ELL 793
Assignment 3
10th Dec

Submitted by:

Sarthak Garg (2017EE30546)
Varun Gupta (2017EE30551)

About ResNets

ResNets use residual blocks as shown below and allow us to build extremely deep neural nets. The idea is that increasing the number of layers won't degrade the performance since those extra layers may just act as the identity function and so effectively the deeper network will be equivalent to a shallow one. On passing an older activation to a future layer, we are precisely doing the same thing. On training, the model can itself choose to train the weights of the extra layers to act as identity function and we are effectively cutting out those layers if required.

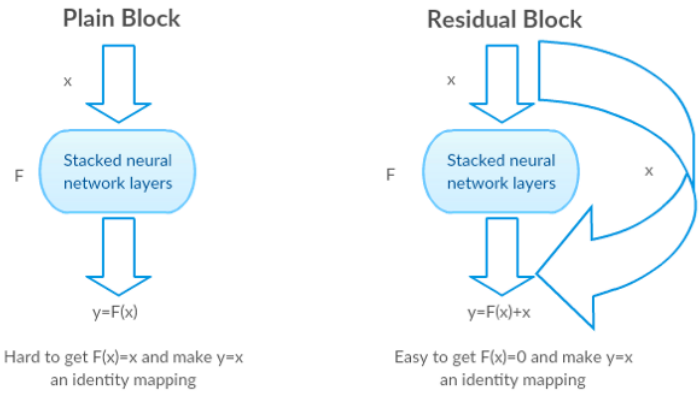


Figure: Comparison of normal CNN block and Residual block

Source: https://medium.com/@14prakash/understanding-and-implementing-architectures-of-resnet-and-resnext-for-state-of-the-art-image-cf51669e1624

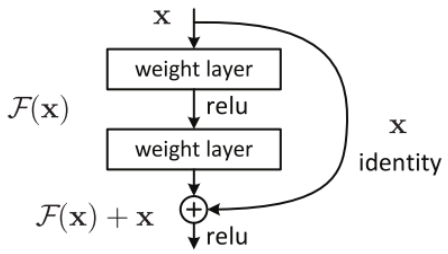


Figure: Representation of the Residual block and identity forward branch

Source: https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035

The Skip Connections between layers add the outputs from previous layers to the outputs of stacked layers. This results in the ability to train much deeper networks than what was previously possible.

Part (a)

Building a CNN with Resnet-18 as the backbone and training it on CIFAR-10 dataset

We implemented a Resnet18 model from scratch using PyTorch. We implemented the basic blocks used in Resnets ourselves and then stacked them to create the final architecture.

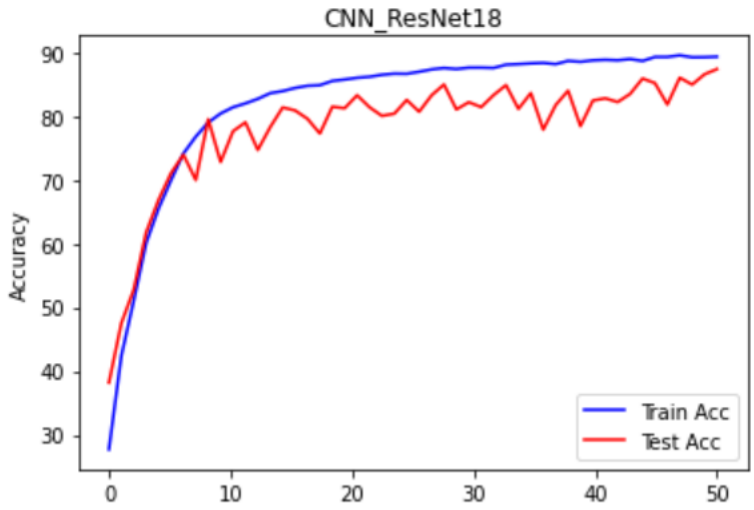
Structure of model

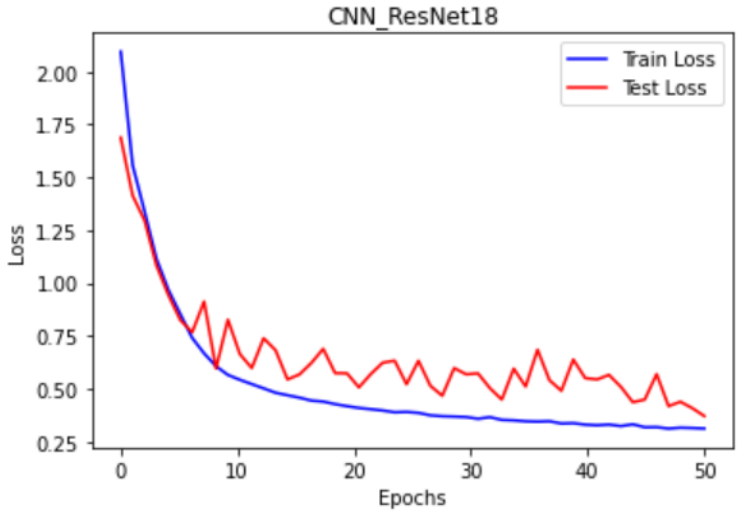
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (shortcut): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
  )
  (linear): Linear(in_features=512, out_features=10, bias=True)
)
```

Training the model

We trained the model on the CIFAR10 dataset for 50 epochs. The hyperparameters chosen are in the attached code file. For each epoch, we evaluated the model performance on the test set as well for a comparative analysis. The plots for training accuracy and loss are shown below:





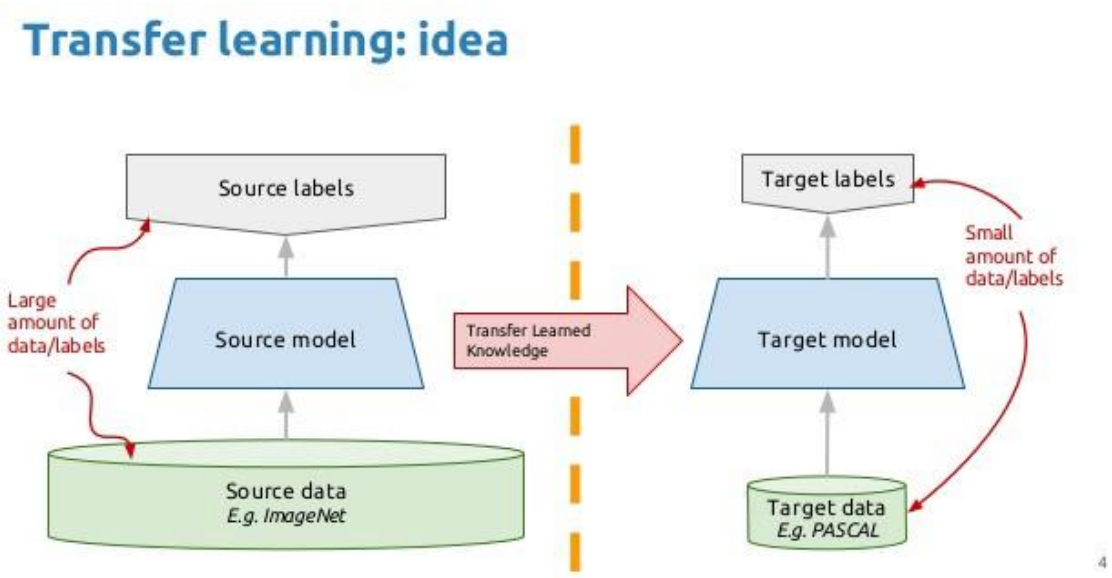
Performance at the end of 50 epochs was:

```
Epoch: 49
Train Epoch: 49 [31200/50000 (100%)]    Acc: 89.290% (44645/50000)      Loss: 0.311361

Test set: Average loss: 0.3705, Accuracy: 8735/10000 (87%)
```

Transfer Learning

Transfer learning is a popular method in computer vision because it allows us to build accurate models in a timesaving way (Rawat & Wang 2017). With transfer learning, instead of starting the learning process from scratch, you start from patterns that have been learned when solving a different problem. This way you leverage previous learnings and avoid starting from scratch. In computer vision, transfer learning is usually expressed through the use of pre-trained models. A pre-trained model is a model that was trained on a large benchmark dataset to solve a problem similar to the one that we want to solve.
Source : https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751



Source : https://towardsdatascience.com/transfer-learning-with-convolutional-neural-networks-in-pytorch-dd09190245ce

Part (b)

Initialize the ResNet-18 network with pretrained weights from ImageNet and use these weights to improve the training for the CIFAR-10 dataset

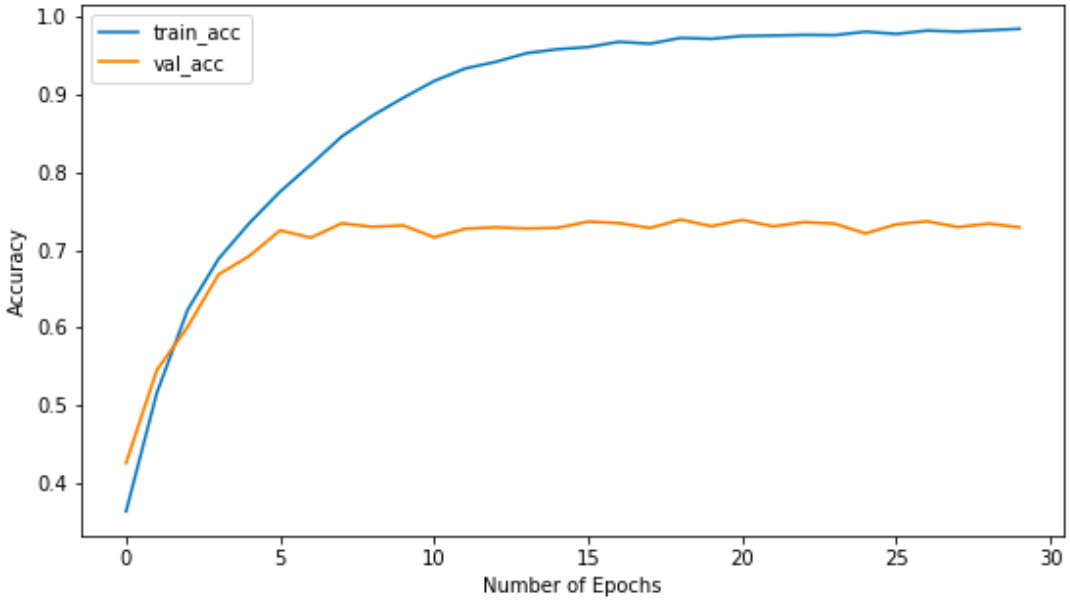
1) Prebuilt model with pretrained = False

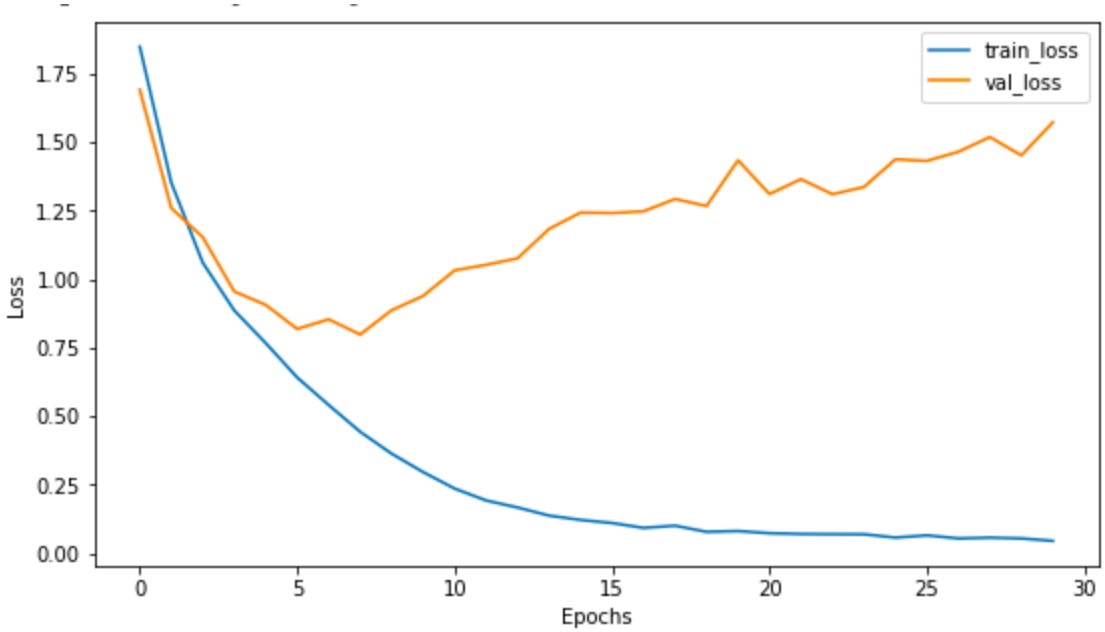
For this problem, the first thing we do is create a base model which we will use to benchmark our other results and tuning.
For this, we import prebuilt the model of ResNet18 in Pytorch and keep pretrained = false

```
model_rn = models.resnet18(pretrained=False)
```

Therefore, this is a raw model with randomly initialised weights and we will train this and store the results as the comparative benchmark for future tuning.

The training and validation plots after training on 30 epochs are as follows:





Final evaluation on train,val and test set:

```
Training  Accuracy:   0.986
Validation Accuracy:  0.739
Testing Accuracy:     0.738
```

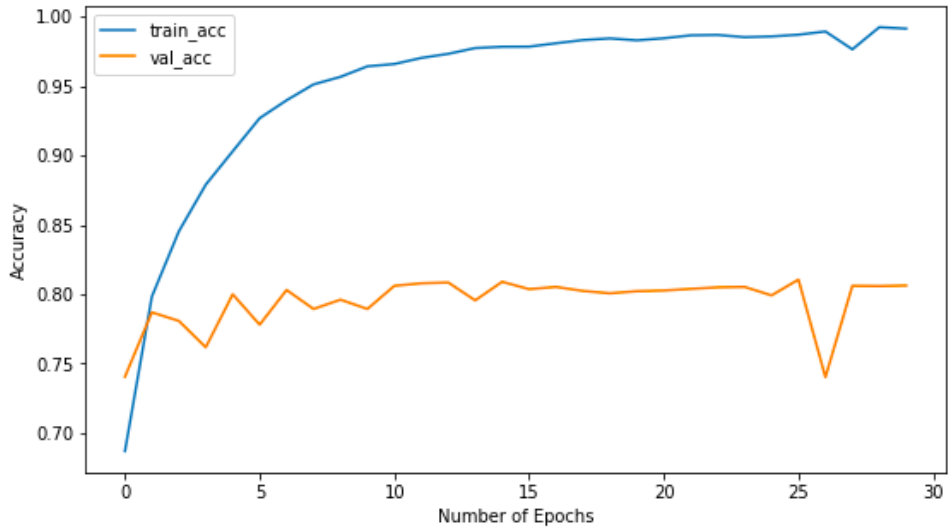
We can see that even before 10 epochs, the validation loss starts increasing and validation accuracy saturates. This shows that there is overfitting occurring on the training set. This also suggests the use of dropout in the future tuning.

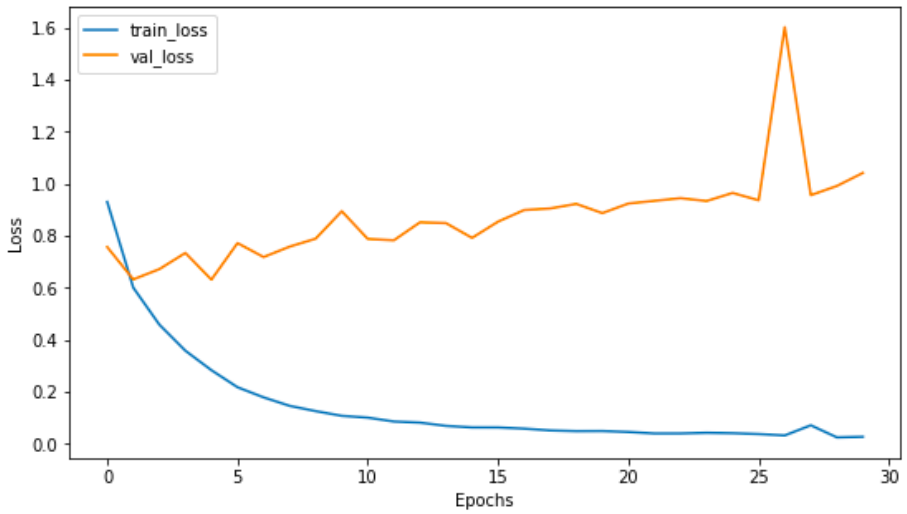
One thing to note is that this model takes more than 9 minutes to train on the 50 epochs.

2) Prebuilt model with pretrained = true and all weights are learnable

Next we use the prebuilt model which has already been pretrained on the ImageNet dataset. This model is made using `models.resnet18(pretrained=True)`
Moreover, in this model, during training we have allowed all weights to be updated using the optimiser. Therefore, even though the model is pretrained on the ImageNet dataset, the weights can further be learned as per the CIFAR10 dataset

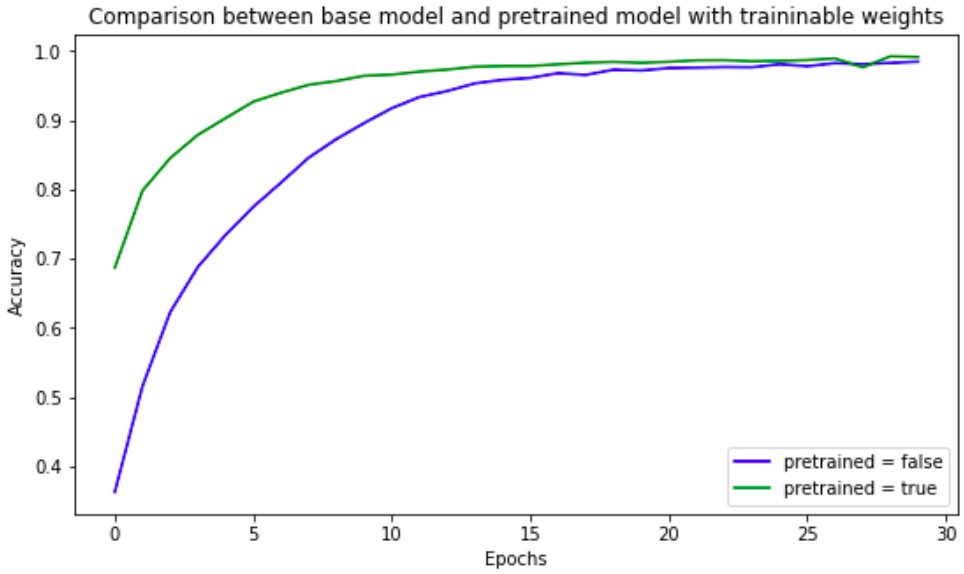
The training plots of 30 epochs on the CIFAR10 dataset are as follows:

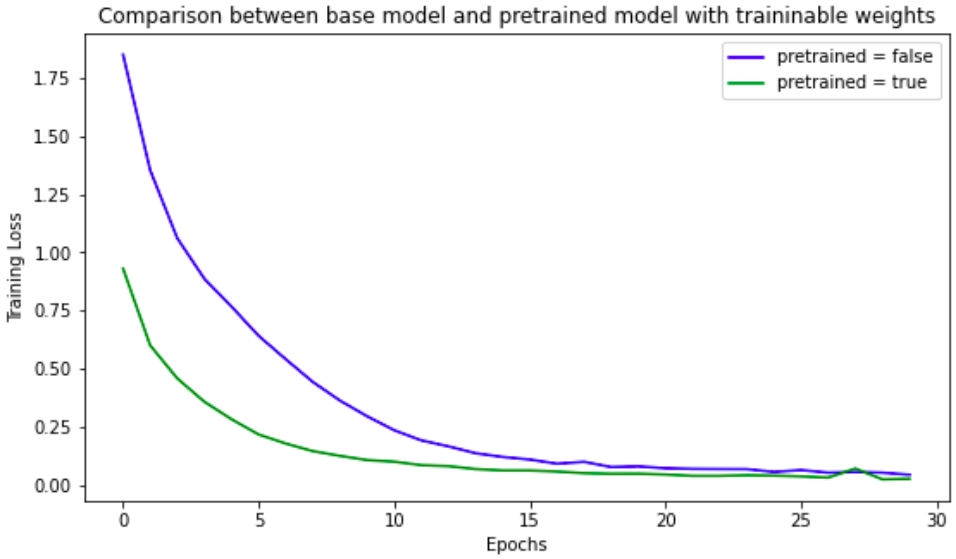





```
Training  Accuracy:   0.995
Validation Accuracy:  0.811
Testing Accuracy:     0.804
```

Comparison between pretrained = True and pretrained = False





As we can see, it is clearly evident that the training accuracy increases and loss decreases very fast in the pretrained model as compared to the fresh model which isn't pretrained. This shows the clear advantage of transfer learning since a lot of the initial layers of the pretrained model detect basic features such as edges and texture which is independent of the dataset and so the same learned layers can be used on other datasets as well. Therefore, the training is faster as there is less learning and the pretrained model very quickly adapts to the CIFAR10 dataset and reaches high accuracy very fast.

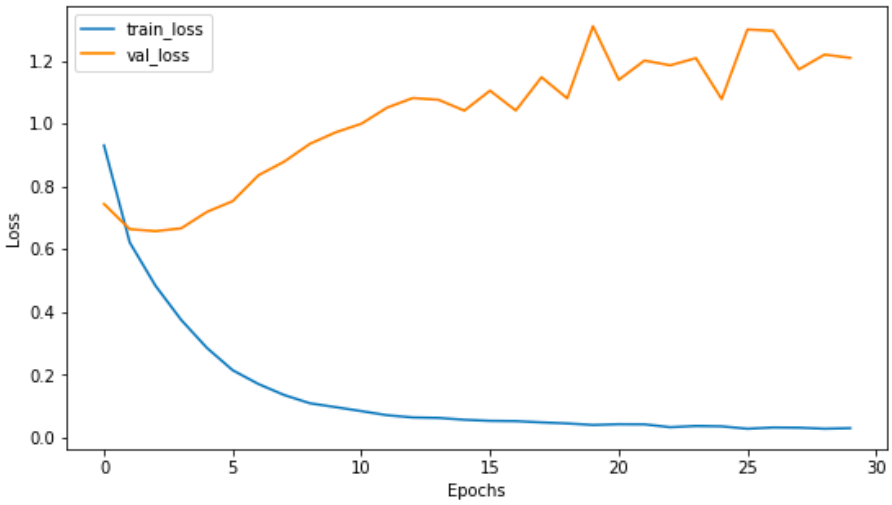
Also, the training on this model happened in 6 minutes 37 seconds.

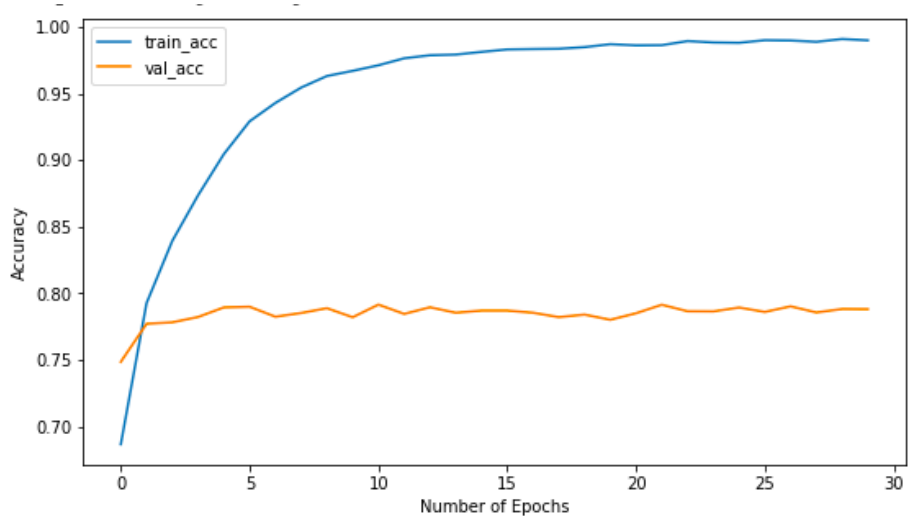
3) Config1: Pretrained Model but freezing first few layers so that their weights aren't changed

In this sub-experiment, we used the same pretrained model as above but this time before training, we fixed the weights of the first 7 layers by setting their "requires_grad" parameter as false. This is done to support the argument that the initial layers' weights trained are independent of the dataset.

As a further step, instead of the simple linear layer at the last step, we add an extra layer and dropout in the last fc layer of this model.
The fc layer added is as follows:

```
(fc): Sequential(
    (0): Linear(in_features=512, out_features=128, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.3, inplace=False)
    (3): Linear(in_features=128, out_features=10, bias=True)
  )
```






```
Training  Accuracy:   0.990
Validation Accuracy:  0.792
Testing Accuracy:     0.790
```

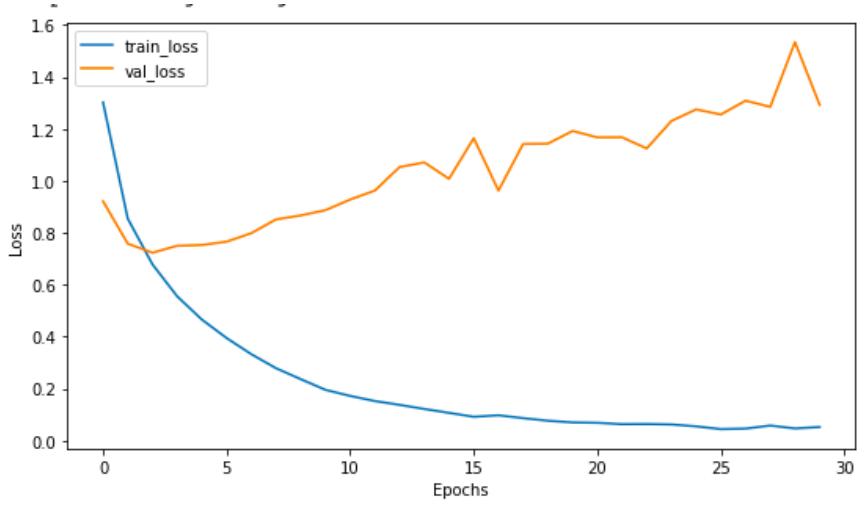
4) Config2: Pretrained Model but freezing first few layers so that their weights aren't changed

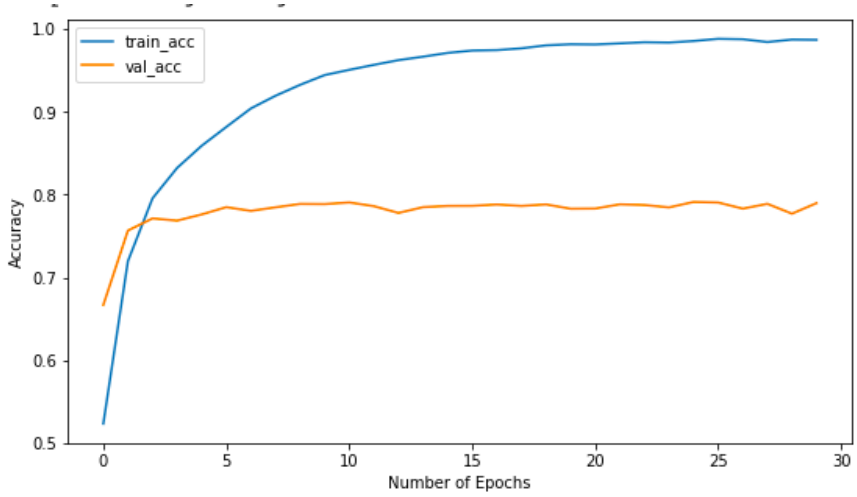
In this configuration, we have done everything similar to the previous configuration (Freezing first 7 layers' weights) but the difference is that this time, we have used the following layer as the last layer:

```
(fc): Sequential(
    (0): Linear(in_features=512, out_features=256, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=256, out_features=128, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.2, inplace=False)
    (6): Linear(in_features=128, out_features=128, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.2, inplace=False)
    (9): Linear(in_features=128, out_features=64, bias=True)
    (10): ReLU()
    (11): Dropout(p=0.2, inplace=False)
    (12): Linear(in_features=64, out_features=32, bias=True)
    (13): ReLU()
    (14): Linear(in_features=32, out_features=10, bias=True)
  )
```

Basically, we have added and entire deep neural network at the end of the resnet architecture. We have also added dropout with dropout parameter 0.2 after each layer to bring in regularisation.

The training on CIFAR10 using 30 epochs is as follows:

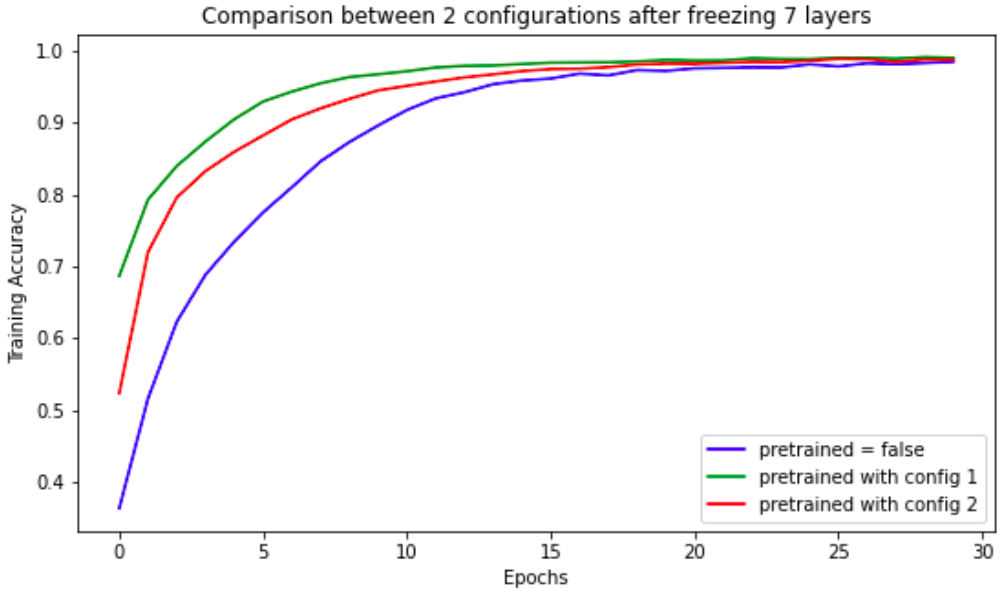


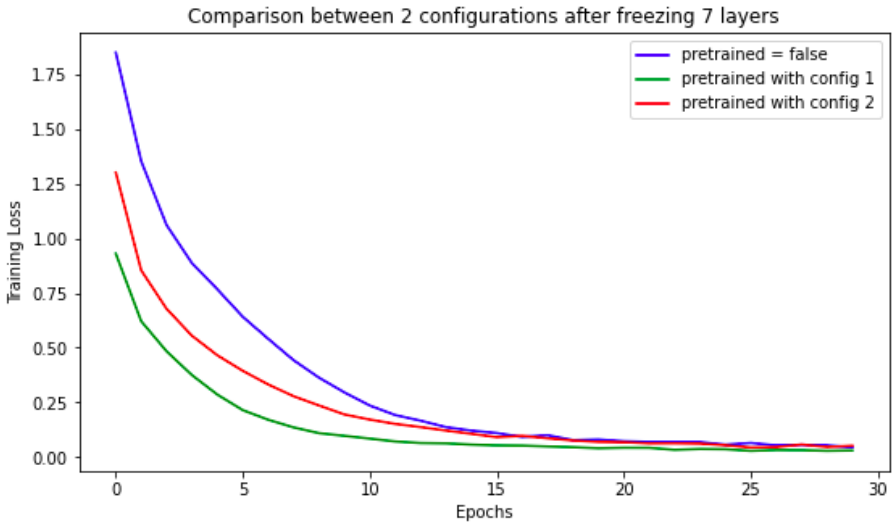



```
Training  Accuracy:   0.995
Validation Accuracy:  0.791
Testing Accuracy:     0.792
```

Comparison between Config1 and Config2 (as compared with the base model which isn't pretrained)

The comparison is summarised using the following training plots:

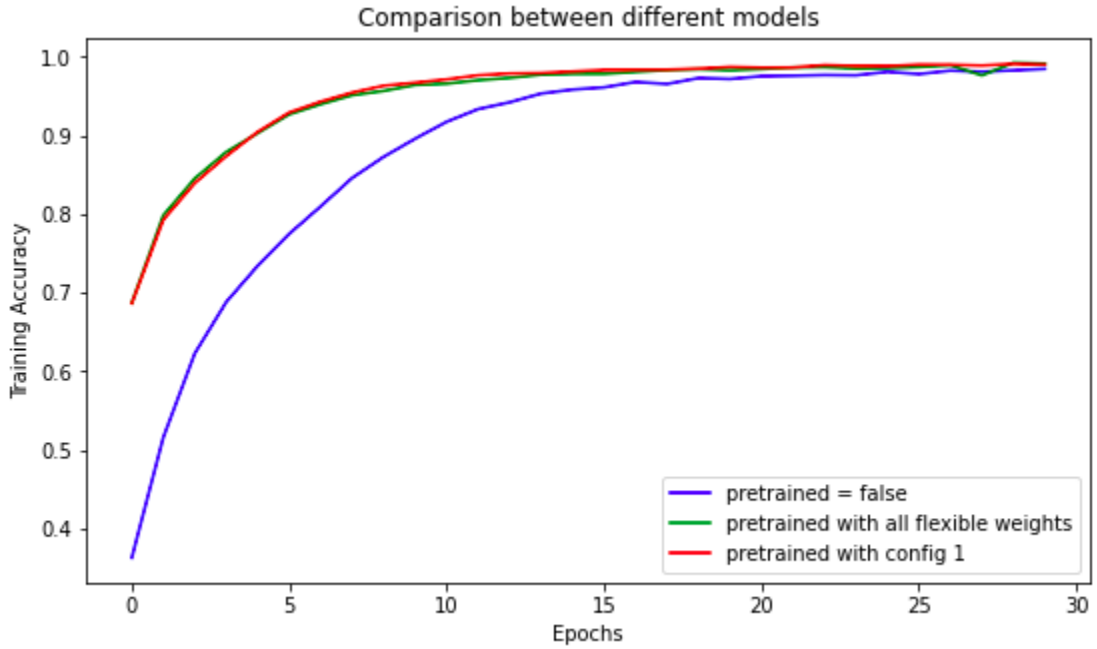


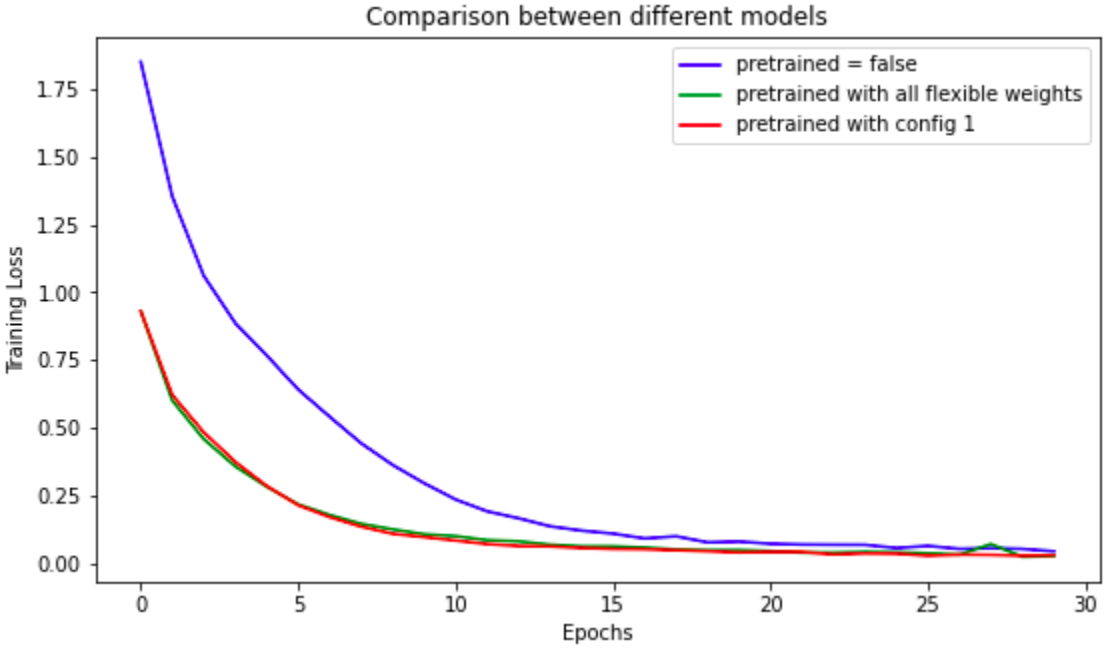


We can see clearly that the config 1 model performs the best and it performs way better than the base model. This is because in config 1, we have just frozen first 7 layers and have just added one linear layer in the end and so the training is very fast and happens over a very low number of epochs. In config 2, since we have added many more layers in the end, more epochs are required to train these new layers and so the training is slower. However, the thing to note is that the saturation accuracy and loss is the same in all the cases. Therefore, by using transfer learning, we aren't losing out on performance.

Final comparison between all the models

Since config 1 performs better than config2, we will use that for the final comparison.





We can see that the model in config1 performs almost similar to the pretrained model where all the weights are learnable. This supports our claim that the initial layers in transfer learning are independent of dataset and freezing or unfreezing them during training doesn't have an effect on the training or final performance of the model.

Part (c) Data Augmentation

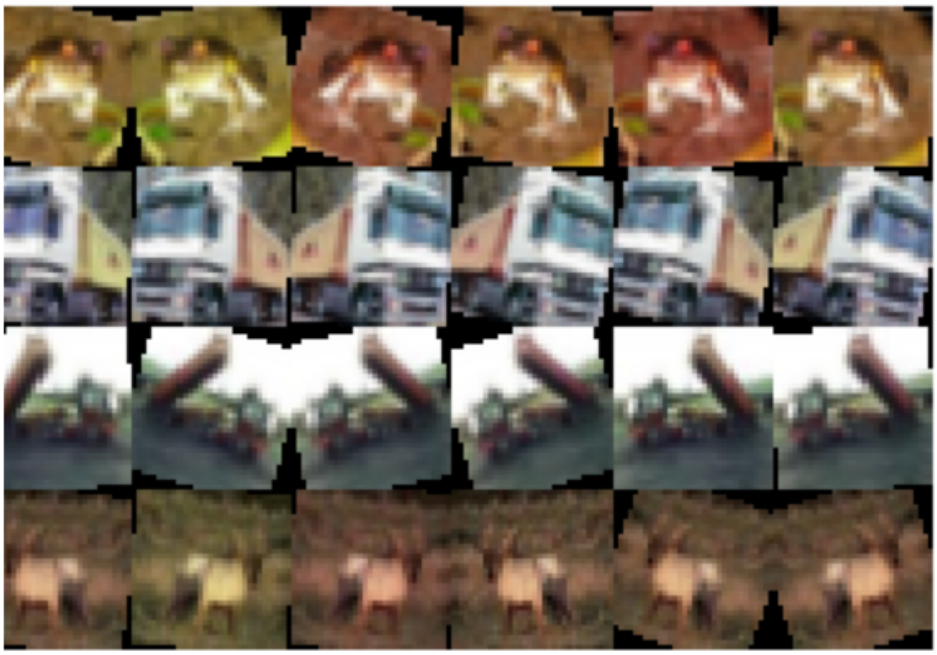
Data augmentation is a technique that enables us to significantly increase the diversity of data available for training models, without actually collecting new data. Different augmentation techniques are used to create variations of the images that can improve the ability of models to fit the training data and generalize what they have learned to new images.Data augmentation techniques like cropping, padding, rotation, and horizontal flipping are commonly used to train large neural networks. This helps deep learning algorithms, such CNN to learn features that are invariant to their location in the image. Augmentation can further help in this transform invariant approach to learning and can aid the model in learning features that are also invariant to transforms such as left-to-right to top-to-bottom ordering, light levels in photographs, and more. But, sometimes, applying heavy augmentations unnecessarily can result in poor accuracy. It is generally advisable to not use data augmentation in test set, as we don't train over test set.

Some of the different data augmentation techniques used and their visualisations are:

1. RandomRotation(degrees) : Rotates the image by angle.
   degrees : Range of degrees to select from. The network has to recognize the object present in any orientation.

2. RandomHorizontalFlip(p) : Horizontally flip the given PIL Image randomly with a given probability p. This scenario is important for the network to remove biasness of assuming certain features of the object are available in only a particular side.

3. ColorJitter(brightness=0, contrast=0, saturation=0, hue=0) : Randomly changes the brightness, contrast and saturation of an image. The parameters describe how much to jitter. This is a very important type of diversity needed in the image dataset not only for the network to properly learn the object of interest but also to simulate the practical scenario of images being taken by the user.



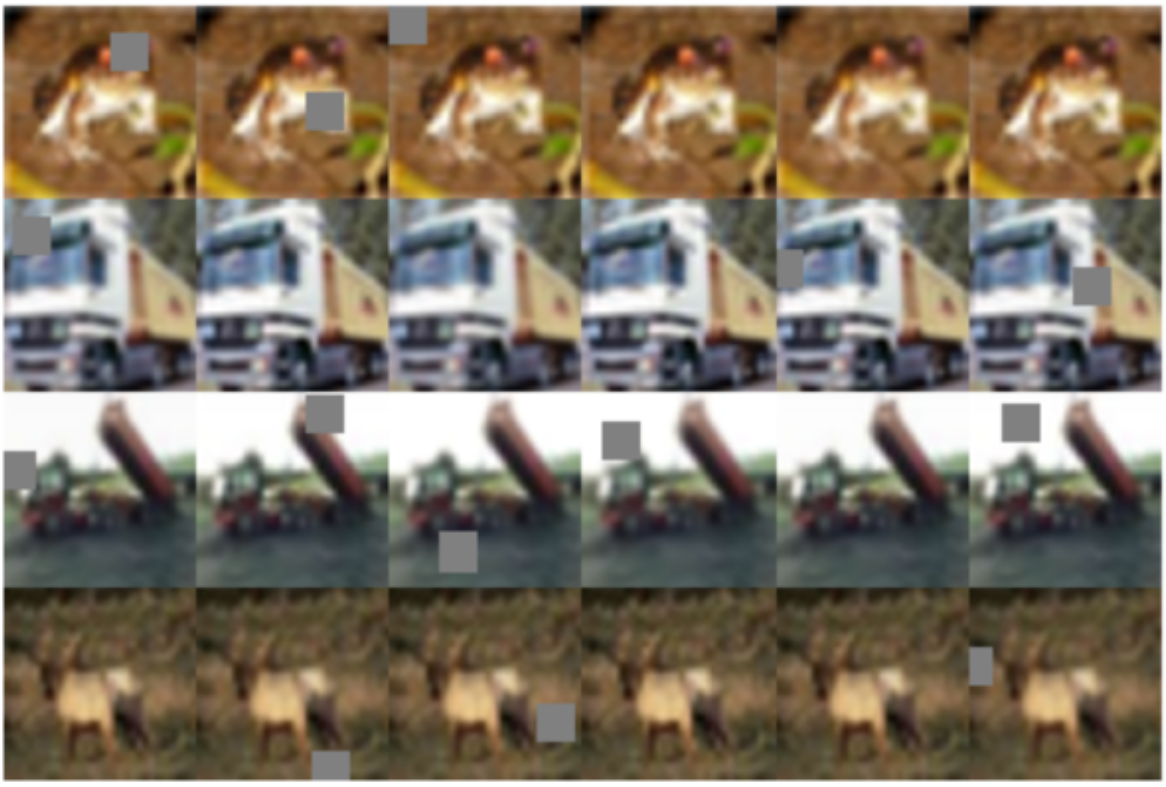
4. RandomResizedCrop(size) : Crop the given Image to random size and aspect ratio. size – expected output size of each edge

5. Resize(size) : Resize the input PIL Image to the given size.

6. CenterCrop(size): Crops the given PIL Image at the center.

7. Cutout is a simple regularization method for convolutional neural networks which consists of masking out random sections of input images during training. This technique simulates occluded examples and helps the model to take more minor features into consideration when making decisions, rather than relying on the presence of a few major features. Cutout has advantages like, we can simulate the situation when the subject is partially occluded. It can also promote the model to make full use of more content in the image for classification, and prevent the network from focusing only on the saliency area, thereby causing overfitting.

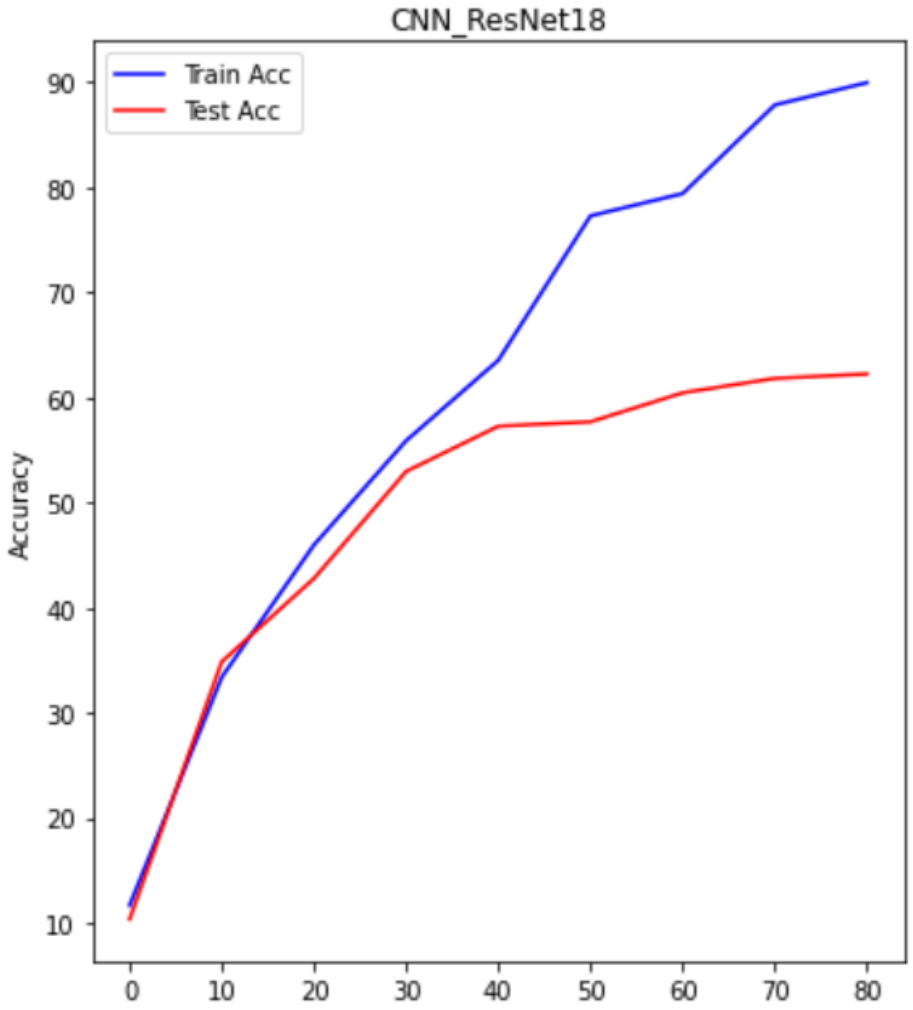
8. Random Gaussian Blur is an image data augmentation technique where we randomly blur the image using a Gaussian distribution. Deliberately introducing imperfections into the datasets is essential to making our machine learning models more resilient to the harsh realities they'll encounter in real world situations. Degrading image quality can be completed in post processing – without the problems of collecting more data and needing to label it.

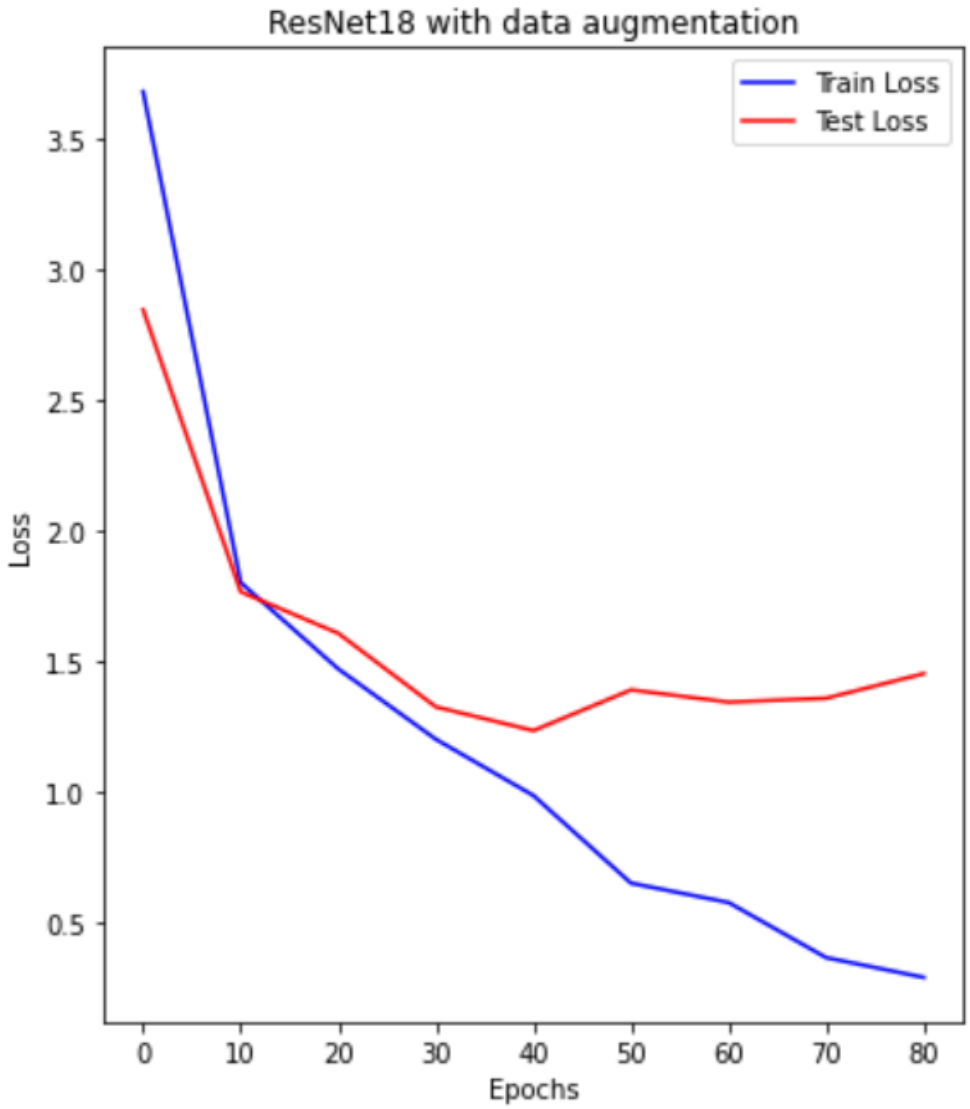


9. Salt and Pepper noise:
   Salt and Pepper noise is the addition of white and black dots to the image. Though this may seem unnecessary, it is important to remember that a general user who is taking images to feed into your network may not be a professional photographer. His camera can produce blurry images with lots of white and black dots. This augmentation helps the such users keeping in mind these realistic conditions.

Results

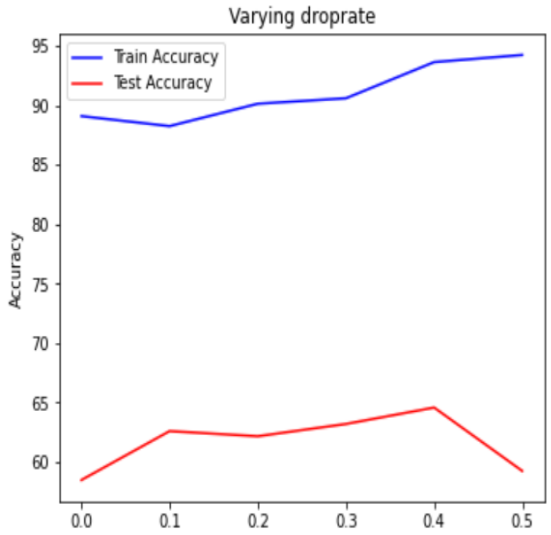
Classification Accuracy and Error on training and test data of the Tiny cifar10 dataset after applying various data augmentation techniques is as follows:

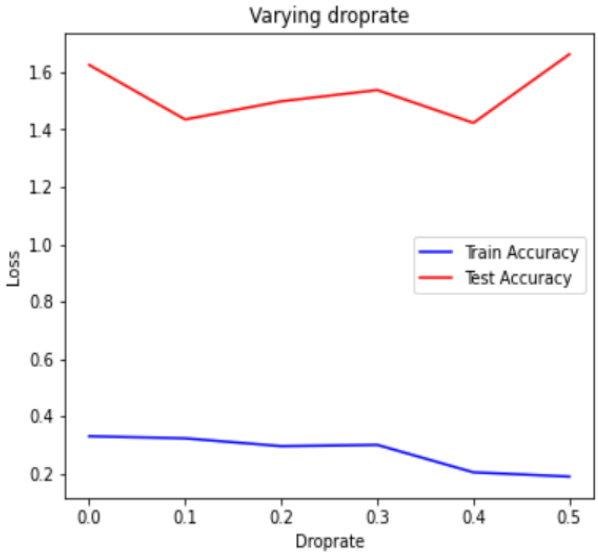




Implementing Dropout:

Dropout after different layers was implemented where rate is the fraction of units to drop. Dropout rate was varied from 0 to 0.5 in steps of 0.1 and the training accuracy and loss was compared after training on 80 epochs. The plots of test accuracy and loss are shown below.



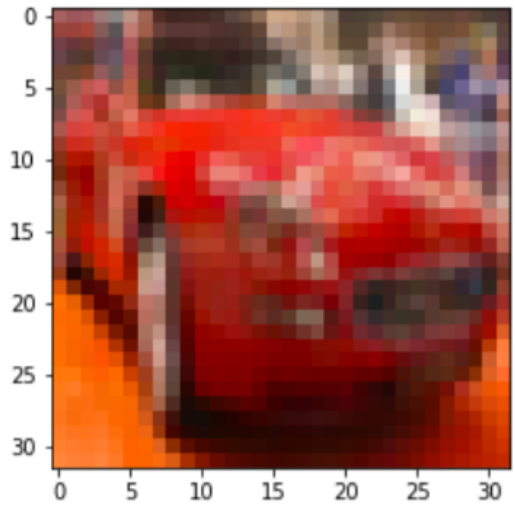


Part (d)

Visualising Activations in different layers

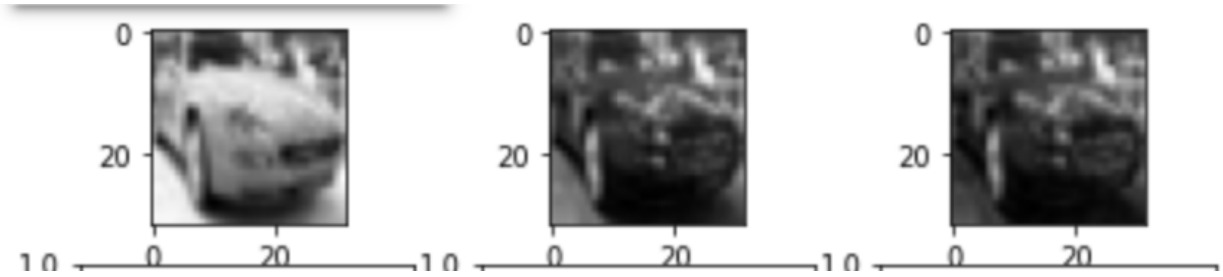
After training on the CIFAR10 model after applying transfer learning, we visualise some activations in different layers

Let’s take a example image of a car in the CIFAR 10 dataset
Visualising the original image:

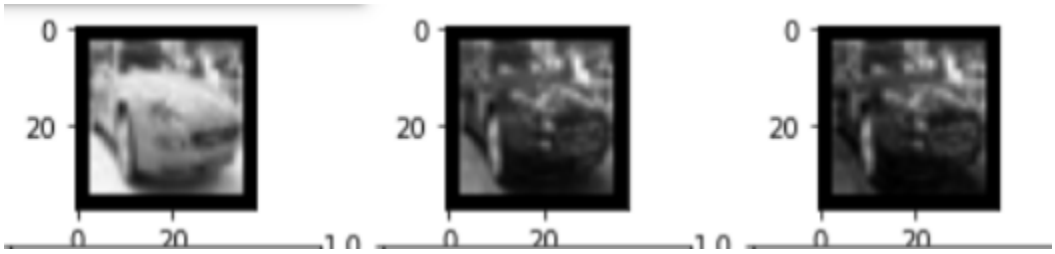


Now we pass this through the network and visualise the activations in the different layers. The activations have been converted to grayscale and have been distributed across channels.

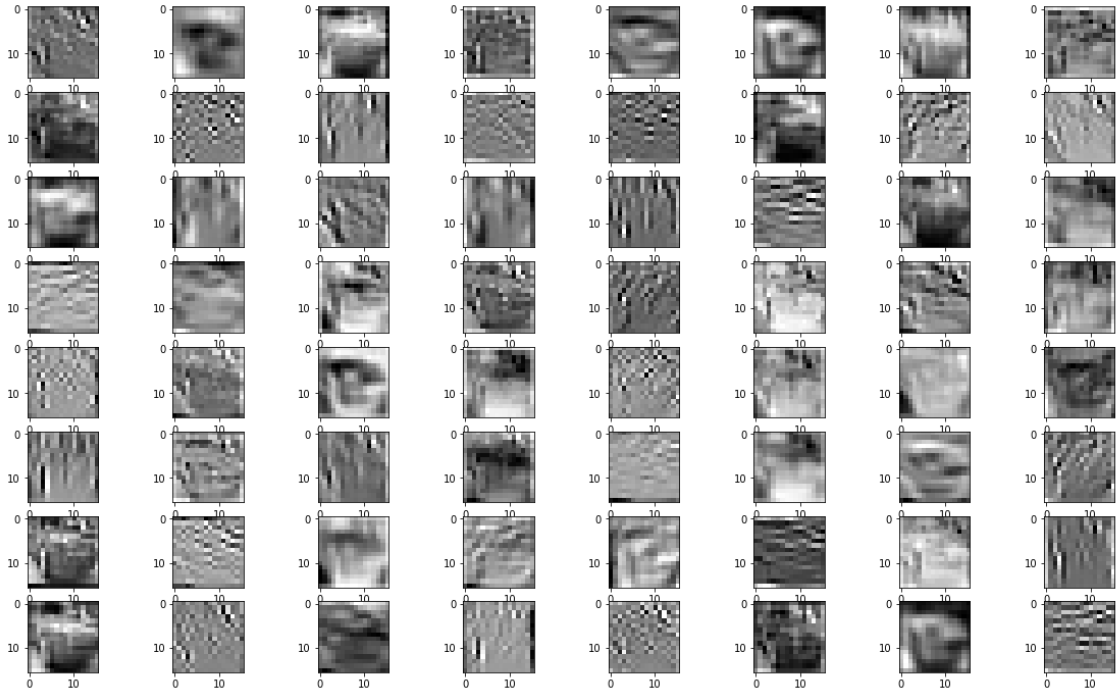
Layer 0: The original image has been divided into its channels



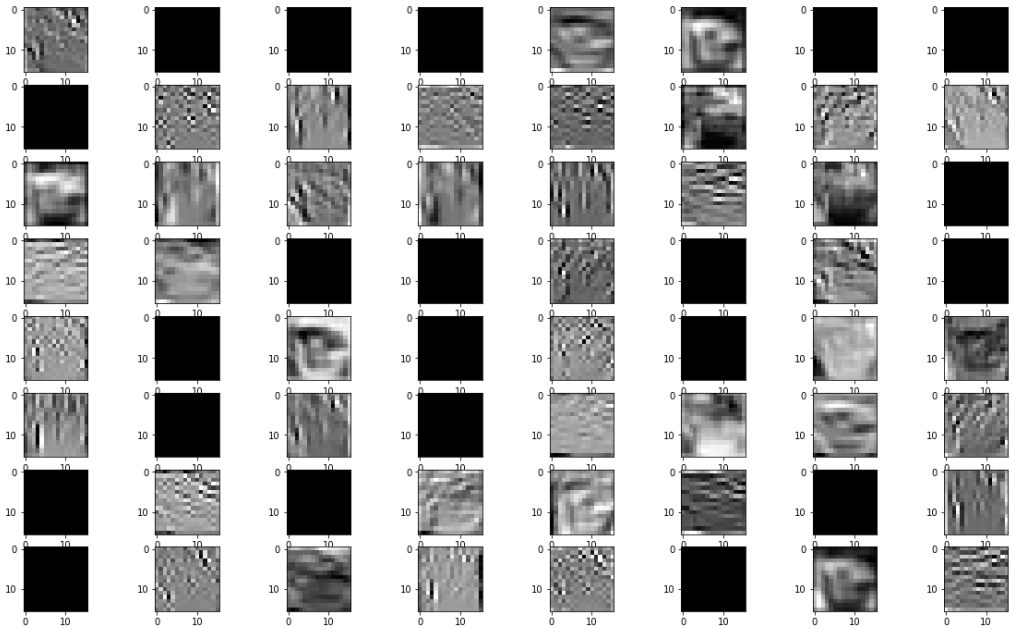
After first zero padding layer:



After first Conv2D layer: The result after the first Conv2D block are as shown (before this there was also batchnorm layer)

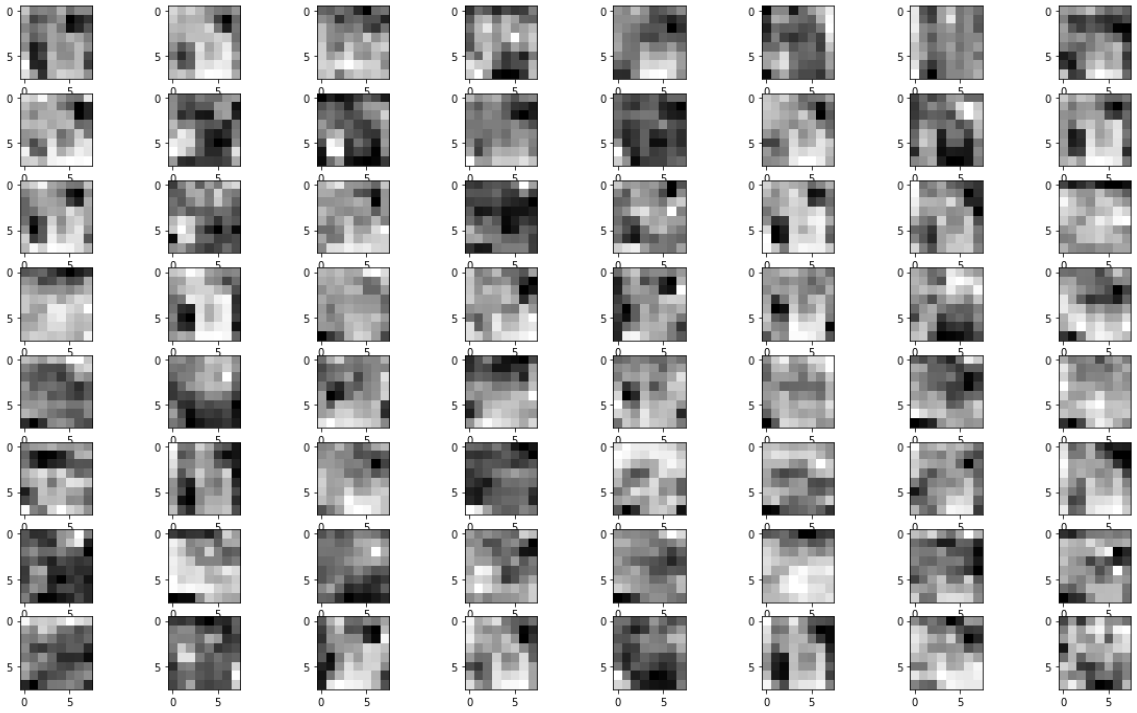


Activations after the first ReLU layer in the first block:

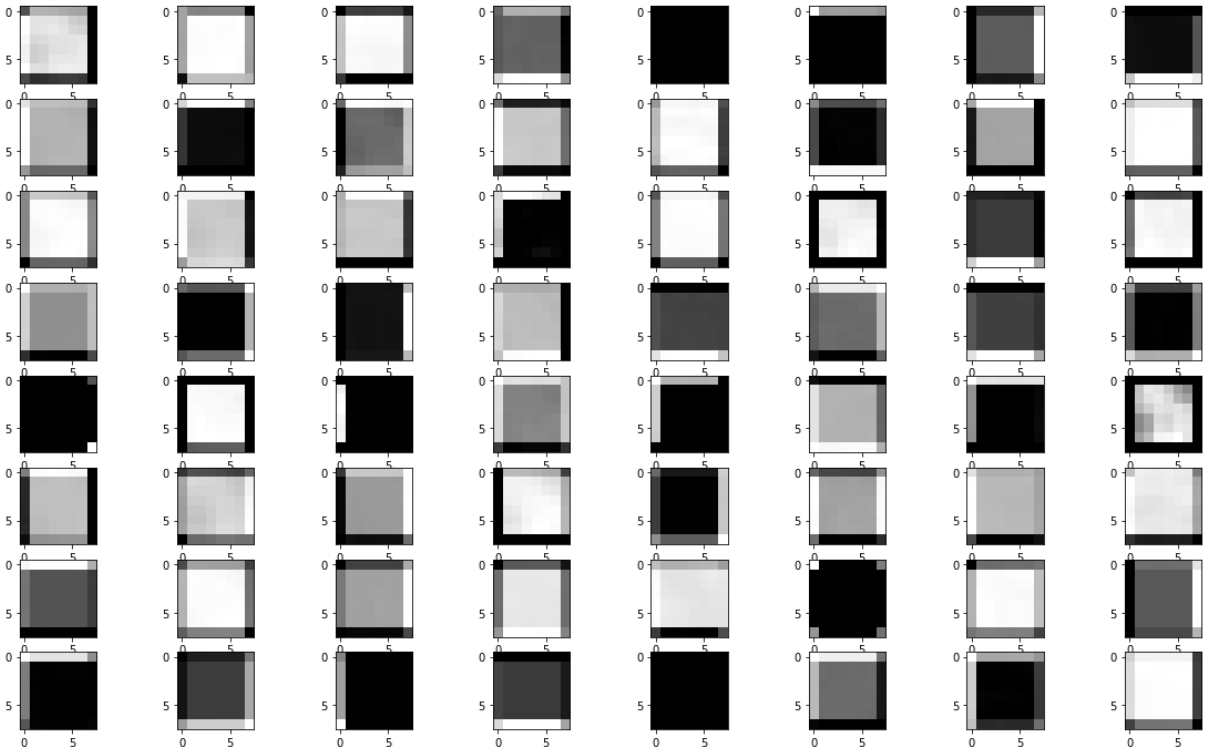


Some of the activations are black due to dead weights in these layers that havent been learned.

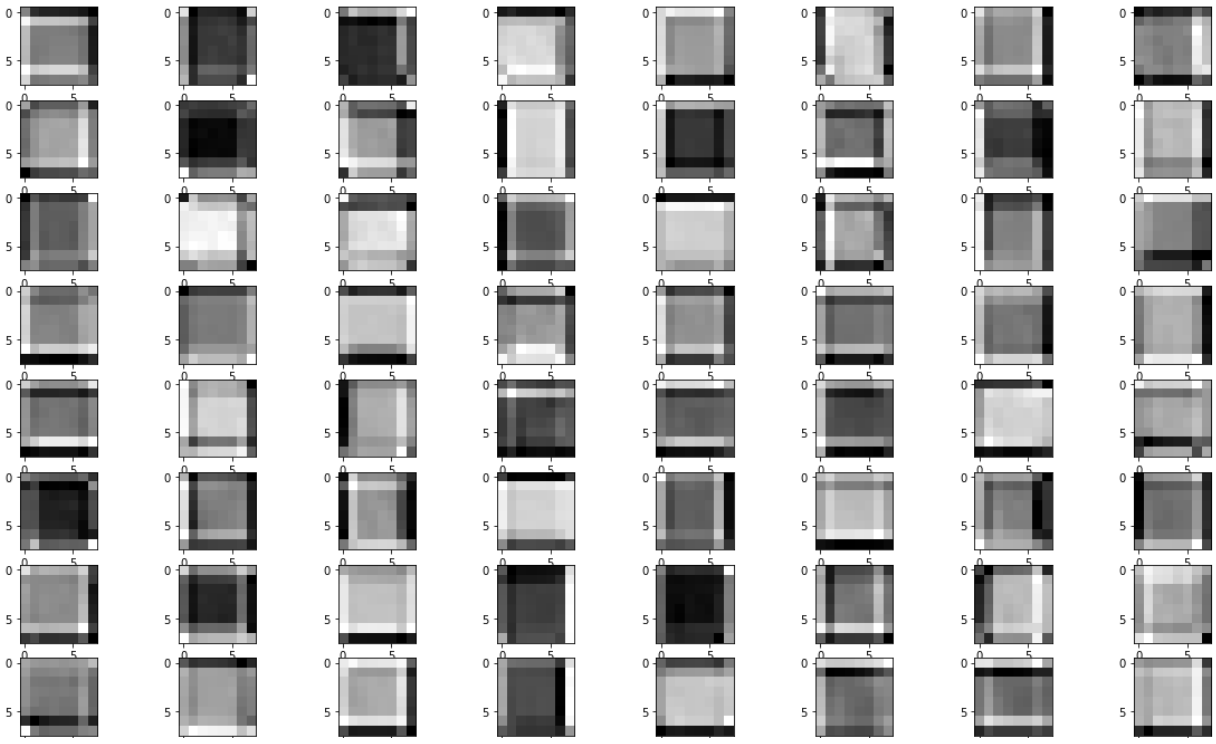
After second Conv2D layer



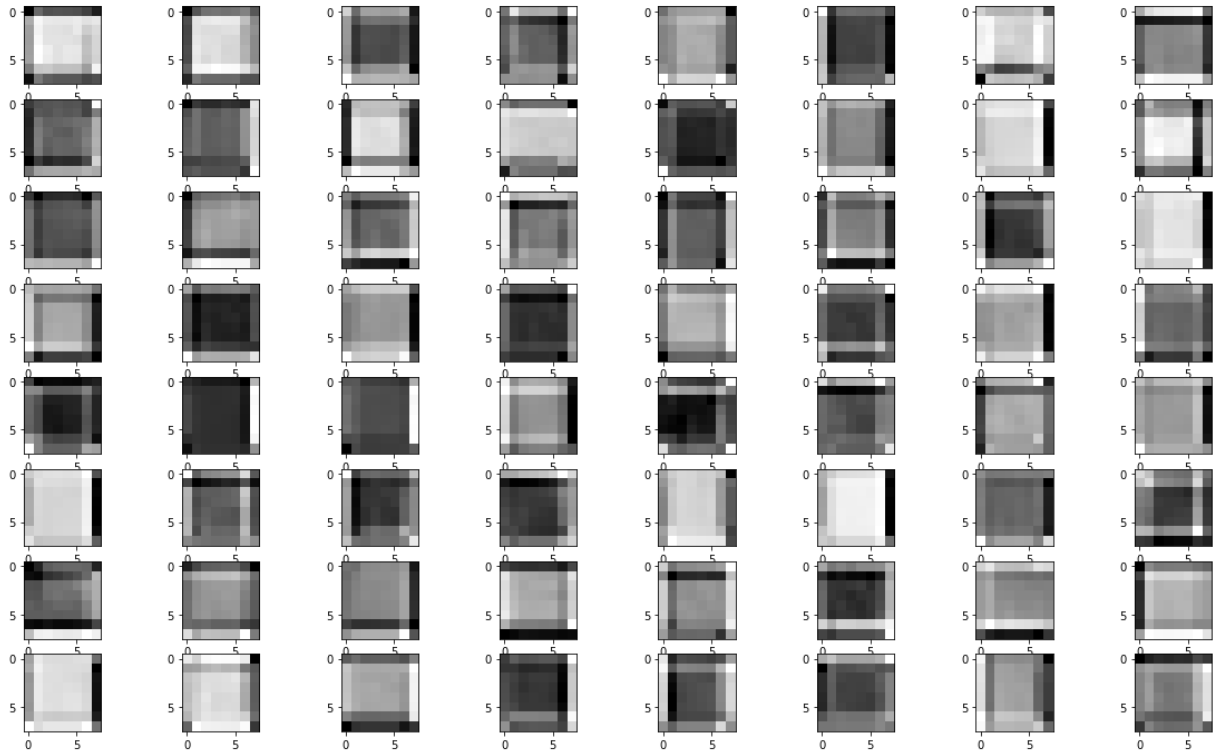
After 2 complete blocks of applying all operations such as maxpooling, batch norm, activations and Conv2D twice:



Activations after 3rd block ReLU:



Activations after the 15th layer:



We can see that the activations in the first few layers have some of the features of the original image but after a few layers the image features have been totally lost and the CNN architecture is learning from very minute level features that are indiscernible to the human eye. This shows the complexity of the entire Resnet model.