REPORT ON

Cifar 10 image classfication using convolutional neural network

MOHD ZAHID FAIZ : MT2019514
SUBARNA KANTI SAMANTA : MT2019523
PRANJAL KUMAR : MT2019079

UNDER GUIDANCE OF:
PROF. DINESH BABU JAYAGOPI

Introduction:

The images in the cifar 10 are really small and pixalated to make any sense using the classical ML driven techniques. Another approach is used for this task. Images are fed to a neural network and let it learn the features out of it. The CNN layer learns the features and the FCC layer classifies those features according to the correct classes. According to the Alexnet paper, with the increase in the depth of the network the accuracy improves but the training time increases. This is what is explored in the following.

Cifar-10 dataset:

The dataset used for this purpose is CIFAR-10 consisting of 50000 of 10 classes with 5000 images per class.

---data -- a 10000x3072 numpy array of uint8s. Each row of the array stores a 32x32 colour image. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue. The image is stored in row-major order, so that the first 32 entries of the array are the red channel values of the first row of the image.

---labels -- a list of 10000 numbers in the range 0-9. The number at index *i* indicates the label of the *i*th image in the array data.

The images are read and reshaped, rescaled and resized with dtype ‘float32’ with the test train split as shown below,

```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

Model architecture:

The base block used is the VGG block. These blocks are stacked over one another to see how the accuracy and model behave in terms of training time. A single VGG block contains a convolution layer with an activation function followed again by the same convolution layer and then a max-pooling layer.

The depth of the kernel increases with the increase in the number of VGG blocks.

Example,

VGG block -	1	2	3	4
Kernel depth-	32	64	128	256

But with the increase in the depth of the layer the number of parameters to be learned increases exponentially, which increases in the training and prediction time. Also, the model could overfit with more layers. The task is to reach those number of layers which takes minimum time to train but still doesn't overfit but on the contradictory generalizes well. To monitor the model performance in all those terms **Tensorboard** is used. The call back created keeps the logs of the training and validation accuracy and time for each epochs. The model is trained with the GPU in google colab, which results in a lot less training time. And no feature engineering and data augumantation is done. But the parameters that were changed are,

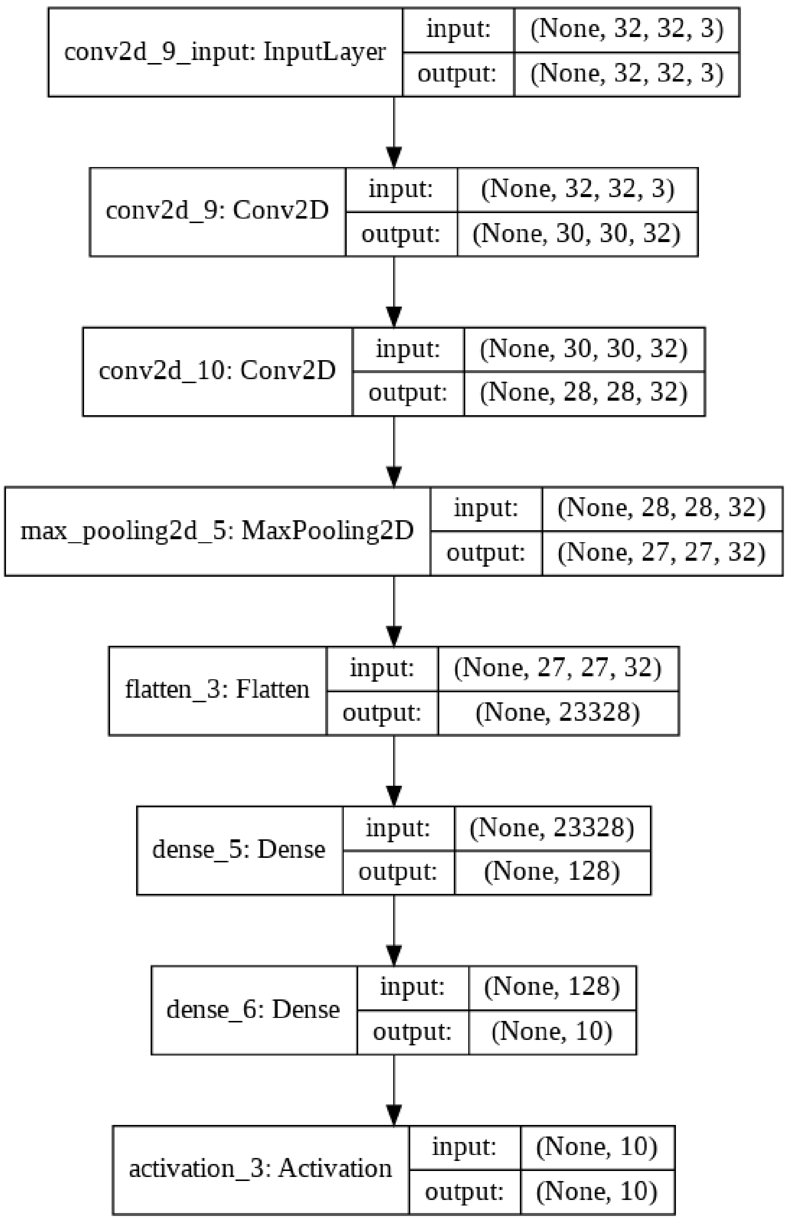
1. Model architecture.

2. Batch norm

3. Optimizers

Selecting the best model architechture,

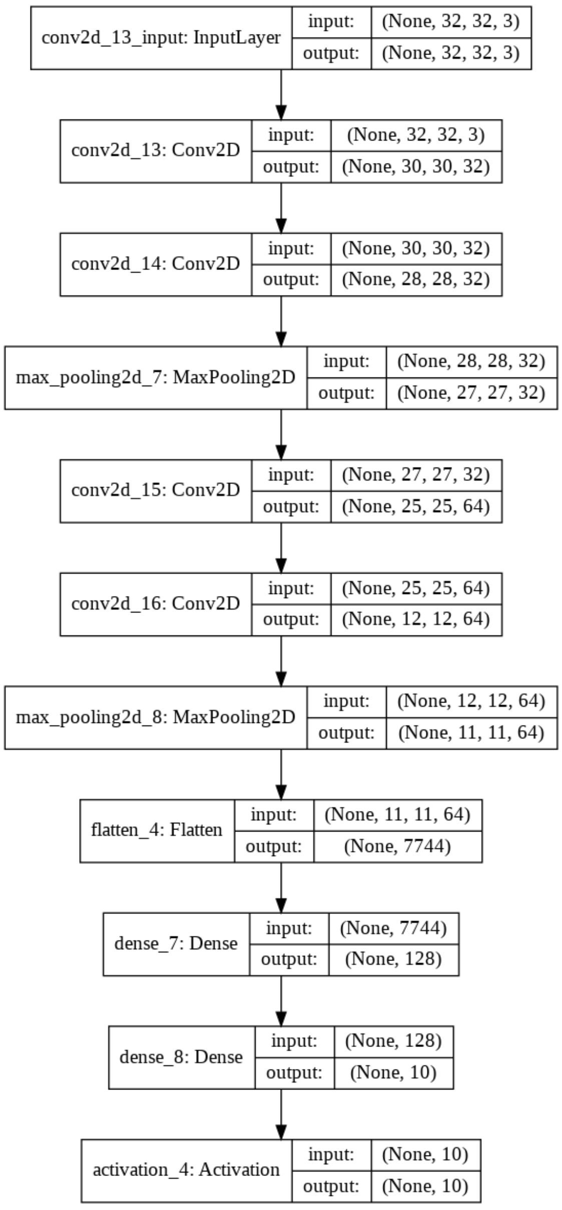
A single block of VGG is selected one FCC layer as shown below,



The model is then run and the accuracy is 66.7% and the model took an average of 5 second to train on 32830 sample ie 520 micro second per sample. The accuracy for the start is good. The model is then checked if stacking more VGG block effects the accuracy or not.

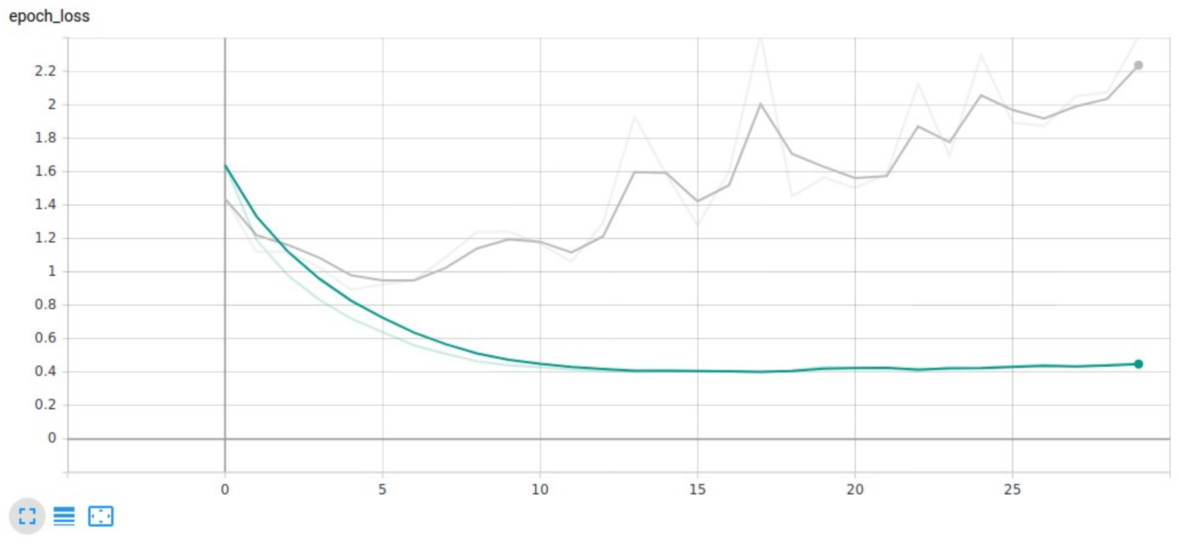
2 blocks of VGG with 2 FCC layers are used

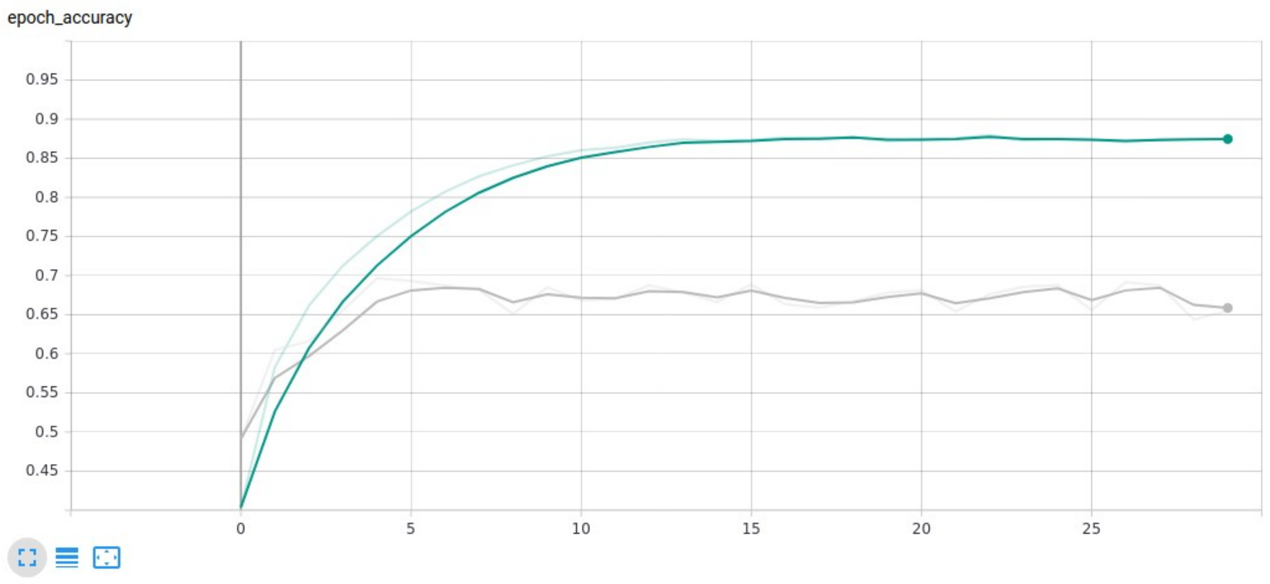
The architecture shown below,


conv2d_13_input: InputLayer | input: (None, 32, 32, 3) | output: (None, 32, 32, 3)
conv2d_13: Conv2D | input: (None, 32, 32, 3) | output: (None, 30, 30, 32)
conv2d_14: Conv2D | input: (None, 30, 30, 32) | output: (None, 28, 28, 32)
max_pooling2d_7: MaxPooling2D | input: (None, 28, 28, 32) | output: (None, 27, 27, 32)
conv2d_15: Conv2D | input: (None, 27, 27, 32) | output: (None, 25, 25, 64)
conv2d_16: Conv2D | input: (None, 25, 25, 64) | output: (None, 12, 12, 64)
max_pooling2d_8: MaxPooling2D | input: (None, 12, 12, 64) | output: (None, 11, 11, 64)
flatten_4: Flatten | input: (None, 11, 11, 64) | output: (None, 7744)
dense_7: Dense | input: (None, 7744) | output: (None, 128)
dense_8: Dense | input: (None, 128) | output: (None, 10)
activation_4: Activation | input: (None, 10) | output: (None, 10)

This architechture takes the same time to train but it has a better profrmance when trained for the same number of epochs, that is 68.5%. Adding **one more VGG block** to the architechture improves the accuracy to 70.569% and the training time almost same per epochs. Now there are three VGG blocks with 6 convolution layers and 3 FCC layers. So, now the model is evaluated in terms of how well it is generaliszing.

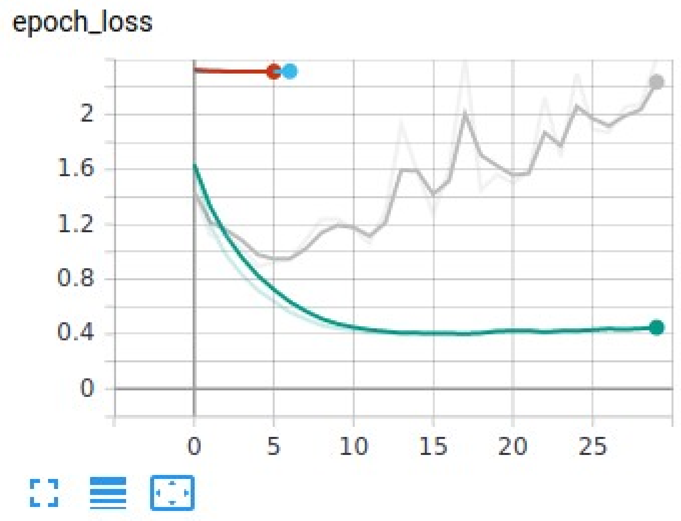
The below graphs shows the training accuracy/error and the validation error for the Three VGG blocks, with per block 2 CNN layers,





The gray line in the graph is for the validation and the blue line is for the training. Clearly the model is overfitting. To rectify this first is to change the activation function and see how well the model behaves,

Tanh activation function:

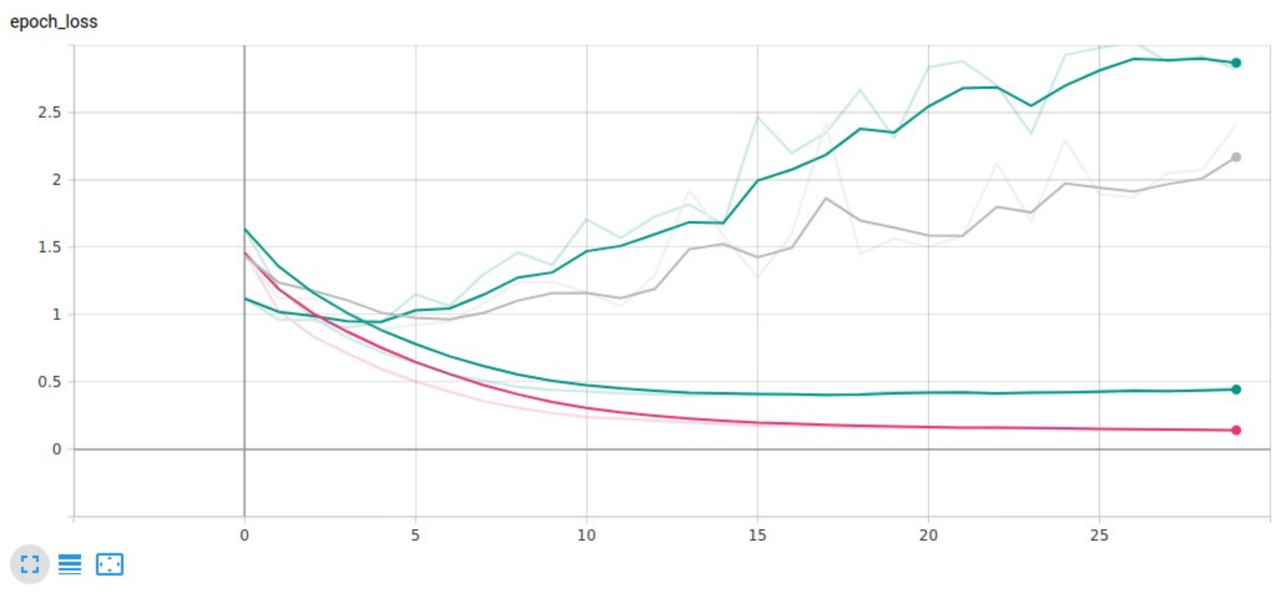


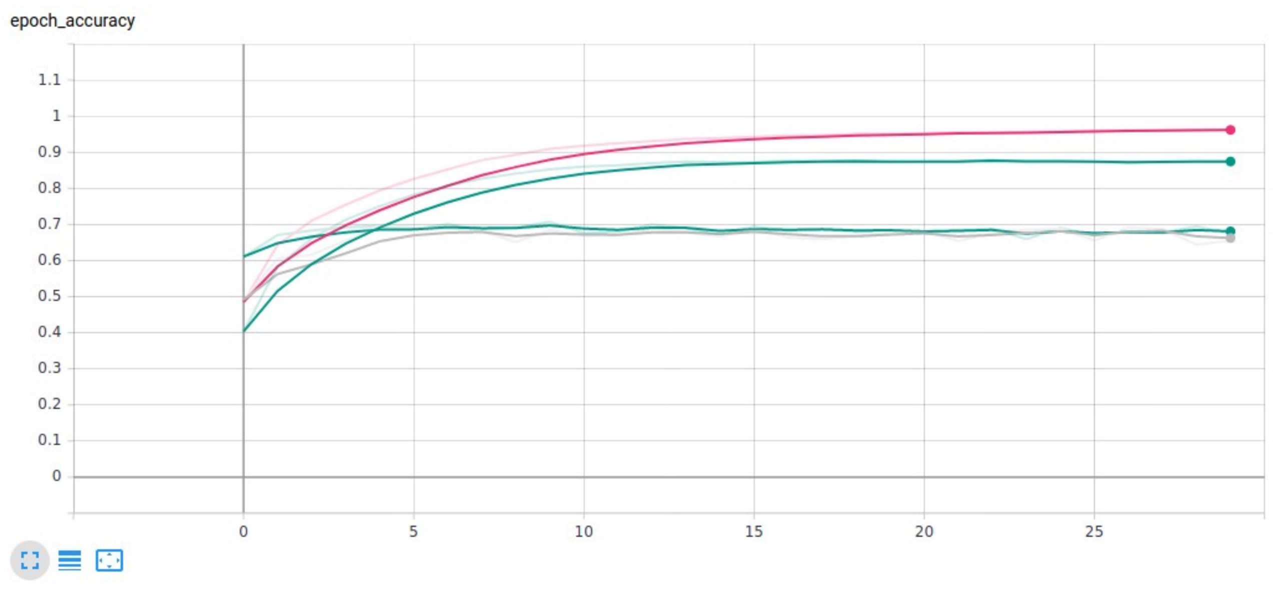
The red line in the top left corner is for the tanh activation function. And the previous graph is plotted with it for the comparison. The model is not able to learn anything at all. So, the activation function is again changed. With the leakyRelu, the model can learn quickly but is still overfitting. To rectify the overfitting problem regularisation is used. Techniques used in this to regularise the model are

1. **Dropout**: Since the model is overfitting to the features which are not always present in the dataset hence dropping out some of the weights helped to tackle this problem.
2. **Kernel weight regularisation**: Weights are regularised to keep its value as small as possible so that the model could generalize well.

The below graphs shows the model performace with activation function as LeakyReLU and the dropout techniques,

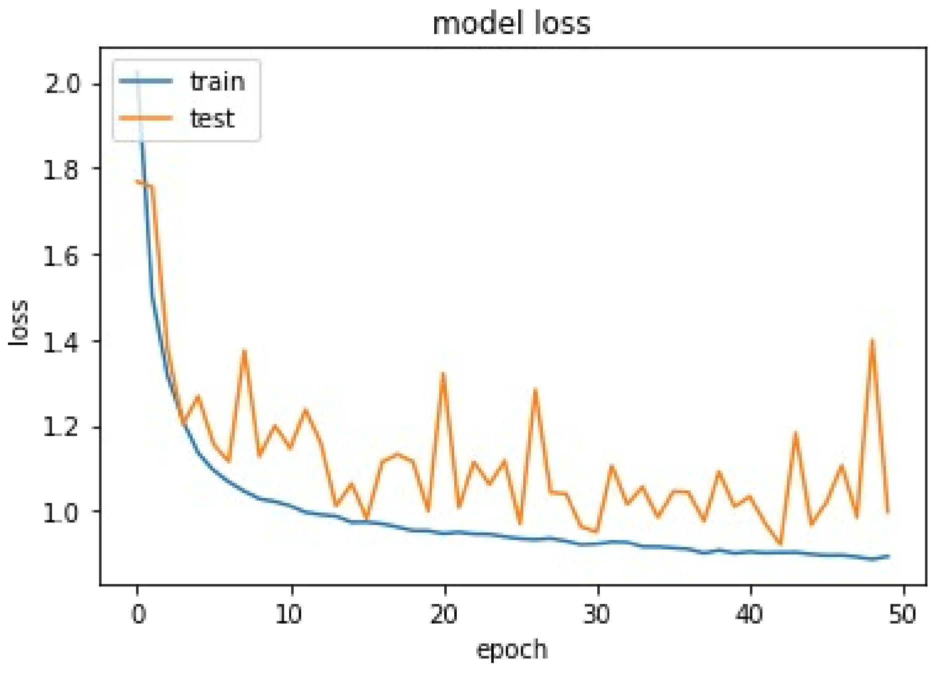
Graphs before applying the regularization.

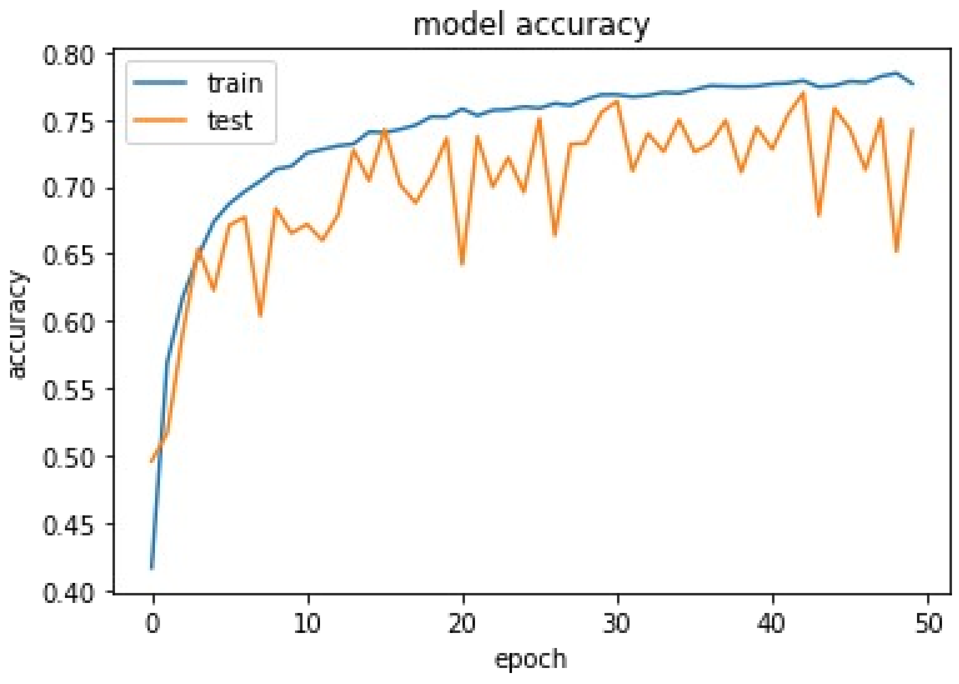




Clearly the with the leaky ReLU the model is able to learn fast but still not able to generalize as there is a big gap between the validation and train accuracy.

Graphs after applying the regularization,

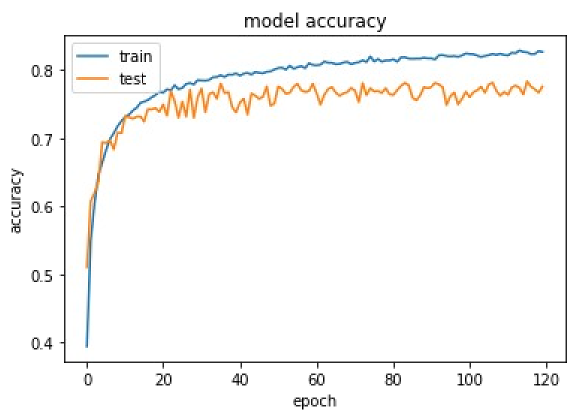


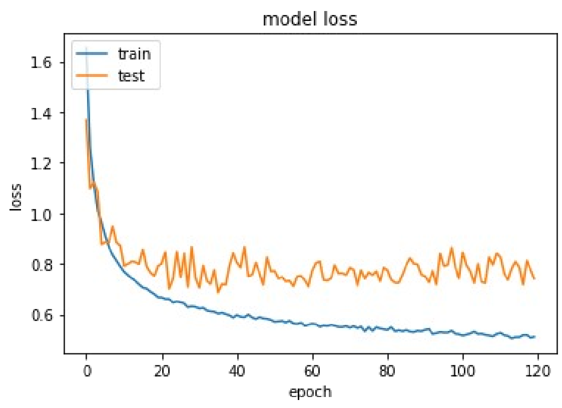


The model is still not able to optimise but it fluctuating, so the model is trained with a better optimizer other then **RMSprop.**

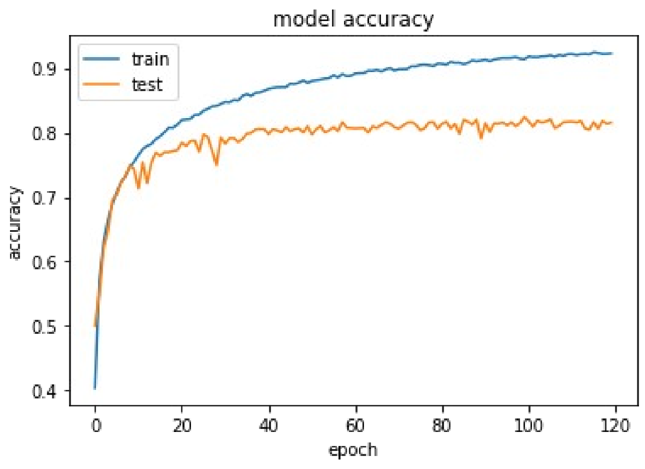
Other then that batch normalization is also introduced to deal with the covarience shift between the CNN layers,

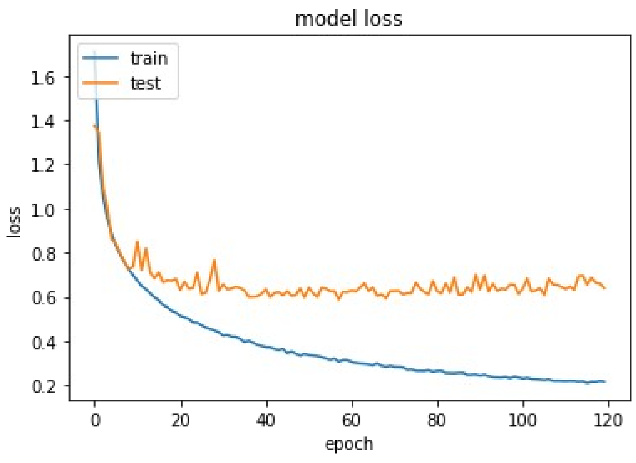
Without batch norm,





With batch norm,



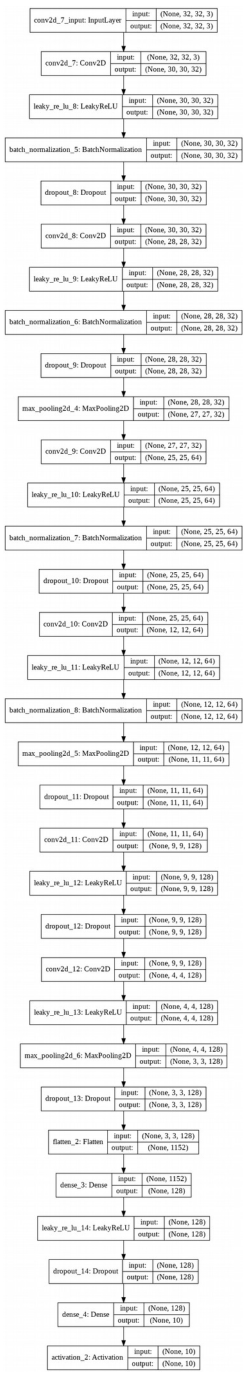


With batch norm the model is able to learn fast, and with the improved optimization of ADAM the model is able to learn smoothly.

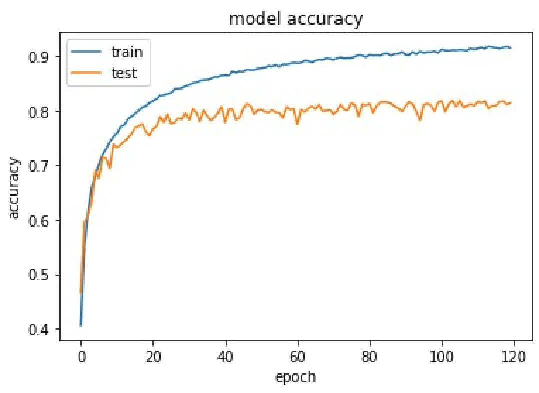
<u>Conclusion:</u>

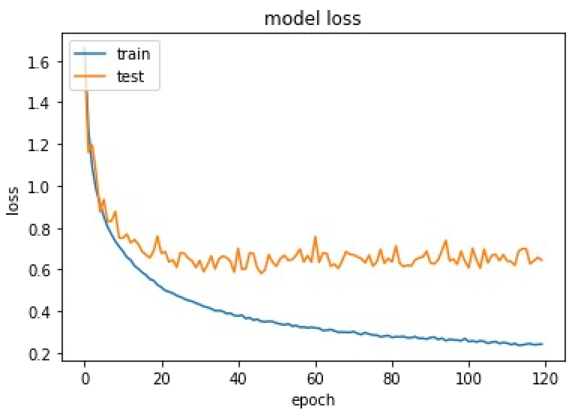
The final architecture for the task is
3 VGG blocks with 2 CNN block each layer with increasing kernel depth 32, 32, 64, 64, 128, 128
with maxpooling layer after 2 CNN blocks and batch norm after each layer.
and 2 FCC layers with 128 , 10 conncetions,

The architechture is shown below, (also included in the code)

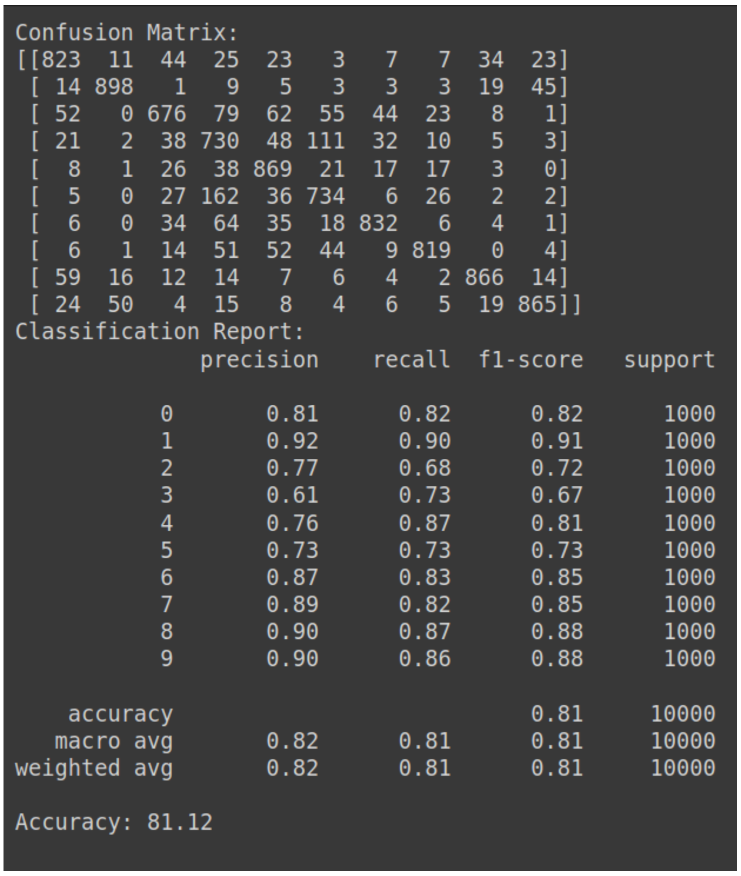

conv2d_7_input: InputLayer input: (None, 32, 32, 3) output: (None, 32, 32, 3)
conv2d_7: Conv2D input: (None, 32, 32, 3) output: (None, 30, 30, 32)
leaky_re_lu_8: LeakyReLU input: (None, 30, 30, 32) output: (None, 30, 30, 32)
batch_normalization_5: BatchNormalization input: (None, 30, 30, 32) output: (None, 30, 30, 32)
dropout_8: Dropout input: (None, 30, 30, 32) output: (None, 30, 30, 32)
conv2d_8: Conv2D input: (None, 30, 30, 32) output: (None, 28, 28, 32)
leaky_re_lu_9: LeakyReLU input: (None, 28, 28, 32) output: (None, 28, 28, 32)
batch_normalization_6: BatchNormalization input: (None, 28, 28, 32) output: (None, 28, 28, 32)
dropout_9: Dropout input: (None, 28, 28, 32) output: (None, 28, 28, 32)
max_pooling2d_4: MaxPooling2D input: (None, 28, 28, 32) output: (None, 27, 27, 32)
conv2d_9: Conv2D input: (None, 27, 27, 32) output: (None, 25, 25, 64)
leaky_re_lu_10: LeakyReLU input: (None, 25, 25, 64) output: (None, 25, 25, 64)
batch_normalization_7: BatchNormalization input: (None, 25, 25, 64) output: (None, 25, 25, 64)
dropout_10: Dropout input: (None, 25, 25, 64) output: (None, 25, 25, 64)
conv2d_10: Conv2D input: (None, 25, 25, 64) output: (None, 12, 12, 64)
leaky_re_lu_11: LeakyReLU input: (None, 12, 12, 64) output: (None, 12, 12, 64)
batch_normalization_8: BatchNormalization input: (None, 12, 12, 64) output: (None, 12, 12, 64)
max_pooling2d_5: MaxPooling2D input: (None, 12, 12, 64) output: (None, 11, 11, 64)
dropout_11: Dropout input: (None, 11, 11, 64) output: (None, 11, 11, 64)
conv2d_11: Conv2D input: (None, 11, 11, 64) output: (None, 9, 9, 128)
leaky_re_lu_12: LeakyReLU input: (None, 9, 9, 128) output: (None, 9, 9, 128)
dropout_12: Dropout input: (None, 9, 9, 128) output: (None, 9, 9, 128)
conv2d_12: Conv2D input: (None, 9, 9, 128) output: (None, 4, 4, 128)
leaky_re_lu_13: LeakyReLU input: (None, 4, 4, 128) output: (None, 4, 4, 128)
max_pooling2d_6: MaxPooling2D input: (None, 4, 4, 128) output: (None, 3, 3, 128)
dropout_13: Dropout input: (None, 3, 3, 128) output: (None, 3, 3, 128)
flatten_2: Flatten input: (None, 3, 3, 128) output: (None, 1152)
dense_3: Dense input: (None, 1152) output: (None, 128)
leaky_re_lu_14: LeakyReLU input: (None, 128) output: (None, 128)
dropout_14: Dropout input: (None, 128) output: (None, 128)
dense_4: Dense input: (None, 128) output: (None, 10)
activation_2: Activation input: (None, 10) output: (None, 10)

The accuracy for the model is 81.12 for the test set and the training time is an average of 23 seconds per epoc with
701 micro second/step. And the validation and training accuracy as shown below,





With just 20 epochs the model is able to reach 75+ accuracy in validation set.This is the last report on how well the model classifies on each class,



```
Confusion Matrix:
[[823  11  44  25  23   3   7   7  34  23]
 [ 14 898   1   9   5   3   3   3  19  45]
 [ 52   0 676  79  62  55  44  23   8   1]
 [ 21   2  38 730  48 111  32  10   5   3]
 [  8   1  26  38 869  21  17  17   3   0]
 [  5   0  27 162  36 734   6  26   2   2]
 [  6   0  34  64  35  18 832   6   4   1]
 [  6   1  14  51  52  44   9 819   0   4]
 [ 59  16  12  14   7   6   4   2 866  14]
 [ 24  50   4  15   8   4   6   5  19 865]]
Classification Report:
              precision    recall  f1-score   support

           0       0.81      0.82      0.82      1000
           1       0.92      0.90      0.91      1000
           2       0.77      0.68      0.72      1000
           3       0.61      0.73      0.67      1000
           4       0.76      0.87      0.81      1000
           5       0.73      0.73      0.73      1000
           6       0.87      0.83      0.85      1000
           7       0.89      0.82      0.85      1000
           8       0.90      0.87      0.88      1000
           9       0.90      0.86      0.88      1000

    accuracy                           0.81     10000
   macro avg       0.82      0.81      0.81     10000
weighted avg       0.82      0.81      0.81     10000

Accuracy: 81.12
```