ELL409
Assignment-1
Linear Regression and Classification

Etiksha Jain(2020EE10491)
Sampan Manna(2020EE10547)

Q1, Q2

MNIST Dataset Description

The MNIST dataset contains a total of 70,000 images. These images are 28x28 pixels in size, and each image represents a single handwritten digit. The digits range from 0 to 9.All images in the MNIST dataset are in grayscale, meaning they have a single channel (black and white).

We split the dataset into test and train dataset. The model was trained on trainset and testes on test set.
Total samples = 70,000
Training samples = 60,000
Test samples = 10,000
Image dimensions = 28x28
Number of Output class = 10
This is modelled as a Linear Classification problem.

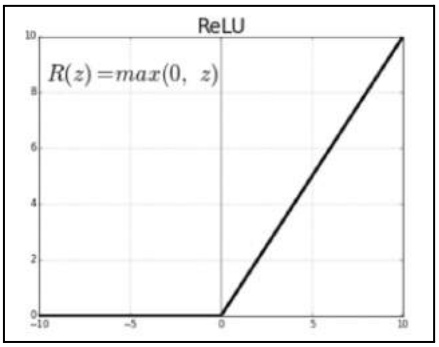
CIFAR10 Dataset Description

The CIFAR10 dataset contains a total of 60,000 images. These images are 3x32x32 pixels in size, and each image represents an object. There are 10 classes.All images in the CIFAR10 dataset are colored.

We split the dataset into test and train dataset. The model was trained on trainset and testes on test set.
Total samples = 60,000
Training samples = 50,000
Test samples = 10,000
Image dimensions = 3x32x32
Number of Output class = 10
This is modelled as a Linear Classification problem.

Model Architecture

As given in the assignment, we used a Linear Classification model. The model consists of a single linear layer followed by an activation ReLu layer.



ReLu function

The model has been implemented using Pytorch's nn module as follows-

```
class Net(nn.Module):
    def __init__(self):
        super(Net,self).__init__()
        self.linear1 = nn.Linear(3*32*32, 10)
        self.relu = nn.ReLU()

    def forward(self, img): #convert + flatten
        x = img.view(-1, 3*32*32)
        x = self.relu(self.linear1(x))
        return x
```

HyperParamters

The set of hyperparamters involved while training the model are as follows-

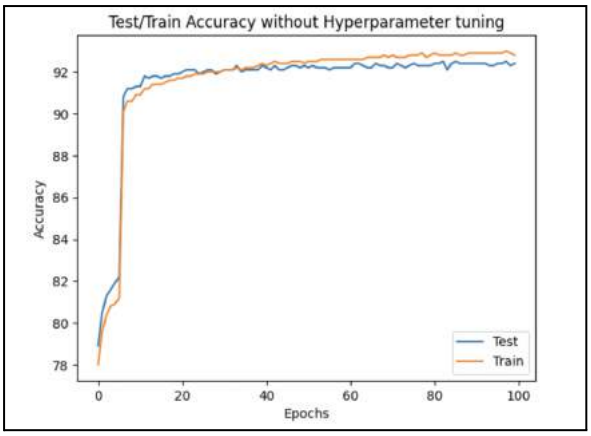
1. Number of epochs
2. Learning Rate
3. LR scheduler
4. Regularisation strength
5. Batch sizes
6. Optimizer

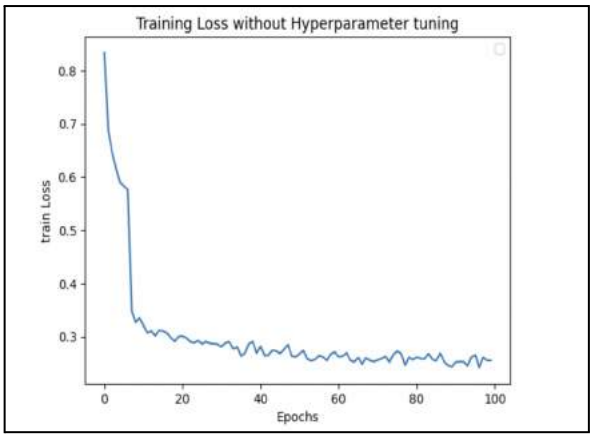
No hyperparameter tuning

HyperParameters Used-

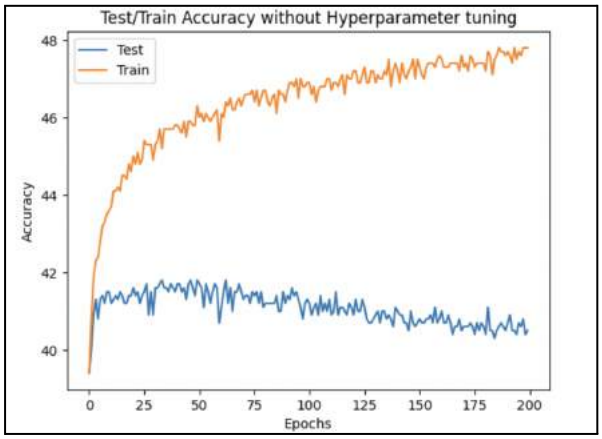
```
batch_size=128
num_epochs=100 #(200 for cifar10)
loss_fn = nn.CrossEntropyLoss()
LR = 0.01
optimizer = t.optim.SGD(net.parameters(), lr=LR)
```

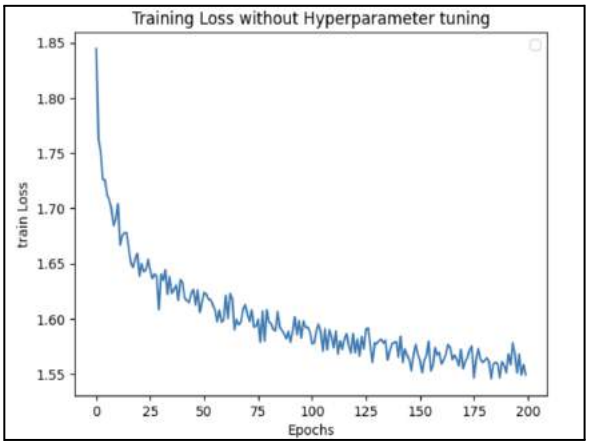
Results(MNIST)





Results(CIFAR10)





Inference-

We can see that the accuracy is good for MNIST dataset but not so much CIFAR dataset since the latter has colored images and linear models are not good for such datasets.

Tuning Learning Rate

Learning rate is a positive scalar value that controls the size of the steps taken when adjusting the model's parameters during optimization.

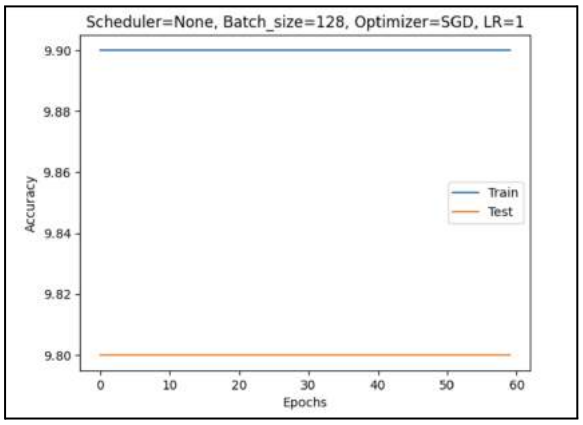
HyperParameters Used-

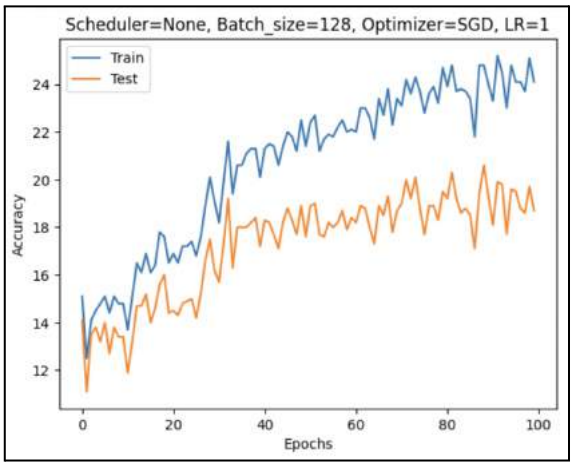
```
batch_size=128
num_epochs=100
loss_fn = nn.CrossEntropyLoss()
optimizer = t.optim.SGD(net.parameters(), lr=LR)
```

Various Learning Rates-

1. LR = 1

Results-



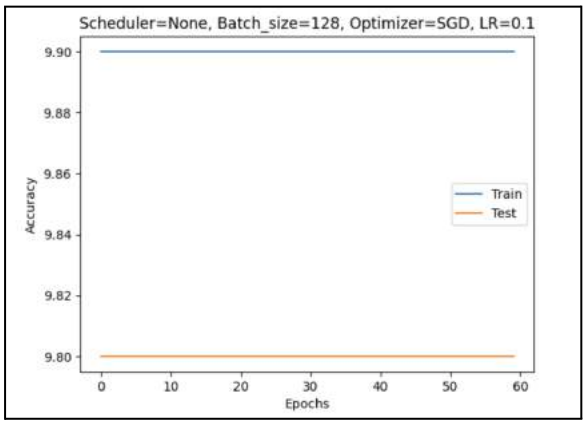


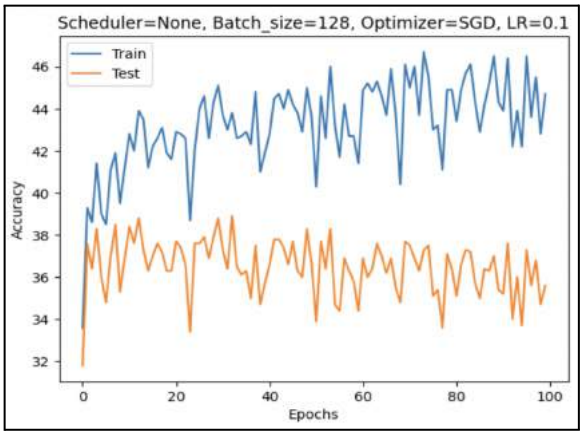
MNIST CIFAR10

Inference- A learning rate of 1 is too high for the model to train in case of MNIST. In case of CIFAR10, the accuracy does increase, but it does not converge to a particular point and keeps oscillating with the number of epochs

2. LR = 0.1

Results-



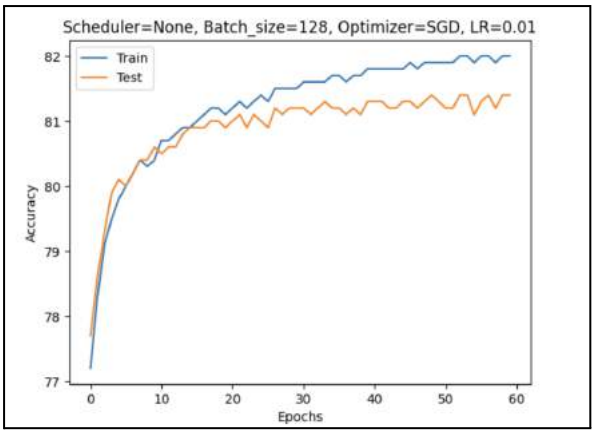


MNIST CIFAR10

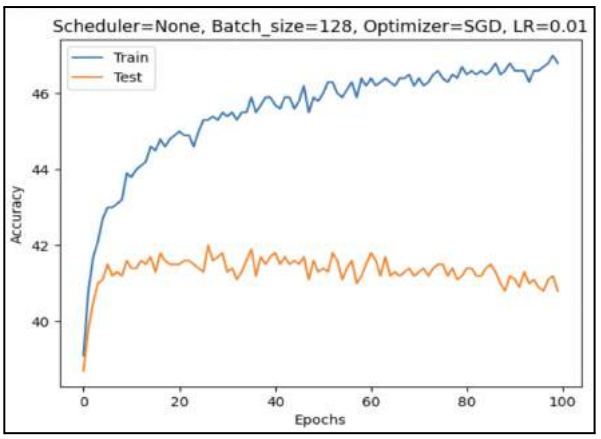
Inference- Similar to the case of LR=1 except that for CIFAR10, the accuracy does not increase appreciably for the test set.

3. LR = 0.01

Results-



MNIST

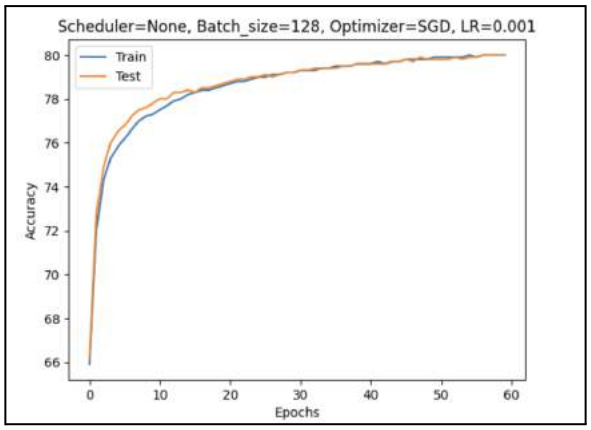


CIFAR10

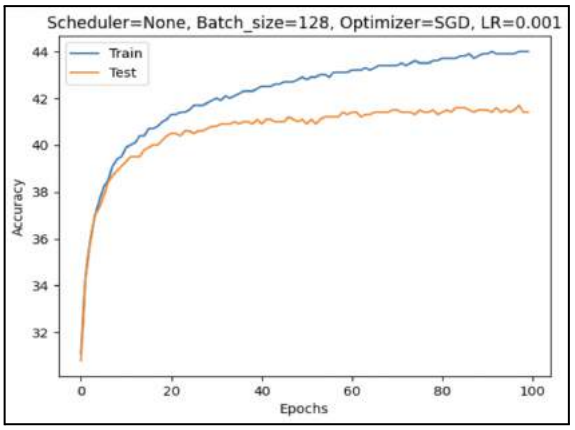
Inference- The accuracy increases for both training and test set for MNIST. For CIFAR10, the accuracy of both training and test set remains in the same range as that of LR=0.1, but has less variations with each epoch now.

4. LR = 0.001

Results-



MNIST

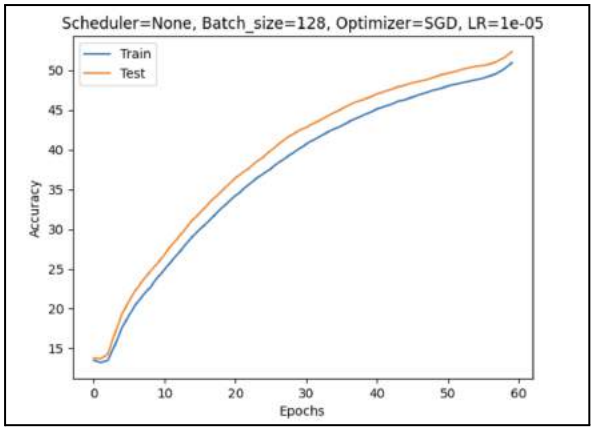


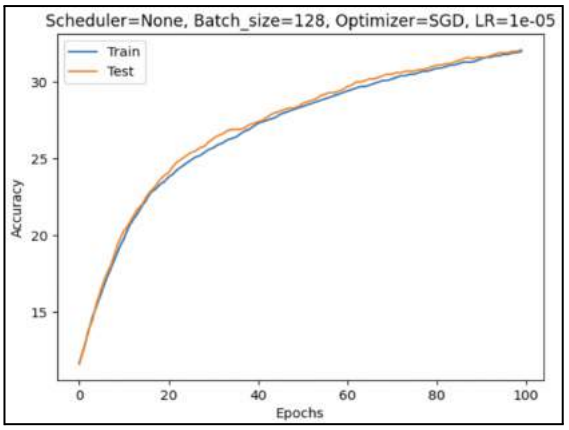
CIFAR10

Inference- Similar to above, the graph of CIFAR10 is even smoother with a lower learning rate

5. LR = 1e-5

Results-



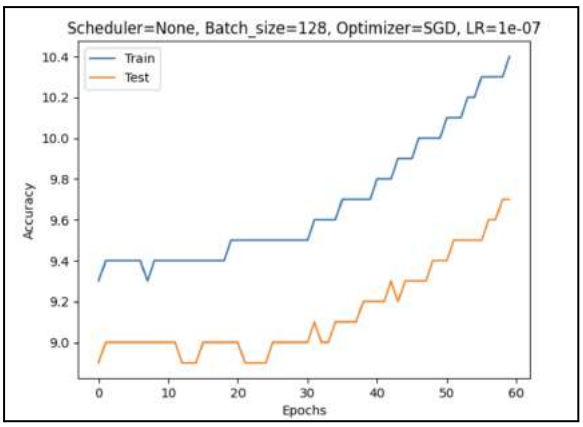


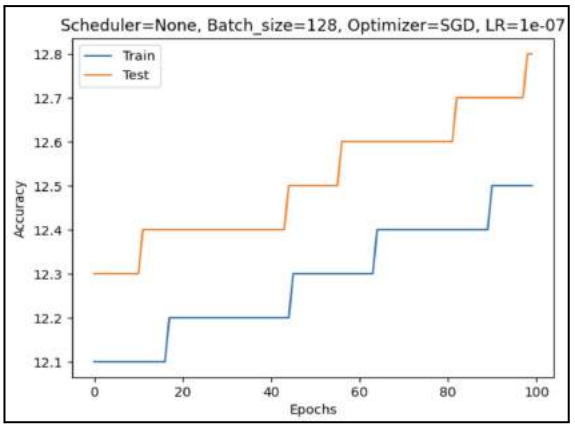
MNIST CIFAR10

Inference- The accuracy steadily increases with every epoch, however, the LR is too small to reach the minima for the given number of epochs. Hence, we get a lower accuracy overall.

6. LR = 1e-7

Results-



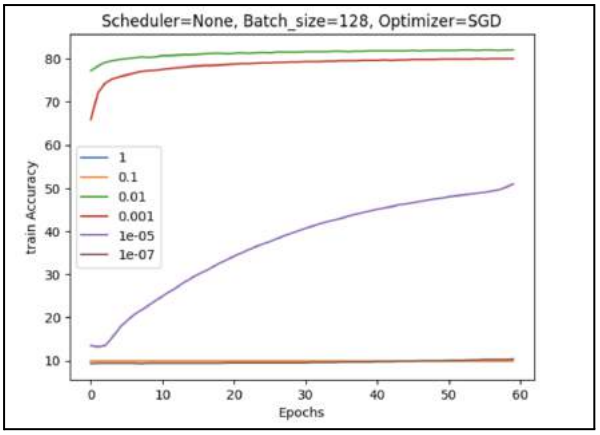


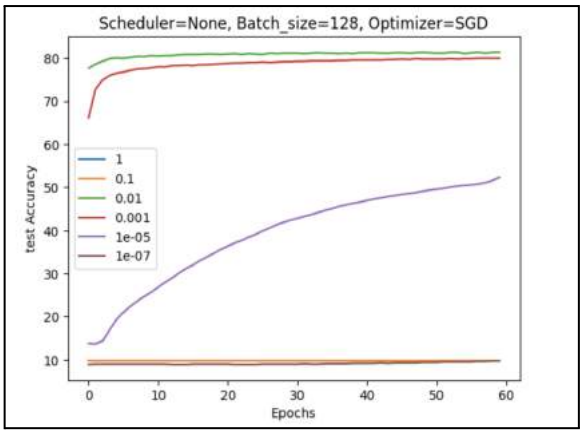
MNIST CIFAR10

Inference- The model trains even more slowly with each epoch because of such a slow learning rate. The accuracy drastically reduces

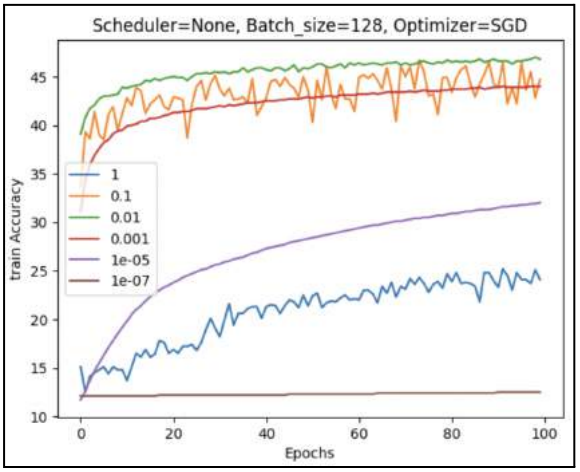
Best Learning Rate-

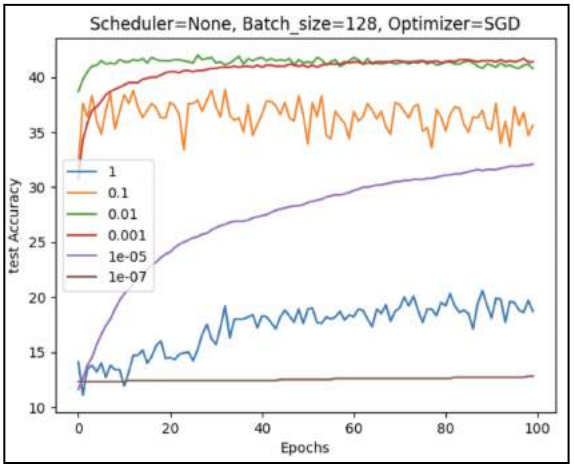
MNIST





CIFAR10





Conclusion-

- If the learning rate is set too high, the optimization algorithm may take excessively large steps, causing the loss function to oscillate or diverge
- If the learning rate is set too low, the optimization algorithm takes very small steps, which can result in slow convergence

As we can see in the graphs, the accuracy is maximum for LR=0.01.

Tuning LR Scheduler

The purpose of a learning rate scheduler is to improve the convergence and performance of the model by changing the learning rate over time in response to the training dynamics.

HyperParameters Used-

```
batch_size=128
num_epochs=100
LR = 0.01
loss_fn = nn.CrossEntropyLoss()
optimizer = t.optim.SGD(net.parameters(), lr=LR)
```

```
scheduler = StepLR(optimizer, 25, gamma=gamma)
```

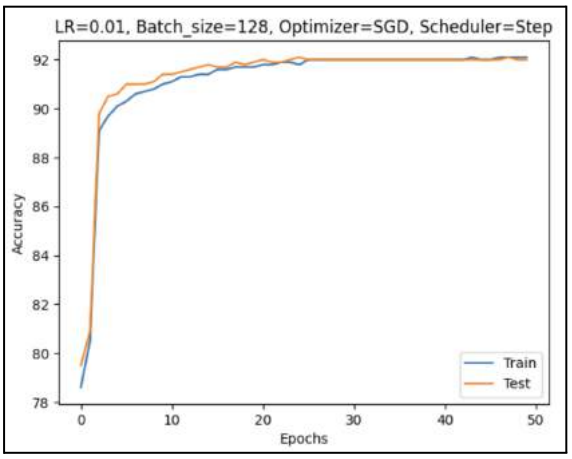
Various LR Schedulers-

1. StepLR

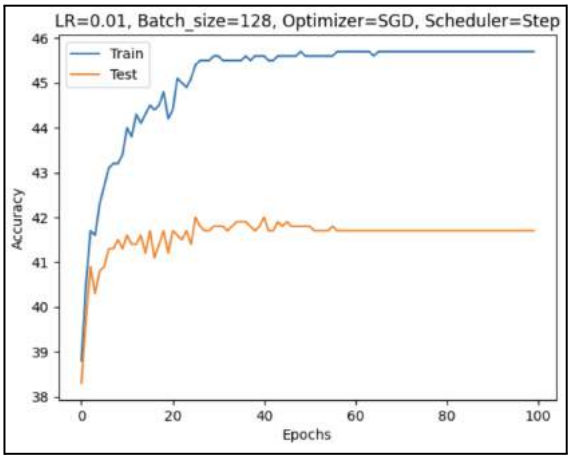
In the StepLR scheduler, the learning rate is reduced by a fixed factor after a fixed number of training epochs. We have used a scheduler that reduces LR by 0.1 after every 25 epochs.

```
from torch.optim.lr_scheduler import StepLR
scheduler = StepLR(optimizer, 25, gamma=0.1)
```

Results-



MNIST



CIFAR10

Inference-

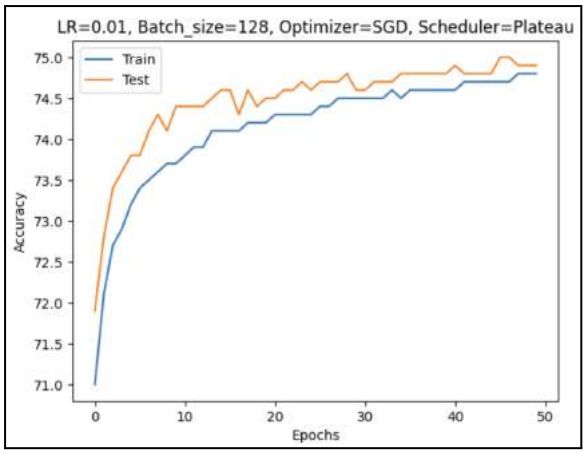
We can see that the accuracy is good for both MNIST and CIFAR10 datasets with less noise and faster convergence.

2. ReduceLROnPlateau

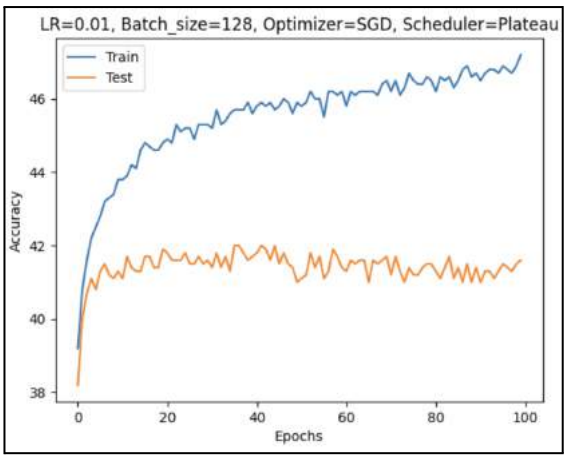
This scheduler monitors a specified metric (validation loss) and reduces the learning rate if the monitored metric stops improving. We have used a scheduler that reduces LR by 0.1 if there's no improvement in the test_loss after for 20 epochs. The min lr it can reach is set to be 1e-5.

```
from torch.optim.lr_scheduler import ReduceLROnPlateau
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=20,
min_lr=1e-5, verbose=True)
```

Results-



MNIST



CIFAR10

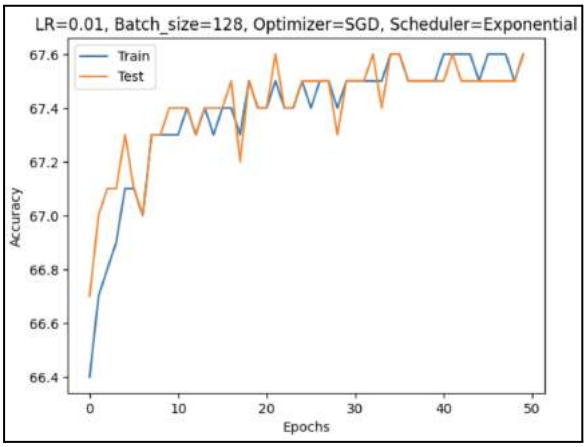
Inference-
We can see that the accuracy is lesser for both MNIST and CIFAR10 datasets with more noisier updates.

3. ExponetialLR

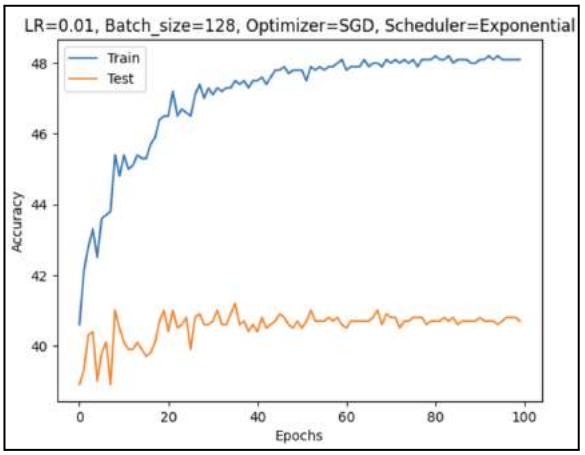
In this scheduler, the learning rate is reduced exponentially over time. We have used a scheduler that reduces LR by 0.95 in every epoch.

```
from torch.optim.lr_scheduler import ExponenetialLR
scheduler = ExponentialLR(optimizer, gamma=0.95)
```

Results-



MNIST

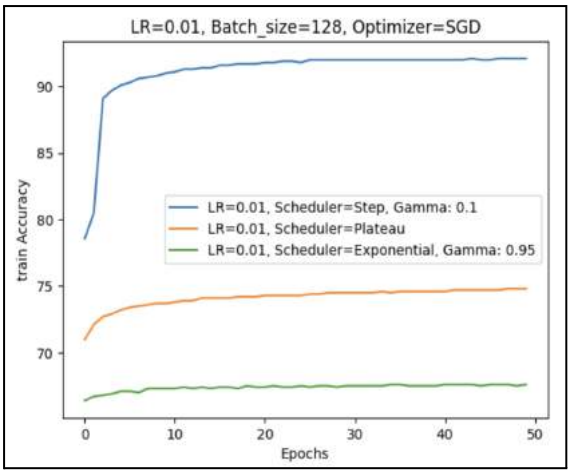


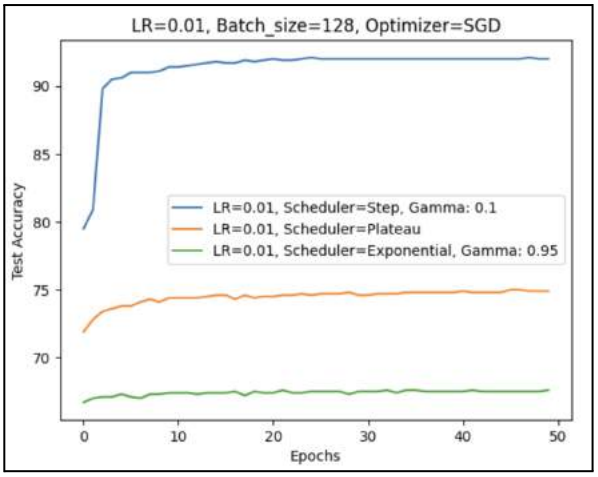
CIFAR10

Inference-
We can see that the accuracy is lesser for MNIST but great for CIFAR10 datasets with more noisier updates and slower convergence.

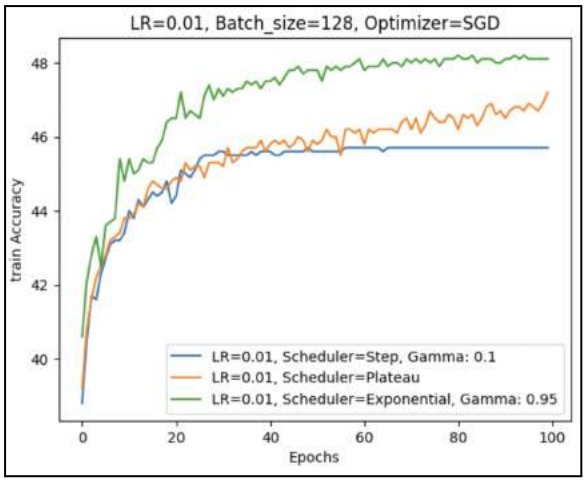
Best LR Scheduler-

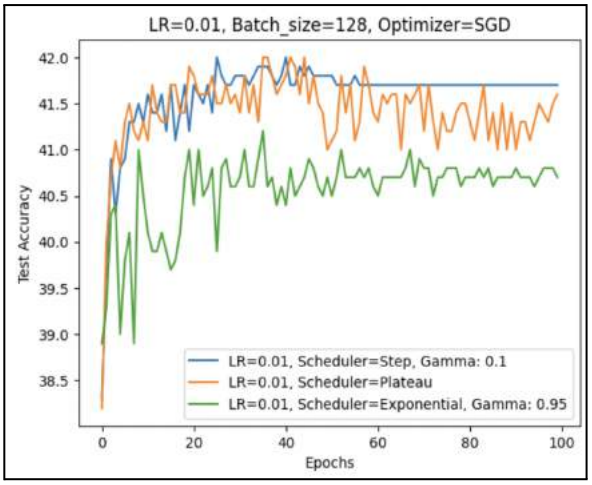
MNIST





CIFAR10





Conclusion-

As we can see in the graphs, that the accuracy is maximum Step LR for btoh datasets.

Adding L2 Regularisation

L2 regularization is a common regularization technique used to prevent overfitting of models.

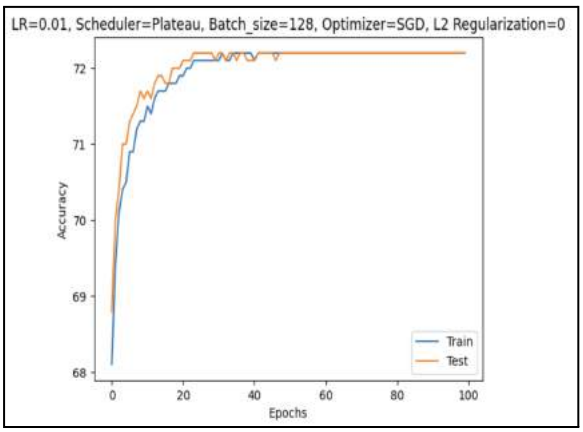
HyperParameters Used-

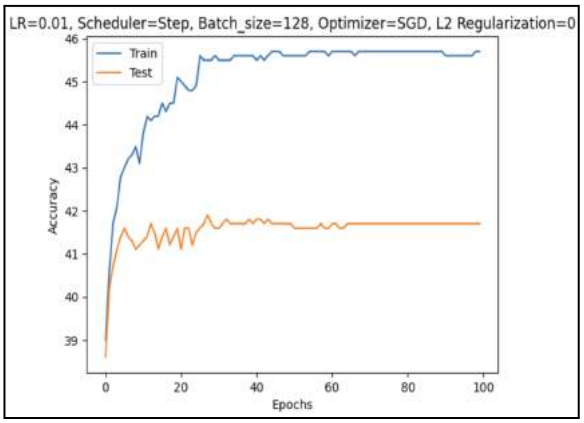
```
batch_size=128
num_epochs=100
LR=0.01
scheduler = StepLR(optimizerr, 25, gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
l2reg=0
optimizer = t.optim.SGD(net.parameters(), lr=LR, weight_decay=l2reg)
```

Various Regularisation Strengths-

1. Regularisation Strength = 0

Results-



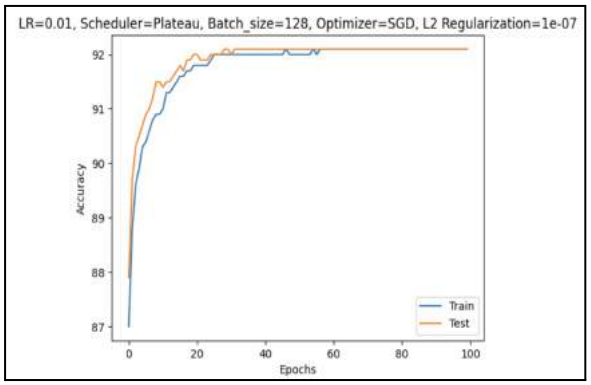


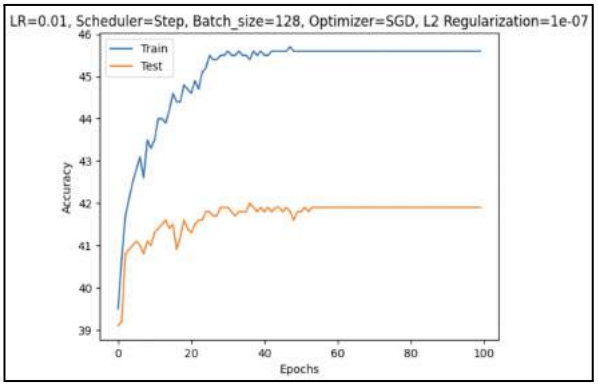
MNIST CIFAR10

Inference- For the MNIST set, there is not much difference between the train and test accuracy. However, there is a notable difference (of 3-4%) between the train and test accuracy for the CIFAR10 set.

2. Regularisation Strength = 1e-7

Results-



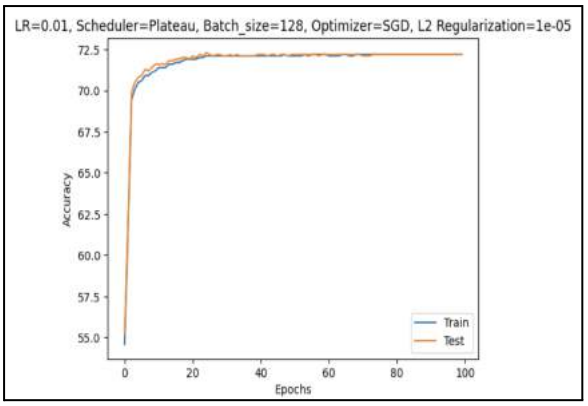


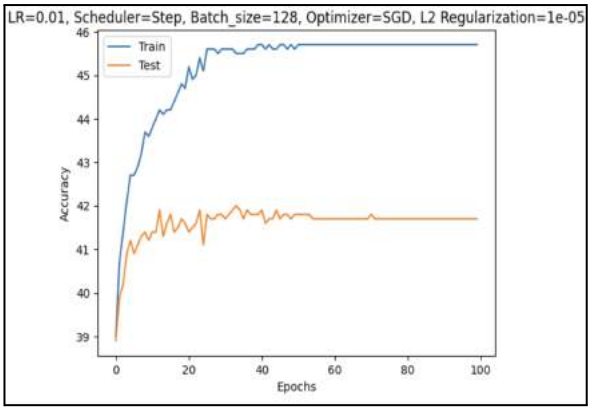
MNIST CIFAR10

Inference- We get similar results as the previous case. The regularisation strength is not high enough to have a good effect on the CIFAR10 dataset.

3. Regularisation Strength = 1e-5

Results-



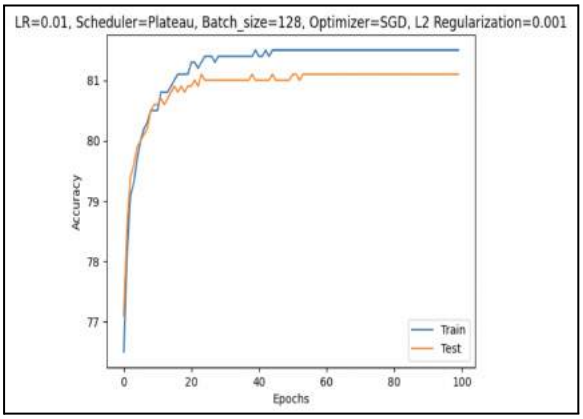


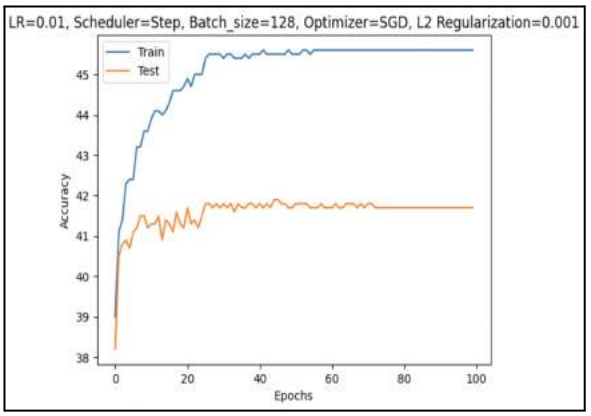
MNIST CIFAR10

Inference- Again, the regularisation strength is too small to have any appreciable effect

4. Regularisation Strength = 1e-3

Results-



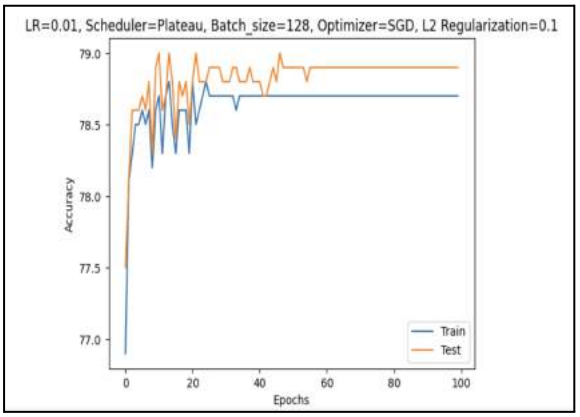


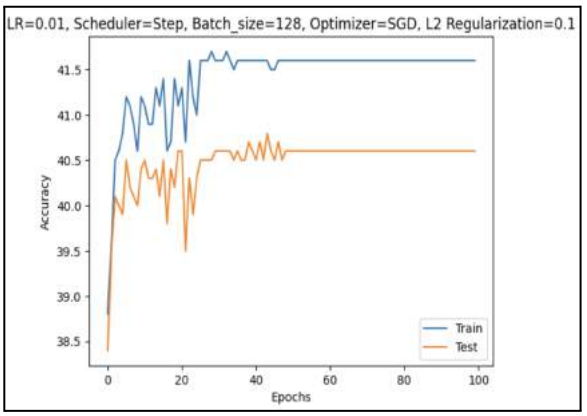
MNIST CIFAR10

Inference- For CIFAR10, we get the same result as the previous case. For the MNIST set however, we get a notably higher accuracy for the train set than the test set.

5. Regularisation Strength = 0.1

Results-



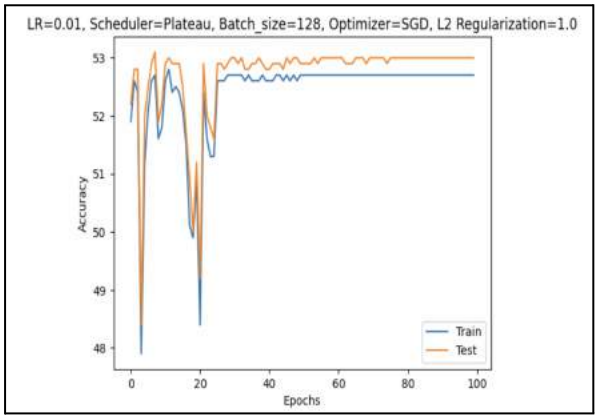


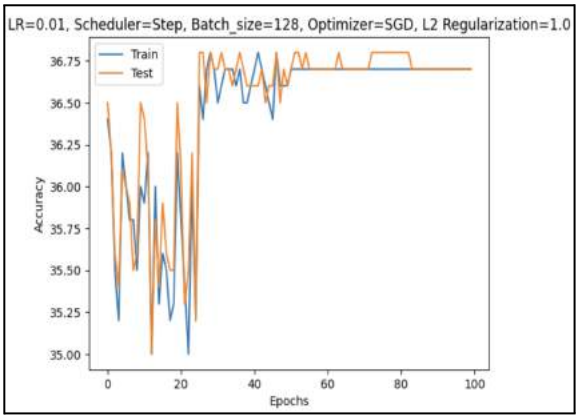
MNIST CIFAR10

Inference- This time, the difference between the training and test set decreases for CIFAR10. However, the accuracy in both these sets also reduces for CIFAR10.

6. Regularisation Strength = 1.0

Results-



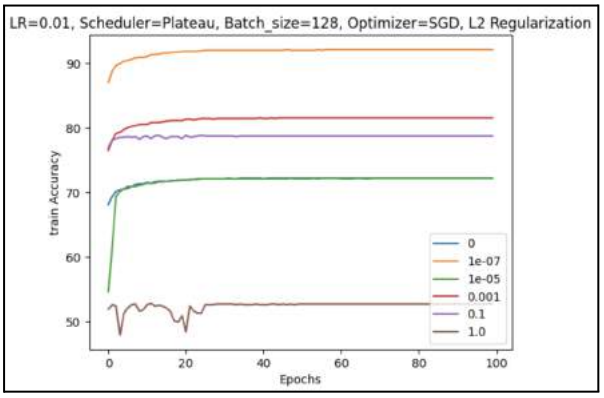


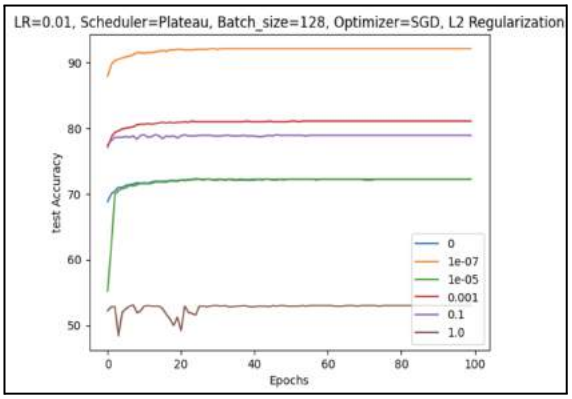
MNIST CIFAR10

Inference- The accuracy falls drastically for all 4 sets across both the datasets. Moreover, we get a much higher degree of variations or oscillations than any of the previous cases.

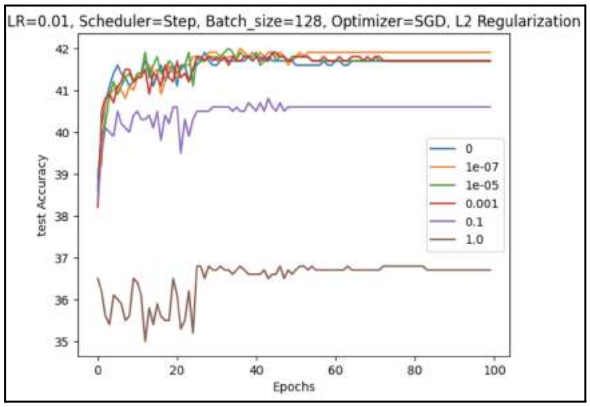
Best Regularisation Strength-

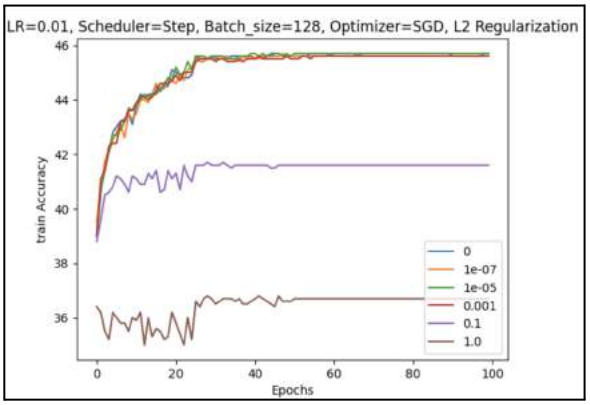
MNIST





CIFAR10





Conclusion-

As we can see in the graphs,for MNIST dataset, that accuracy is maximum for regularisation strength = 1e-7
As we can see in the graphs, for CIFAR10 dataset that accuracy is maximum for regularisation strength = 1e-5

Varying batch Size

Batch size determines the number of data samples that are processed together in each forward and backward pass during one iteration of training.

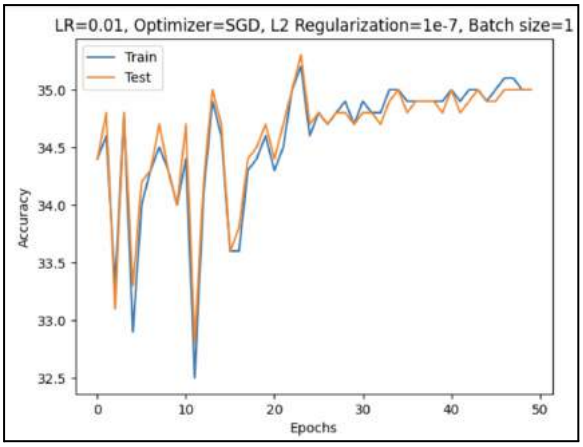
HyperParameters Used-

```
batch_size=128
num_epochs=100
LR=0.01
scheduler = StepLR(optimizerr, 25, gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
l2reg=1e-7
optimizer = t.optim.SGD(net.parameters(), lr=LR, weight_decay=l2reg)
```

Various Batch Sizes-

1. Batch Size = 1

Results-

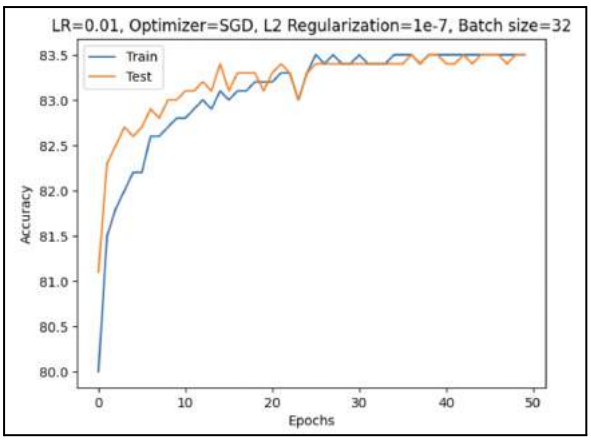


MNIST

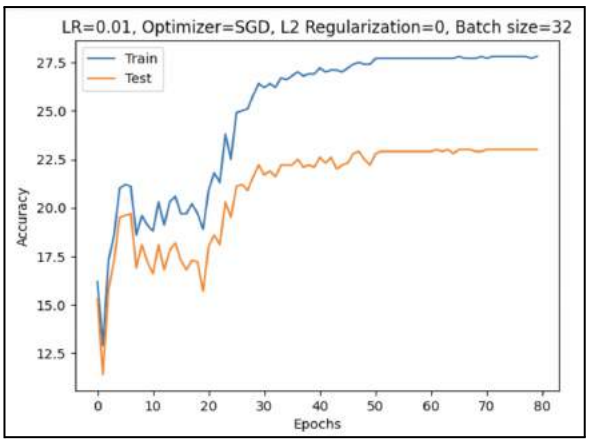
Inference-
We can see that the updates are very noisy since they occur at every datapoint and batch size=1 does not give a good accuracy. The convergence is also slow. Although, such small batchsize is computationally efficient.

2. Batch Size = 32

Results-



MNIST

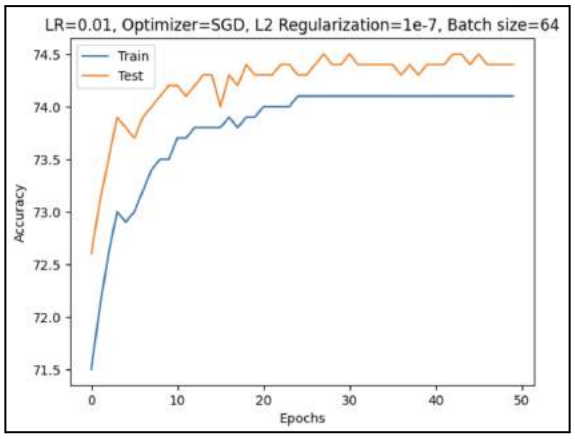


CIFAR10

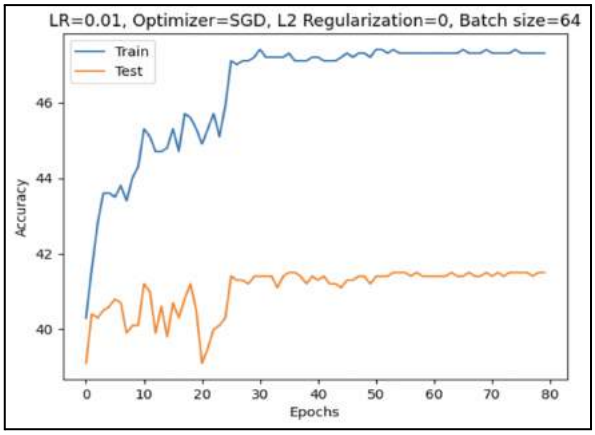
Inference-
We can see that the updates are less noisier and batch size=32 gives better accuracy than before but not the best. The convergence is slow.

3. Batch Size = 64

Results-



MNIST

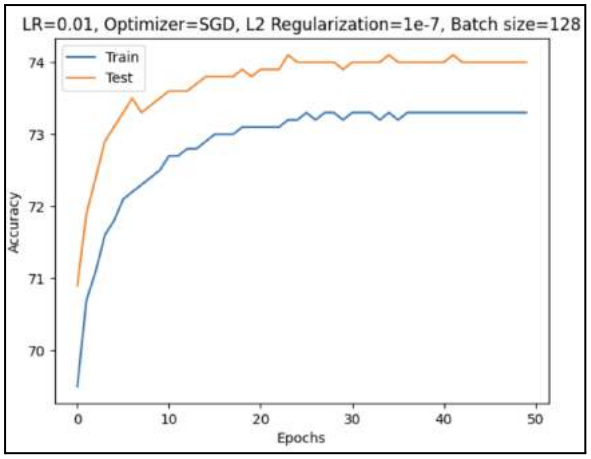


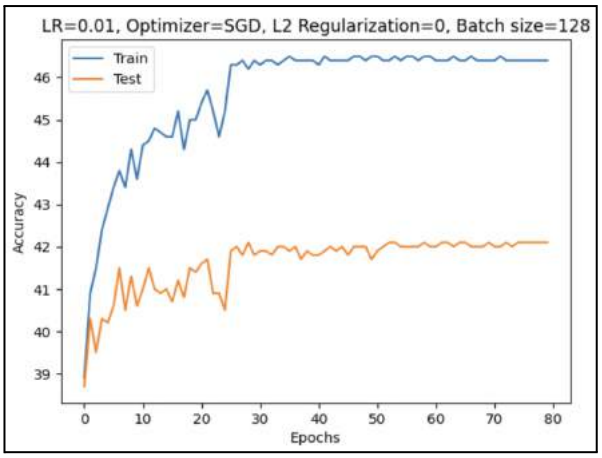
CIFAR10

Inference-
We can see that the updates are less noisier for both datasets and batch size=64 gives good accuracy for CIFAR but not MNIST.

4. Batch Size = 128

Results-



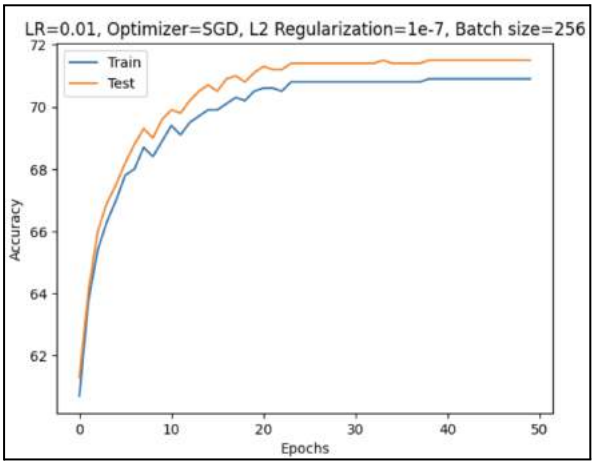


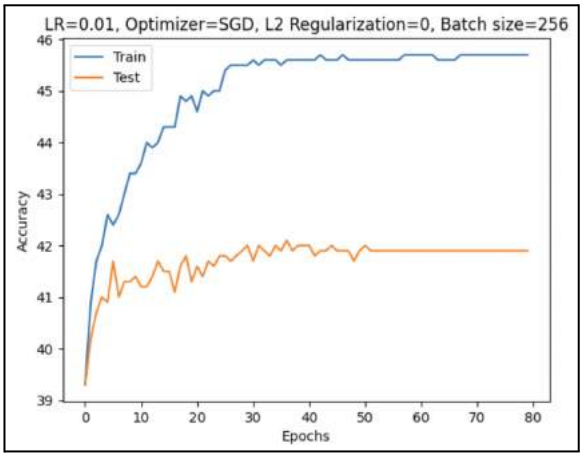
MNIST CIFAR10

Inference-
The results are similar to the previous batch size but less noisier.

5. Batch Size = 256

Results-



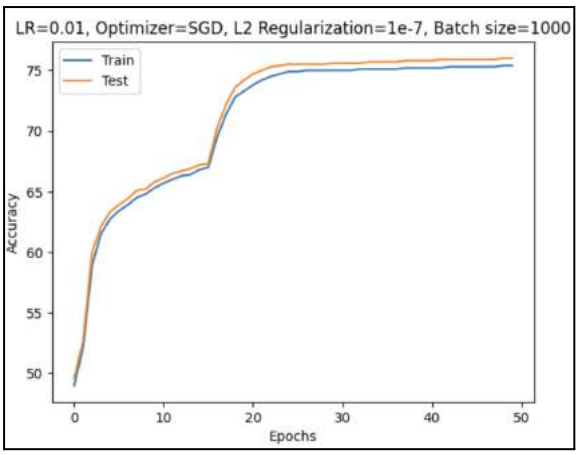


MNIST CIFAR10

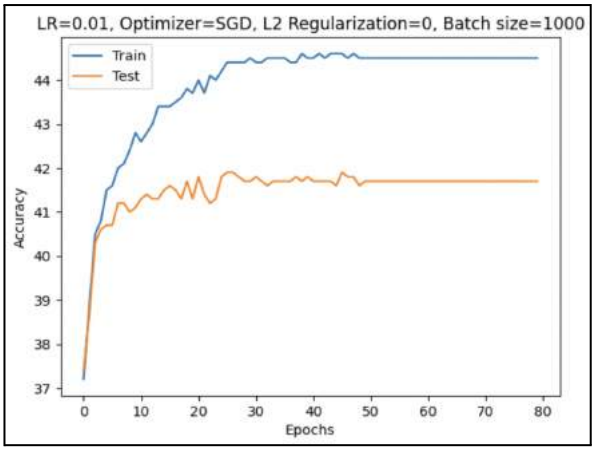
Inference-
The gradients have become even smoother. In MNIST dataset, overfitting has reduced with a slight drop in accuracy. Results are same as the previous batch size for CIFAR daatset.

6. Batch Size = 1000

Results-



MNIST



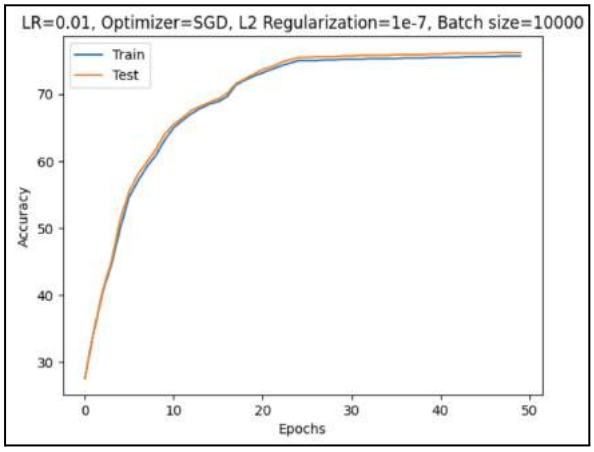
CIFAR10

Inference-

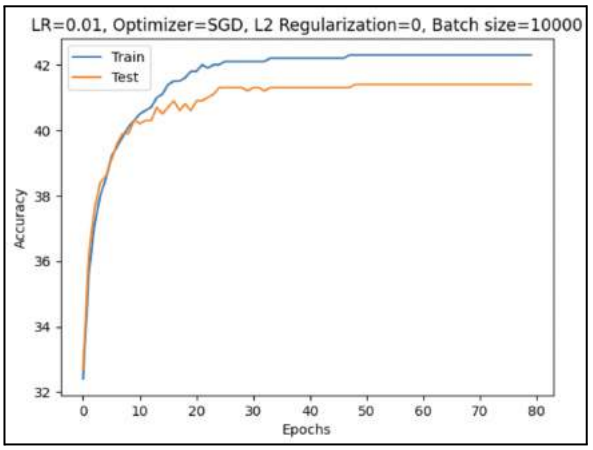
Overfitting has reduced a lot in both datastes and gradients have become more smooth but with a drop in accuracy.

7. Batch Size = 10000

Results-



MNIST



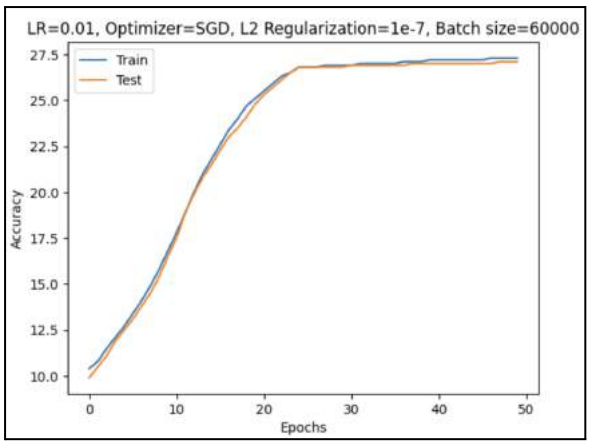
CIFAR10

Inference-

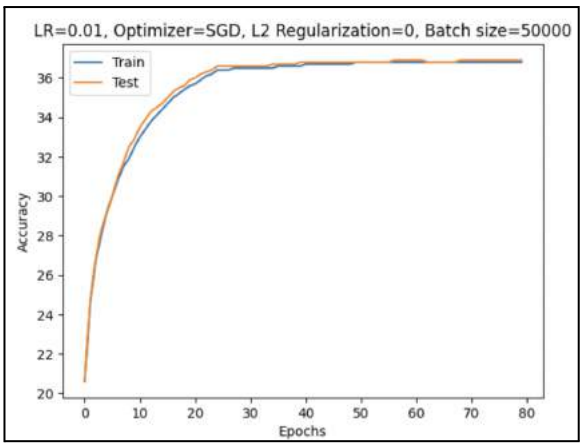
Overfitting has reduced a lot in both datastes, almost zero in MNIST and gradients have become even smoother but with a high drop in accuracy.

8. Batch Size = 60000

Results-



MNIST

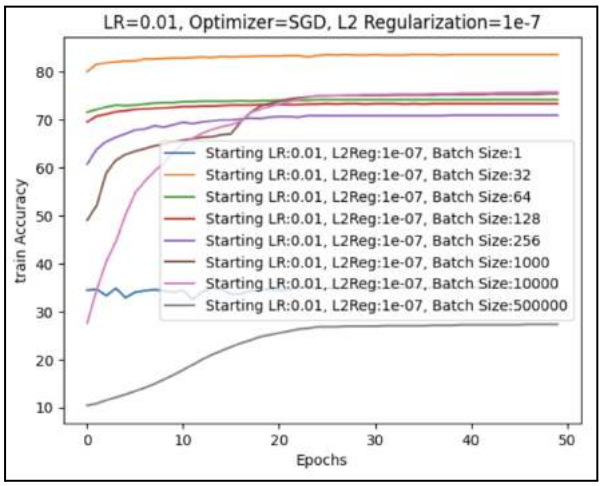


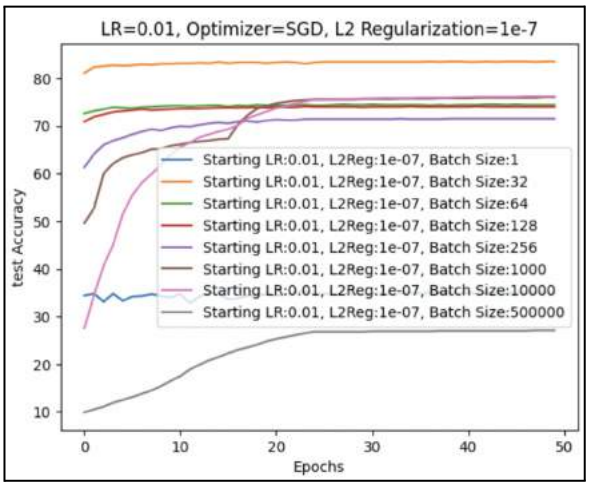
CIFAR10

Inference-
Overfitting has reduced a lot in both datastes, almost zero in both of them and gradients have become very smooth but with very low accuracy. Convergence is slower.

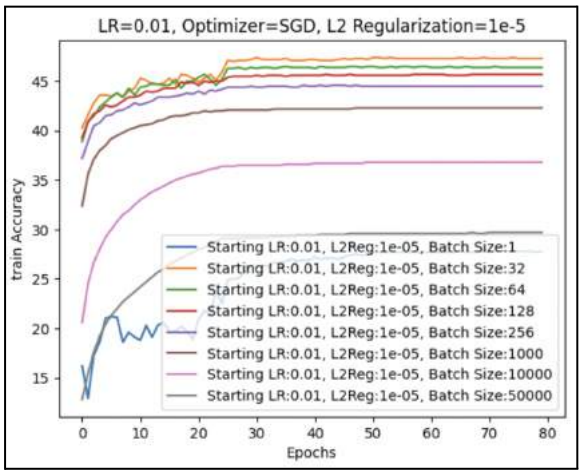
Best Batch Size-

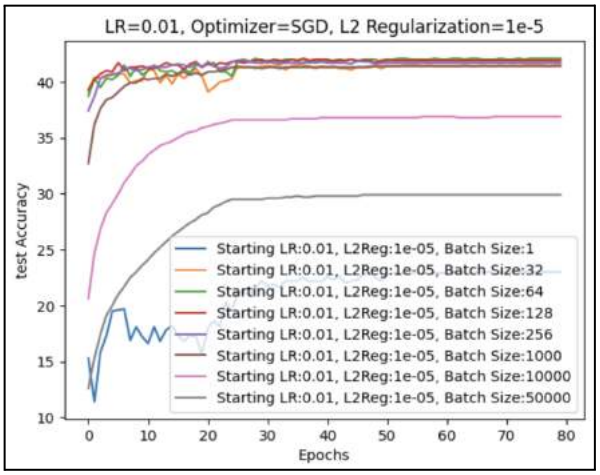
MNIST





CIFAR10





Conclusion-

As we can see in the graphs, the best choice for batch size=128
As batch size increases,

- the gradients become smoother
- Training becomes computationally heavy
- Convergence is slower

Neither too small not too big batch sizes are good for training.

Optimizers

Optimizers are algorithms or methods used to update the parameters (weights and biases) of the model during training in order to minimize a chosen loss function.

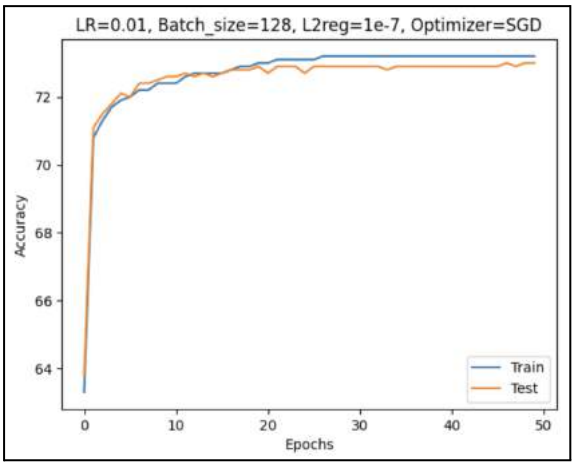
HyperParameters Used-

```
batch_size=128
num_epochs=100
LR=0.01
scheduler = StepLR(optimizer, 25, gamma=0.1)
loss_fn = nn.CrossEntropyLoss()
l2reg=1e-7
optimizer = t.optim.SGD(net.parameters(), lr=LR, weight_decay=l2reg)
```

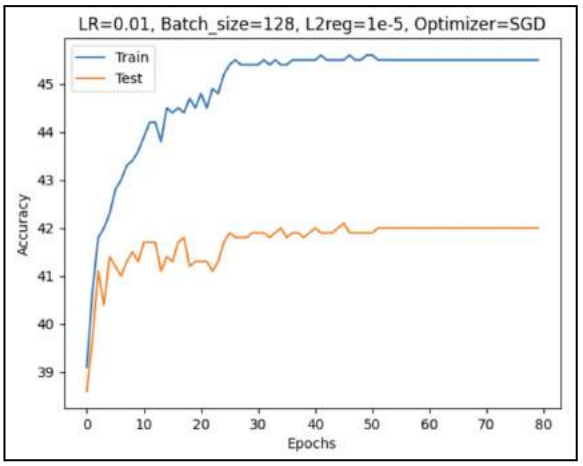
Various Optimizers-

1. SGD- It updates model parameters by computing gradients of the loss with respect to each parameter and adjusting the parameters in the direction that reduces the loss.

Results-



MNIST



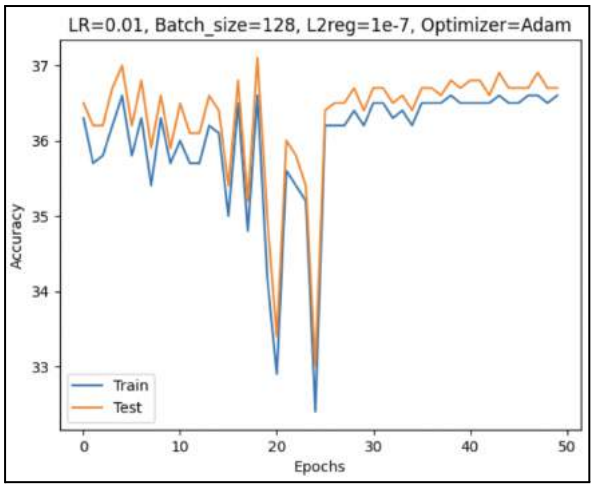
CIFAR10

Inference-

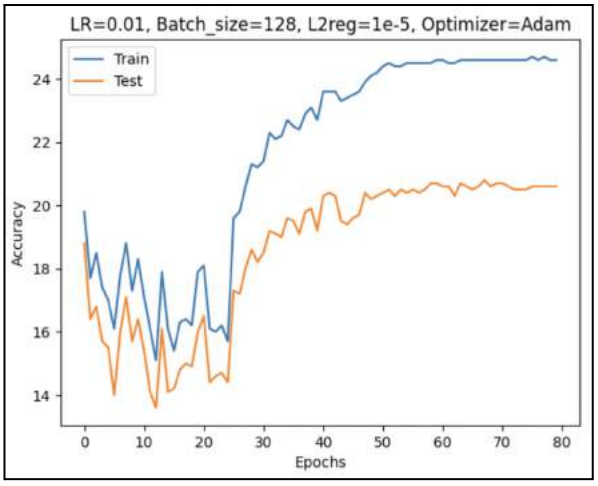
We can see that SGD converges to a stable value very fast, around after 20 epochs with good accuracy. Also, the updates are quite smooth.

2. Adam - It maintains moving averages of both gradients and squared gradients. It is one of the most popular and widely used optimizers in deep learning due to its good performance and adaptive learning rate properties.

Results-



MNIST



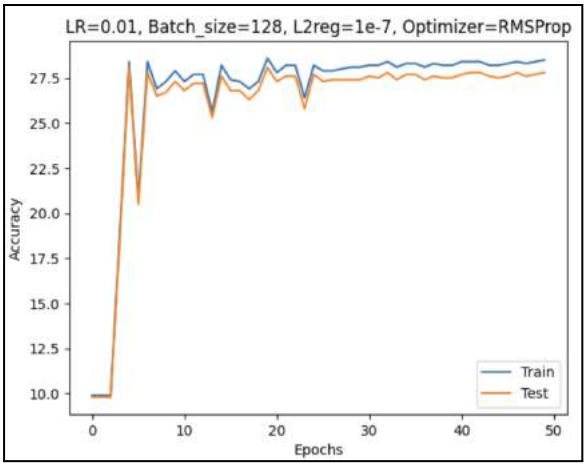
CIFAR10

Inference-
We can see that Adam optimizer solves the overfitting problem but the accuracy is lesser than SGD and updates are noisier.

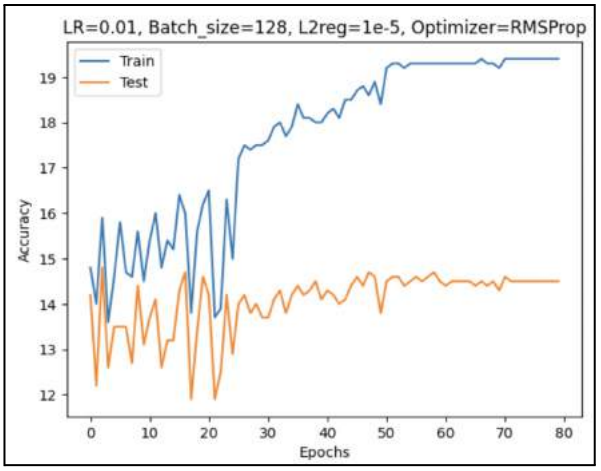
3. RMSProp

RMSprop is another adaptive learning rate optimizer. It addresses the vanishing learning rate problem using a moving average of squared gradients.

Results-



MNIST

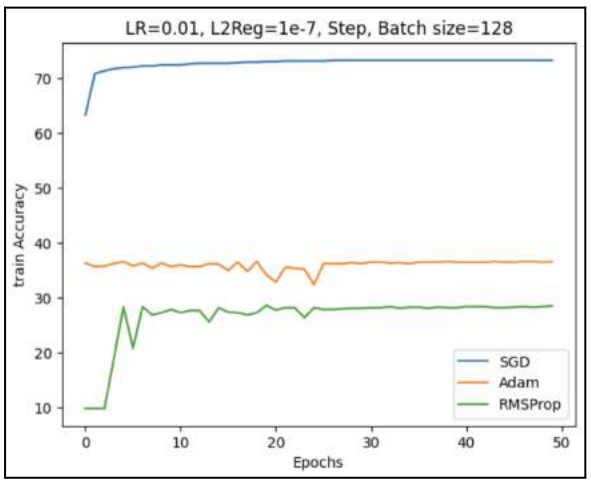


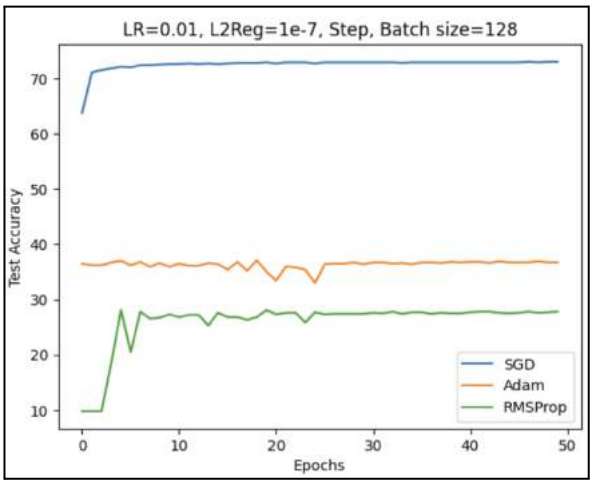
CIFAR10

Inference-
We can see that RMSProp leads to even lesser accuracy and is not a good choice in this scenario though it converges faster with noisy updates.

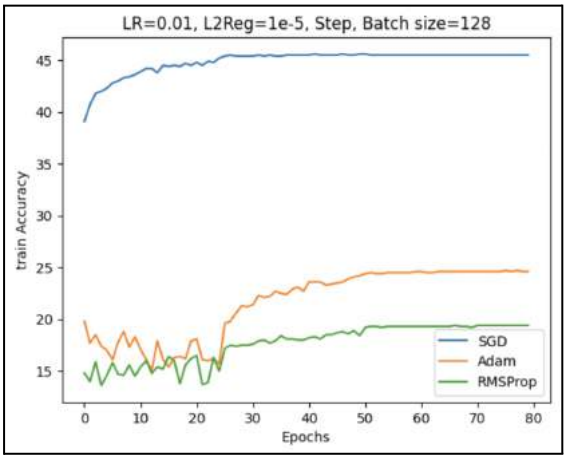
Best Optimizer-

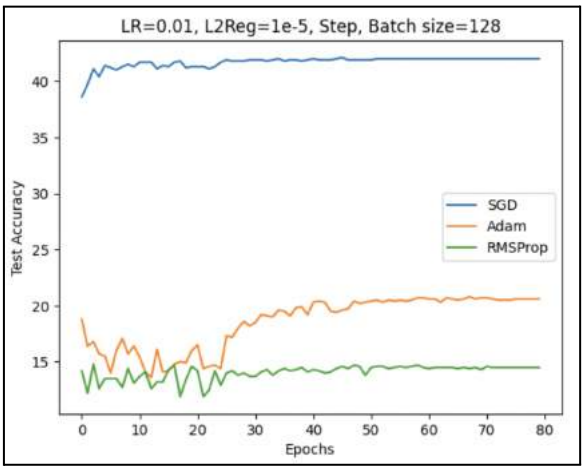
MNIST





CIFAR10





Conclusion-

As we can see in the graphs, the best optimizer is SGD as it gives the best accuracy without overfitting.

Q3.

Predicting sales revenue

3.1. Data Preprocessing

After loading the data from the CSV file, we encode the categorical feature X3 into three features X3_A, X3_B and X3_C using one hot encoding.

3.2. Exploratory Data Analysis (EDA)

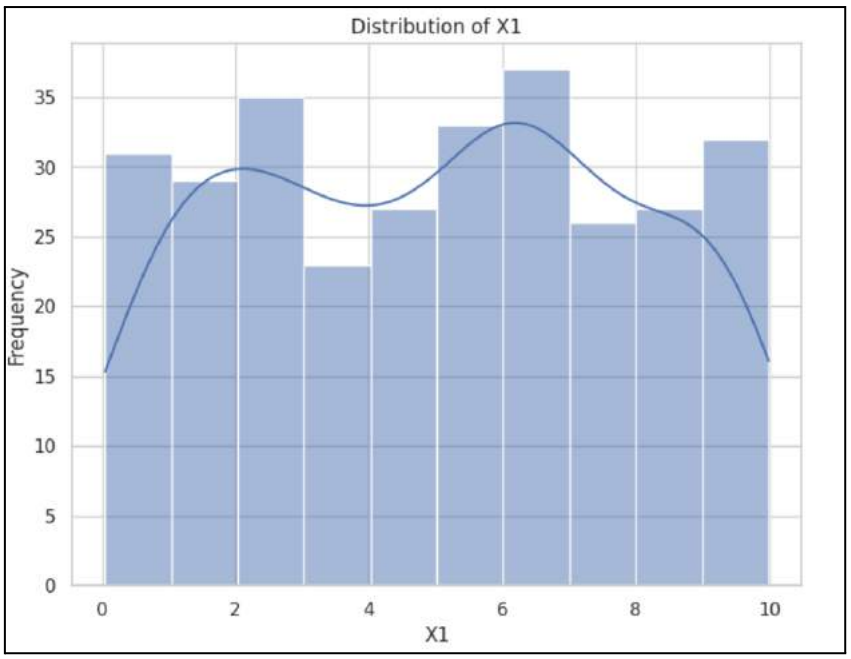
	X1	X2	y
count	300.000000	300.000000	300.000000
mean	5.035025	0.516667	16.201970
std	2.904740	0.500557	9.217629
min	0.046955	0.000000	-3.924092
25%	2.476713	0.000000	8.048621
50%	5.225482	1.000000	16.214571
75%	7.303055	1.000000	23.707521
max	9.988470	1.000000	34.671262

Features of the numerical columns

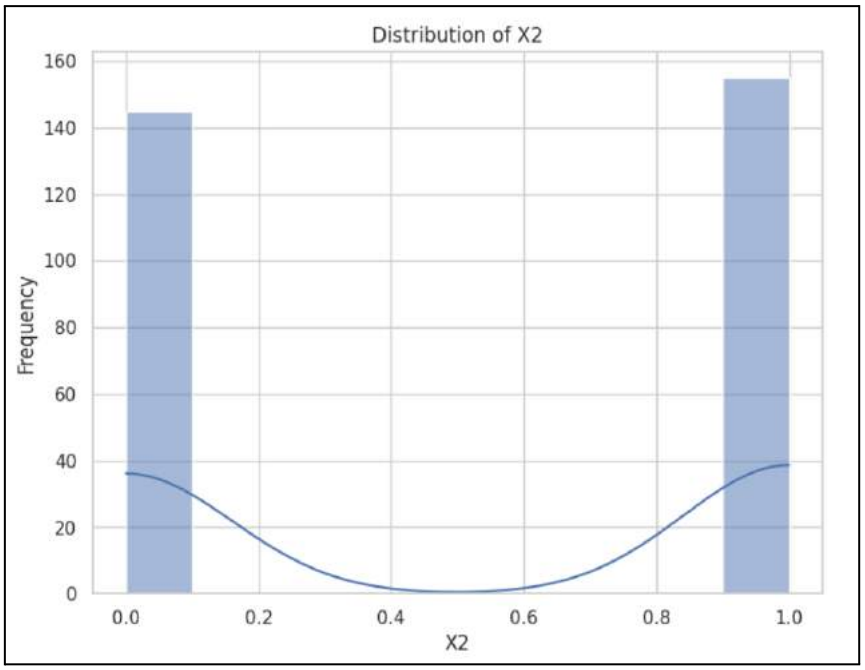
	X3
count	300
unique	3
top	A
freq	105

Features of the categorical column

We used the IQR method to count the number of outliers in the numerical data and got the following results.
Number of outliers in X1 = 0
Number of outliers in Y = 0

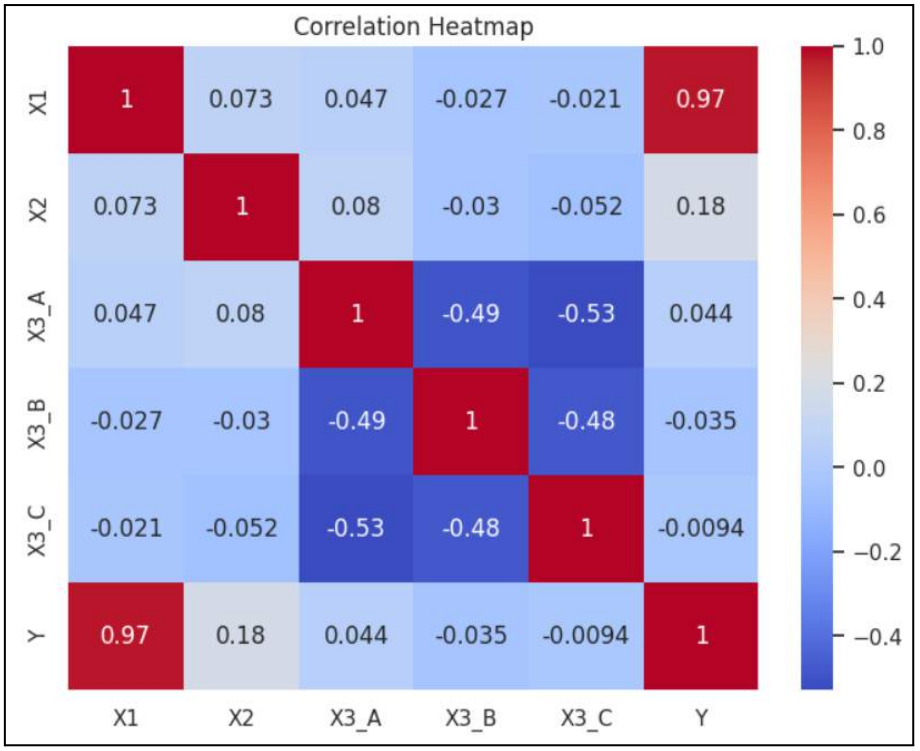


Histogram showing distribution of X1



Histogram showing distribution of X2

3.3. Feature Selection



Correlation Heatmap

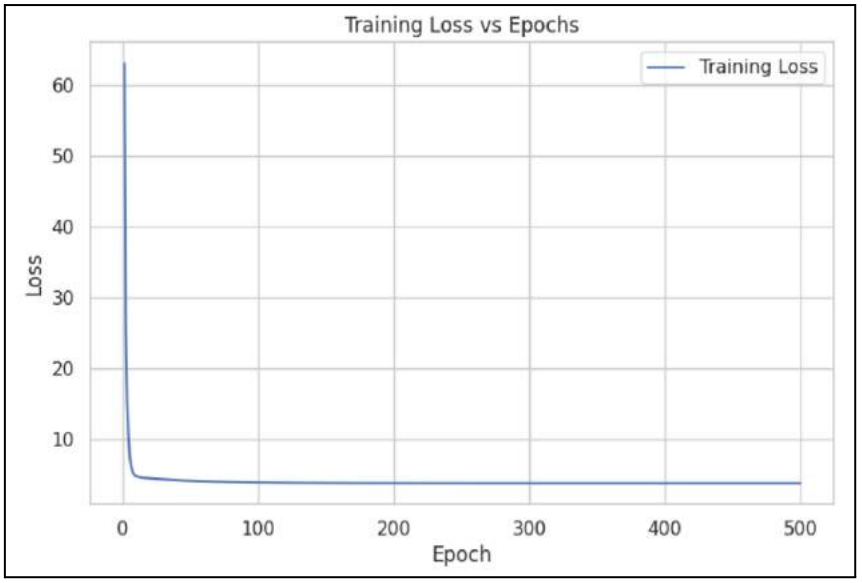
Clearly, numerical feature X1 is the one most highly correlated with the output Y with an autocorrelation of 0.97.
This is followed by categorical feature X2 with an autocorrelation of 0.18
The feature X3 is the feature least strongly correlated with the output Y

So, while training the linear regression model, we can drop the feature X3.
In our submission, we shall train the model twice, once including X3, and once without it. The model including feature X3 shall be referred to as M1, and the one without it as M2.

3.4 Linear Regression Model

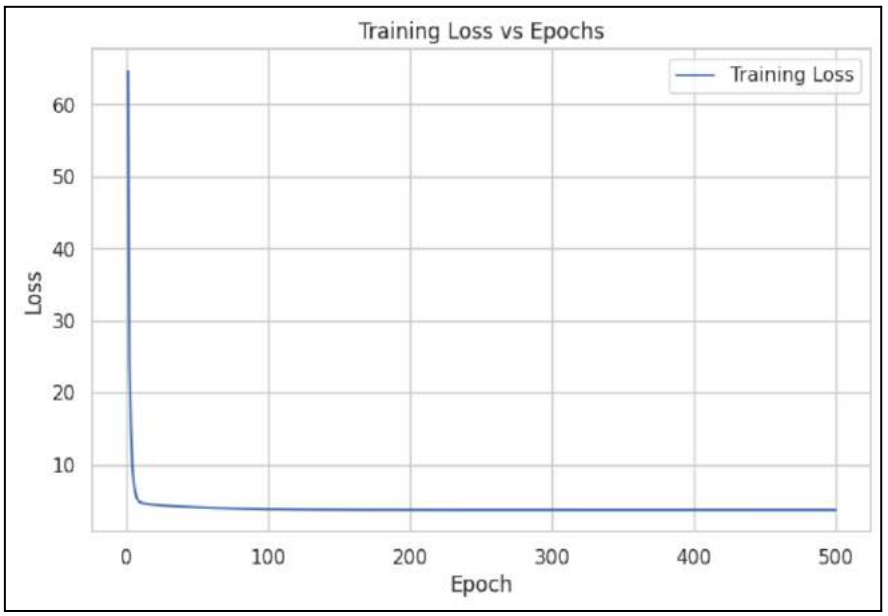
3.4.1. Model M1

Learning Rate = 0.01
Loss function criterion used: Mean Squared Error (MSE)
Number of epochs run for = 500
Optimizer: Stochastic Gradient Descent (SGD)



Training set loss after 500 epochs = 3.7589

3.4.2 Model M2
Learning Rate = 0.01
Loss function criterion used: Mean Squared Error (MSE)
Number of epochs run for = 500
Optimizer: Stochastic Gradient Descent (SGD)

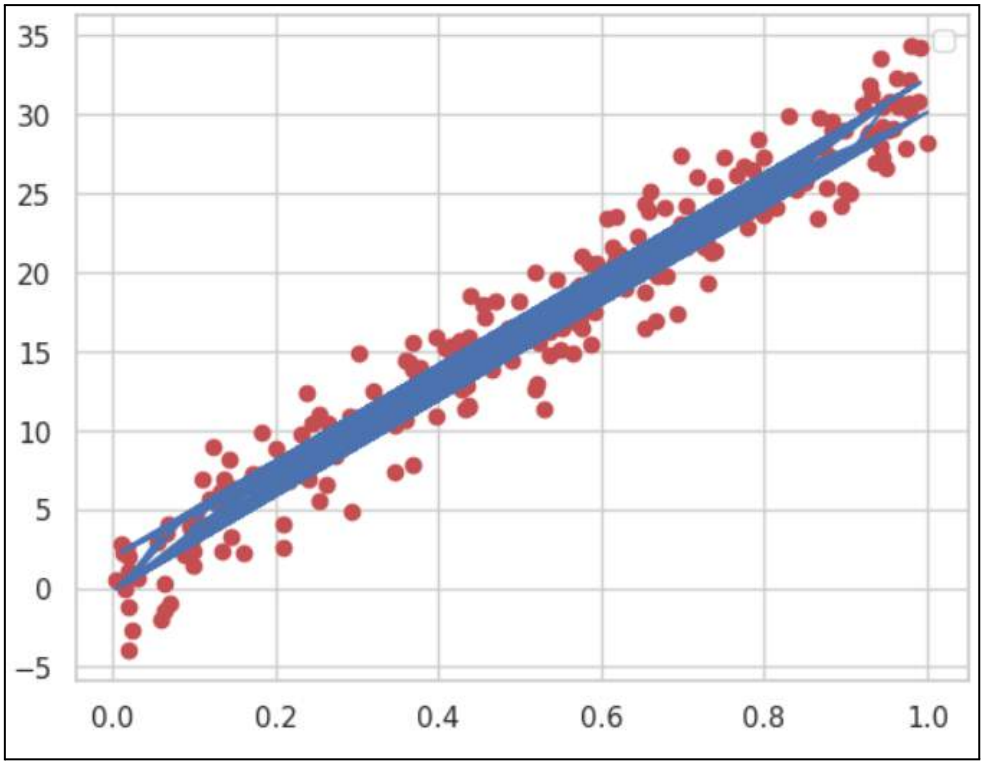


Training set loss after 500 epochs = 3.7482

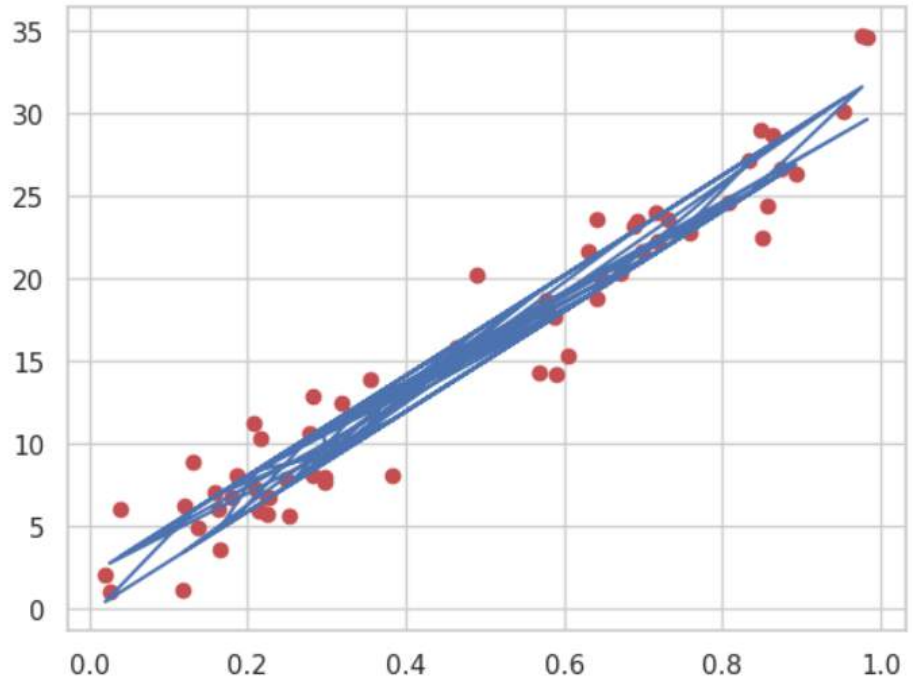
3.5 Model Evaluation

Loss function	M1	M2

Mean Squared Error (MSE)	4.5026	4.4313
Mean Absolute Error (MAE)	1.7147	1.7281
R-squared error	0.9428	0.9437



Red: Target variable values in the training set
Blue: Target variable values produced by the model M2 on the training set



Red: Target variable values in the test set
Blue: Target variable values produced by the model M2 on the test set

3.6. Model Interpretation

3.6.1. Model M1

Weights of the parameters are given as follows

Feature	Weight
X1	30.318
X2	21.543
X3_A	-0.0848
X3_B	0.01636
X3_C	0.3689

As expected, the model depends the most heavily on X1 followed by X2.
It is almost independent of X3

3.6.2 Model M2

Feature	Weight
X1	30.3257
X2	21.5756

Thus, even after removing the feature X3, we observe that the weights of the features X1 and X2 remain unchanged.

3.7. Testing new scenarios to make predictions

We used the model M2 for making new predictions since it appeared to be more accurate. In order to do so, we first prepared a random collection of data points with features given as

X1	X2	X3
7.46	0	C
8.42	1	B
3.57	0	A

2.29	0	A
9.18	1	B

We then loaded the model, normalised the data and used the model to make predictions. The predictions obtained were as follows

X1	**X2**	**X3**	**Y**
7.46	0	C	22.4400
8.42	1	B	27.5150
3.57	0	A	10.6297
2.29	0	A	6.7435
9.18	1	B	29.8224