model with the Batch normalization.



Inferences from the graph:

Here in the initial phases of the epoch the model was trained for the data well and attained max accuracy in the initial epochs itself. Later, accuracy haven't changed along with the time.

Therefore, we have observed that with the base model:

Optimizer: SGD

Loss function: constant entropy loss

Activation function: leaky relu

Initializer: Xavier initializer

Dropout: 0.2

In order to increase the accuracy of the model, we have used certain techniques as mentioned above, the techniques are:

Early stopping: 77.27% Accuracy -- less interval of execution time.

Learning rate scheduler: 77.27% Accuracy --- Takes more execution time than the early stopping.

Sliding gradient: 81.81 % Accuracy.

Batch Normalization: 73.37 % Accuracy.

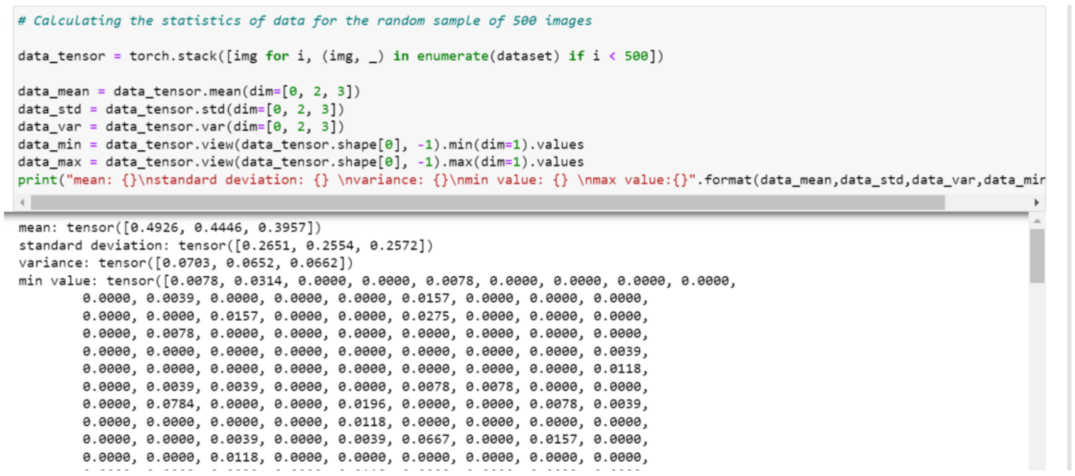
The Model has attained highest accuracy using the Sliding gradient technique.

<u>PART -3</u>

The dataset we have used here is a sample convolution neural network dataset provided on our ublearns. It contains 3 classes namely dogs, cars, and food. We have to build a multiclass classification model to classify the given dataset. This model is trained with 30,000.
images i.e. each class has 10,000 images. These images are of size 64 x 64 and have,
been resized to 224 x 224 to increase the computation speed.

The data set is converted to pixels data using the pytorch commands and later the preprocessing steps are applied on it to prepare the data to feed to the model.

Main stastics for the pixel tensors dataset is:



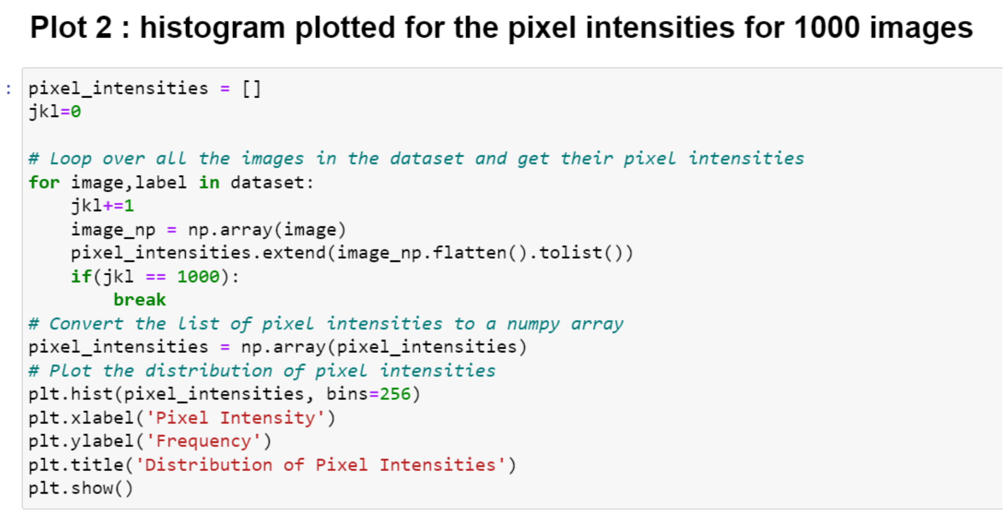
```
# Calculating the statistics of data for the random sample of 500 images

data_tensor = torch.stack([img for i, (img, _) in enumerate(dataset) if i < 500])

data_mean = data_tensor.mean(dim=[0, 2, 3])
data_std = data_tensor.std(dim=[0, 2, 3])
data_var = data_tensor.var(dim=[0, 2, 3])
data_min = data_tensor.view(data_tensor.shape[0], -1).min(dim=1).values
data_max = data_tensor.view(data_tensor.shape[0], -1).max(dim=1).values
print("mean: {}\nstandard deviation: {} \nvariance: {}\nmin value: {} \nmax value:{}".format(data_mean,data_std,data_var,data_mir
```

```
mean: tensor([0.4926, 0.4446, 0.3957])
standard deviation: tensor([0.2651, 0.2554, 0.2572])
variance: tensor([0.0703, 0.0652, 0.0662])
min value: tensor([0.0078, 0.0314, 0.0000, 0.0000, 0.0078, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0039, 0.0000, 0.0000, 0.0000, 0.0157, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0157, 0.0000, 0.0000, 0.0275, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0078, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0039,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0118,
        0.0000, 0.0039, 0.0039, 0.0000, 0.0000, 0.0078, 0.0078, 0.0000, 0.0000,
        0.0000, 0.0784, 0.0000, 0.0000, 0.0196, 0.0000, 0.0000, 0.0078, 0.0039,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0118, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0039, 0.0000, 0.0039, 0.0667, 0.0000, 0.0157, 0.0000,
        0.0000, 0.0000, 0.0118, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
```

To understand the data well, and to know the distribution of data in each class we require graphs.

Visualization Graphs:

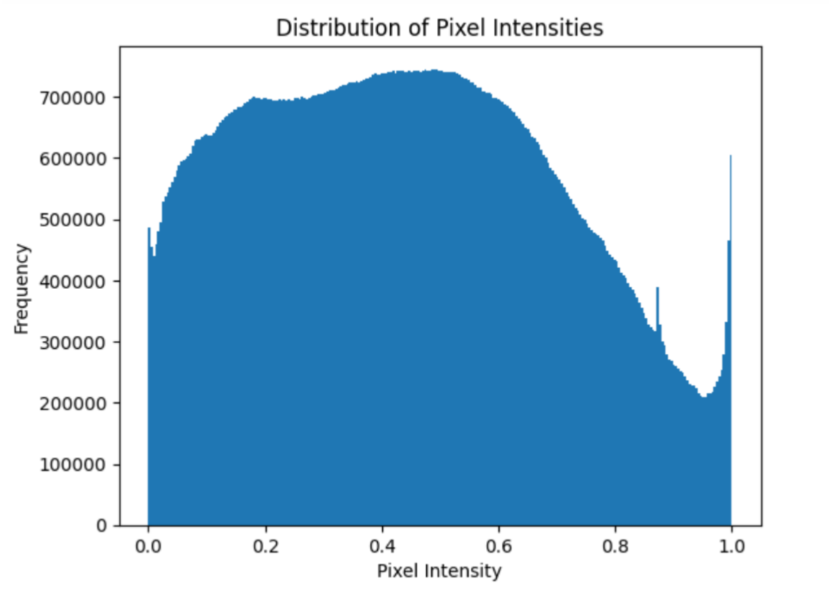
Plot:1



Plot 2 : histogram plotted for the pixel intensities for 1000 images

```
pixel_intensities = []
jkl=0

# Loop over all the images in the dataset and get their pixel intensities
for image,label in dataset:
    jkl+=1
    image_np = np.array(image)
    pixel_intensities.extend(image_np.flatten().tolist())
    if(jkl == 1000):
        break
# Convert the list of pixel intensities to a numpy array
pixel_intensities = np.array(pixel_intensities)
# Plot the distribution of pixel intensities
plt.hist(pixel_intensities, bins=256)
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.title('Distribution of Pixel Intensities')
plt.show()
```

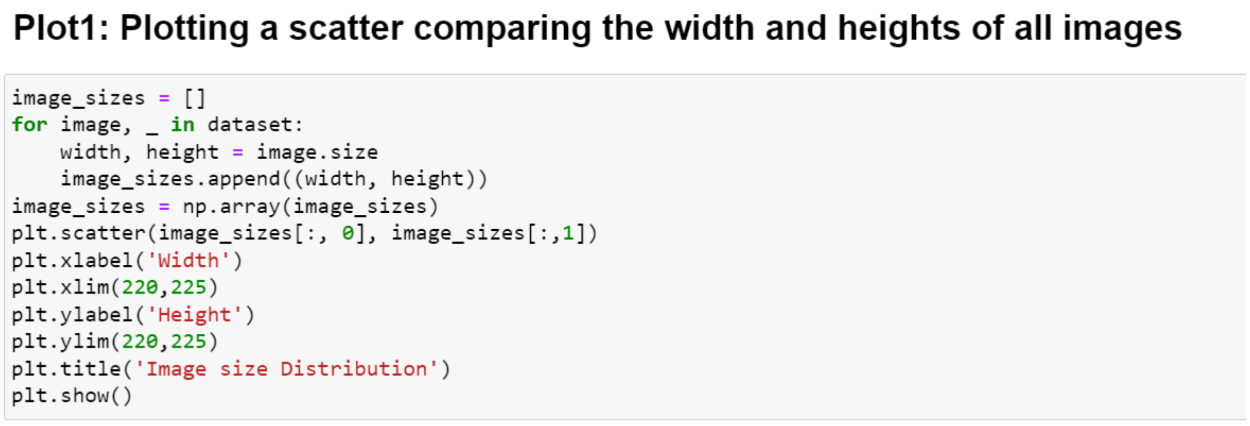
Graph:



Inferences from the graph:

The image depicts that the frequency with 0.5 pixel intensity has maximum area, between the pixel intensities 0.8 and 0.9 least and again the intensity of 1 has more area. This graph conveys to us how the image pixels intensity is distributed. The image pixel intensity is compared with the frequency of that pixel in the image dataset.

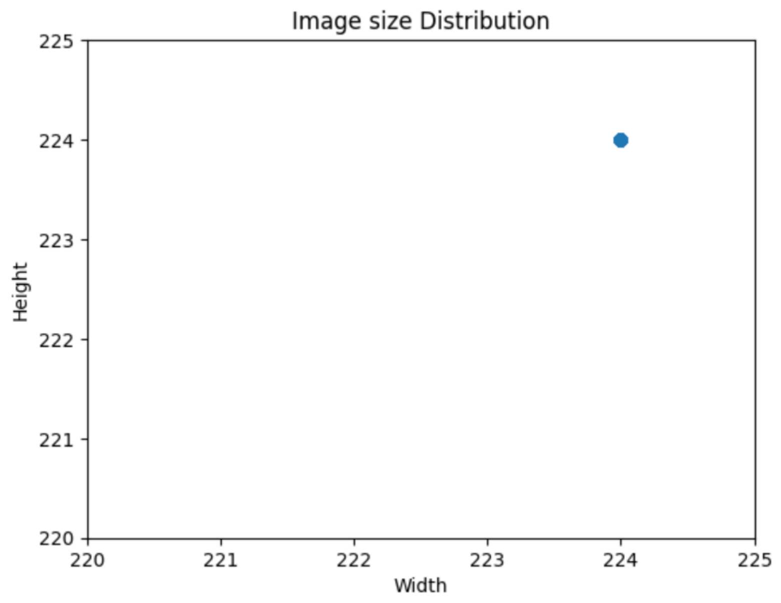
Plot 2:



Plot1: Plotting a scatter comparing the width and heights of all images

```python
image_sizes = []
for image, _ in dataset:
    width, height = image.size
    image_sizes.append((width, height))
image_sizes = np.array(image_sizes)
plt.scatter(image_sizes[:, 0], image_sizes[:,1])
plt.xlabel('Width')
plt.xlim(220,225)
plt.ylabel('Height')
plt.ylim(220,225)
plt.title('Image size Distribution')
plt.show()
```

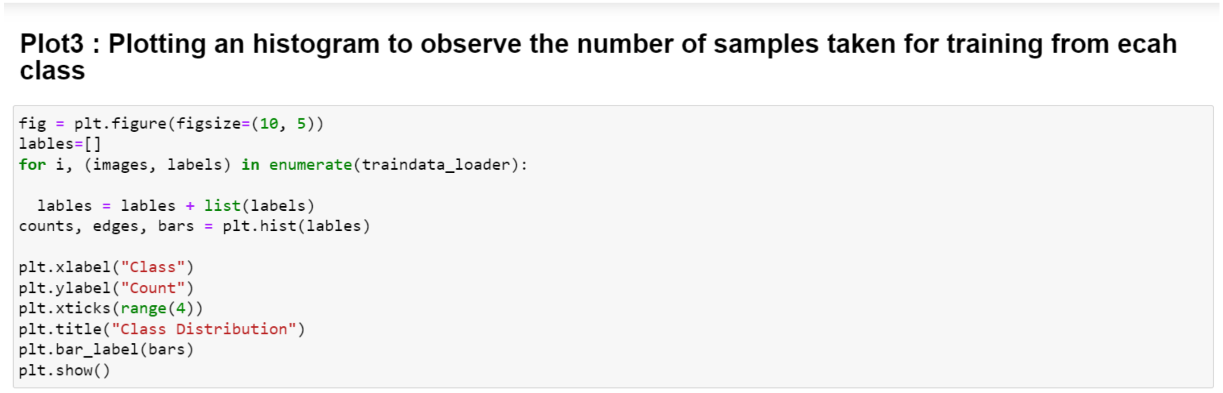
Plot:



This graph conveys to us that all the images in the dataset are all of same size , as we know that previously after reading the image dataset , we have converted all the image data in the 64X64 to 224x224 width and height. So that the data will be uniform and will be perfect to apply the model on it, and it gives accurate results.

This graph is usually useful for us to check whether the data is in uniform manner or not.

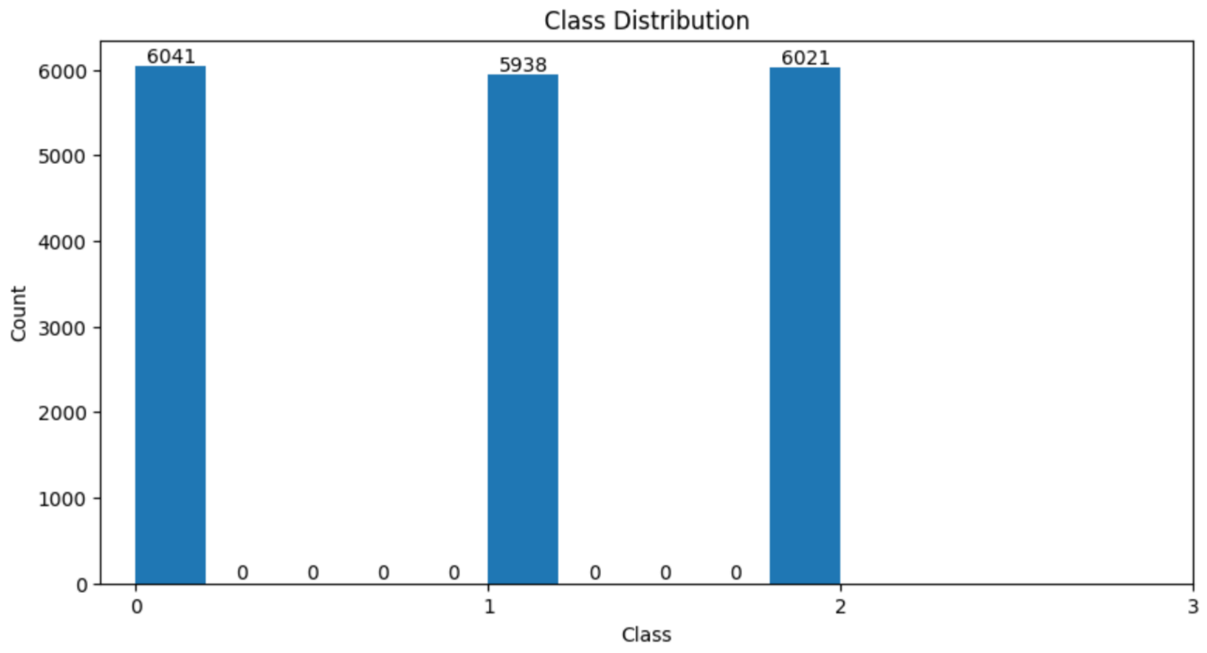
Plot3:



Plot3 : Plotting an histogram to observe the number of samples taken for training from ecah class

```
fig = plt.figure(figsize=(10, 5))
lables=[]
for i, (images, labels) in enumerate(traindata_loader):

  lables = lables + list(labels)
counts, edges, bars = plt.hist(lables)

plt.xlabel("Class")
plt.ylabel("Count")
plt.xticks(range(4))
plt.title("Class Distribution")
plt.bar_label(bars)
plt.show()
```

Graph 3:



These graph convey us, how many samples of images from each class are taken in the training dataset randomly.

From class 1: 6041 images are feed to training dataset.

Class2: 5938 images are feed to training dataset.

Class 3: 6021 images are feed to training dataset.

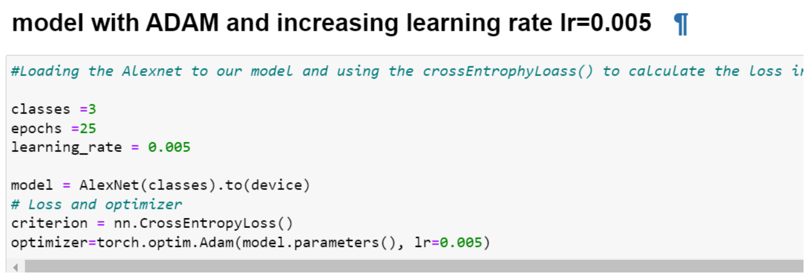
<u>Model Building</u>:

We have build the Alex net model to feed our data, and to do the multiclass classification.

Model:

The alexnet that we have implemented contains 5 convolutional layers, with different sized kernels(11x11, 5x5, 3x3, 3x3, 3x3). We have applied max pooling to every convolutional layer. We have also applied padding to every convolutional layer other than the 1st one. The last convolutional layer has been flattened and connected to a fully connected layer with 4096 neurons. We have a total of 3 fully connected layers with the last one being for predicting the classes hence the output layer. It has 3 neurons. Each fully connected layer has a dropout rate of 0.5.

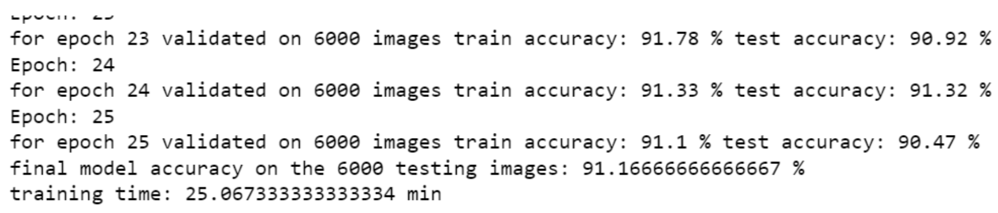
Accuracies of the Model with hyperparameters are shown in the below image:



model with ADAM and increasing learning rate lr=0.005 ¶

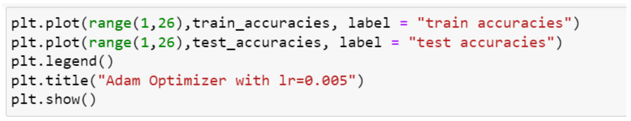
```
#Loading the Alexnet to our model and using the crossEntrophyLoass() to calculate the loss i

classes =3
epochs =25
learning_rate = 0.005

model = AlexNet(classes).to(device)
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer=torch.optim.Adam(model.parameters(), lr=0.005)
```

For this hyperparameters we have obtained an accuracy of:

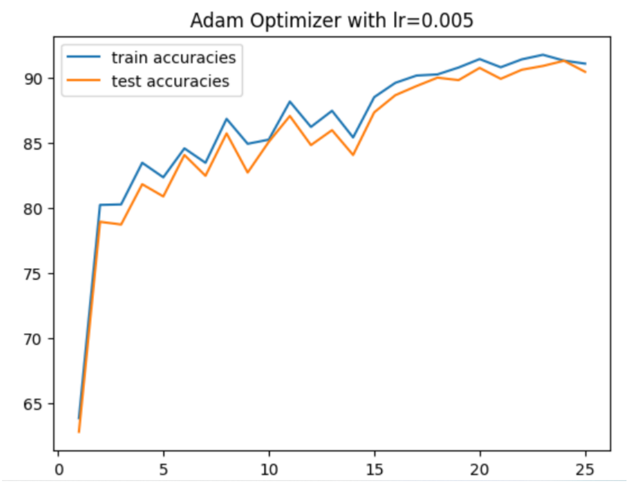


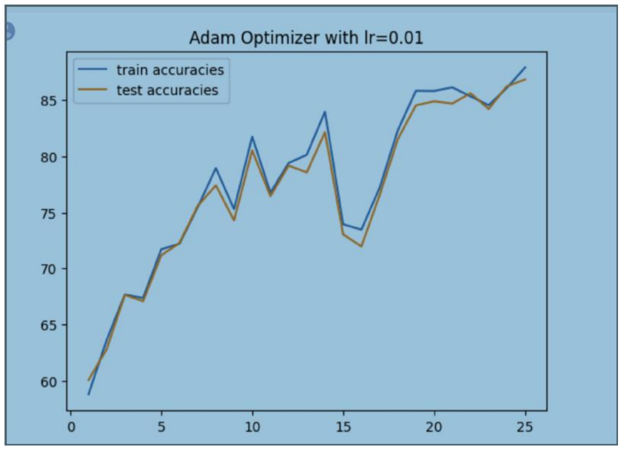
```
Epoch: 23
for epoch 23 validated on 6000 images train accuracy: 91.78 % test accuracy: 90.92 %
Epoch: 24
for epoch 24 validated on 6000 images train accuracy: 91.33 % test accuracy: 91.32 %
Epoch: 25
for epoch 25 validated on 6000 images train accuracy: 91.1 % test accuracy: 90.47 %
final model accuracy on the 6000 testing images: 91.16666666666667 %
training time: 25.067333333333334 min
```

Test and Train Accuracies are:



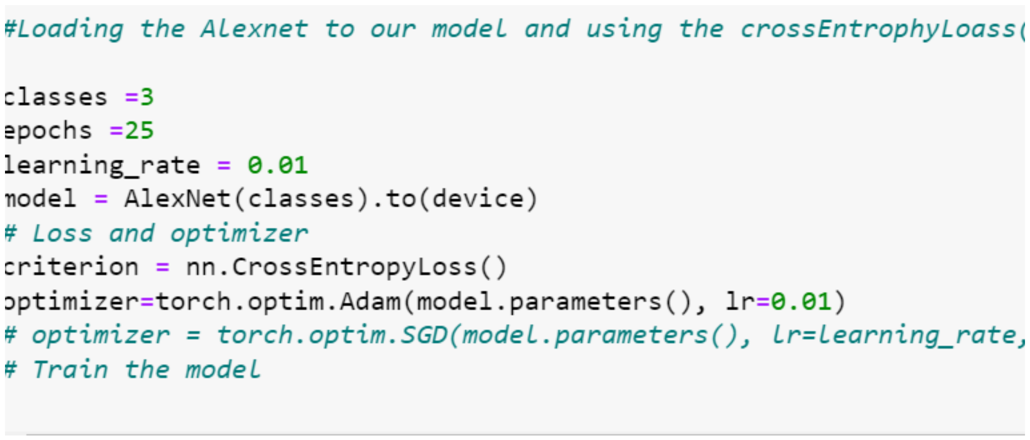
```
plt.plot(range(1,26),train_accuracies, label = "train accuracies")
plt.plot(range(1,26),test_accuracies, label = "test accuracies")
plt.legend()
plt.title("Adam Optimizer with lr=0.005")
plt.show()
```



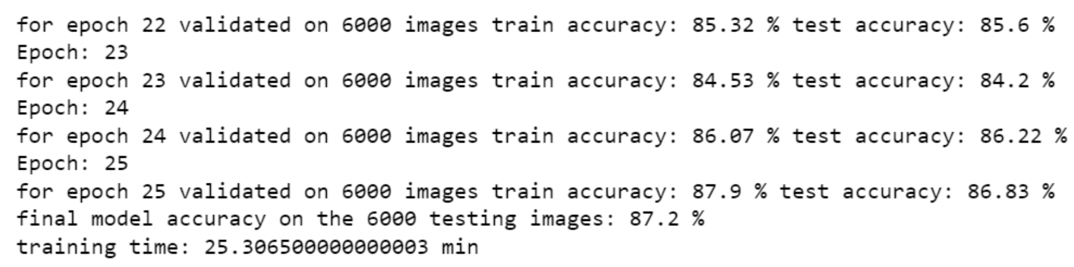


Inferences from the graph:

From the above graph we have observed that the model is not overfitting test and train accuracies of the data are increased in the initial epochs in the learning rate 0.05 and for the 0.01 learning it increased in a slight slow manner and got deceased at the 16 epoch for the 0.01. But for the learning rate it increased uniformly.



```
#Loading the Alexnet to our model and using the crossEntrophyLoass(

classes =3
epochs =25
learning_rate = 0.01
model = AlexNet(classes).to(device)
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer=torch.optim.Adam(model.parameters(), lr=0.01)
# optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate,
# Train the model
```



```
for epoch 22 validated on 6000 images train accuracy: 85.32 % test accuracy: 85.6 %
Epoch: 23
for epoch 23 validated on 6000 images train accuracy: 84.53 % test accuracy: 84.2 %
Epoch: 24
for epoch 24 validated on 6000 images train accuracy: 86.07 % test accuracy: 86.22 %
Epoch: 25
for epoch 25 validated on 6000 images train accuracy: 87.9 % test accuracy: 86.83 %
final model accuracy on the 6000 testing images: 87.2 %
training time: 25.306500000000003 min
```

To improve the Accuracy of the Model, we have used the Other optimizing techniques like:

We have added additional convolutional layers to improve the existing model.
We have added 2 more convolutional layers to existing convolutional layers to see the difference in the performance.
 We have also seen the improvement in performance with changing the activation function to LeakyRelu and Tanh in the 3rd and the 4th convolutional layers.
We have observed that the dropout rate of 0.5 is causing the model to underfitting, which can be solved by changing the dropout to 0.3 probability.
This will help model from overfitting, and yet preserve most of the information.
Moreover, the original model only contains only 3 fully connected layers. We have observed that by adding a few more fully connected layers, we were able to improve the accuracy by 10 percent.

But the Accuracy of the model increased by the SGD optimizer and also by using Sliding gradient optimizing technique, we have discussed about it previously in the part 2 report.

SGD optimizer:

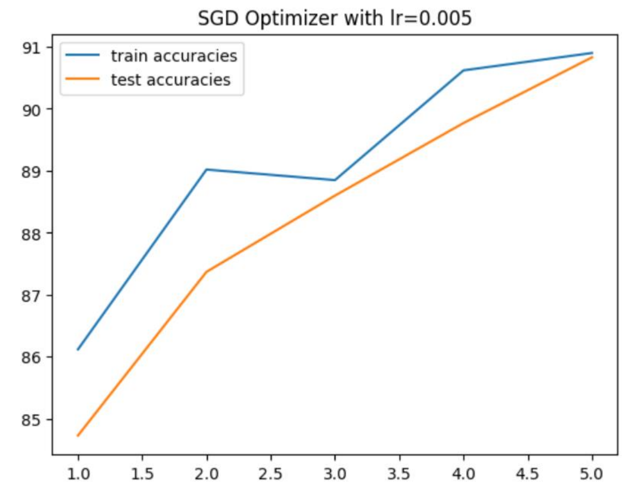
```
classes =3
epochs =5

model = AlexNet(classes).to(device)
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
# optimizer=torch.optim.Adam(model.parameters(), lr=0.01)
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay = 0.005, momentum = 0.9)
# Train the model
```

```
        correct1 += (predicted == labels).sum().item()
        del images, labels, outputs
print('final model accuracy on the {} testing images: {} %'.format(total1,100*correct1/total1))
print("training time: {} min".format(round(end_time-start_time,2)/60))
```

```
Epoch: 1
for epoch 1 validated on 6000 images train accuracy: 86.12 % test accuracy: 84.73 %
Epoch: 2
for epoch 2 validated on 6000 images train accuracy: 89.02 % test accuracy: 87.37 %
Epoch: 3
for epoch 3 validated on 6000 images train accuracy: 88.85 % test accuracy: 88.6 %
Epoch: 4
for epoch 4 validated on 6000 images train accuracy: 90.62 % test accuracy: 89.77 %
Epoch: 5
for epoch 5 validated on 6000 images train accuracy: 90.9 % test accuracy: 90.83 %
final model accuracy on the 6000 testing images: 90.7 %
training time: 5.005833333333333 min
```

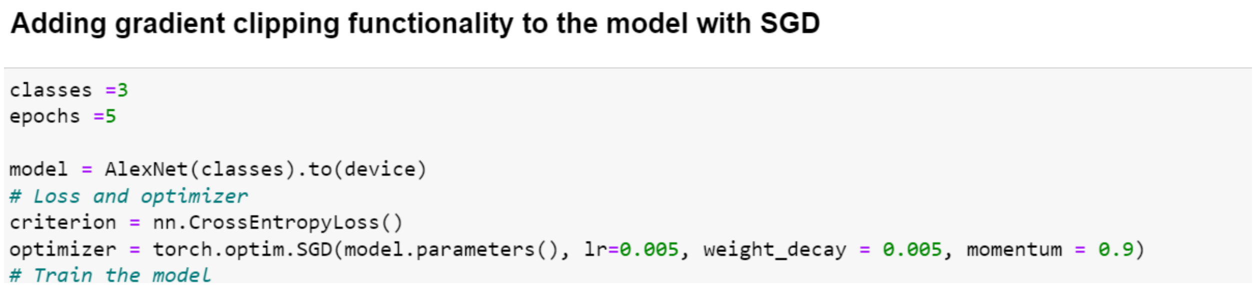
Plot :

```
plt.plot(range(1,6),test_accuracies, label = "test accuracies")
plt.legend()
plt.title("SGD Optimizer with lr=0.005")
plt.show()
```



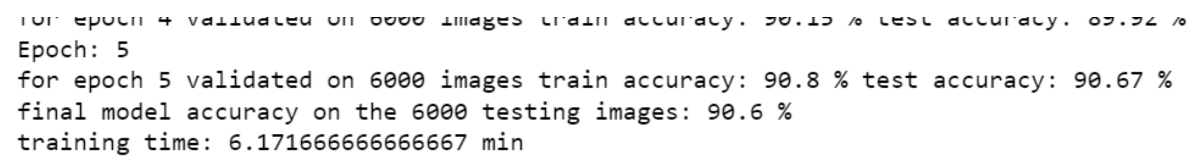
We have observed from the graph that the model have learned until the epoch 2 for the train data and later on it got decreased slightly and increased and again decreased att 3 epoch and later on increased. Test data increased uniformly after the 2 epoch.

Sliding gradient:

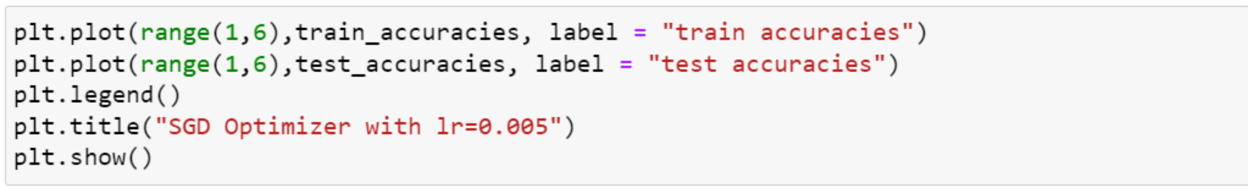


Adding gradient clipping functionality to the model with SGD

```
classes =3
epochs =5

model = AlexNet(classes).to(device)
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.005, weight_decay = 0.005, momentum = 0.9)
# Train the model
```

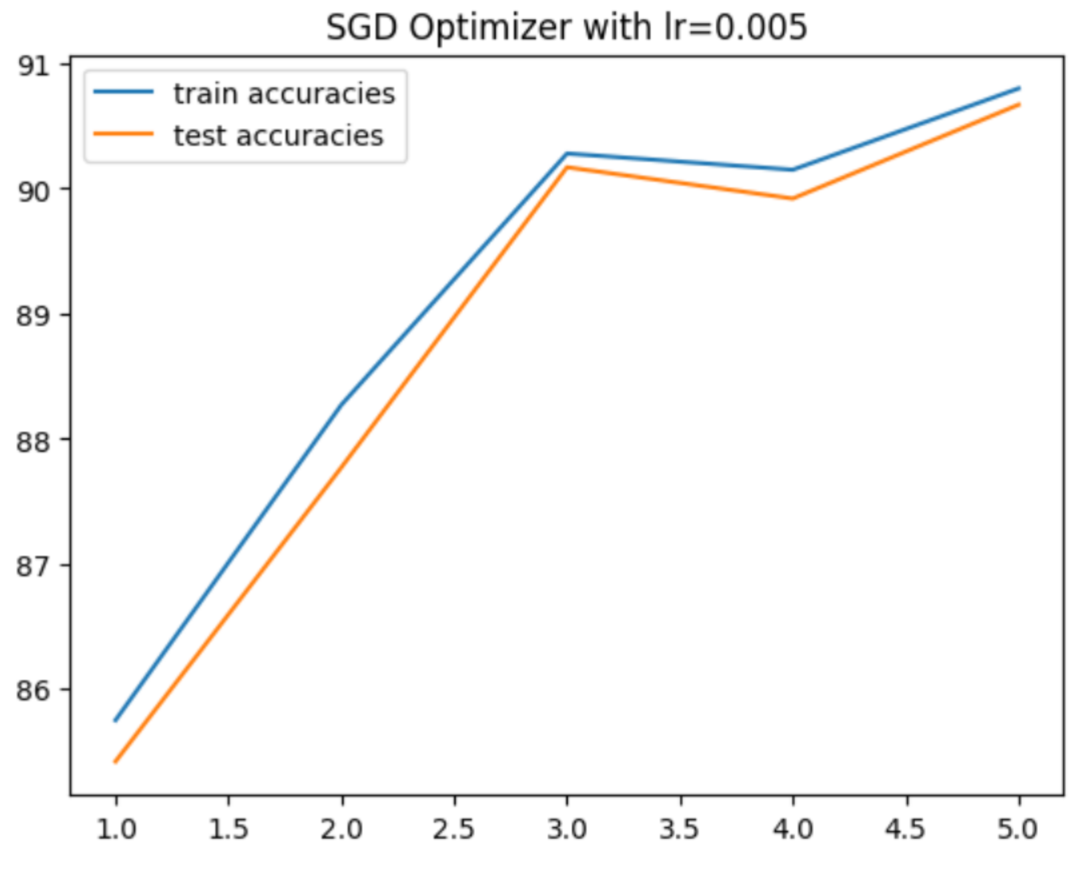


```
for epoch 4 validated on 6000 images train accuracy: 90.15 % test accuracy: 89.92 %
Epoch: 5
for epoch 5 validated on 6000 images train accuracy: 90.8 % test accuracy: 90.67 %
final model accuracy on the 6000 testing images: 90.6 %
training time: 6.171666666666667 min
```



```
plt.plot(range(1,6),train_accuracies, label = "train accuracies")
plt.plot(range(1,6),test_accuracies, label = "test accuracies")
plt.legend()
plt.title("SGD Optimizer with lr=0.005")
plt.show()
```

Plot:



Inferences from the graph:

Here we have observed that the accuracy of the model uniformly increased until the 3rd epoch , and later on the for both the data (test and train) the accuracy decreased at 4th epoch and increased later on. Both the test and the train data acted similarly.

About the dataset:

SVHN dataset is one of the most popular datasets in the py-torch datasets library used for the image processing models. SVHN mean Street View House Numbers. This dataset consists a set of images containing the house numbers which are captured with the help of Google's Street view.

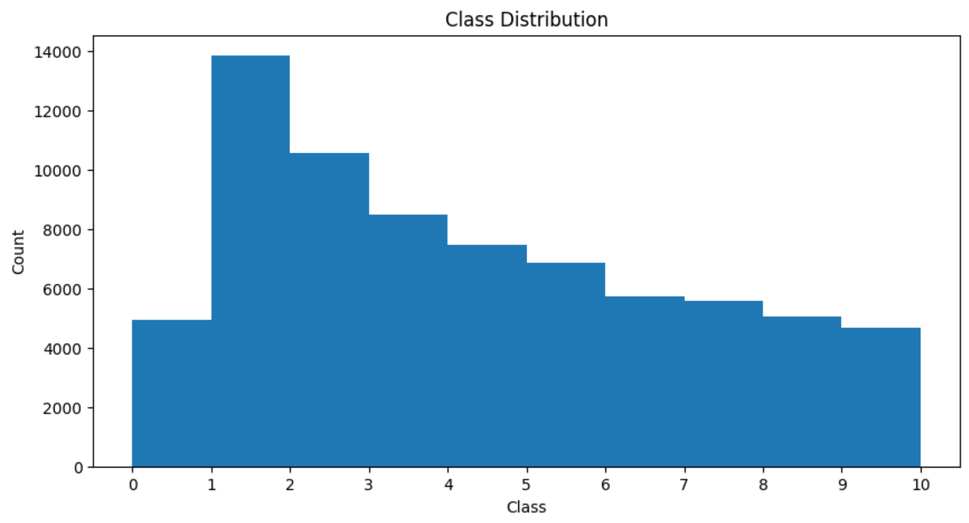
The dataset has a total of around 604,388 images. Out of these 73,257 images are for the training and 26,032 images are for the testing purposes. All the remaining images are the additional images which are used for the training purposes if necessary.

Also, this dataset includes the labels for all the images. The images in the dataset are divided into 10 classes and the all the images are assigned to one of the class or commonly known as the label. Here we have 10 labels ranging from 0 to 9 and the images are assigned to these labels according to their digits.

This SVHN dataset includes 3 variables: image data, label data, metadata.

Data visualization:

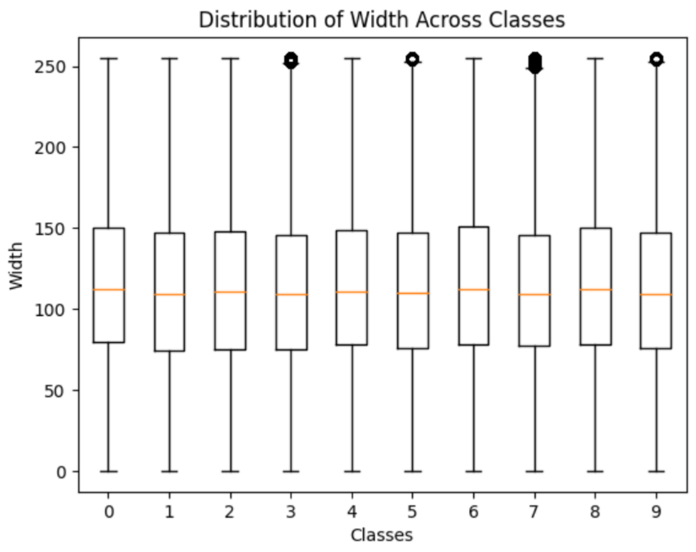
Plot1: Class distribution for all the images



This plot gives an overview of the number of images available for each label in the training dataset which is being chosen for the model training.

From the Plot it is observed that the images belonging to the class 1 (images for numerical 1) are more in count and class 2 stands next to it. Also, the images belonging to the class 0 are least in count.

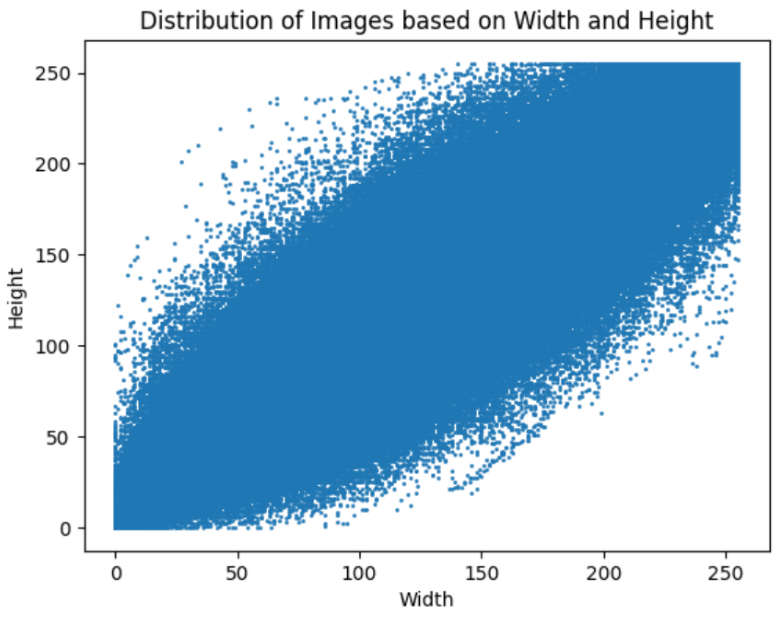
Plot 2: Width of the images for each class



This plot shows the distribution of the width of the SVHN dataset images across the 10 different classes. The significance of this plot lies in its ability to help us understand the variation of image width across different classes, which can provide insights into the nature of the dataset and inform the choice of image preprocessing and model architecture.

This helps us understand the variation of images width across the different classes. With the help of this visualization, we can consider the step to resize or cropping the images to standard size. This will make sure that the model learns about the patterns rather than getting affected by the issues with the size of the images.

Plot 3: Distribution of Images according to the sizes



The above scatter plot gives us an idea about the distribution of images according to the sizes in the SVHN dataset. This can assist in preprocessing steps or adjustments to the model architecture to ensure that the model can handle the range of image sizes and shapes present in the dataset.

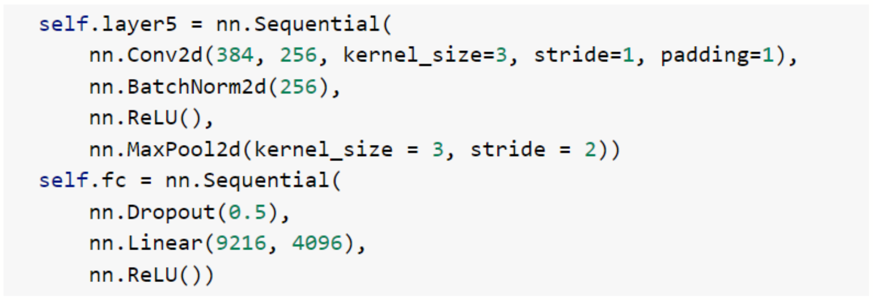
In short, the scatter plot provides a visual representation of the distribution of images in the SVHN dataset, which can inform our understanding of the dataset and guide our approach to building a machine learning model for classification.

CNN Model for the SVHN data set:

From all the models used for training the neural network training in part 3, considering the model with the Stochastic gradient descent as the optimizer as this is the model, we are getting the best possible accuracy with less training time.

As the model is mostly designed for training of the any dataset with any number of classes importing the model directly to our environment.

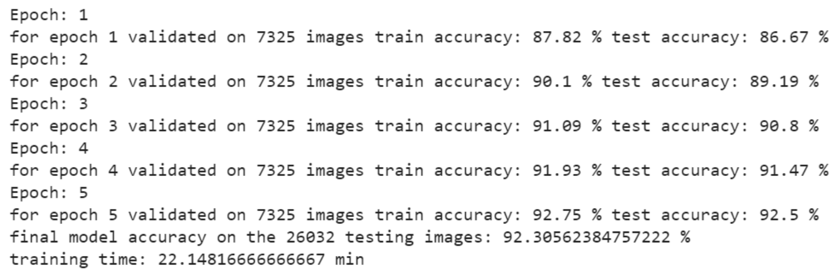
One main change in the model is that we need to change the input size for the linear network depending on the size of the output from the 5th convolution layer. As this might change depending on the size of input tensor. Here the after applying the 5th convolution layer with stride 1 and padding 1 we obtain the out matrix of size (64 X 9216). So, we will be passing the input of the size 9216 to the linear layer.



```
self.layer5 = nn.Sequential(
    nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
    nn.BatchNorm2d(256),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size = 3, stride = 2))
self.fc = nn.Sequential(
    nn.Dropout(0.5),
    nn.Linear(9216, 4096),
    nn.ReLU())
```

Model Implementation without Data augmentation:

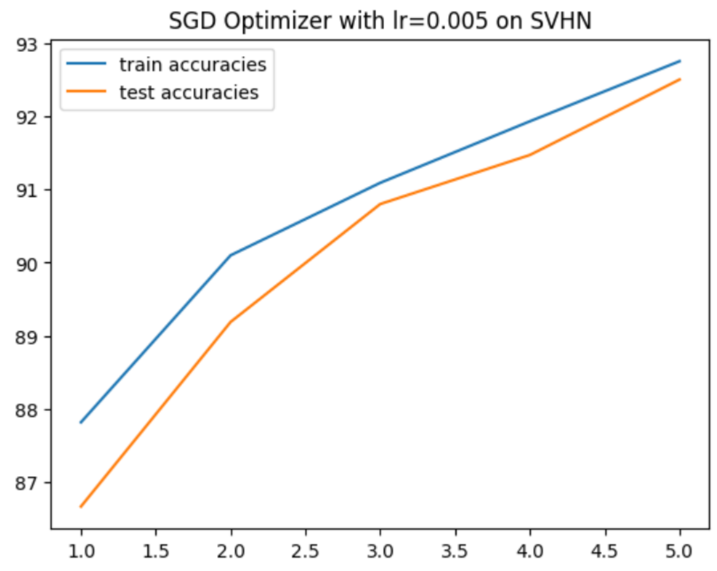
Training the model on the dataset with out making any sort of changes to the dataset. For this data we obtained an accuracy of the 92.3%.



```
Epoch: 1
for epoch 1 validated on 7325 images train accuracy: 87.82 % test accuracy: 86.67 %
Epoch: 2
for epoch 2 validated on 7325 images train accuracy: 90.1 % test accuracy: 89.19 %
Epoch: 3
for epoch 3 validated on 7325 images train accuracy: 91.09 % test accuracy: 90.8 %
Epoch: 4
for epoch 4 validated on 7325 images train accuracy: 91.93 % test accuracy: 91.47 %
Epoch: 5
for epoch 5 validated on 7325 images train accuracy: 92.75 % test accuracy: 92.5 %
final model accuracy on the 26032 testing images: 92.30562384757222 %
training time: 22.14816666666667 min
```

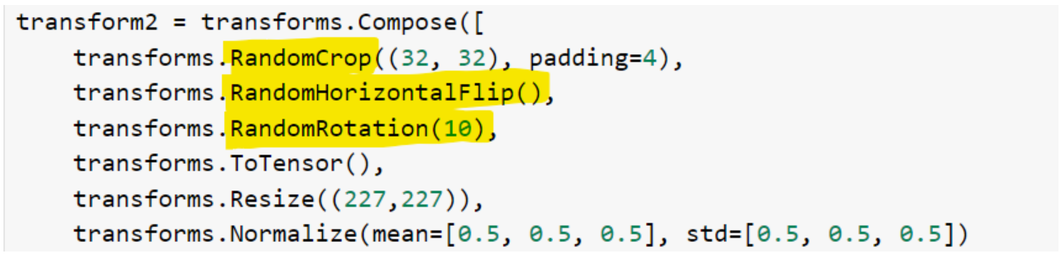
Plot between train accuracy and test accuracy:

But while plotting the graphs for the test accuracy and train accuracy it is observed that the model is overfitting over the data.



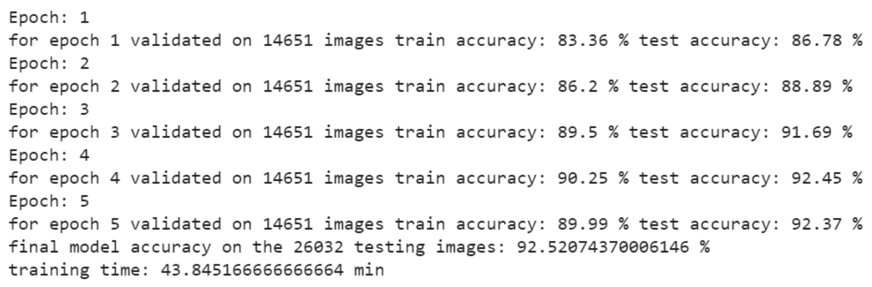
Model 2: Implementation with the data augmentation techniques:

Here we are using combination of data augmentation techniques namely, random cropping, random horizontal flip, random rotation for the random images of the training data and adding this image to the training dataset. By doing this we are doubling the size of dataset with the same images with some changes to the images. This helps in reducing the overfitting of the data.



```
transform2 = transforms.Compose([
    transforms.RandomCrop((32, 32), padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Resize((227,227)),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
```

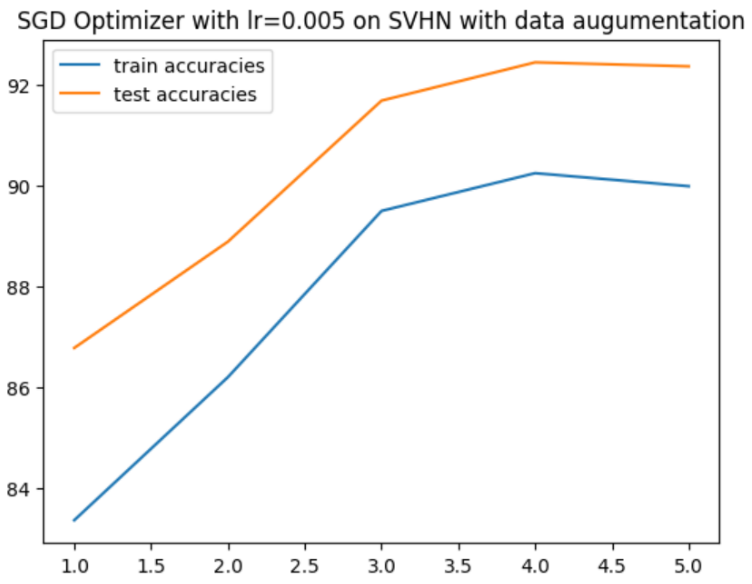
Accuracy of the model trained with the train data which is procced with the data augmentation techniques.



```
Epoch: 1
for epoch 1 validated on 14651 images train accuracy: 83.36 % test accuracy: 86.78 %
Epoch: 2
for epoch 2 validated on 14651 images train accuracy: 86.2 % test accuracy: 88.89 %
Epoch: 3
for epoch 3 validated on 14651 images train accuracy: 89.5 % test accuracy: 91.69 %
Epoch: 4
for epoch 4 validated on 14651 images train accuracy: 90.25 % test accuracy: 92.45 %
Epoch: 5
for epoch 5 validated on 14651 images train accuracy: 89.99 % test accuracy: 92.37 %
final model accuracy on the 26032 testing images: 92.52074370006146 %
training time: 43.845166666666664 min
```

There is an improvement in the accuracy of the model which is very small and there is very much increase in the training time. This is mainly due to the increase in the size of the data set which is doubled before the training.

Plot between train accuracy and the test accuracy:



From the above plot we can observe that the test accuracies are greater than the train accuracies. This means that the model is fitting correctly, and no overfitting is observed. Also, the accuracy is around 92.5% which is good and desirable.

Contribution table:

Contribution table

Team member	Assignment Part	Contribution
Tharun Sai Malireddy (tharunsai)	Part 1	50%
Sai Korupolu (saikorup)	Part 1	50%
Tharun Sai Malireddy (tharunsai)	Part 2	50%
Sai Korupolu (saikorup)	Part 2	50%
Tharun Sai Malireddy (tharunsai)	Part 3	55%
Sai Korupolu (saikorup)	Part 3	45%
Tharun Sai Malireddy (tharunsai)	Part 4	45%
Sai Korupolu (saikorup)	Part 4	55%

References:

https://www.aboutdatablog.com/post/how-to-successfully-add-large-data-sets-to-google-drive-and-use-them-in-google-colab

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

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

https://www.educative.io/answers/what-is-early-stopping

https://machinelearningmastery.com/how-to-avoid-exploding-gradients-in-neural-networks-with-gradient-clipping/#:~:text=Gradient%20clipping%20involves%20forcing%20the,to%20as%20%E2%80%9Cgradient%20clipping.%E2%80%9D

Class recording and PPts.