Chinmay Mittal
(2020CS10336)

Assignment 4 | COL341
Machine Learning

April 29, 2023

Implementing CNN from the Scratch

I implemented classes for all layers and completed their forward and backward pass. The implementation was a little slow. So I subsampled the dataset to 1/10th its size both for the training and validation sets. I trained the model for 20 EPOCHS using the Cross Entropy Loss (self-implemented) and Adam Optimizer (self-implemented) with learning rate 3e-4. The results are as follows:

Training and Validation Loss plots

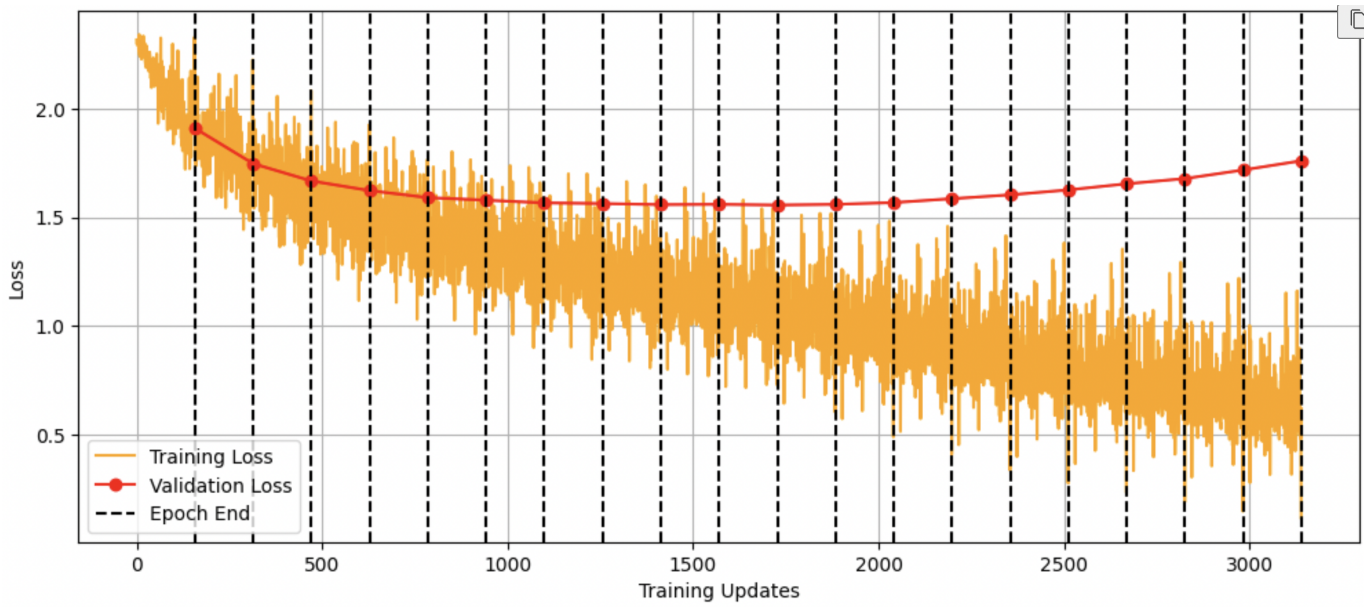

Figure 1: Loss Curves

Training and Validation Accuracy plots

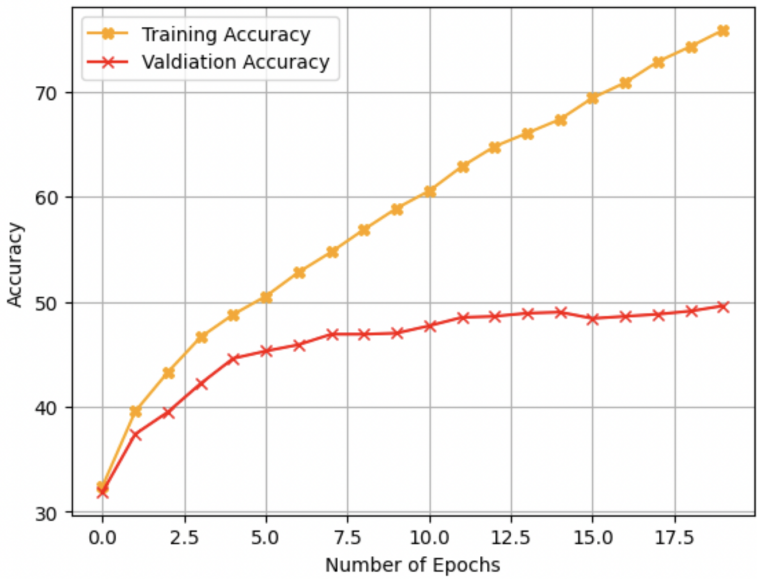

Figure 2: Accuracy Curves

Class Wise Accuracy

Class	Training Accuracy	Validation Accuracy
Plane	87.18%	67.82 %
Car	89.41%	60.00 %
Bird	60.79%	40.74%
Cat	51.95%	27.10%
Deer	83.04%	50.53%
Dog	90.75%	62.11%
Frog	77.30%	52.00%
Horse	53.16%	31.37%
Ship	83.44%	60.78%
Truck	78.86%	49.04%

Observations Due to the relatively small size of the dataset, the model overfits to the training dataset with poor generalization.
Implementing the Adam Optimizer helped learn quicker, with the accuracy increasing in a steep fashion, because Adam helps to adapt the learning rates, parameter wise for more stable gradient updates.

PyTorch based CNN implementation

Hyperparameter Tuning

Learning Rate

In this experiment I fixed the batch size (32), optimizer (adam), number of epochs (10) and the loss function (cross entropy). I did not use any data augmentation or learning rate scheduling. I trained the model for different learning rates (1e-6, 1e-3, 5e-3) and tried to see the effects on the loss curves, accuracy curves and the class wise accuracies.

Class Wise Training and Validation accuracy

Learning Rate	**Class**	**Training Accuracy**	**Validation Accuracy**
0.001	Plane	82.14 %	72.60 %
0.001	Car	87.92 %	80.70 %
0.001	Bird	71.26 %	55.40 %
0.001	Cat	74.60 %	62.70 %
0.001	Deer	75.06 %	60.40 %
0.001	Dog	69.04 %	60.10 %
0.001	Frog	81.90 %	76.80 %
0.001	Horse	84.82 %	74.50 %
0.001	Ship	94.16 %	86.30 %
0.001	Truck	89.22 %	82.20%
1e-6	Plane	43.58 %	43.10 %
1e-6	Car	31.26 %	31.60 %
1e-6	Bird	0.64 %	0.30 %
1e-6	Cat	8.08 %	7.30 %
1e-6	Deer	2.02 %	2.10 %
1e-6	Dog	21.94 %	22.00 %
1e-6	Frog	56.70 %	58.30 %
1e-6	Horse	44.36 %	43.50 %
1e-6	Ship	40.56 %	40.60 %
1e-6	Truck	41.30 %	42.90 %
5e-3	Plane	70.78 %	68.00 %
5e-3	Car	75.98 %	70.50 %
5e-3	Bird	32.82 %	28.60 %
5e-3	Cat	51.80 %	47.70 %
5e-3	Deer	49.50 %	45.80 %
5e-3	Dog	35.96 %	34.10 %
5e-3	Frog	68.86 %	68.40 %
5e-3	Horse	62.50 %	57.40 %
5e-3	Ship	84.26 %	81.40 %
5e-3	Truck	70.94 %	68.20 %

Loss Curves

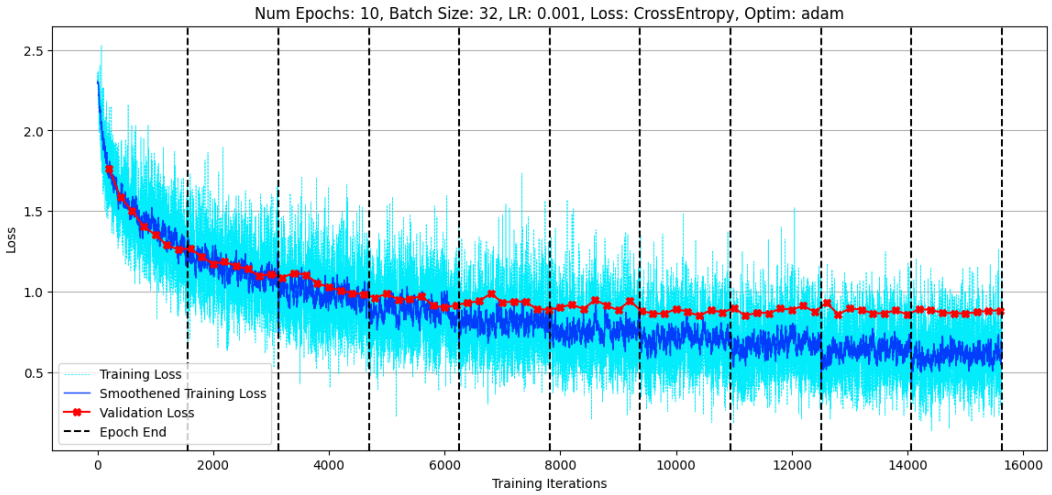

Figure 3: Learning Rate 1e-3

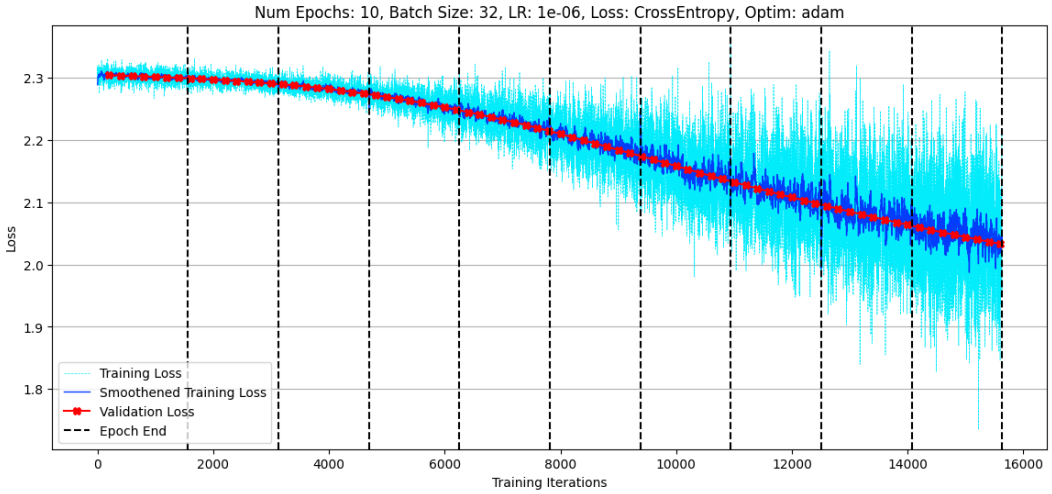

Figure 4: Learning Rate 1e-6

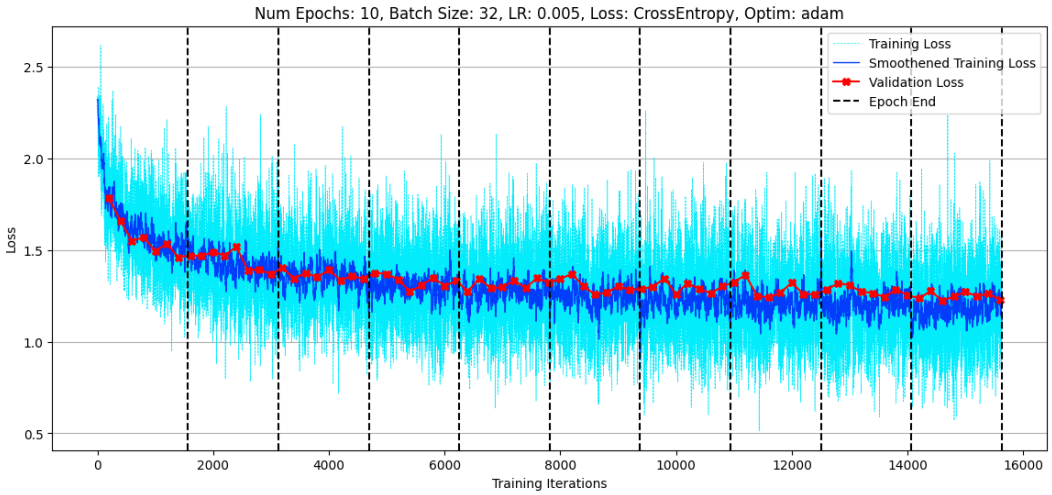

Figure 5: Learning Rate 5e-3

Accuracy Curves

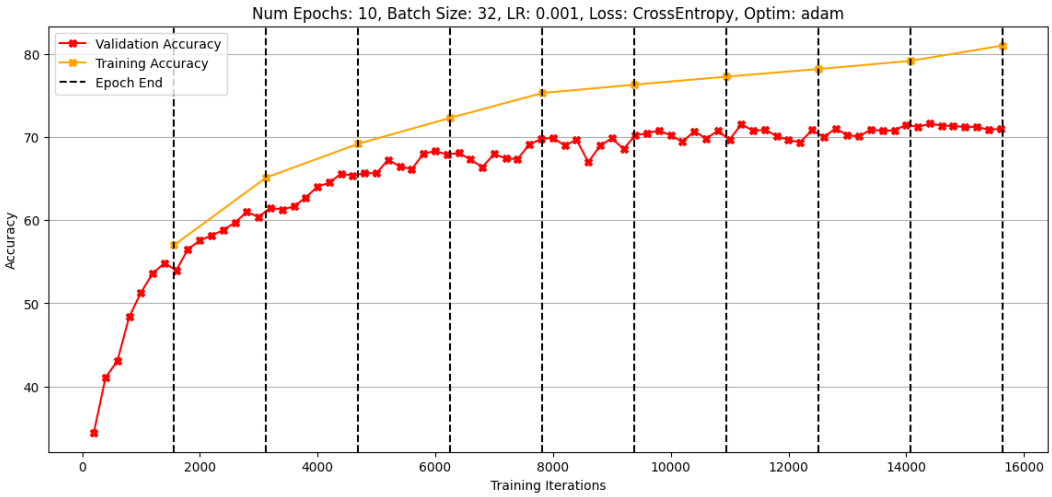

Figure 6: Learning Rate 1e-3

Figure 7: Learning Rate 1e-6

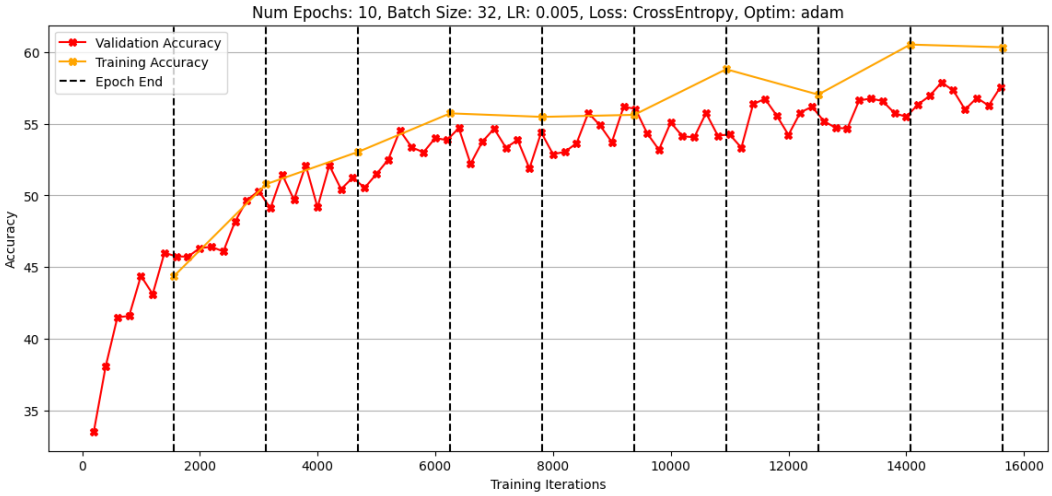

Figure 8: Learning Rate 5e-3

Observations As seen a very large or very small learning rate can be harmful for the model training. When the learning rate is very small, each update does not change the weights much. Gradient descent is optimal when the learning rate tends to 0, hence the smallest learning rate leads to most stable training. However this

also leads to slowest training (a lot of these small steps are required). As seen when the learning rate is small, the training is the most stable (steady decrease in loss and steady increase in accuracies) with less scattered training loss and training and validation curves close to each other.
If the learning rate is too large, then initially training is fast with steep decrease in training and validation loss which corresponds to large steps along directions which correspond to steep decrease in the loss function. However after a time the loss starts stagnating at a higher value before convergence. This happens because very close to the minima we are taking large steps overshooting the minima each time. Ideally close to the minima the learning rate should be very small so that we directed steps towards the minima.
Hence like other hyperparameters there exists a sweet spot for the learning rate. A very large or a very small learning rate can lead to poor training.
The class-wise accuracy trend follows, the overall accuracy in general. Some classes are harder than others and thus have particularly lower accuracies. These classes suffer a lot (eg. Cat, Deer) when the learning rate is small since no learning occurs for these classes.

Learning Rate Scheduler

I kept other parameters same, batch size(32), optimizer (ADAM), number of epochs (32), Loss function (Cross Entropy). In one set of experiments I used a fixed learning rate of 1e-5. In the next experiment I used a Learning Rate Scheduler (StepLR). Step LR decays the learning rate by a constant multiplicative factor (set to 0.7). I started by a larger learning rate (5e-3) and kept on decreasing the learning rate as the number of epochs increased eventually taking it close to the learning rate of the first experiment. *Loss Curves*

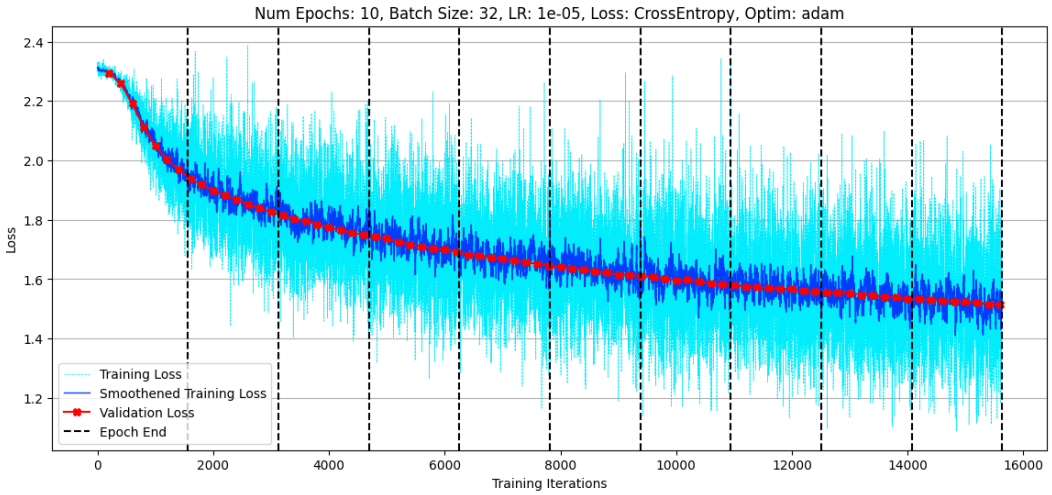

Figure 9: No Learning Rate Scheduler

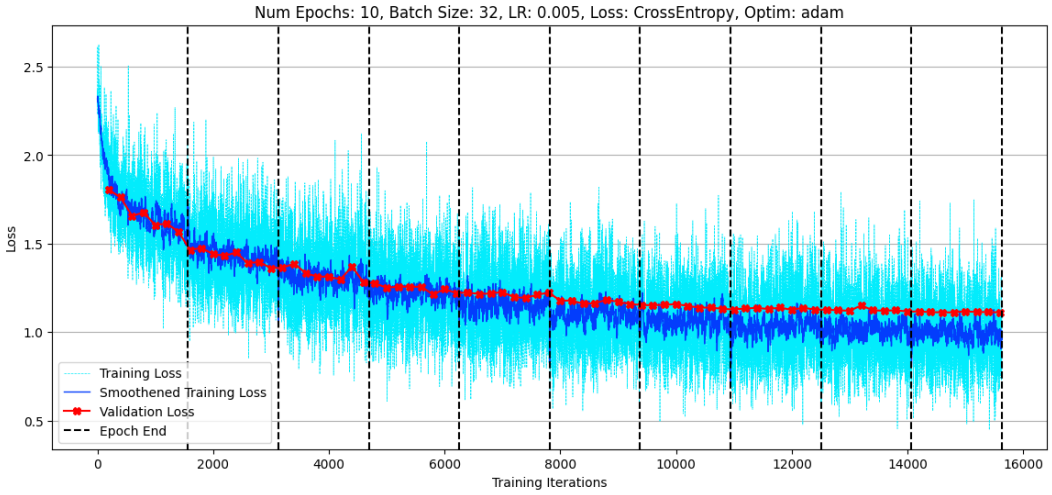

Figure 10: Learning Rate Scheduler StepLR

Accuracy Curves

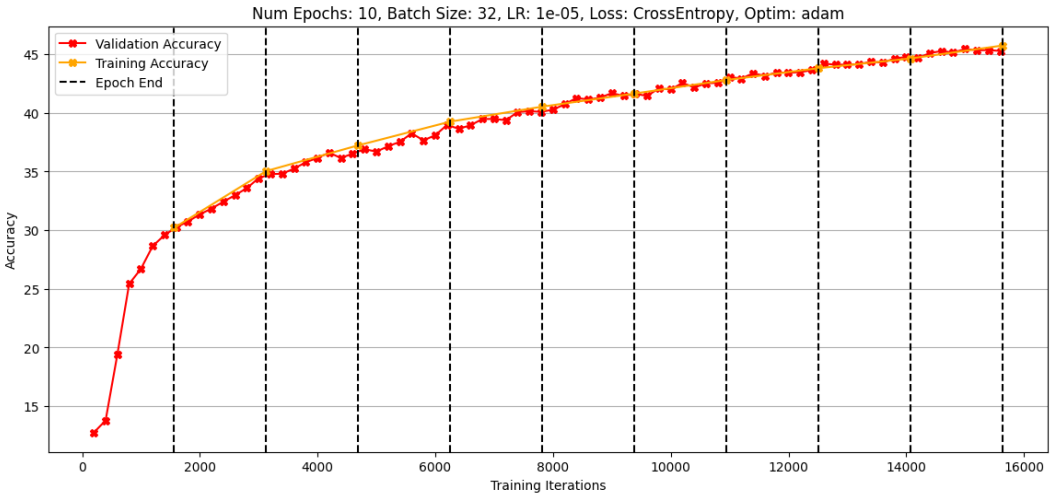

Figure 11: No Learning Rate Scheduler

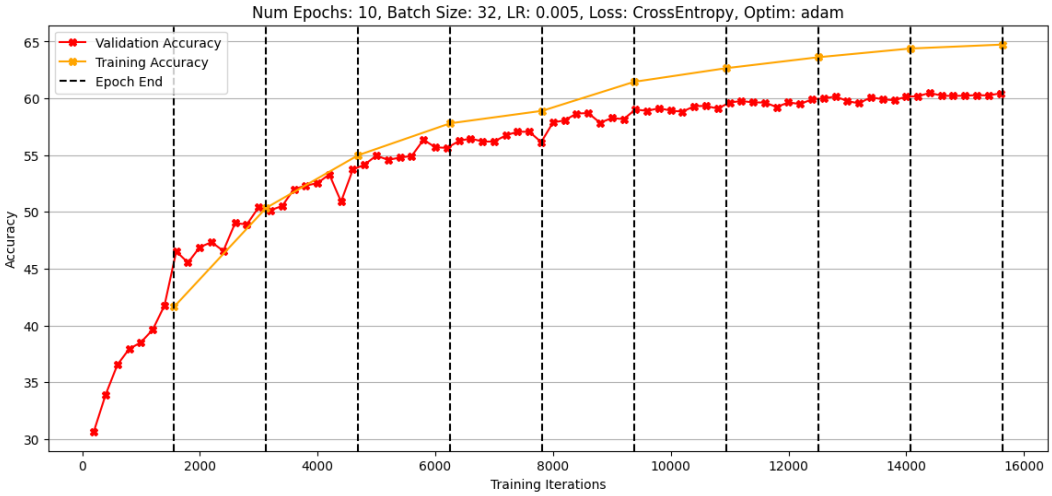

Figure 12: Learning Rate Scheduler StepLR

Class Wise Accuracies Without LR Scheduler

Class	**Training Accuracy**	**Validation Accuracy**
Plane	46.24 %	46.70 %
Car	61.20%	59.90 %
Bird	38.50%	36.00%
Cat	22.42%	20.80%
Deer	24.98%	23.30%
Dog	47.56%	46.90%
Frog	60.80%	62.90%
Horse	50.08%	53.50%
Ship	57.76%	57.50%
Truck	47.88%	47.00%

With LR Scheduler

Class	Training Accuracy	Validation Accuracy
Plane	68.44%	63.50 %
Car	84.20%	79.40 %
Bird	48.94%	42.80%
Cat	41.44%	39.40%
Deer	51.42%	43.70%
Dog	53.06%	51.10%
Frog	76.28%	72.70%
Horse	66.72%	65.00%
Ship	81.84%	76.30%
Truck	74.98%	70.70%

Observations As seen by the training and validation curves a learning rate scheduler often works better than a static learning rate. In the original few epochs, we can afford to take larger steps in the direction of the gradient because of the steep nature of the loss function away from the minima. As the number of epochs increases and we reach close to the minima we need to take smaller steps to avoid overshooting the minima, hence there is a need to decrease the learning rate. StepLR allows us to start with a larger learning rate and eventually as we train it keeps decreasing the learning rate to a small value.

Number of Training Epochs

In this experiment I fix the batch size (32), learning rate (3e-4), optimizer (Adam), Loss function(Cross Entropy). I did not use any data augmentation or learning rate scheduling. I trained the model for a large number of epochs and tried to see the effect on the loss curves and accuracy curves for both training and validation set. The loss and accuracy curves is as follows:

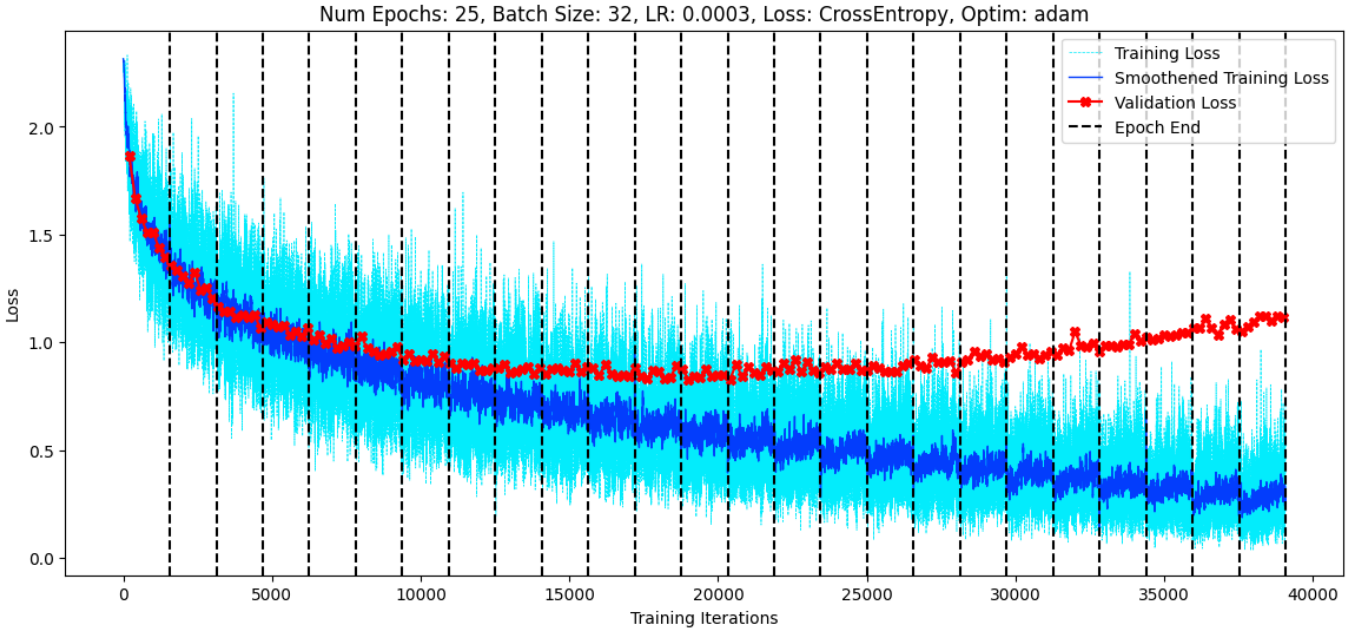

Figure 13: Training and Validation Loss vs Training Iterations

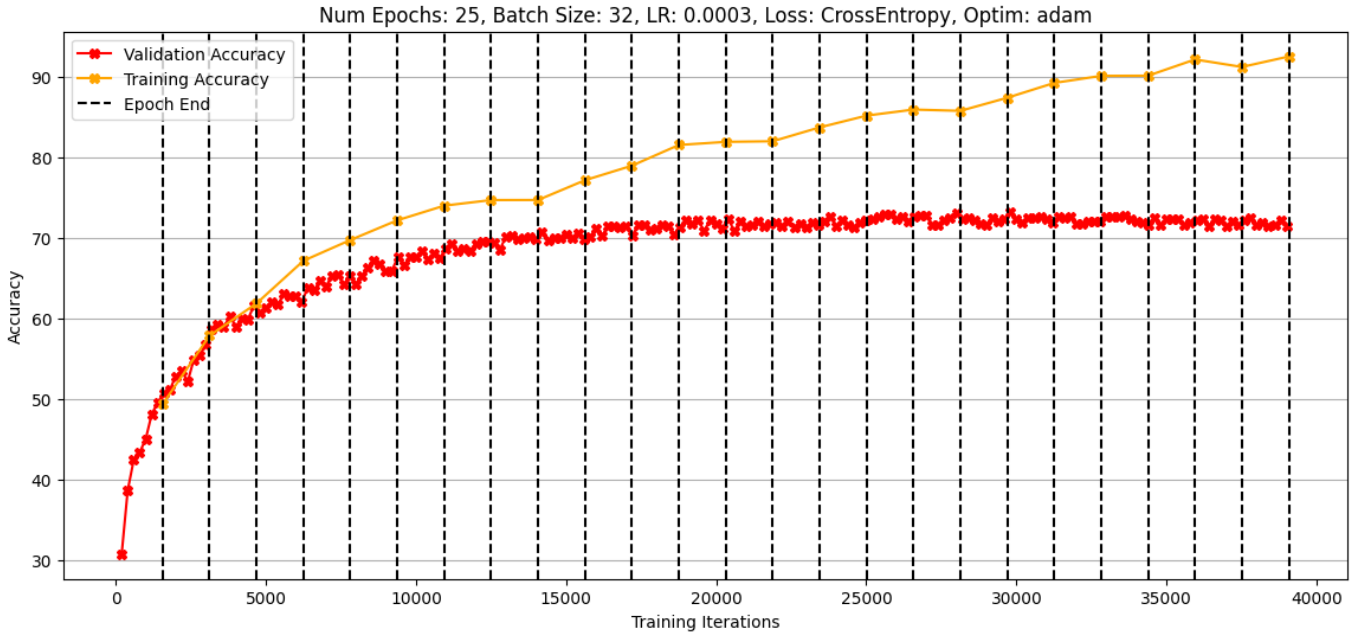

Figure 14: Training and Validation Accuracy vs Training Iterations

Observations: Increasing the number of epochs, keeping other things the same implies increasing the number of model updates. As we increase the number of model updates the model learns more and gets better at predicting images from the training dataset, this can be seen from the increasing training accuracy and decreasing training loss (although the increase/decrease will saturate over time).
It is not necessary that increasing the number of epochs is always better. As we increase the epochs initially the model learns and validation accuracy increases and the loss keeps decreasing. But if we increase the number of epochs further we can see that the validation accuracy starts declining gradually and the loss starts increasing. This is because of overfitting, beyond a point training the network leads to the network leading the noise in the training set to fit it better, but this leads to poor generalization and the validation accuracy keeps decreasing gradually due to poor generalization.
But if we train the model for very few epochs than the model under-fits even the training data and does not perform well on the validation set. Hence there exists a sweet spot where the model has not underfit and neither does it overfit.

Batch Size

In this experiment I fixed the number of epochs (10), learning rate (3e-4), optimizer (Adam), Loss function(Cross Entropy). I did not use any data augmentation or learning rate scheduling. I trained the model with different batch sizes and saw the effect on the loss curves and accuracy curves for both training and validation set.

Training Time

Batch Size	Time
4	703s
8	642s
16	876s
32	1202s

Loss Curves

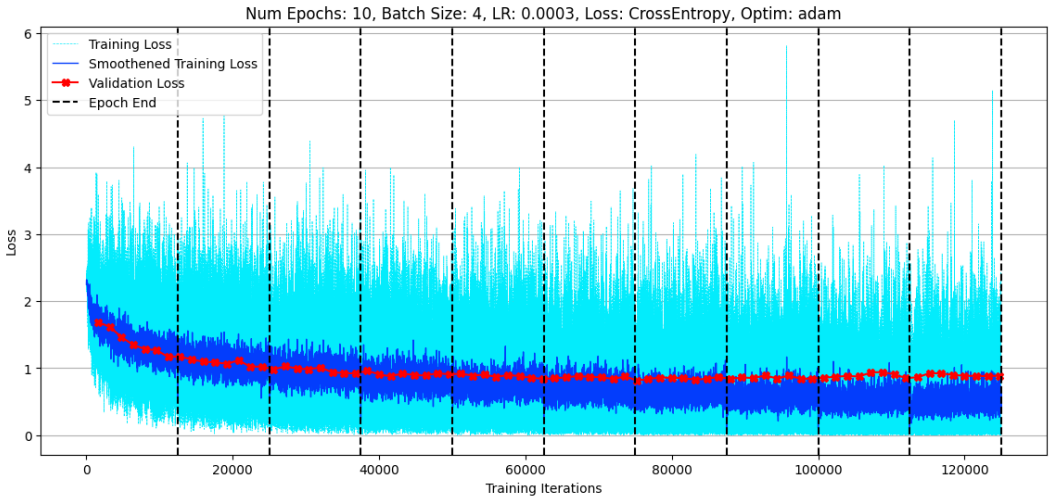

Figure 15: Batch Size 4

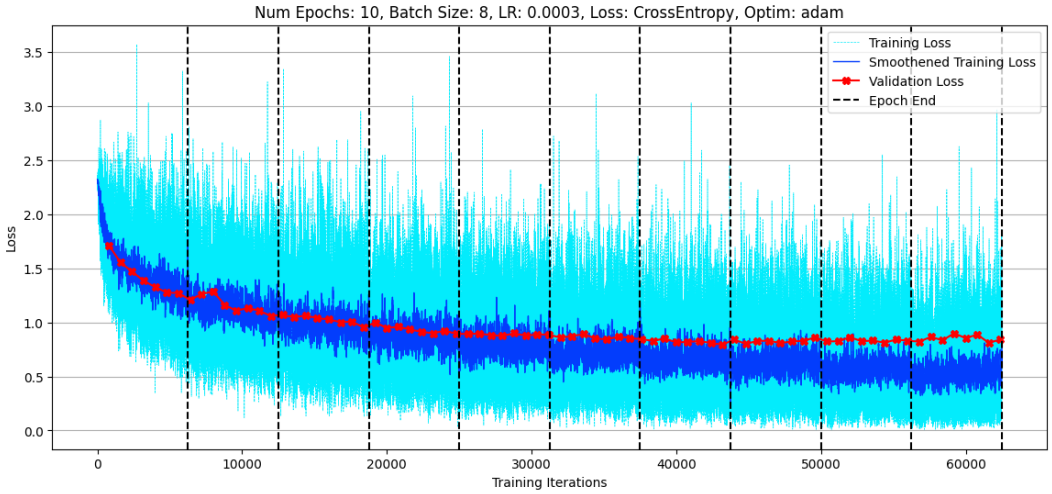

Figure 16: Batch Size 8

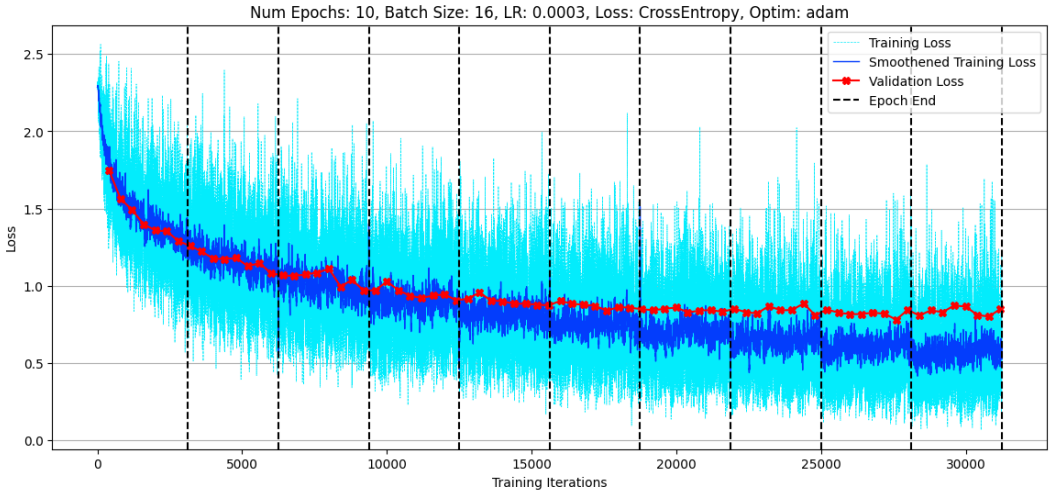

Figure 17: Batch Size 16

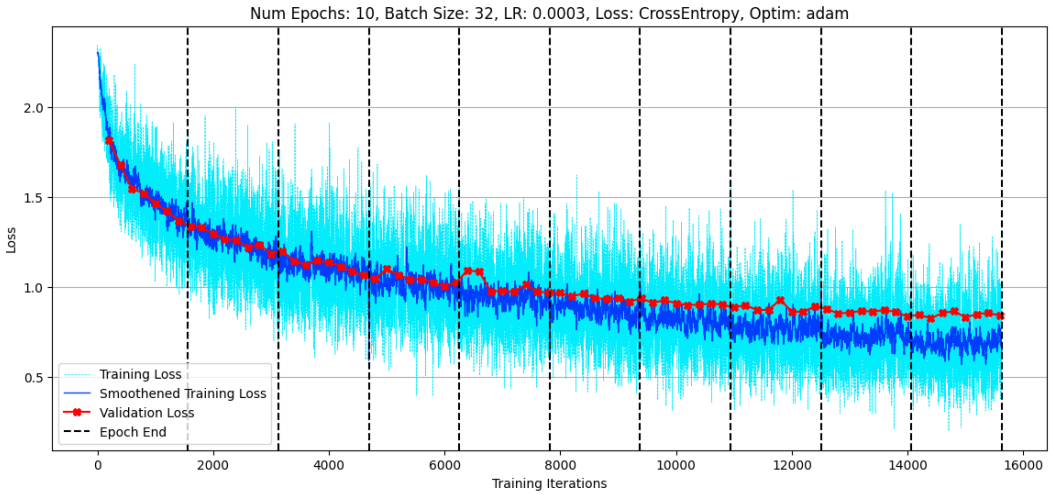

Figure 18: Batch Size 32

Accuracy Curves

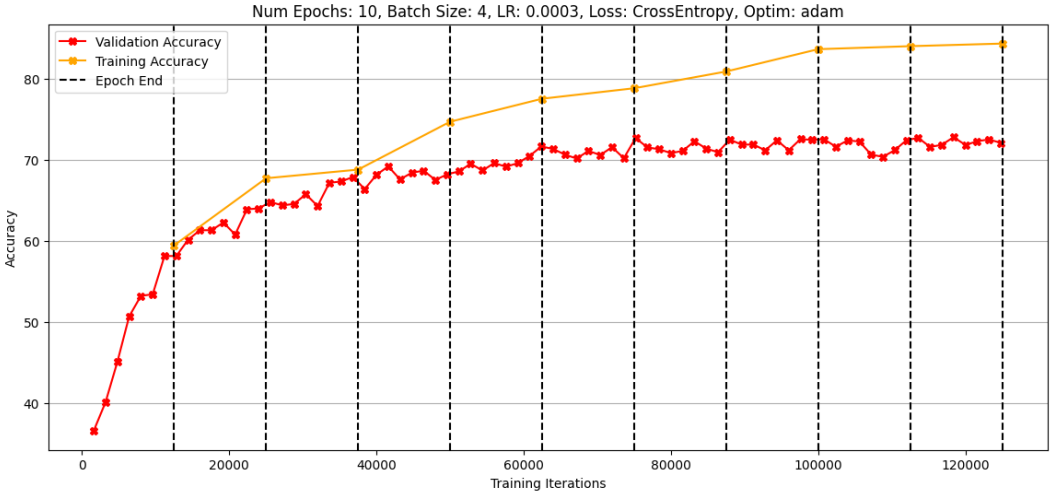

Figure 19: Batch Size 4

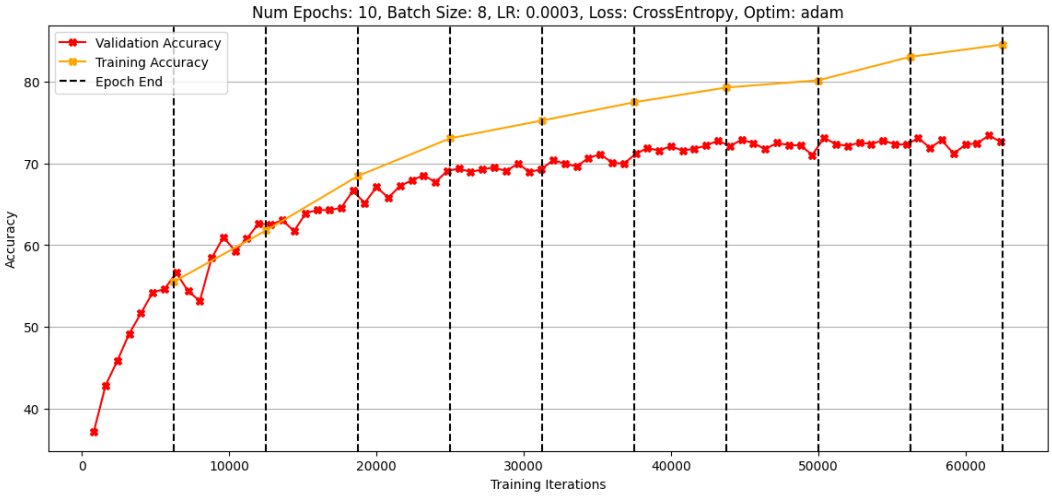

Figure 20: Batch Size 8

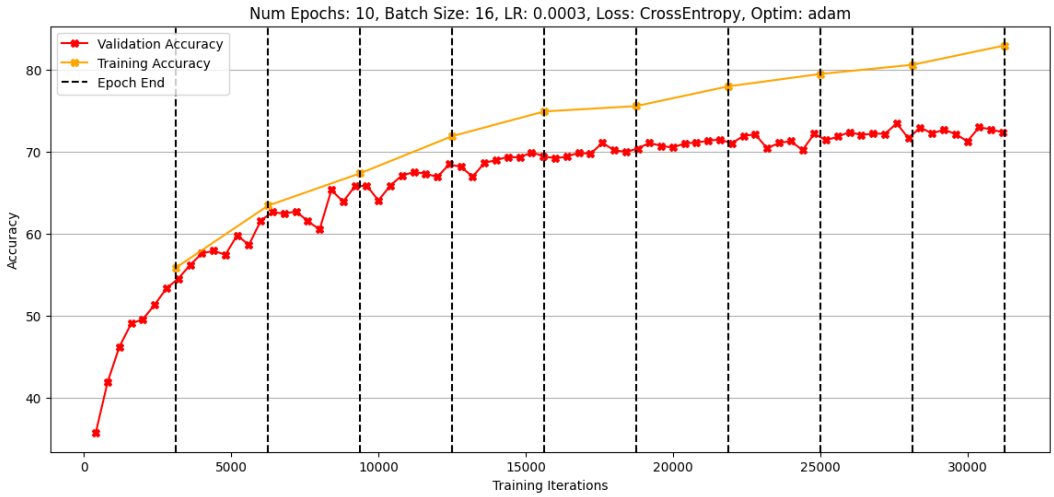

Figure 21: Batch Size 16

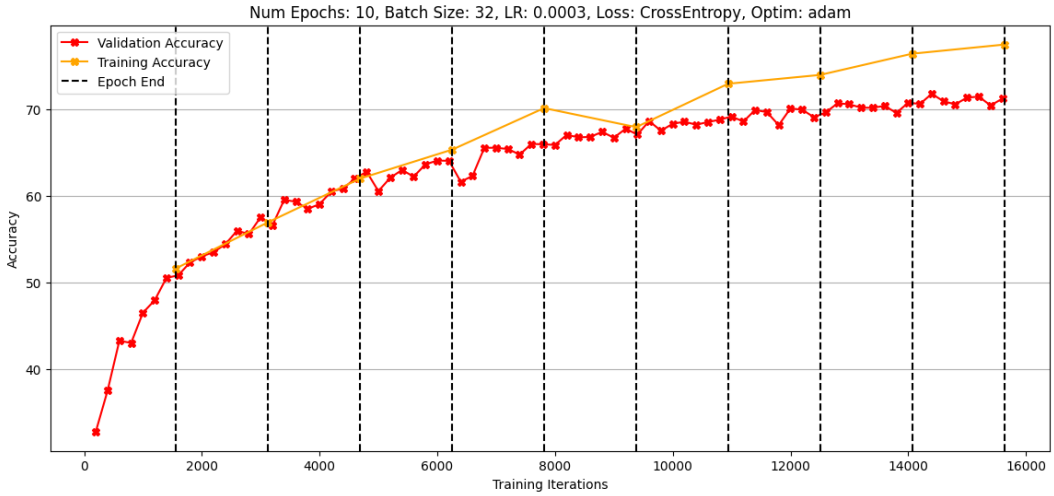

Figure 22: Batch Size 32

Observations Keeping the number of epochs the same, decreasing the batch size leads to an increase in number of parameter updates which corresponds to more training and hence is better. We can see that when batch size is 32 the model has not trained enough, (lower train accuracy compared to other cases)
But as the batch size decreases, each batch has fewer samples and the average gradient from this batch is noisy and less representative of the total training dataset gradient. This is seen by the spread in the batch losses (spread in the light blue curve increases as batch size decreases). This noise in the gradient affects training, and makes training less stable and thus training takes more weight updates.
Thus a very large batch size is good for training since each update is valuable. But number of updates become very small and training time increases for convergence. Memory limitations might also prevent large batches. Smaller batches allow for quicker updates but each update is noisy and thus several updates are required for convergence which can increase convergence time and can make the training less stable.
Due to vectorization, time taken for one epoch is faster with larger batch sizes.
Thus there exists a sweet spot for the optimal batch size which is around 16 for our case.

Effect of Loss Function

Keeping the other hyperparameters same I tried two loss functions (KL-Divergence, Cross Entropy). Other hyperparameters are as follows: Batch Size (32), Learning rate (3e-4), Optimizer (Adam) and 10 training epochs.

Loss Curves

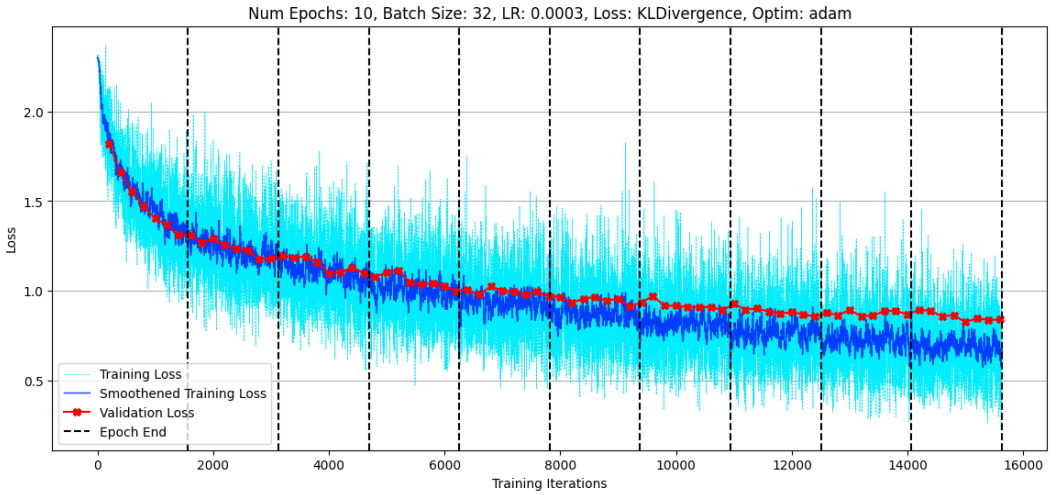

Figure 23: KL Divergence

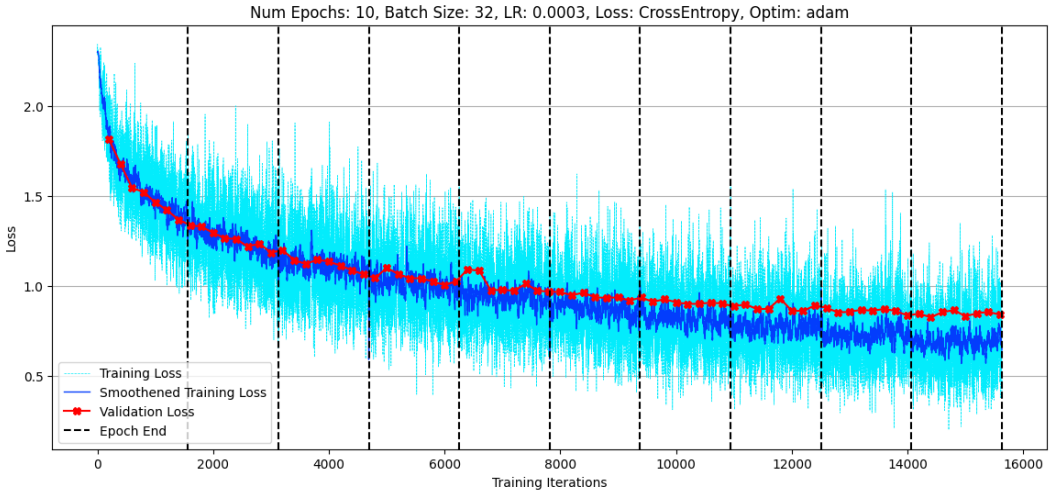

Figure 24: Cross Entropy Loss

Accuracy Curves

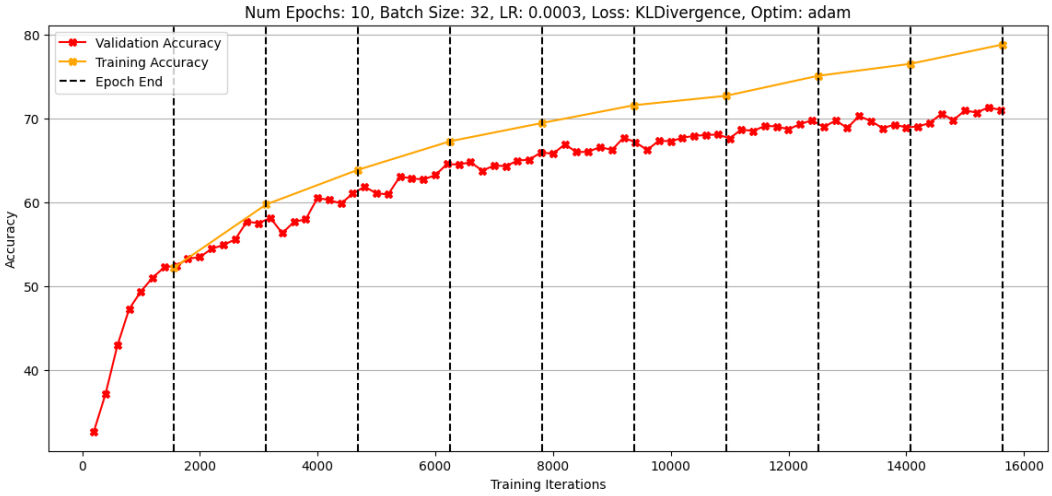

Figure 25: KL Divergence

Figure 26: Cross Entropy Loss

Observation As we can see that both the curves look almost similar. Loss curves / class accuracies / Convergence is similar for both the Cross Entropy Loss and the KL-Divergence Loss. This happens because in case of one-hot targets both of them turn out to be the same mathematically.

$$H_{\text{cross-entropy}}(p|q) = D_{\text{KL}}(p||q) + H_{\text{entropy}}(p)$$

In case of one hot targets, the entropy is zero and hence,

$$H_{\text{cross-entropy}}(p|q) = D_{\text{KL}}(p||q) + 0$$

which leads to exactly similar loss curves.

Data Augmentation

In this experiment I train two sets of models the only difference being, one was trained with data augmentation on and other was trained with data augmentation off. Other training hyper parameters were fixed as follows, Number of Epochs (10), Learning Rate (3e-4), Loss function (Cross Entropy), Optimizer (Adam).

The following transforms were used for data augmentation.

1. Random Horizontal Flipping (p=0.25)
2. Random Vertical Fliiping (p=0.25)
3. Random Rotation (between -30 and +30 degrees)
4. Random Gaussian Blur (with a (3,3) kernel and standard deviation from (0.1 to 0.25))

A sample batch with and without data augmentation look as follows

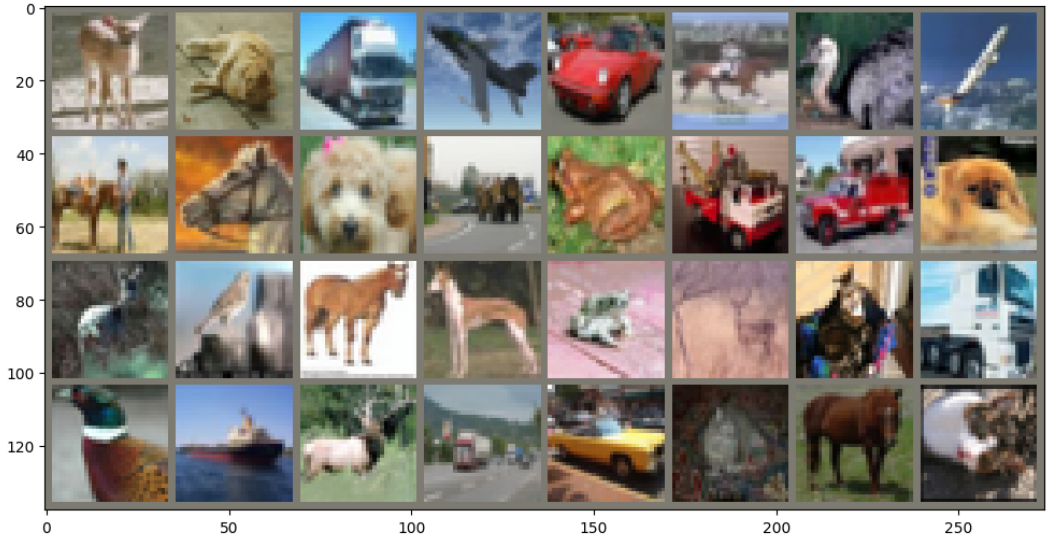

Figure 27: Without Data Augmentation

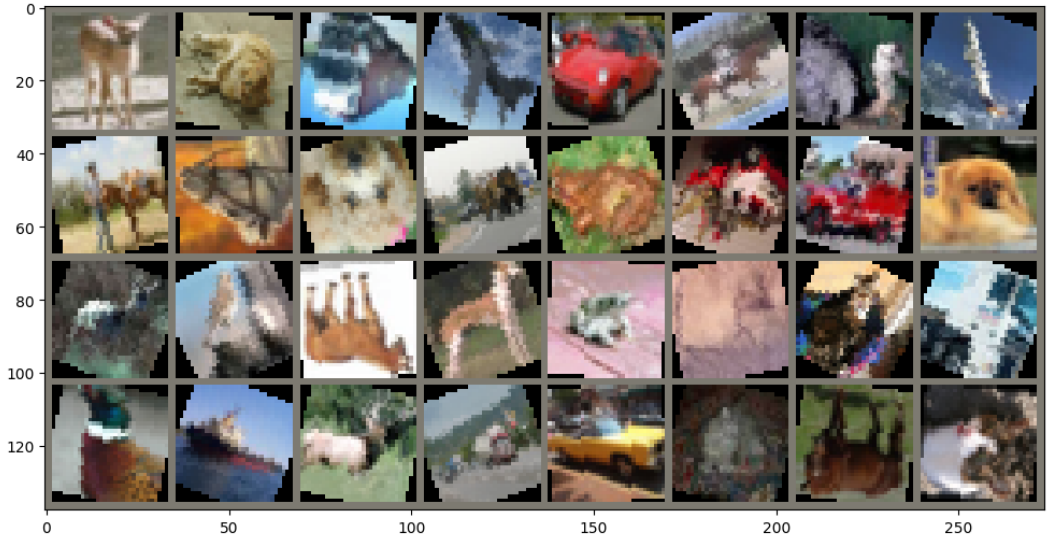

Figure 28: With Data Augmentation

The loss and accuracy curves are as follows:
Loss Curves

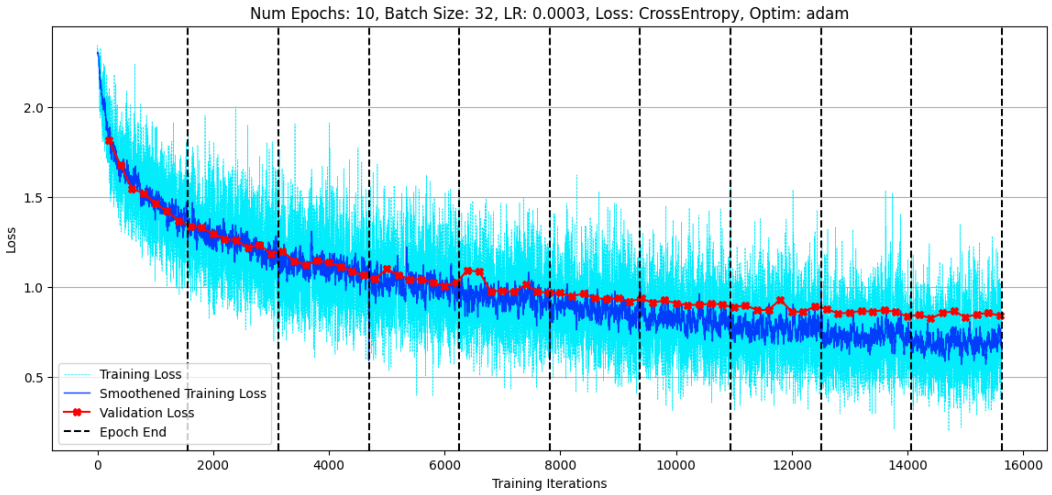

Figure 29: Without Data Augmentation

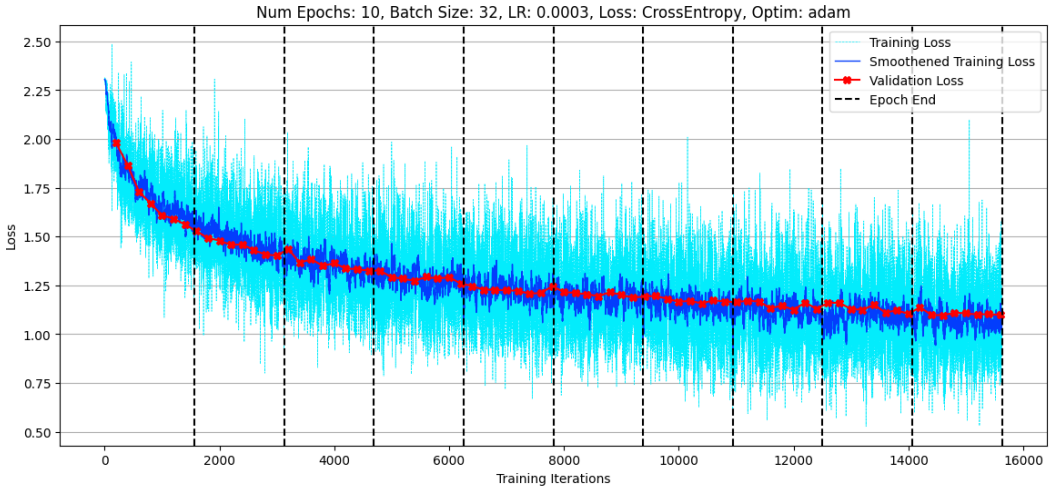

Figure 30: With Data Augmentation

Accuracy Curves

Figure 31: Without Data Augmentation

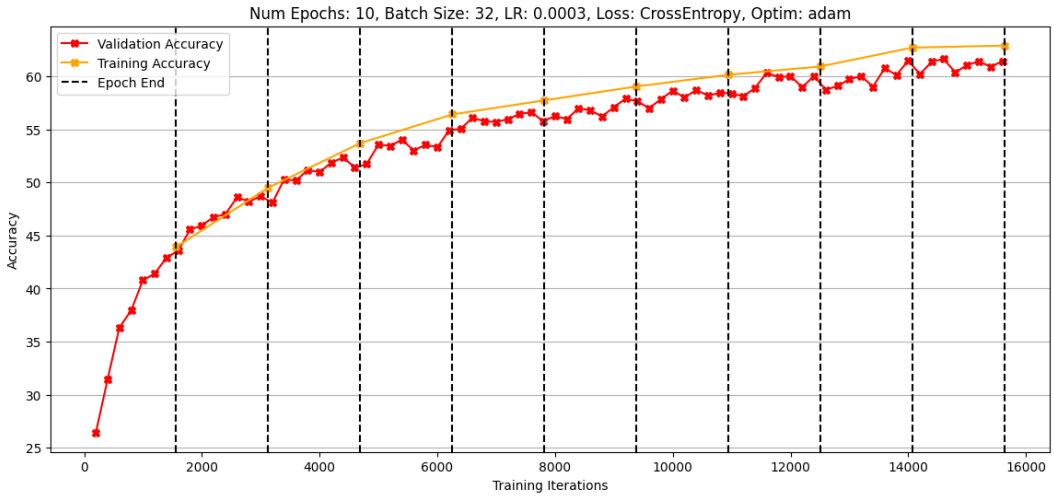

Figure 32: With Data Augmentation

Observations When we augment data, we add noise to it and hence the model learns to denoise the image and only focus on the relevant features. Thus this prevents the model from focusing on irrelevant features, and tends the model towards focusing on only relevant features. This leads to regularization in some sense. The benefit is in generalization, since data augmentation prevents the model from over-fitting (by the denoising explanation mention above and increasing the effective dataset size), we expected the gap in the training and validation accuracies/losses to be lower. These can be seen from both the accuracy and the loss curves, they tend to be close to each other when data augmentation is on.
The cost of data augmentation is that training loss tends to be lower, because the augmented dataset is strictly tougher than the original dataset. If we apply too much augmentation than the model will find the denoising objective too hard and might not learn from the training data a lot.
Hence the amount of noise added has a sweep spot, too little might lead to overfitting and too much might lead to underfitting.

Improving the CNN

There is scope of improvement in fitting the training data, given that the number of parameters in the current model is relatively small (126K), an obvious next step will be to add and try more layers.
There is also an issue of poor generalization in our model, we need to apopt techniques to avoid overfitting the noises in the dataset.
Some observations

1. Need to maintain spatial resolution of image, the current model downsamples the image pretty quickly making it difficult to detect spatially granular features. Thus we need to maintain the size of the image and downsample less frequently.
2. Add more convolution layers, this will increase the parameter count and will help fit the training data better
3. Pooling should be less frequent. Pooling downsamples the image quickly and also leads to a loss of information, instead of pooling after each convolution there is a need to pool less frequently. We will try to pool after several convolution layers. This will prevent loss of information in the initial stages of processing.
4. Smaller filters should be applied initially (find smaller features) and larger filters should be applied later onwards(detect more diffused and larger features).
5. Increasing the number of filters can help us capture more features.
6. Padding the images, given that the images are small in size, convolving without padding the images with zeros leads to very few slides of the kernel across the image (particularly large kernels). Padding will allow us to increase the number of different positions that the kernel can be kept in and help us maintain the spatial resolution of the image.
7. Adding Dropout can lead to regularization and can help us prevent overfitting.
8. Batch Normalization can help us speed up training by reducing internal covariate shift.
9. Increasing the Batch Size, will help us lead to more stable training.
10. As seen from the loss curves, given we are increasing the batch size we can also afford to increase the number of training epochs.
11. Global pooling across channels using (1, 1) filters are effective pooling strategies

    All of these ideas have been incorporated into the following architecture

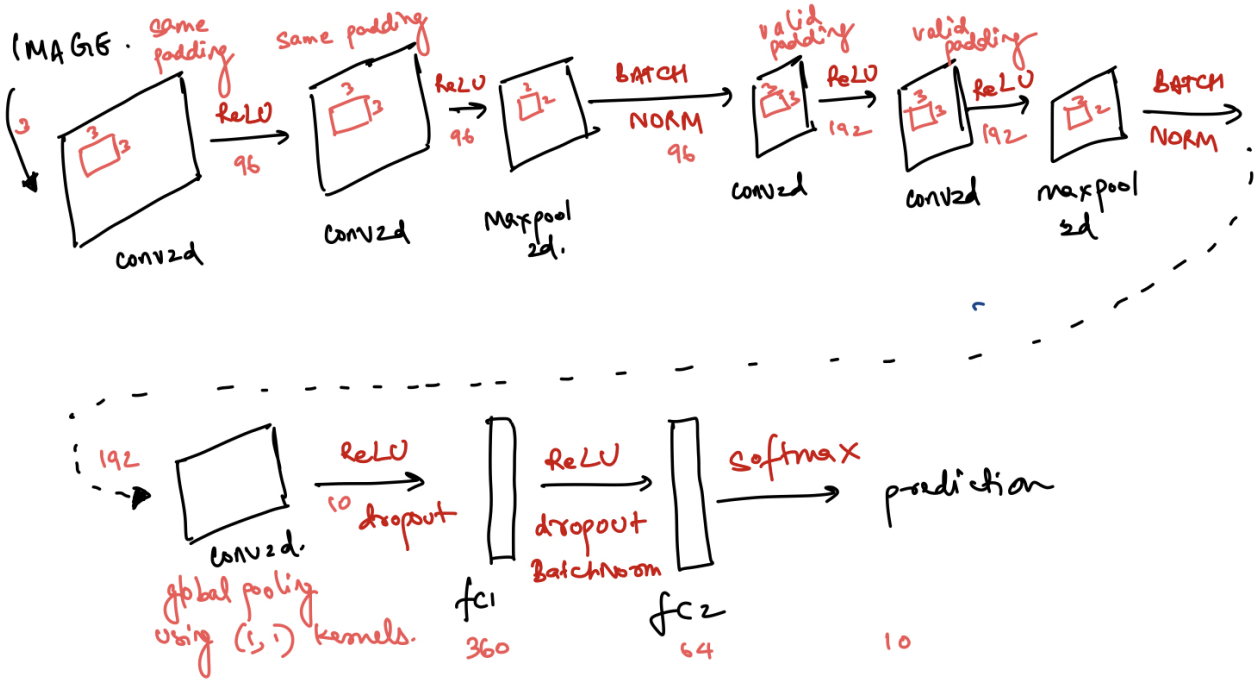

Figure 33: Improved Architecture

Loss Curves

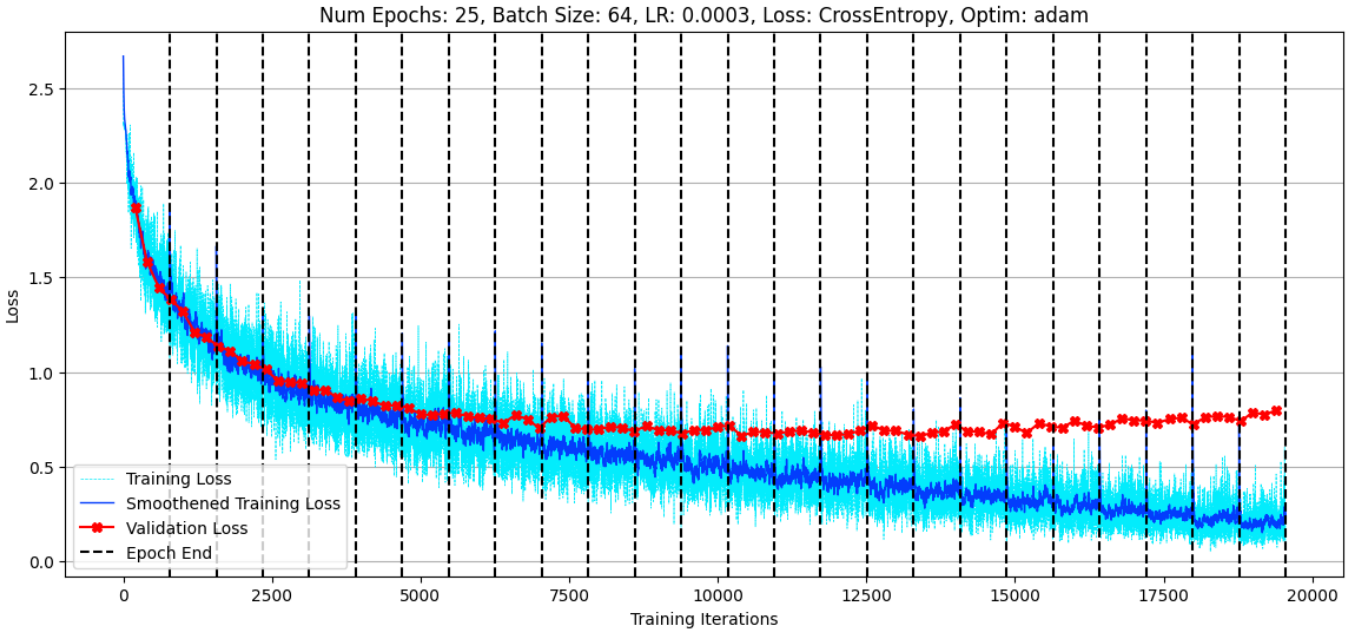

Figure 34: Loss Curve

Accuracy Curves

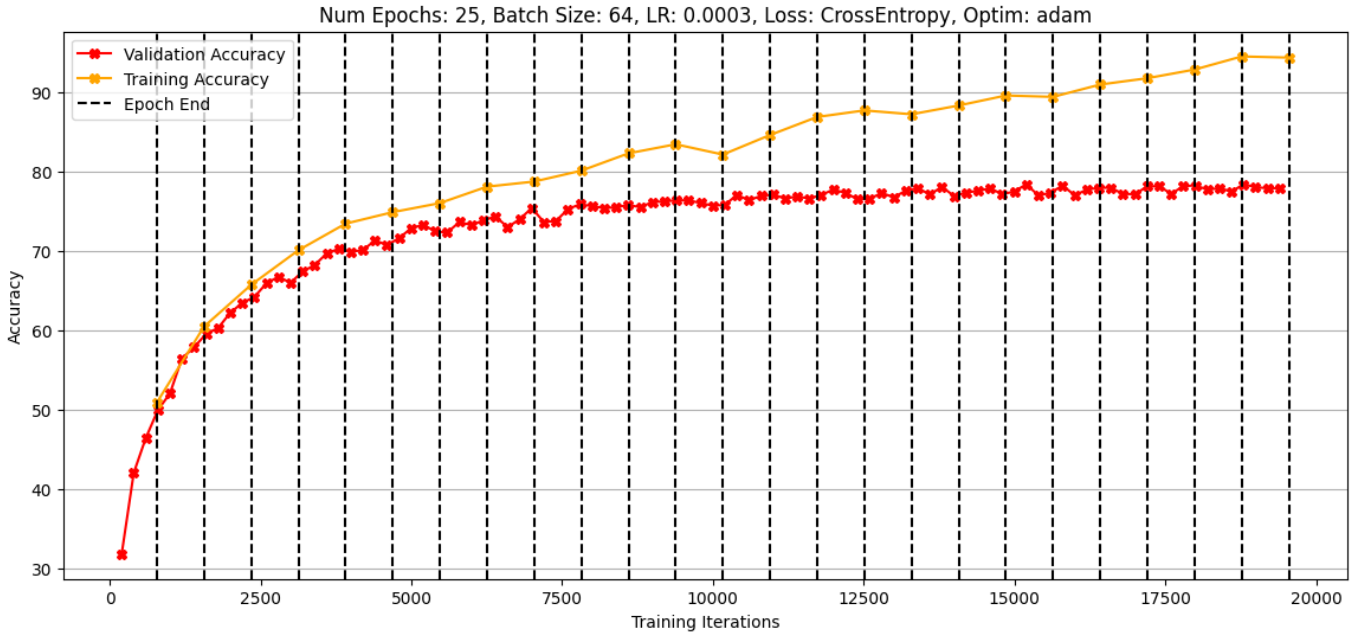

Figure 35: Accuracy Curve