Divyanshu Mund
A53321339

1 Part1

1.1 1

CIFAR-10 dataset is used for the classification task. The dataset contains colored 50000 train images and 10000 test images of $32x32$ sizes. There are total of 10 classes. The task is to train a Neural network with the aim of correctly labelling a given test image to one of these classes.

1.2 2

For data pre-processing, we are dividing each pixel value in by 255 and converting it to 'float32' so that values do not go to zero because of integer division. We scaling the input data to be in the [0-1] which helps the network train faster.

Our network is predicting a vector of length 10, with each element corresponding to prediction probability for that class. To make use of categorical cross entropy loss function, we need to create a label vector of length 10, in which all elements except the correct class index is 1. This is what is meant by one-hot encoding. Example, if the correct label for an input image is 4, its one hot encoding is $[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]$ (labels are 0-indexed).

1.3 3

The categorical cross entropy loss function is given by:

$$C = -\frac{1}{n}\sum_{x}\sum_{c}[yln(a) + (1-y)ln(1-a)] \tag{1}$$

where n is the total number of training examples, a is the prediction from our NN, summation over x and c represent sum over all training data and all classes within prediction vector respectively. For a training example, if i^{th} element is 1 in the label one hot encoded vector, the expression in square bracket becomes $-[ln(a_i) + \Sigma_{j\neq i}ln(1-a_i)]$ which we want to minimize. This is will be minimized when $a_i == 1$ and all other $a_j's == 0$. So we are incentivizing our network to give correct prediction.

1.4 4

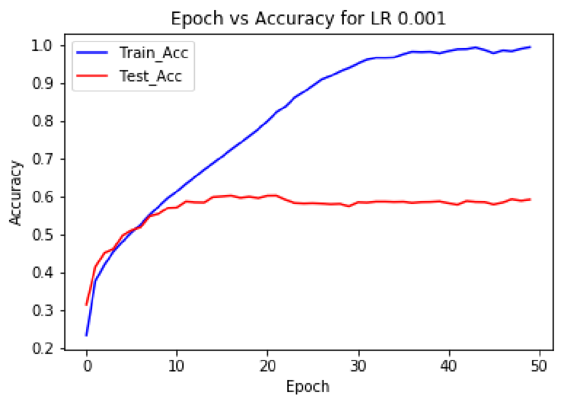
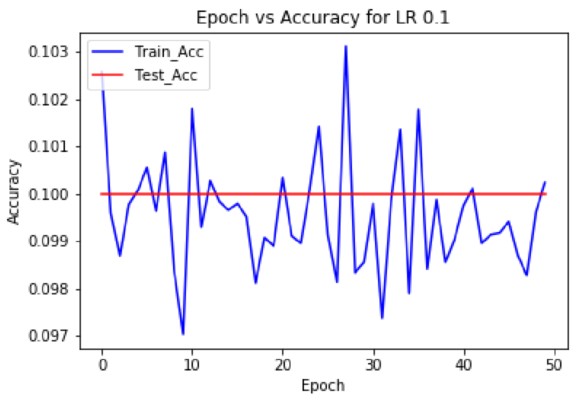
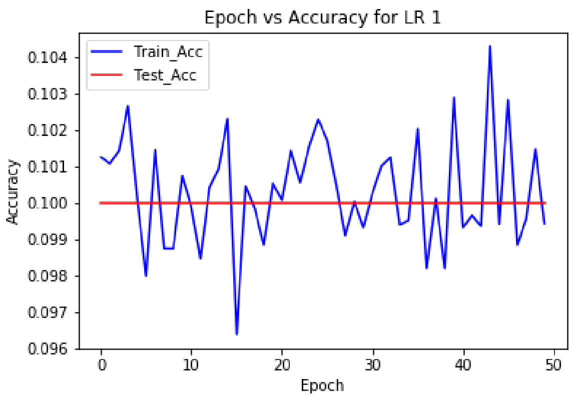


Figure 1: Plot for step 4, Lr = 0.001

As can be seen from the test accuracy plot, We can say the model saturates in learning after 15 epochs. The final test accuracy is 59%.



(a) Lr 0.1



(b) Lr 1

Figure 2: Plots for Part 5

We can see from the above plots that model is not even able to train properly when we use these learning rates. This suggests that learning rates 0.1 and 1 are too large for the model to learn anything.

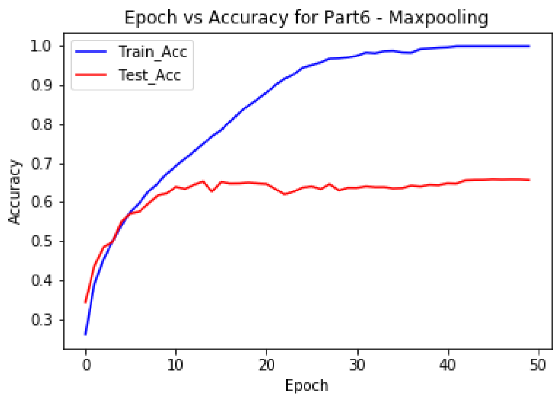


Figure 3: Plot for step 6, Lr = 0.001 and Maxpooling

Comparing the Figure 3 with Figure 1 reveals that after adding Maxpooling layers, the model learns faster and is able to get a little higher test accuracy also. This can seen from the fact that at epoch no. 10, Figure 3 is getting > 60 test accuracy whereas Figure 1 gets < 60 accuracy. Also the final accuracy for Figure 3 is more than Figure. 1.

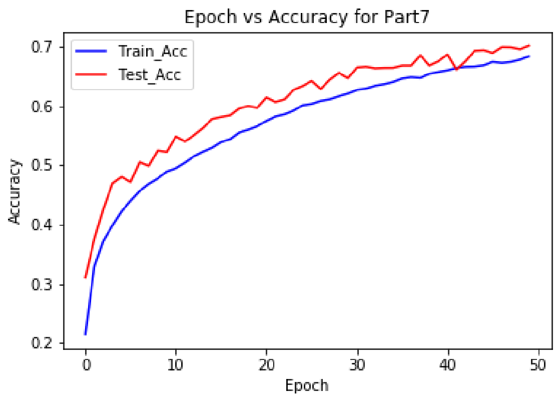


Figure 4: Plot for step 7, Lr = 0.001 and Data augmentation

We can see from the Figure 4 that data augmentation is best at dealing with overfitting. The test accuracy does not saturate even after 50 epochs and the model gets final accuracy of around 68% which is much larger than previous values.

2 Part2

2.1

Neural networks are memory and computation intensive as you have to store millions of parameters. To reduce the storage and computation overhead, we can remove some 'unimportant' weights which do not contribute significantly towards accuracy performance. Pruning is one such method where we remove some unimportant connections/weights. Since we are also making the network less powerful, there are lesser chances of over-fitting and pruning can improve generalization. For this project, we are pruning weights which have very small absolute value. However, since we are not retraining the network, there will be some accuracy drop.

2.2

One drawback of pruning is to choose the criteria to select weights to drop out from the network. This is more of a heuristic which has to be implemented and tested. Plus, as we will see from the experiments below, pruning weights from different layers have different impact on final performance and it can be difficult to choose the layer to prune weights from.
Also, after the network has been pruned, the network will obviously be sparse which will not have the efficiency gains over a fully dense network when trained on GPU.

2.3

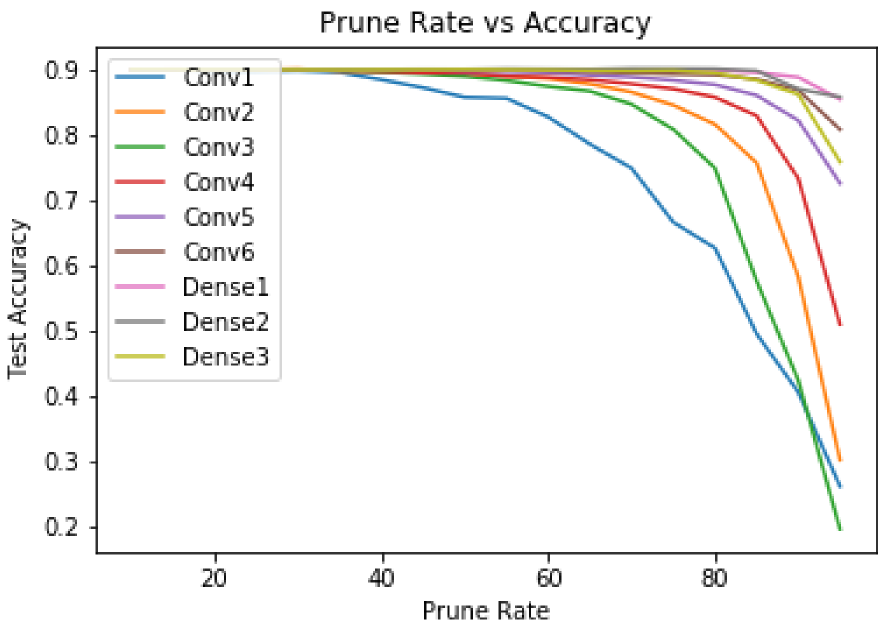


Figure 5: Accuracy vs prune rate for various layers

As can be seen from the network, pruning weights from the shallower layers lead to much larger performance degradation as compared to pruning weights from deeper layers. This makes sense because impact of shallower weights on the final output is much more as compared to deeper weights.

3 Part3

3.1

In this part, we are trying to compress a neural network so that it is less memory intensive and time intensive during forward prop. This has its applications when we want to deploy a large pre-trained model on mobile devices. We achieve this by using Tucker decomposition to decompose the weight tensors of a model into smaller tensors. We wish to achieve similar level of accuracy performance.

3.2

Just like SVD is a type of matrix decomposition technique in 2D, Tucker decomposition is higher order generalization of SVD. It decomposes a tensor into a core tensor by multiplied by a matrix along each mode. The core tensor can be much smaller compared to original tensor which leads to very compact representation of original tensor.

As given in the project, tucker decomposition on W of size $k\text{x}k\text{x}c\text{x}f$ gives us:

$$W \sim I * core * O \tag{2}$$

where $core \sim k\text{x}k\text{x}R1\text{x}R2$, $I \sim c\text{x}R1$, $O \sim R2\text{x}f$
So we have decomposed our original convolution operation into 3 conv2D operations. Original padding and stride is incorporated into the conv2D corresponding to 'core' layer. Original bias and activation function is incorporated into the conv2D corresponding to 'O' layer.

3.3

3.4

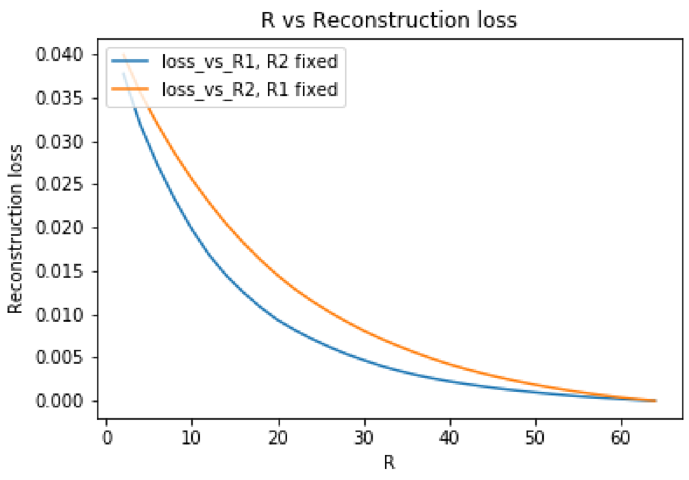


Figure 6: Reconstruction loss vs R variation

From the figure we see that as we increase R, in both R1 and R2 cases, the reconstruction loss decreases. This makes sense because we are allowing a larger core matrix which can capture more information from the original matrix. We also observe that keeping R1 fixed to 64 and varying R2 leads a higher reconstruction loss across all values of R. This is probably because R2 has direct impact on outgoing connections.

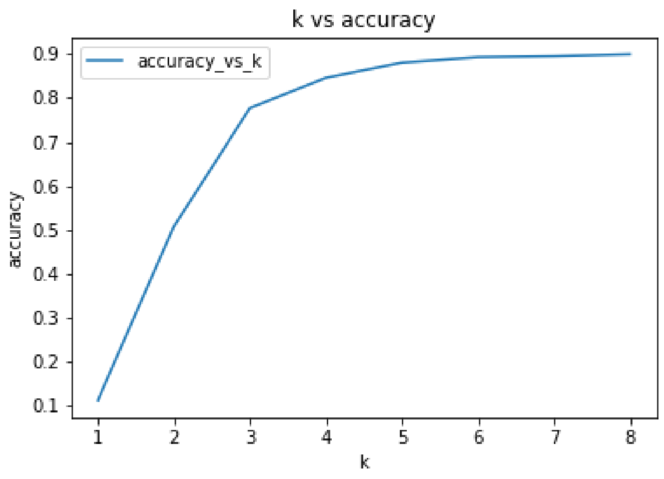
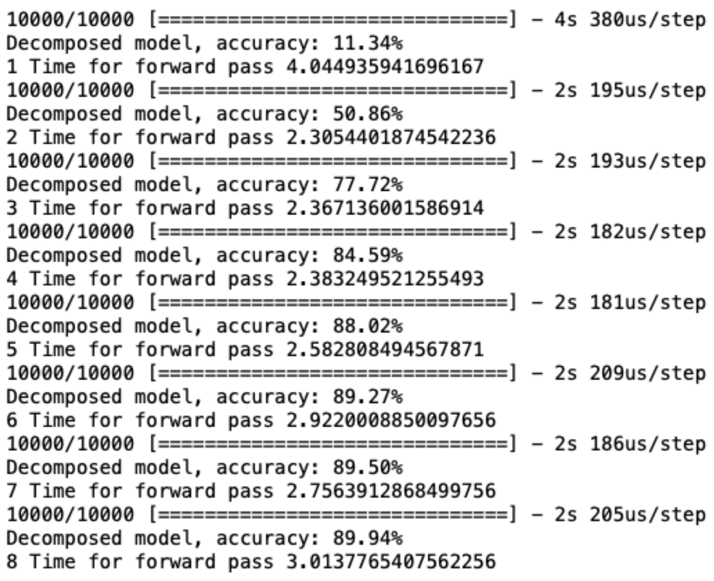


Figure 7: Accuracy vs k variation

From the figure we see that as we increase k, the test accuracy increases and at k-8, the model reaches original accuracy of around 90%. This is intuitive because in this case $R1 = c$ and $R2 = f$ and the model fully captures original tensor information.

From the figure[8] we see that, at $k = 4$ we get accuracy of around 88.02% which is competitive with original



```
10000/10000 [==============================] - 4s 380us/step
Decomposed model, accuracy: 11.34%
1 Time for forward pass 4.044935941696167
10000/10000 [==============================] - 2s 195us/step
Decomposed model, accuracy: 50.86%
2 Time for forward pass 2.3054401874542236
10000/10000 [==============================] - 2s 193us/step
Decomposed model, accuracy: 77.72%
3 Time for forward pass 2.367136001586914
10000/10000 [==============================] - 2s 182us/step
Decomposed model, accuracy: 84.59%
4 Time for forward pass 2.383249521255493
10000/10000 [==============================] - 2s 181us/step
Decomposed model, accuracy: 88.02%
5 Time for forward pass 2.582808494567871
10000/10000 [==============================] - 2s 209us/step
Decomposed model, accuracy: 89.27%
6 Time for forward pass 2.9220008850097656
10000/10000 [==============================] - 2s 186us/step
Decomposed model, accuracy: 89.50%
7 Time for forward pass 2.7563912868499756
10000/10000 [==============================] - 2s 205us/step
Decomposed model, accuracy: 89.94%
8 Time for forward pass 3.0137765407562256
```

Figure 8: Time taken for forward prop

performance and it only takes 77% of time taken by the original network.