# COL341 : MACHINE LEARNING (Spring 2023)

**Name:** Vatsal Jingar
**Entry Number:** 2020CS50449

**Date:** April 29, 2023
**Assignment:** 4

NEURAL NETWORKS ASSIGNMENT

## 1 Scratch Implementation of CNN (Using Numpy)

### 1.1 Implementation

```
For implementing 4 dimension nested loops, I have used a function np.einsum().
This function inproved the efficieny from 1 min 10 sec per 32 samples to
7 seconds for 10 epochs.
Time taken for Training : 1216 min
I have created my architecture in strcuture format.
To implement any layer, user would have to call the forward layer methods and
backward layer methods.
All the CNN backpropagations are similar to Fully connected Layer except there
was convolution operation involve rather than multiplication.
To implement CNN back propagation with respect to previous input : We have
reduced our differentiated expressions in the form of convolution operations.
```

### 1.2 Analysis

**Observations :**

- As it is visible that training loss is decreasing rapidly while the validation loss is almost saturated. Validation accuracy for this model has also saturated after 10 epochs.
- To compare the training and validation loss trends in one graph I have normalized the data with number of samples passed.

```
Overall Validation Accuracy : 61.041%
```

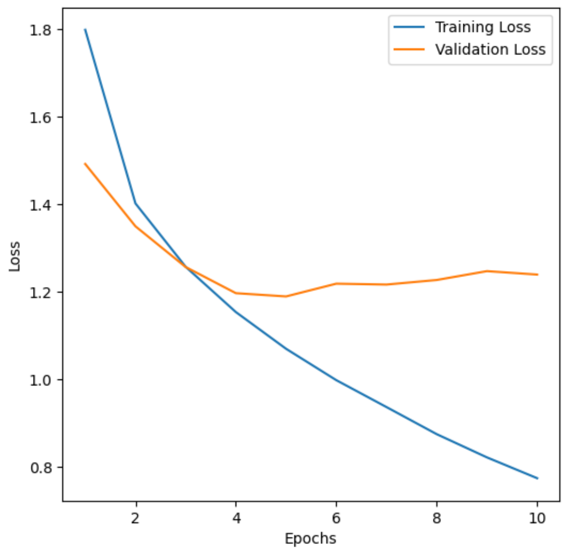

**Figure 1:** Validation and Training Loss

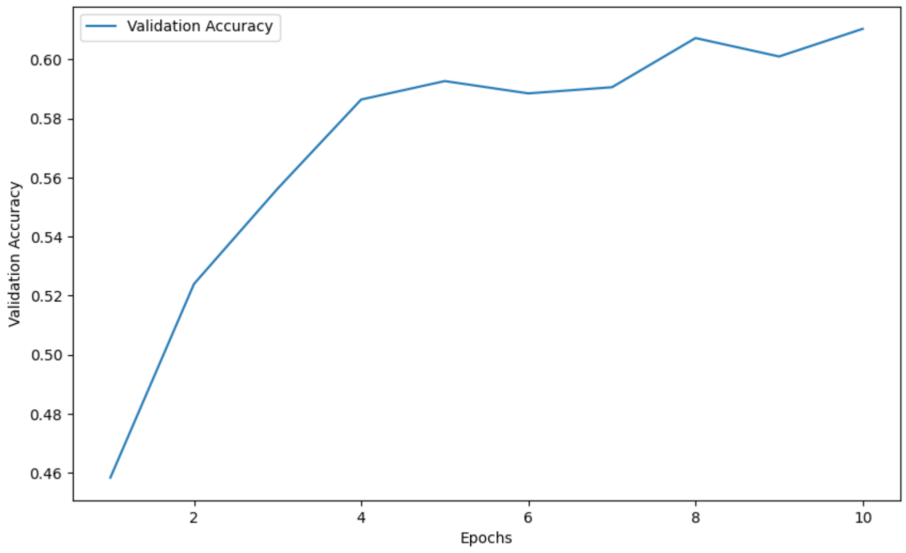

**Figure 2:** Validation Accuracy

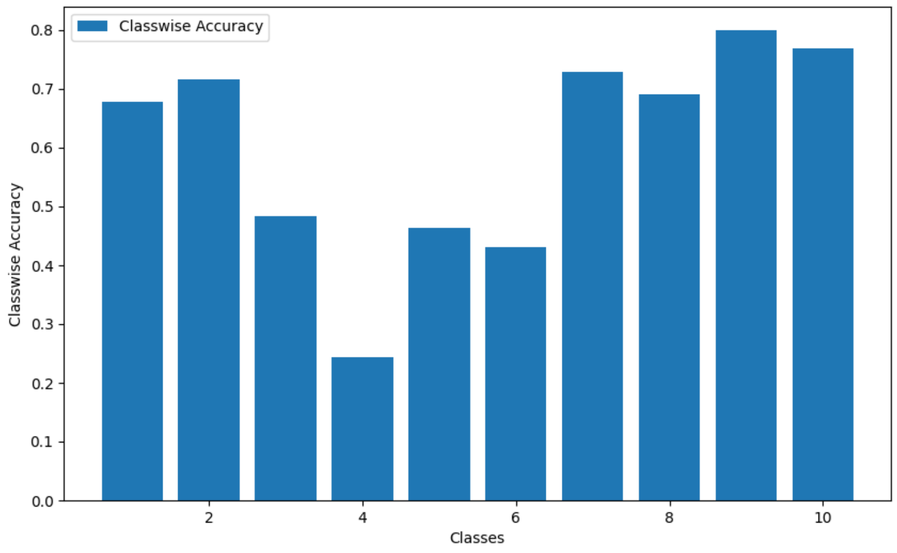

**Figure 3:** Classwise Accuracy

# 2 Pytorch Implementation of CNN:

## 2.1 Hyperparameter Tuning (with Data Augmentation):

<u>USING OPTIMIZER = ADAM</u>

### 2.1.1 Learning Rate (fixed):

Learning rate = 0.1

```
LR =0.1, epochs = 20
Val acc = tensor(0.1030)
```

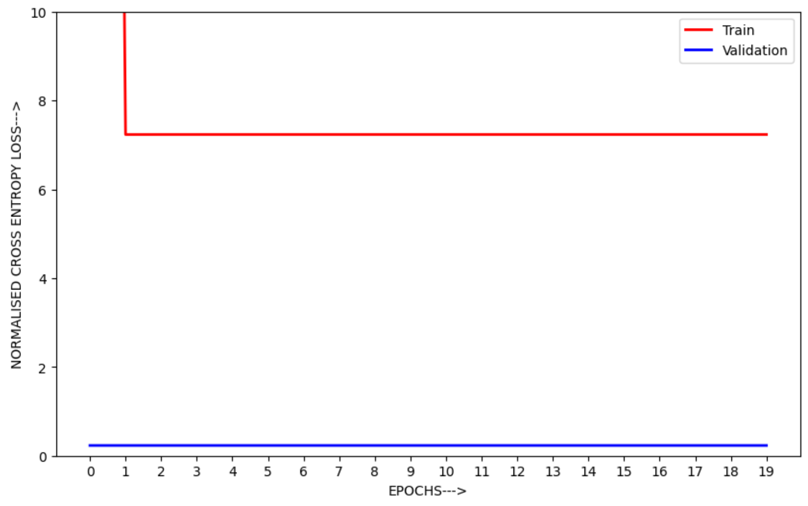

**Figure 4:** Loss curves with lr = 0.1

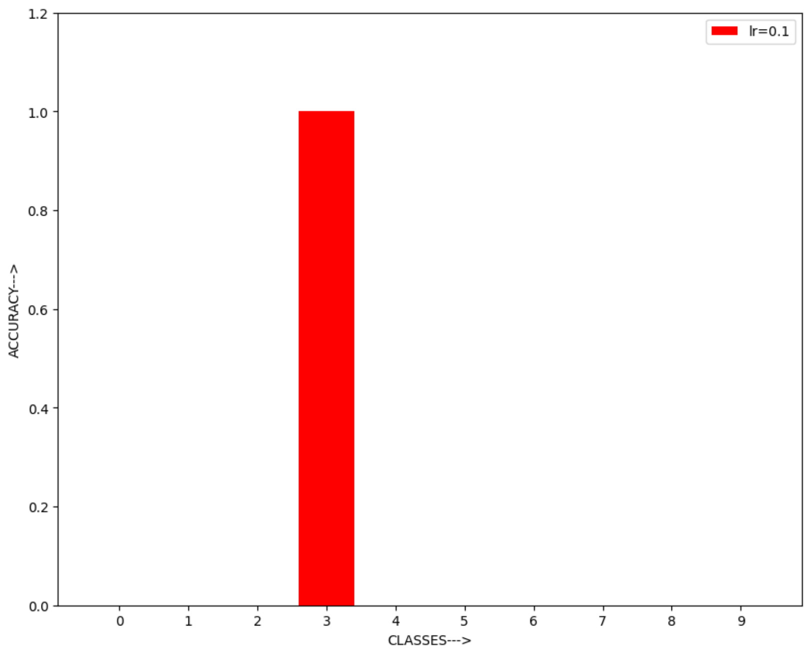

**Figure 5:** Classwise accuracy curves with lr = 0.1

Learning rate = 0.001

```
LR = 0.001, epochs = 40
Val acc = tensor(0.6950)
```

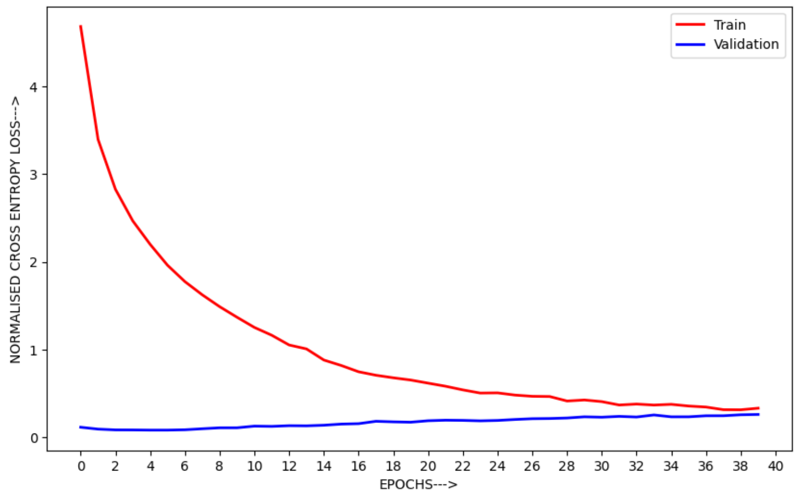

**Figure 6:** Loss curves with lr = 0.001

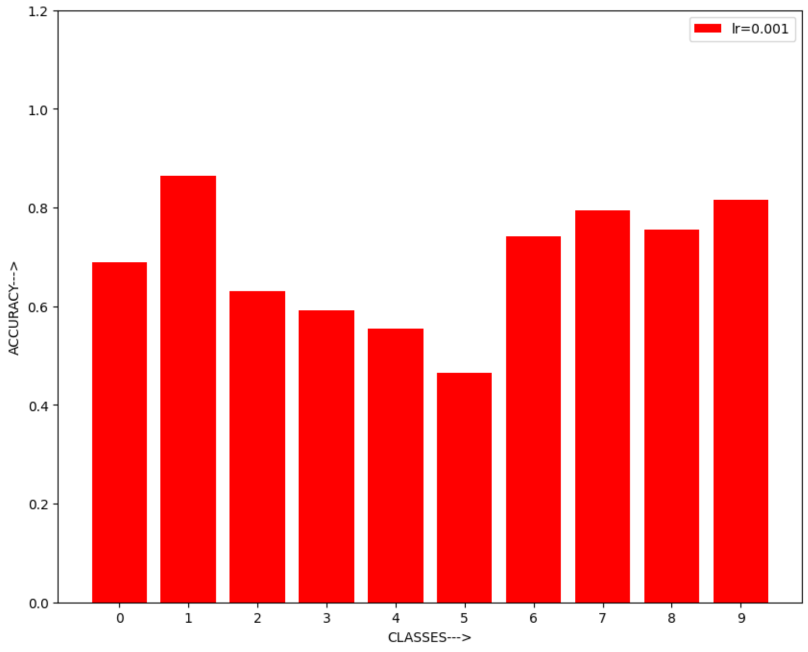

**Figure 7:** Classwise accuracy curves with lr = 0.001

## Learning rate = 0.00001

```
LR = 0.00001, epochs = 15
Val acc = tensor(0.4890)
```

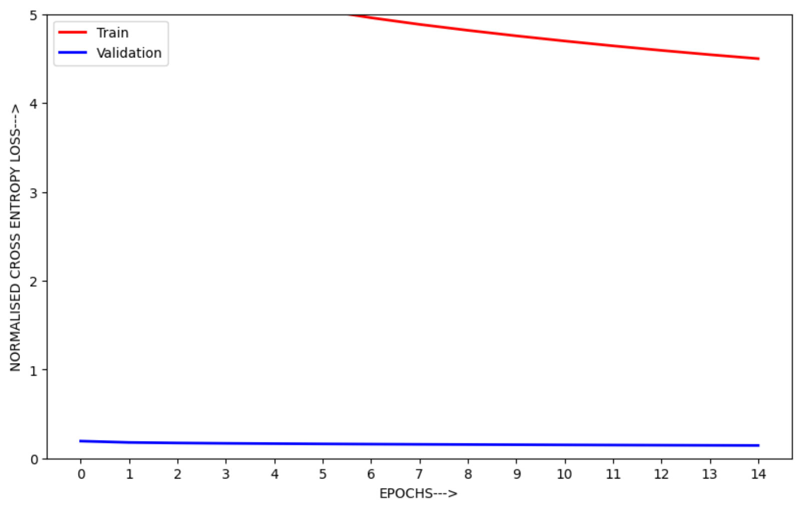

**Figure 8:** Loss curves with lr = 0.00001

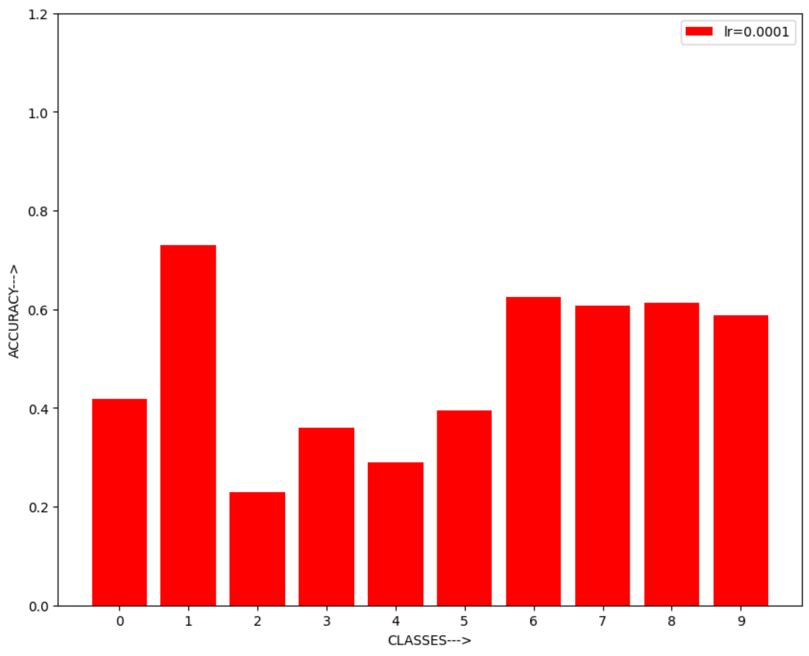

**Figure 9:** Classwise accuracy curves with lr = 0.00001

## Comparision among learning rates:

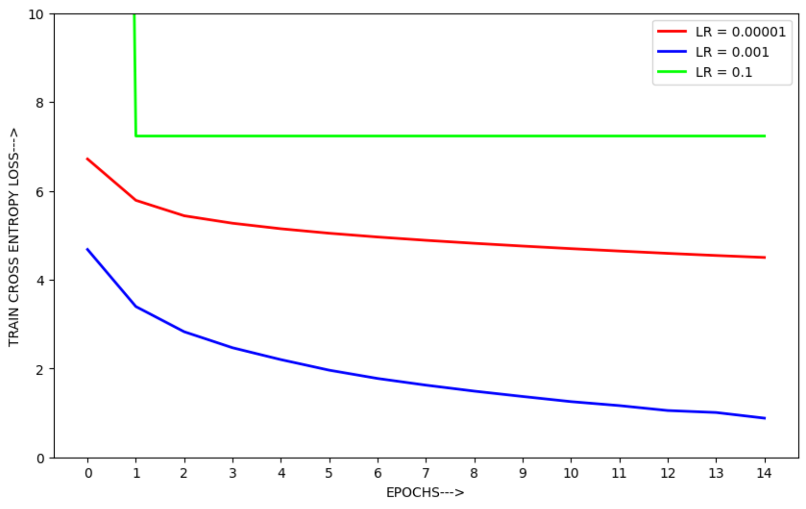

**Figure 10:** Training Loss curves convergence

**Observations :** Talking about the cross entropy loss, it is clear from plot 10, that when the learning rate was really high, the loss was not decreasing. It was looping around a local minima. Also when we choose the learning rate very small, the loss was converging really slow. This is because very small steps are taken now. And it is taking large number of epoch to converge. For a perfect learning rate for this data set which is 0.01 the loss is converging smoothly.

### 2.1.2 LR Scheduler:

```
EXPONENTIAL
LR 0.02, GAMMA = 0.99
Val acc = tensor(0.6540)
```

**LR Scheduler with exponential variation starting from and gamma =**

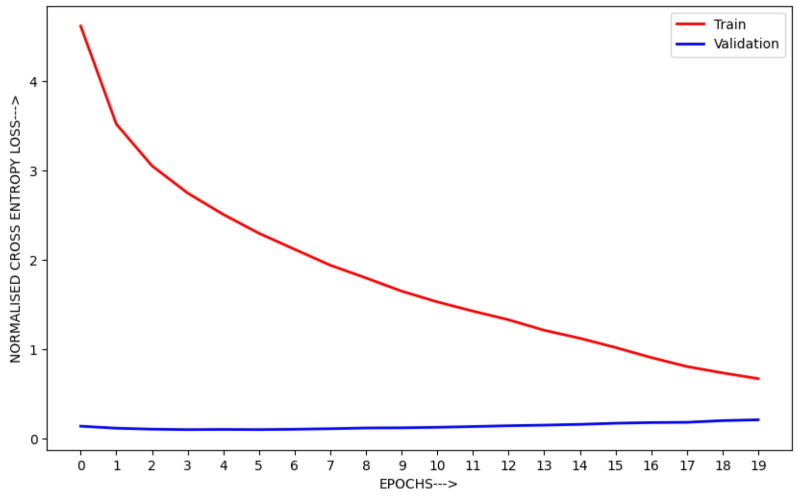

**Figure 11:** Loss curves with variable LR

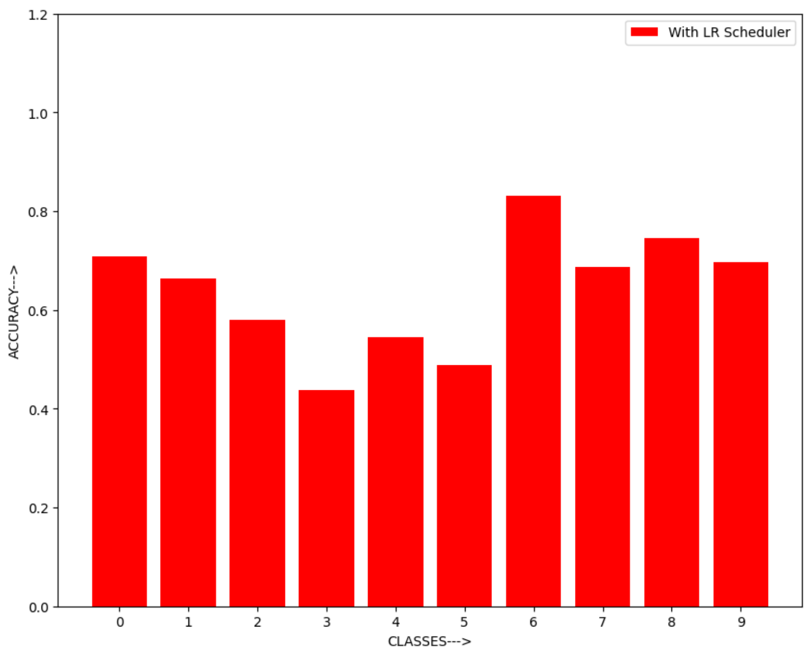

**Figure 12:** Classwise accuracy curves with variable LR

**Comparision with fixed LR:**

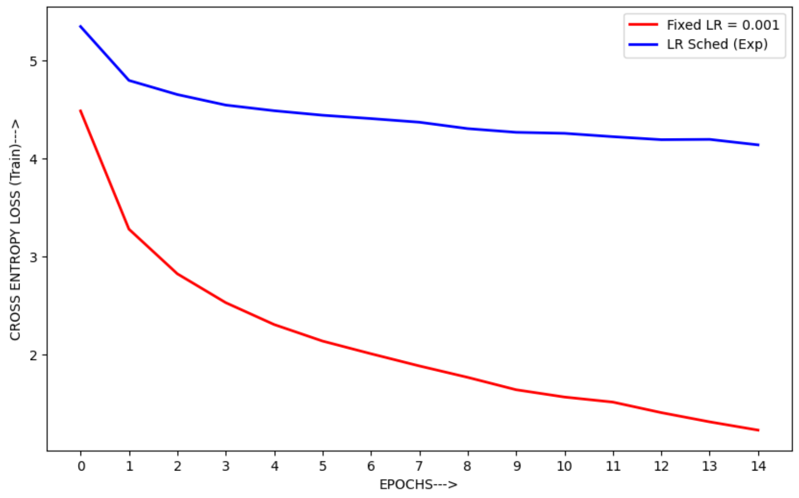

**Figure 13:** Loss curves on Train data with variable LR

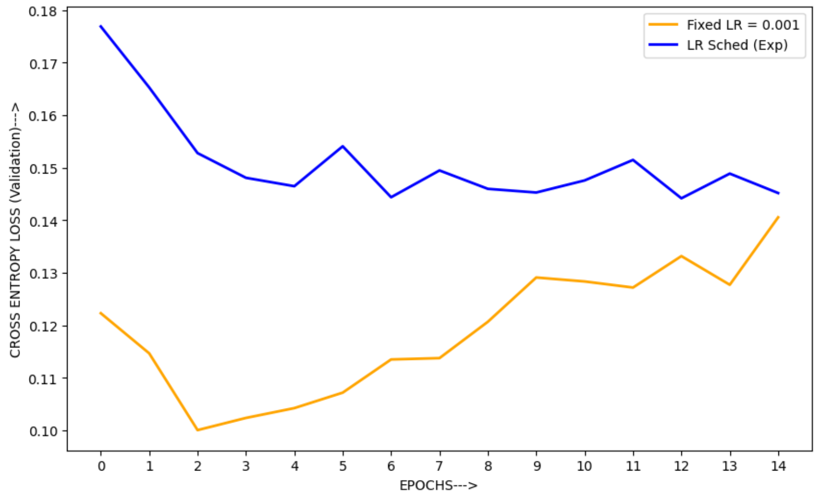

**Figure 14:** Loss curves on Validation data with variable LR

**Observations :** LR scheduler is using a decay function to decrease the value of learning rate with increase of epochs. It is expected that as the epochs will increase the loss convergence in LR scheduler becomes slower. From the plot 13, it is visible that both started with the same pace, but the slope of LR scheduler is decreasing and hence the convergence is slowing down. By training it on around 50 epochs, it could hardly able to reach the loss of Fixed LR. But if used with appropriate settings, the loss would able to converge more after a very large number of iterations than fixed LR.
Also using LR sched, the validation loss is decreasing while for Fixed LR it is increasing. This is because if we will decrease the LR slowly, then for the iterations after a ceratin time, the learning rate will be very low. This will led to decrease in high change in weights and bias by the effect of some particular data points. Hence it is generalizing more better.

### 2.1.3 Number of Epochs:

```
LR = 0.001, epochs = 40
Val acc = tensor(0.6950)
```

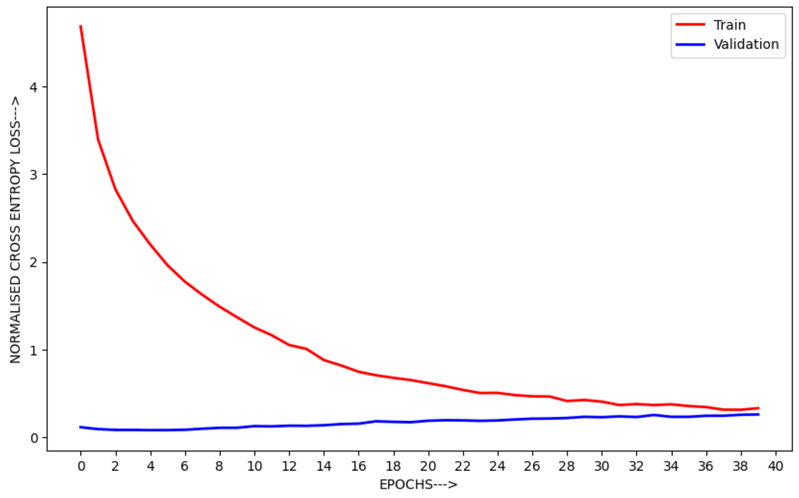

**Figure 15:** Loss curves with 40 epochs

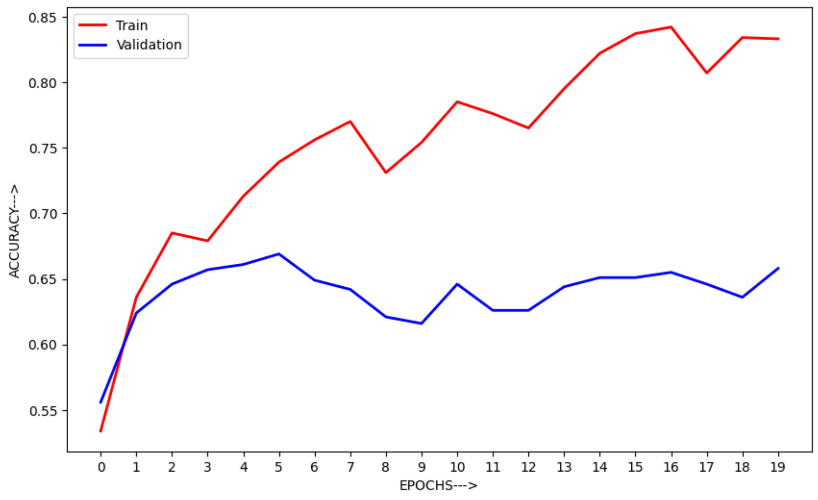

**Figure 16:** Accuracy curves with 20 epochs

**Observations(Does increasing the epochs decreases the loss)** : For the training loss, it could be confirmed only on the condition that the gradient never enters in a local optimal. Apart from this, if the learning rate would be high then it would oscillate and it could happen that the loss could increase.
For the validation loss, it is tougher to guarantee, because for some values of weights and biases it could be more generalized than others.
If the learning rate could be chosen appropriately, then the loss may always converge.

### 2.1.4 Varying Batch size:

```
Validation Accuracies :
    Batch 32 : 0.6515
    Batch 16 : 0.6400
    Batch 8  : 0.6105
    Batch 4  : 0.6375
```

**Accuracy curves with varying batch size = 4,8,16,32:**

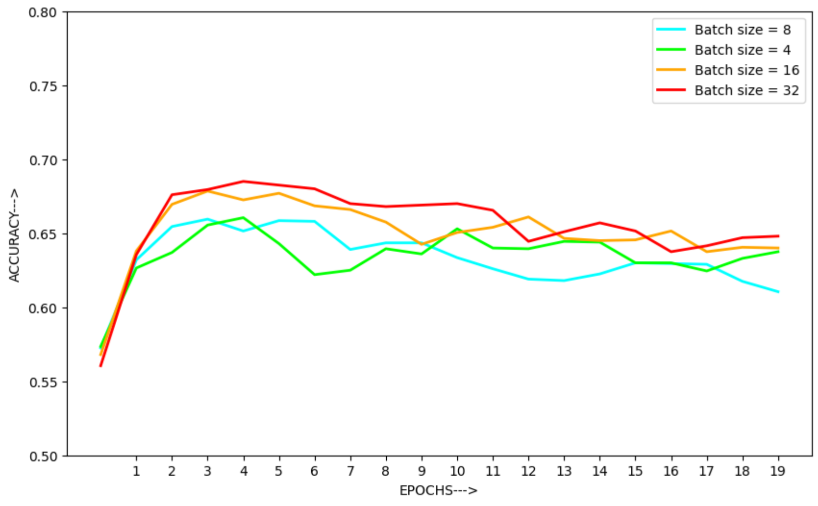

**Figure 17:** Accuracy curves on Validation data for different batch sizes

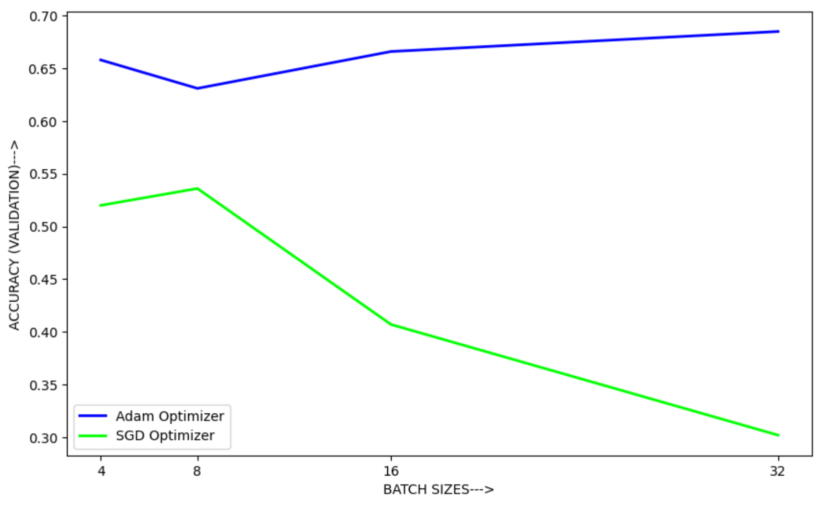

**Figure 18:** Accuracy curves on Validation data for different batch sizes

## EFFECT OF SGD OPTIMIZER:

### LR = 0.001, EPOCHS = 20

### Comparision of SGD and ADAM Optimizer

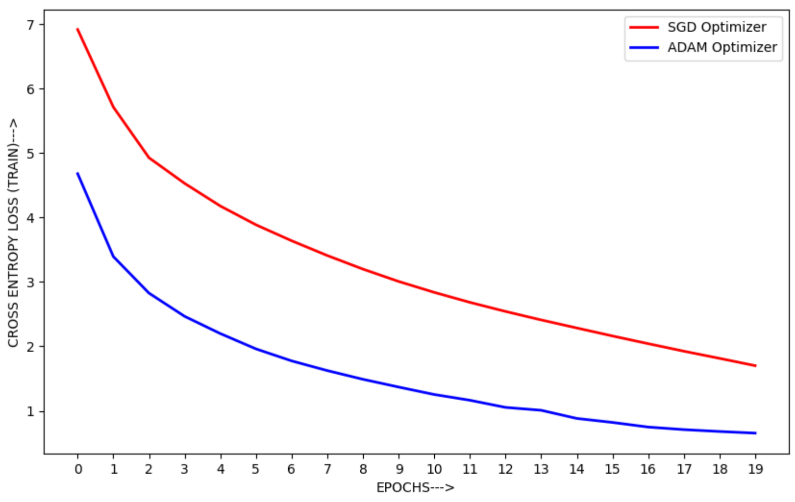

**Figure 19:** Train Loss curves for different Optimizers

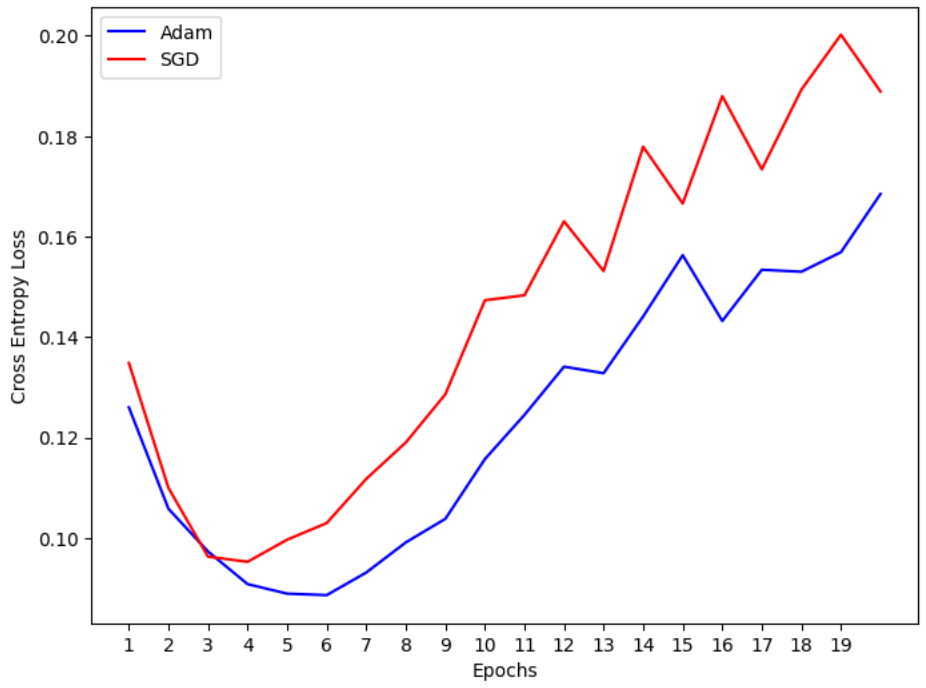

**Figure 20:** Validation Loss curves for different Optimizers

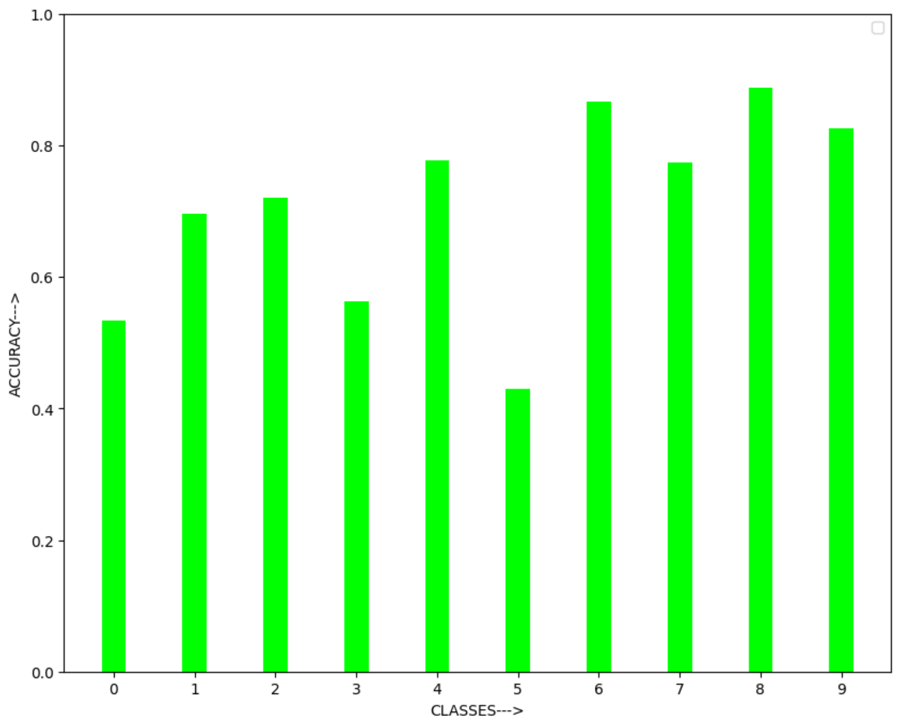

**Figure 21:** Class wise Validation accuracy

**Observations :** It is observable that the SGD optimizer is not converging faster. This is because Adam uses adaptive learning rate to converge faster. Also it avoids overfitting by balancing the learning rates where-ever it is required. So that the weights would be more changed when required and less change when we are around local optimals. Apart from this, Adam removes bias from samples. Hence perform better on generalized data also.

## 2.2 KL Divergence Loss Function:

```
LR = 0.001
Val accuracy: tensor(0.6643)
```

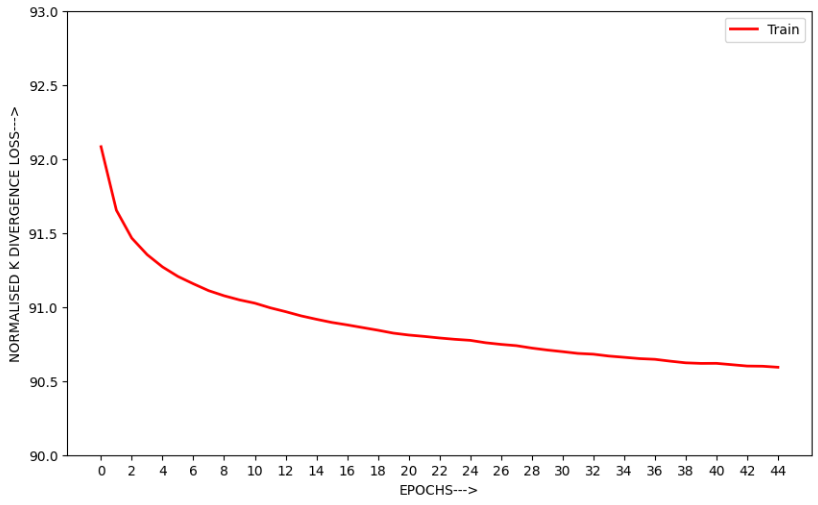

**Figure 22:** Train loss curves with KD loss function

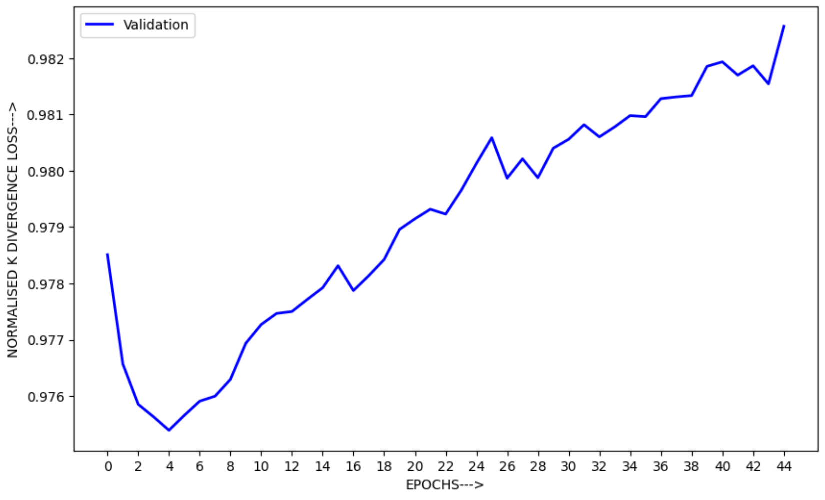

**Figure 23:** Validation loss curve with KD loss function

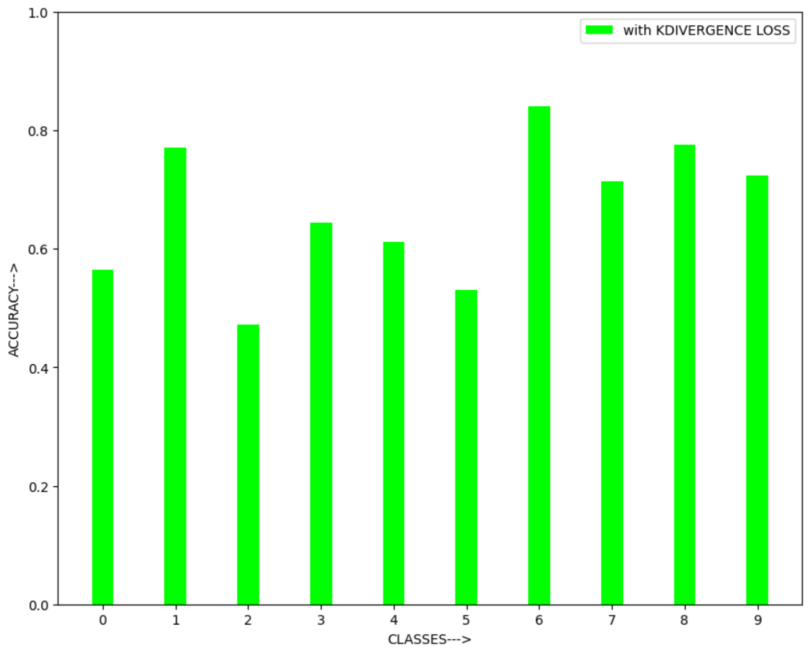

**Figure 24:** Classwise accuracy graph with KD Loss function

**Observations** : Cross Entropy loss in general provides more stable gradients. By including loss term, model will be penalized if it would be farther from our targetted distribution. We are increasing prior probability for target here. So hypothesis set will reduce and it is regularizing. Overall the loss function itself is using a penalizing term for overfitting. Hence it has higher validation accuracy and it is not overfitting.

## 2.3 Without Data Augmentation:

```
val acc = tensor(0.7030)
```

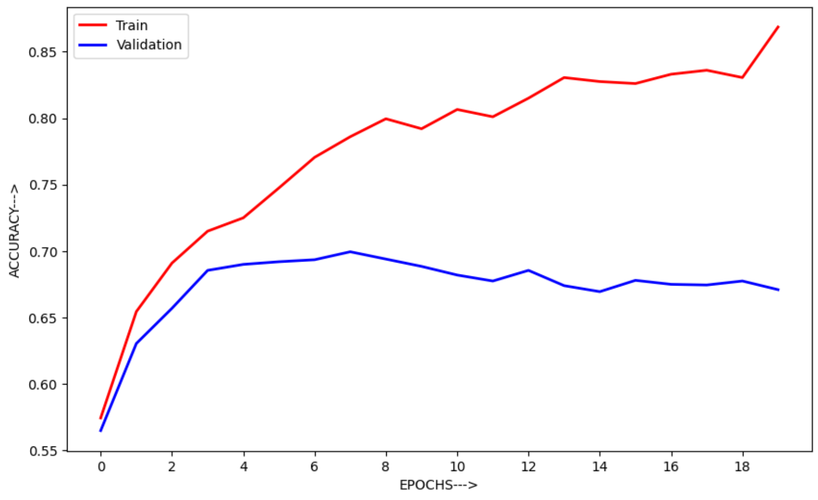

**Figure 25:** CAccuracy curves with DATA AUGMENTATION = OFF

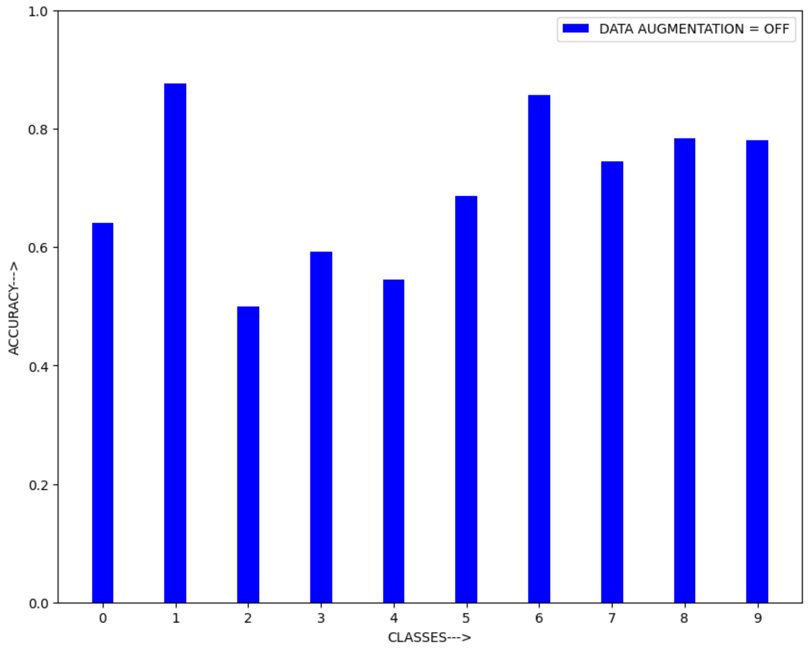

**Figure 26:** Classwise accuracy graph with Data augmentation = OFF

**Comparision with Data Augmentation = ON**

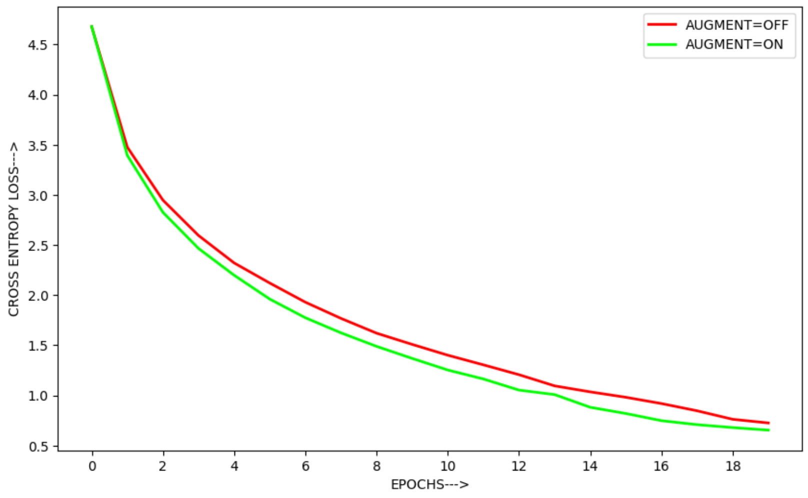

**Figure 27:** Loss curves (on Train data) with augmentation=on/off

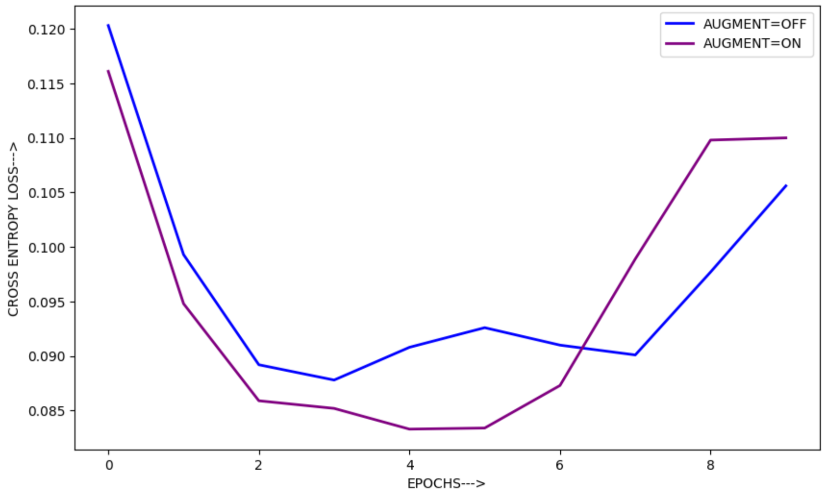

**Figure 28:** Loss curves (on Validation data) with augmentation=on/off

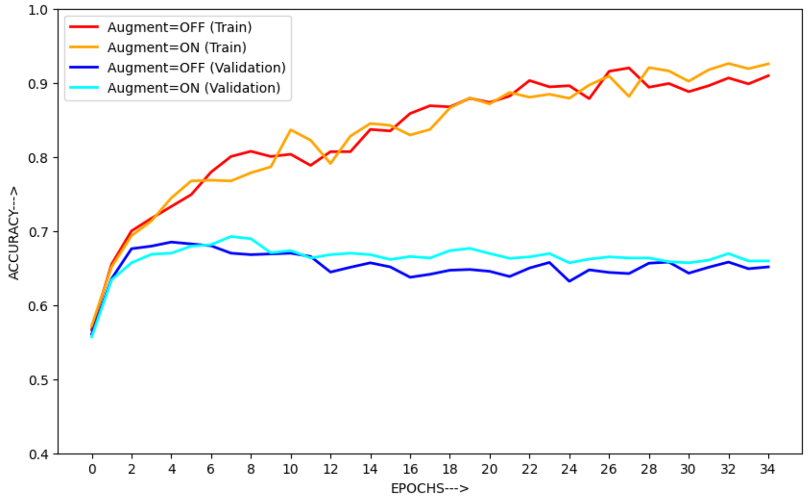

**Figure 29:** Accuracy curves with augmentation=on/off

**Observations :**

Due to variability in the positions and orientation of image, the network is actually learning the important features rather than over fitting it. Also as the input space is more complex in case of augmentation, training loss is converging some what slower than when it is off. From

the validation also, for certain epochs, and at early stops, model with augmentation on is performing better in case of generalized data.
Talking about class accuracies : In this case it is visible that the accuracy is more balanced class wise than in case when augmentation is off. Although not much difference is visible, augmentation is helping in performing better on generalized versions of every class. And hence validation accuracies are more balanced class wise when augmentation is on.

## 3 Improving the CNN Model (Bonus):

**Improved Strategy** :

- Added dropout layers
- Added batch normalization layer
- Increased number of channels for every convolutional layer.

**Why I choose this architecture choices :**

- As for all the above models the training accuracy was increasing with epochs but the validation accuracy was either saturated or started decreasing after some epochs. I thought to regularize the network using dropout. The dropout layer will deactivate some neurons with a particular probability p. This will help the architecture to learn the data by eradicating the noises. The dropout layer can be considered as choosing only some features for the prediction randomly.
- This learning will help in reducing complexity. So if there would some features which are specially contributing to the over complexities of model, then if we would train removing these features, we would get better output.
- Now after this I have also increased the channels for each convolution layer. If we consider a fully connected layer to be equivalent to a convolutional layer, channels could be considerd as edges of fully connected layer. When we increase the edges the input nodes and output nodes also increases. If we consider this layer as an feature map for the next layer, more nodes will help in attaining more complex functions. So we are increasing the potential of model or Hyptothesis set for choosing model. Simultaneously to tackle the overfitting, we will set dropout layers at some points.
- Apart from this, By researching on different articles, I found that there is a common problem which occurs during training of CNNs with input images. This problem is hardcoding of input with output, due to which Architecture becomes highly overfitted. For example, if there are two images of aeroplane but with different positions then if only one is used in train, the neural network will learn to predict aeroplane only when the aeroplane is in position for which it is trained. So to overcome this, we need model to be independent of these type of features. Batch normalization by using a specific way of normalization removes this dependency. Hence The model is more robust now.

- Overall my model has a overall validation accuracy of around 78.89 %. Although I could not get a high training accuracy for this.

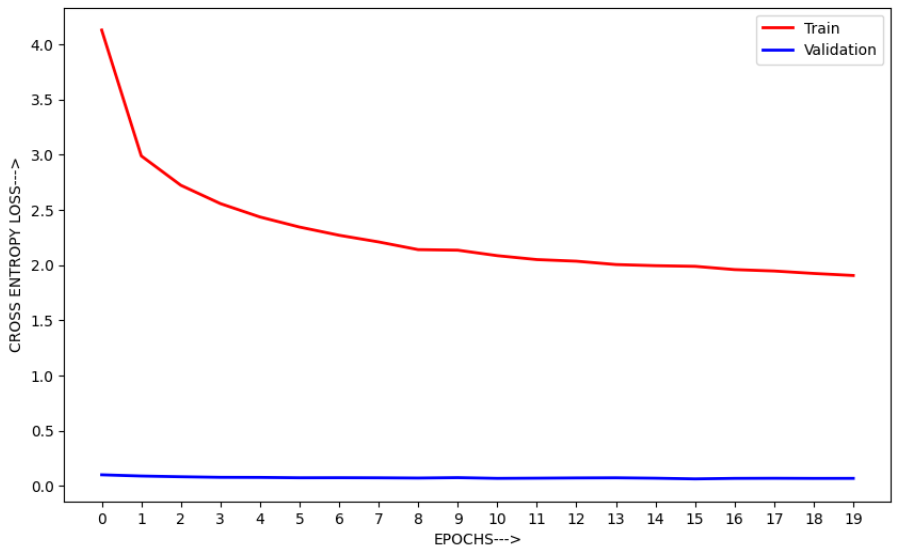

**Figure 30:** Training and Validation Loss

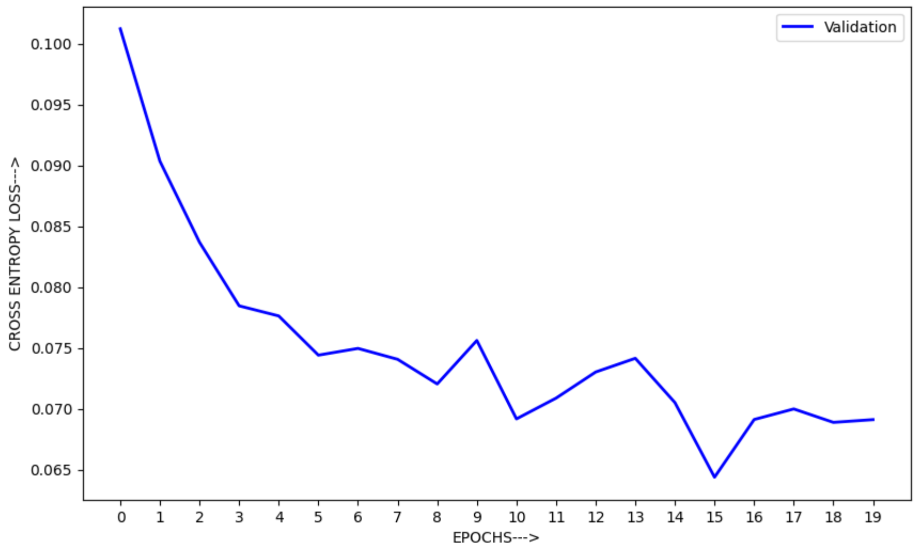

**Figure 31:** Validation Loss

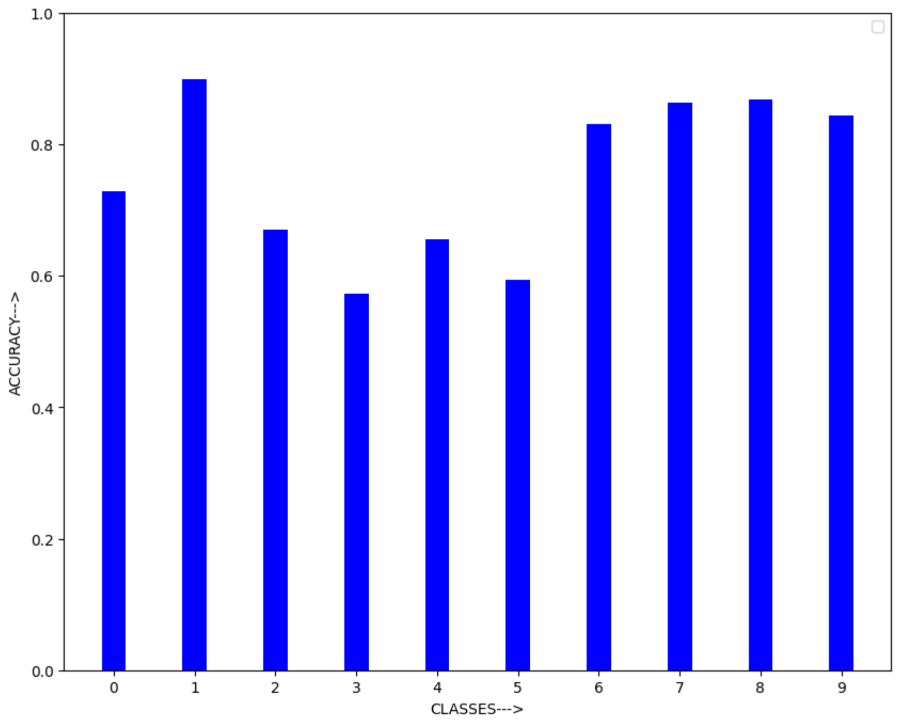

**Figure 32:** Classwise Accuracy

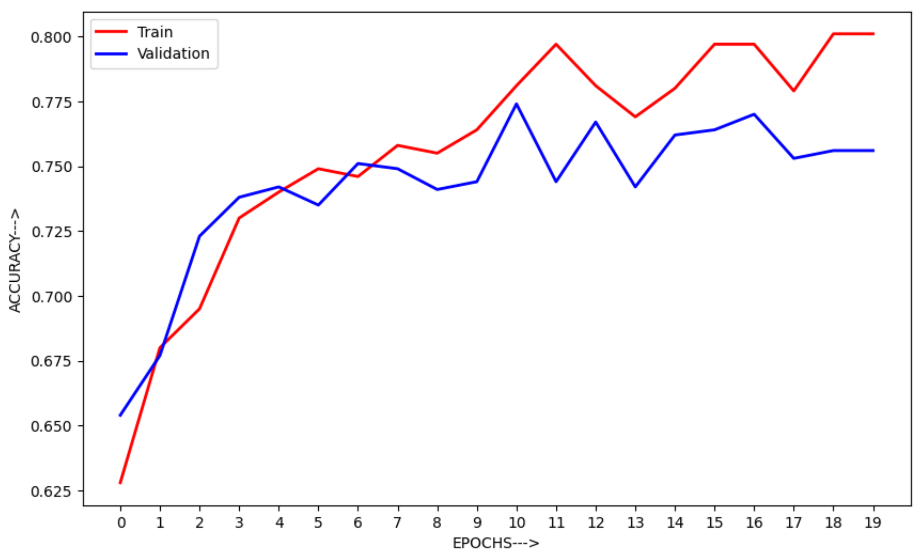

**Figure 33:** Validation Accuracy