CSE 676 Project 1

Sandesh Kumar Srivastava(sandeshk)

Introduction:

This project provides us with an opportunity to explore Convolutional Neural Networks(CNNs) architectures **VGGNet 16**, **ResNet 18** and **Inception V2**. By implementing these three different types of architectures, I have come to understand the different ways in which CNNs are built using the various elementary layers. Developing an architecture from scratch provides a better understanding of the individual fundamental layers like MaxPool2D, Activation, Conv2D etc. and how they are combined to form the more complex architectures like VGGNet, Resnet, Inception.The project also provides an opportunity to explore two very popular machine learning optimizers **Stochastic Gradient Descent(SGD)** and **Adam** and also appreciate the effects of two techniques called **Batch Normalization** and **Dropout** on improving the CNNs efficiency and faster training. In order to avoid overfitting and thus train the model faster, we also implement **Early Stopping** for each of the models.
We have used the **CIFAR-100** dataset for training and testing the model implemented.

Following is a brief introduction of different topics covered in this project:

1. **VGGNet 16**: VGGNet 16 was introduced by K. Simonyan and A. Zisserman in 2014[1]. It is a simple architecture of 16 layers and consists of convolution, max pooling followed by fully connected layers and softmax classifier at the end.
2. **Resnet 18**: Introduced by He et al. in 2015[3], Resnet is based on the concept of skip or residual connections also called micro-architecture[2]. Here, the input to a layer of CNN is in parallel stored and re-introduced into the CNN flow later to produce the effect of short switching or residual network.
3. **Inception V2**: Introduced by Szegedy et al in 2014[4], Inception is based on passing the input to a series of filters of sizes 1x1, 3x3 and 5x5 at the same level and concatenating the resultant with a max pooled variant of the input.
4. **Batch Normalization**: It is a technique for training the CNN faster where the output of the previous activation layer is adjusted for mean and standard deviation before passing to the next layer.
5. **Dropout**: It is a technique which helps to address the overfitting problem in a CNN by randomly removing nodes from the network so that the model is more generalized.
6. **SGD**: It is a workhorse algorithm for optimizing a machine learning model where in each step we update the parameters of all training samples.
7. **Adam**: It is another highly popular optimizing algorithm which is based on calculating an exponential moving average of the gradient and the squared gradient[5].
8. **Early Stopping**: It is a mechanism to stop training the model further in case the validation data parameters like accuracy, loss are not improving with further learning. It helps to estimate when a model starts overfitting and is thus beneficial for fast training.

9. **Cifar-100 dataset**: It is an image classification dataset which consists of 100 classes and contains 600 images of each class out of which 500 are used for training and 100 for testing.

Implementation details:

1. I have implemented all the architectures and the variants using Keras and followed the same pattern throughout. I begin by importing the libraries and defining some of the hyper-parameters like `NUMBER_OF_CLASSES, epochs, batch_size, path_best and path_train` where most of the names are self-explanatory apart from `path_best` which stores the path of best weights seen so far and is used for validation and `path_train` is used to store the optimal weights seen during current training phase. ***For verification of the results, we need to set the "path_best" to the actual weight file path and then execute all cells except for the "Training" part.***
2. This is followed by loading the cifar-100 dataset and performing some preprocessing on it by subtracting mean and dividing by standard deviation. In order to train the model better, I have implemented data augmentation using `ImageDataGenerator` and utilized some of the available features like `horizontal_flip, rotation_range, width_shift_range` and `height_shift_range` mainly.
3. All these steps are exactly the same in all the implementations and the difference starts once the CNN model is defined. The input to each model is an image of size (32,32,3) where 3 indicates that it is an RGB image.
4. Following are the details of implementation of the specific architectures:
    a) **VGGNet 16:** The main method for creating the VGGNet 16 model is `create_VGG16Model`() which is a very simple and straightforward implementation of the model. It consists of 16 layers which are formed using Conv2D, Maxpool2D, Flatten and Dense layers and using 'relu' activation for all inner layers and 'softmax' at the output layer.
    b) **Resnet 18:** The main method for creating the model is `createResentModel`() which starts by performing a single Conv2D and MaxPool2D on the input and then passing this to a helper function `resnetBuilder`() which implements one unit of residual or short-switching network. The basic function of `resnetBuilder`() is to pass the input to 2 subsequent Conv2D layers and then `add()` the original rescaled input with the output. The network is constructed by calling the `resnetBuilder`() repeatedly and then performing AveragePolling2D followed by a final Dense output layer using 'softmax' activation.
    c) **Inception V2:** The main method for creating the model is `create_InceptionV2Model`() which also utilizes a helper method called `inceptionBuilder`(). This utility function passes the input to 1x1, 3x3, 5x5 kernel size Conv2D layers and MaxPool2D and then concatenates these outputs together. We invoke the `inceptionBuilder`() module three times to add the micro-architecture thrice in our main architecture. We then perform GlobalAveragePolling2D followed by a final Dense output layer using 'softmax'

activation. More complex architecture can be formed by increasing the calls to the `increptionBuilder`() module with required number of filter parameters.

5. `BatchNormalization()` and `Dropout()` layers are added to the architecture separately for the corresponding implementations. I have kept the implementations of both separate which means that I have not used both the techniques together. The number of `BatchNormalization` and `Dropout` layers and their position I have determined by experiments and observing the performance metrics.
6. Model is then created and a summary of the implementation is displayed.
7. Model is then compiled using the optimizer i.e. either `SGD` or `Adam`.
8. We then start the training portion by instantiating `ModelCheckpoint()` which is used to store the best model in the `path_train` location based on the metric specified in the `monitor.`
9. We also implement `EarlyStopping()` by defining the metric specified in the `monitor` and also a `patience` value which specifies for how many iterations we should observe the `monitor` value for no improvement and then stop training the model.
10. We then start training the model by calling `model.fit()`. Important things to consider here are that we specify the `validation_data` and also the `callbacks` declared in steps 8 and 9 above.
11. After training is completed, I am plotting the model accuracy and loss for both training and test data[6]. As can be seen in the results the plots show training for only max 200 epochs with patience as 20. The lower value of patience is only for capturing the current training loss. While calculating the best model, I trained the model with patience as high as 500 and repeatedly saved the best model and resumed training from the saved model.
12. We then start the Prediction part where we load the model with the best model weights obtained by prolonged training. The model is used to predict the test data values and calculate the precision, recall and accuracy.

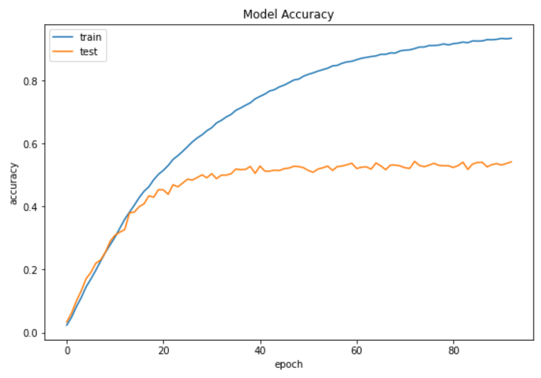
Results:

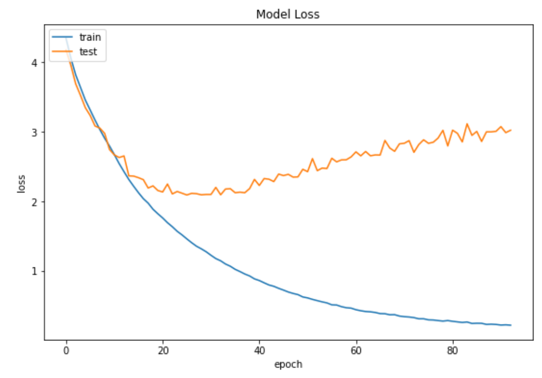
The results are shown in two portions:

a) In the first portion I have shown the accuracy and loss plots obtained during training for both training and test data for a singular smaller run with patience=20. Even though the training is simple but it depicts clearly the way accuracy and loss vary as the training progresses.

VGGNet16:

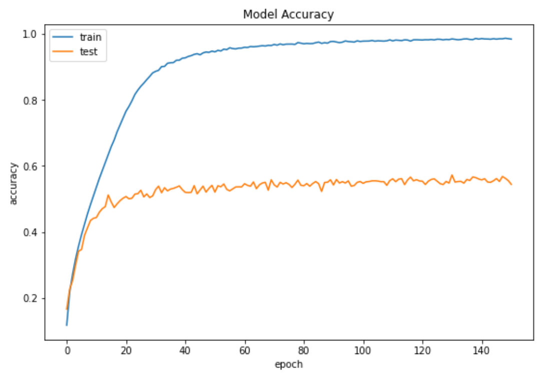
i) VGGNet16_ADAM_NoRegularization

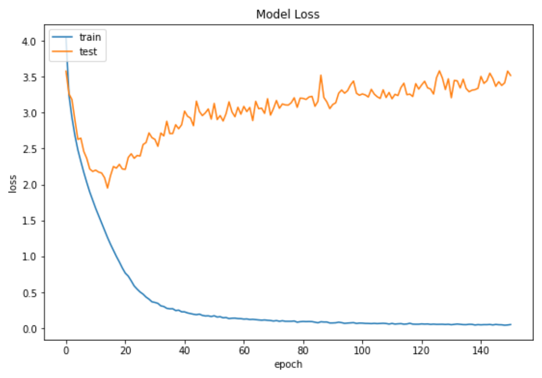




Test accuracy does not improve much after reaching its maximum value even though training accuracy keeps increasing and almost reaches 1. Also, Loss for test data decreases initially but then starts increasing both of which indicate that the model is overfitting and so we stop training using EarlyStopping.

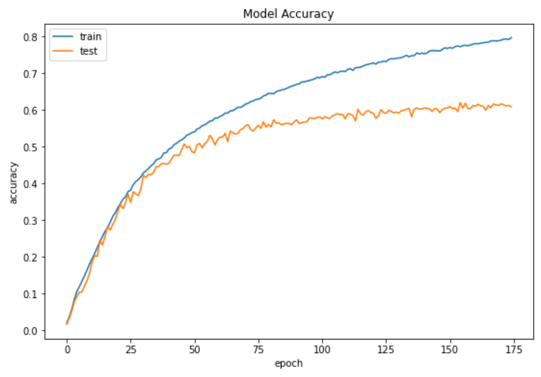
ii) VGGNet16_ADAM_Batch

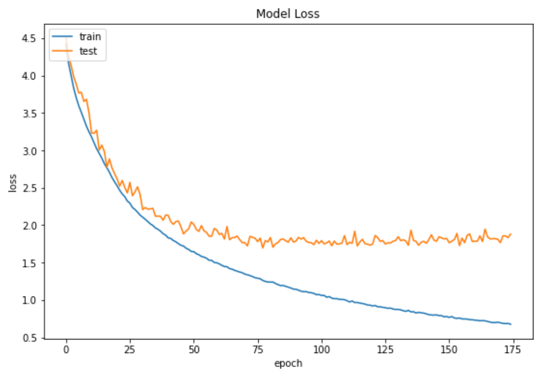




Same behaviour as in (i).

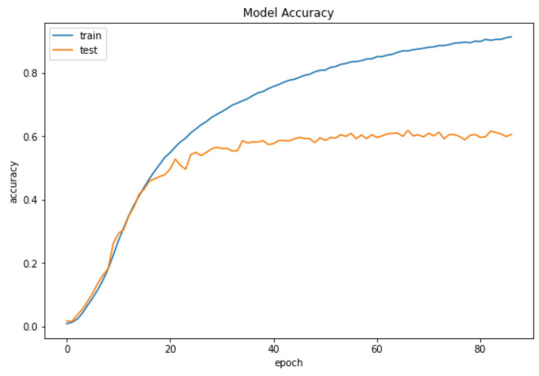
iii) VGGNet16_ADAM_Dropout

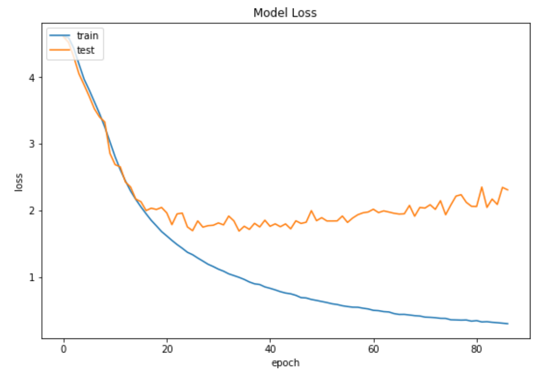




Even though loss for training data does not increase later but there is not much improvement in validation data accuracy indicating overfitting of the model.

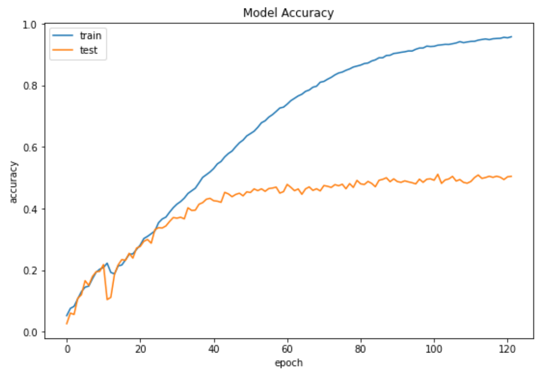
iv) VGGNet16_SGD_NoRegularization

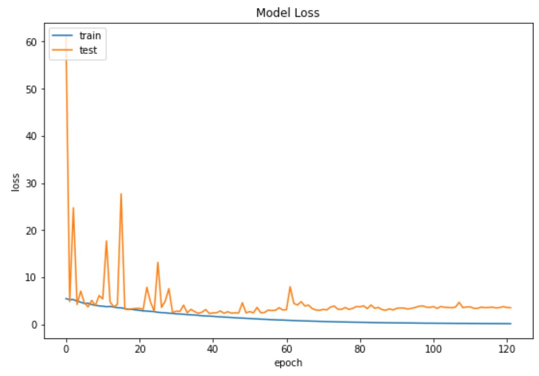




Same behaviour as in (i).

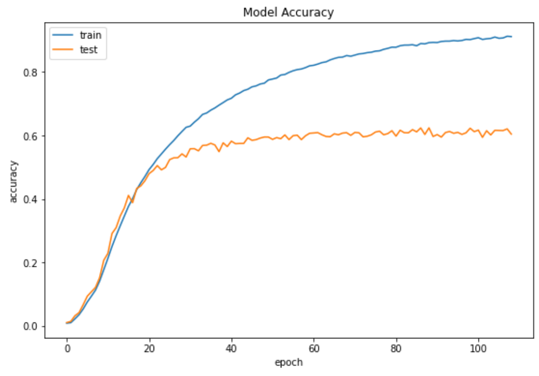
v) VGGNet16_SGD_Batch

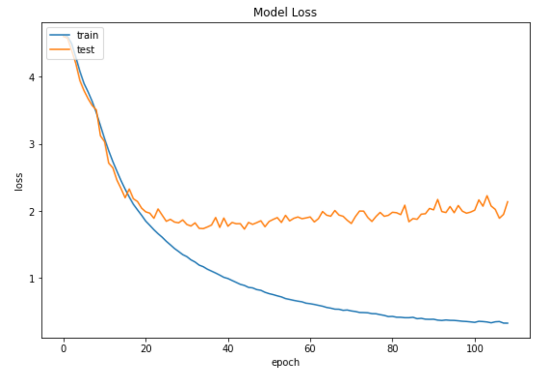




There is great fluctuation in loss initially but later the loss on both training and test data is very low. However, the validation data accuracy does not increase even though training data accuracy increases indicating overfitting.

vi) VGGNet16_SGD_Dropout

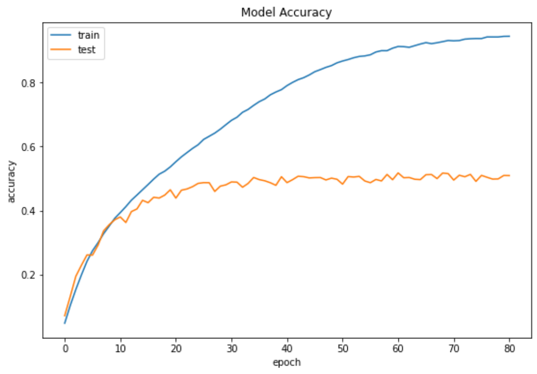


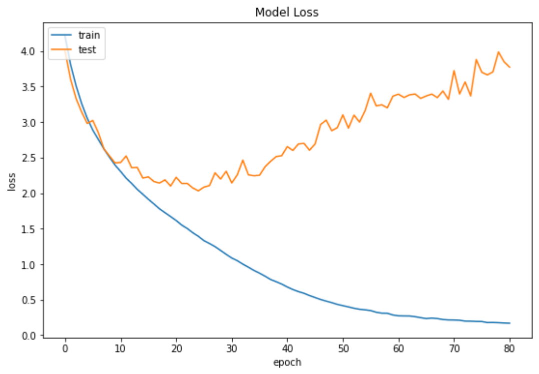


Same behaviour as in (i).

ResNet18:

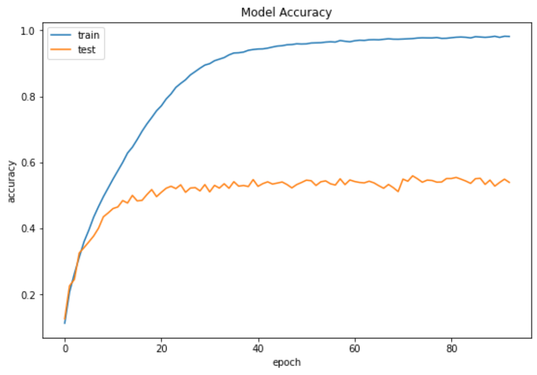
i) Resnet18_ADAM_NoRegularization

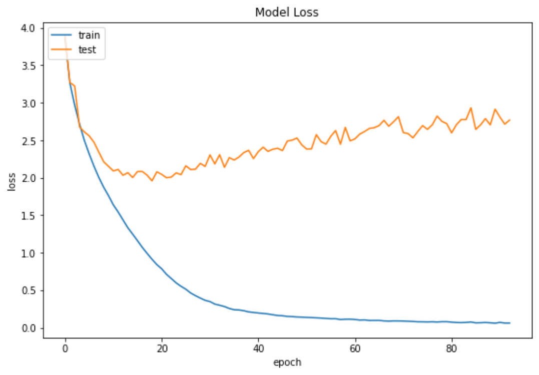




Test accuracy does not improve much after reaching its maximum value even though training accuracy keeps increasing and almost reaches 1. Also, Loss for test data decreases initially but then starts increasing both of which indicate that the model is overfitting and so we stop training using EarlyStopping.

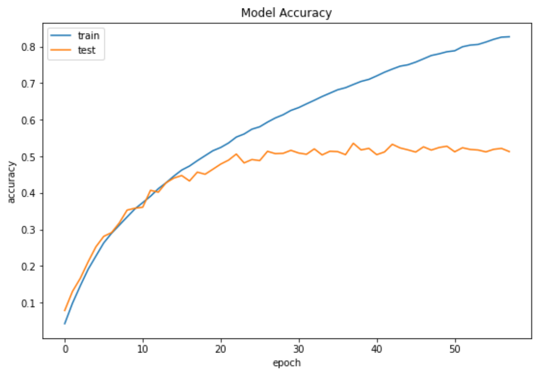
ii) Resnet18_ADAM_Batch

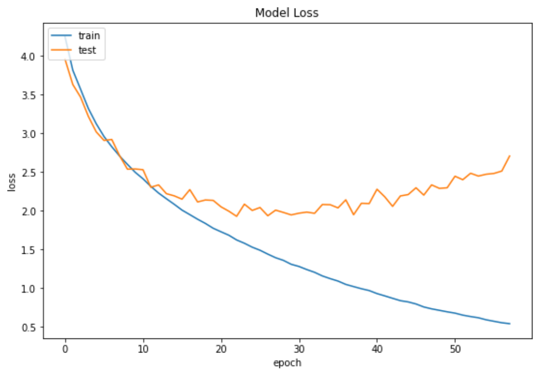




Same behaviour as in (i) even though loss does not increase that sharply.

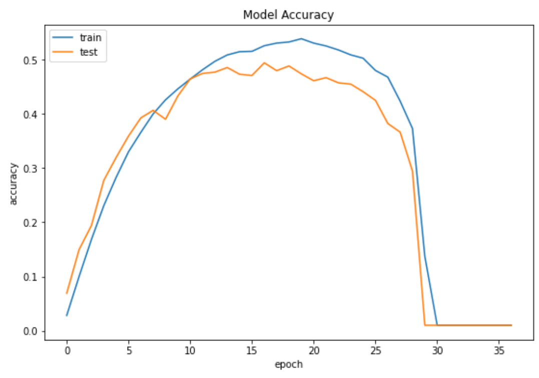
iii) Resnet18_ADAM_Dropout

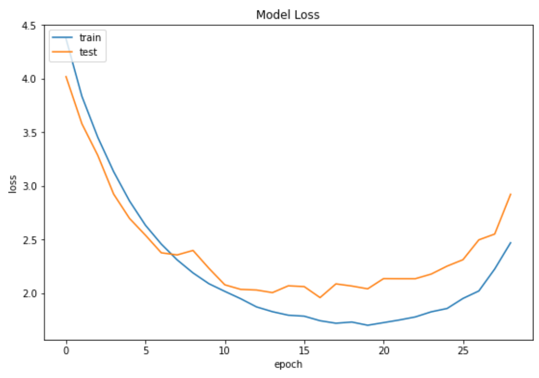




Same behaviour as in (i) even though loss does not increase that sharply.

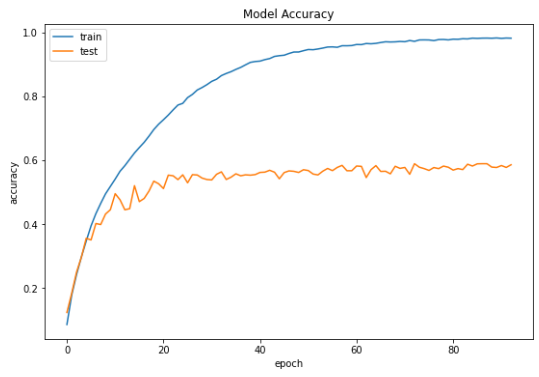
iv) Resnet18_SGD_NoRegularization

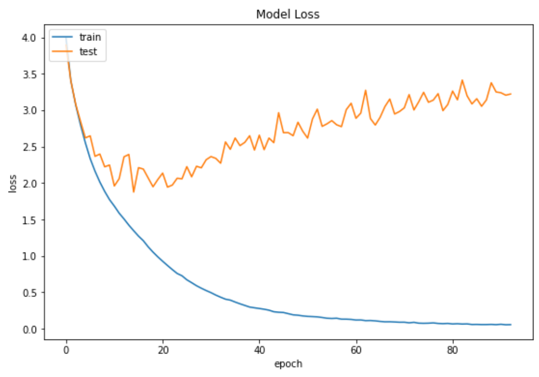




A very peculiar graph which shows that initially the model was learning well as accuracy was increasing and loss was decreasing for both training and test data. However, as training progresses, the accuracy drops sharply and the loss starts increasing for all data. It indicates that the model is not able to improve after this and this is where Checkpoint helps us. I have observed in my experiments that using the best model weights saved so far using Checkpoints and restarting training can help to handle such scenarios. Also, modifying the learning rate of the SGD optimizer with the trained weights helps in improving performance. It might be due to the fact that with higher learning rate, the SGD is not able to find the minima and keeps oscillating around resulting in such results.

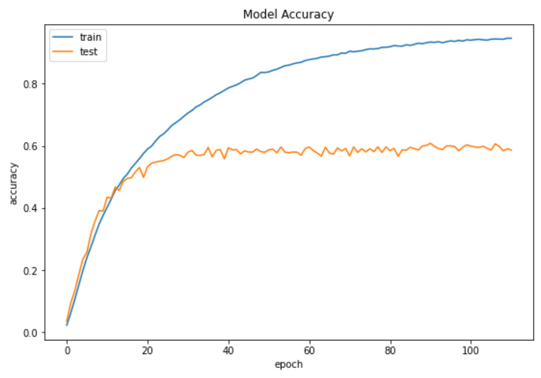
v) Resnet18_SGD_Batch

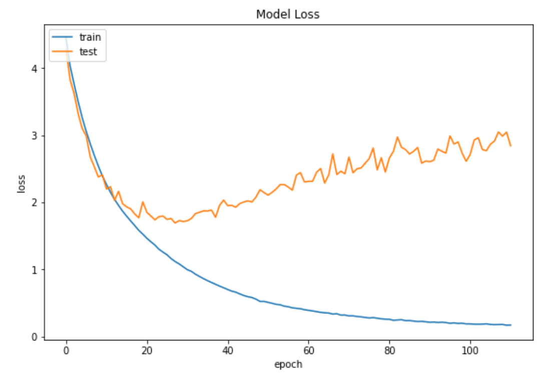




Same behaviour as in (i)

vi) ResNet18_SGD_Dropout

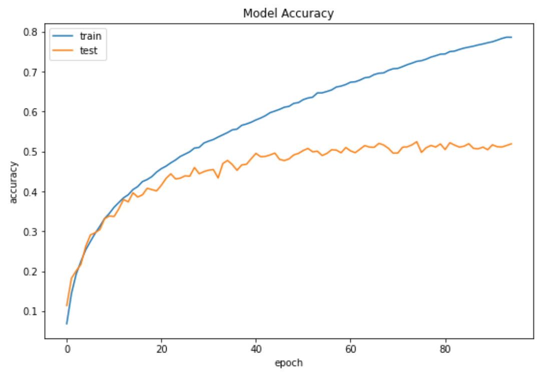


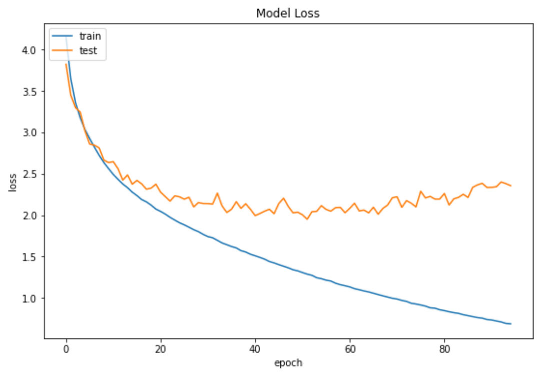


Same behaviour as in (i)

InceptionV2:

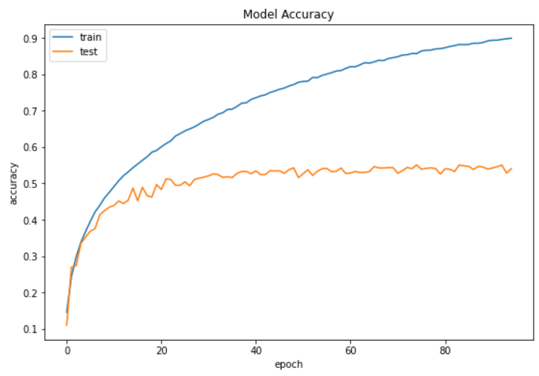
InceptionV2_ADAM_NoRegularization

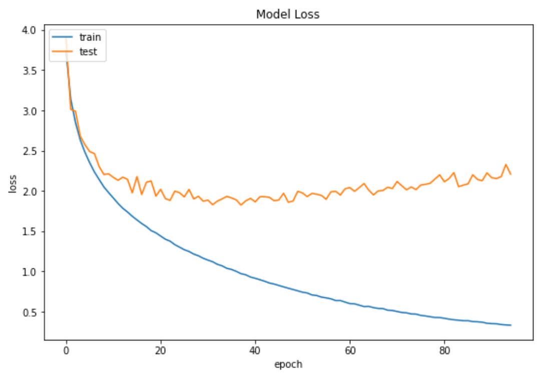




Test accuracy does not improve much after reaching its maximum value even though training accuracy keeps increasing and almost reaches 1. Also, Loss for test data decreases initially but then becomes stagnant, both of which indicate that the model is overfitting and so we stop training using EarlyStopping.

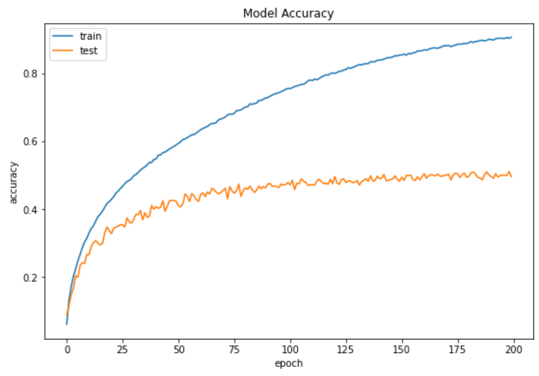
ii) InceptionV2_ADAM_Batch

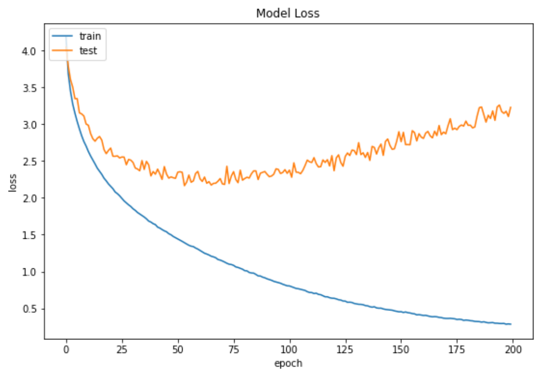




Same behaviour as in (i)

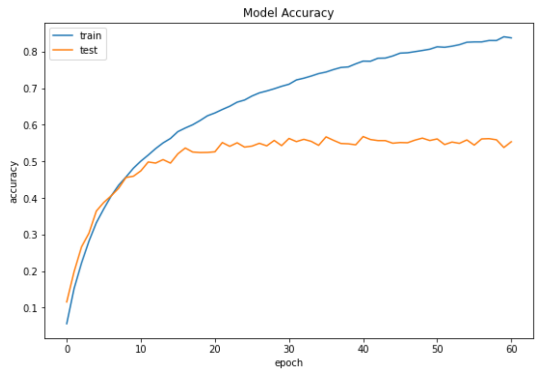
iii) InceptionV2_ADAM_Dropout

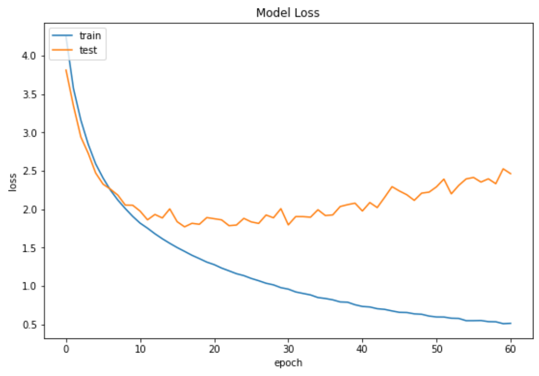




Test accuracy does not improve much after reaching its maximum value even though training accuracy keeps increasing and almost reaches 1. Also, Loss for test data decreases initially but then starts increasing both of which indicate that the model is overfitting and so we stop training using EarlyStopping.

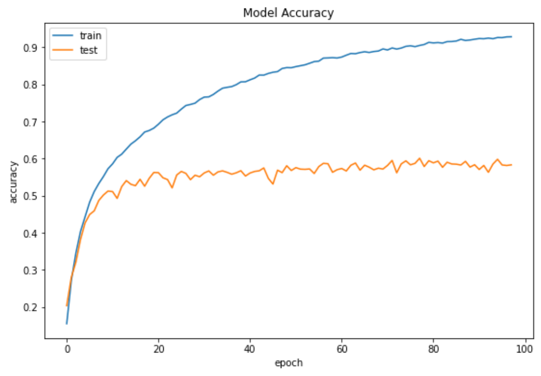
iv) InceptionV2_SGD_NoRegularization

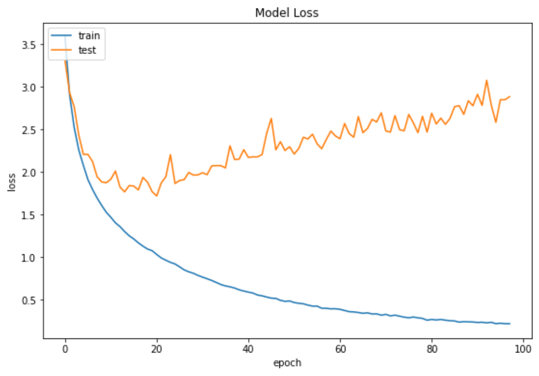




Same behaviour as in (iii)

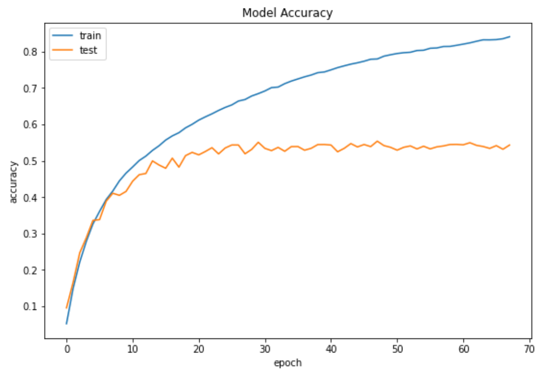
v) InceptionV2_SGD_Batch

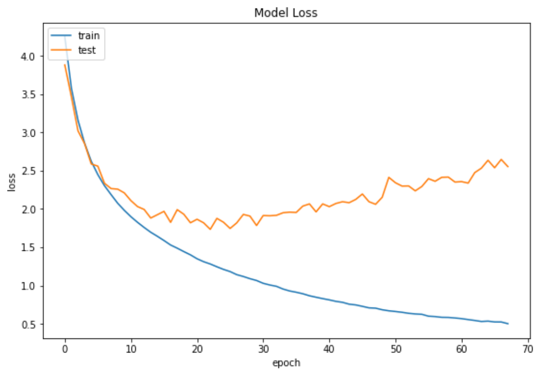




Same behaviour as in (iii)

vi) InceptionV2_SGD_Dropout





Same behaviour as in (iii)

b) In the second portion, I am showing the results obtained with the best model weights that I could train the model on. This training has been done with a much larger patience like 500 and repeatedly resuming training with the saved best model. In order to get better predictions I have also experimented with randomly changing the learning rate of the Optimizers used once I get some stable model. These experiments help in arriving at the best model weights for each 18 architectures. The below table shows the precision, accuracy and recall obtained on the test data using the best model weights.
As we can see from the data in the table below, the values of precision, recall and accuracy are lowest for the “No Regularization” case and improve whenever we use Batch Normalization or Dropout layers to the original CNN architecture. The only place where the improvement is not significant happens for the InceptionV2 case with Adam optimizer using Dropout. The performance metrics in this case are almost the same as the “No Regularization” case.

	Arch	VGG-16			ResNet 18			Inception v2		
Optimizer	Score / Setting	Precision	Recall	Accuracy	Precision	Recall	Accuracy	Precision	Recall	Accuracy
SGD	With Batch Norm	0.52191	0.5111	0.5111	0.60584	0.598	0.598	0.61072	0.6011	0.6011
	With Dropout	0.64449	0.6323	0.6323	0.61918	0.6078	0.6078	0.57493	0.5536	0.5536
	No Regularization	0.51596	0.4674	0.4674	0.51694	0.4945	0.4945	0.506	0.4998	0.4998
Adam	With Batch Norm	0.58342	0.5726	0.5726	0.57868	0.5569	0.5569	0.57729	0.5647	0.5647
	With Dropout	0.63296	0.619	0.619	0.55924	0.5513	0.5513	0.55312	0.5234	0.5234
	No Regularization	0.56114	0.5433	0.5433	0.5412	0.5287	0.5287	0.54533	0.5214	0.5214

Conclusion:

By implementing the CNN models from scratch, it helps to understand how we stitch together the elementary layers like Conv2D, MaxPool2D etc. to form more complex architectures like VGGNet16, Resnet18 and InceptionV2. Through working on this project, I also realized the effect of optimizers SGD and Adam and it appears that for certain architecture one performs better than the other while for some settings the situation is reversed. However, it is also very clear that both Batch Normalization and Dropout help in improving the model by increasing the performance metrics and also training the model faster. Lastly, the experiments performed and the results obtained on cifar-100 dataset also show the importance of CheckPoint and EarlyStopping for developing efficient CNN architectures.

References:

[1]. https://arxiv.org/abs/1409.1556
[2]. https://www.pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception-xception-keras/
[3]. https://arxiv.org/abs/1512.03385
[4] https://arxiv.org/abs/1409.4842
[5] https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
[6] https://machinelearningmastery.com/display-deep-learning-model-training-history-in-keras/
[7] https://towardsdatascience.com/step-by-step-vgg16-implementation-in-keras-for-beginners-a833c686ae6c
[8] https://www.analyticsvidhya.com/blog/2018/10/understanding-inception-network-from-scratch/
[9] https://github.com/geifmany/cifar-vgg/blob/master/cifar100vgg.py
[10] https://towardsdatascience.com/step-by-step-vgg16-implementation-in-keras-for-beginners-a833c686ae6c
[11] https://machinelearningmastery.com/how-to-implement-major-architecture-innovations-for-convolutional-neural-networks/
[12] https://github.com/jeffheaton/t81_558_deep_learning/blob/master/t81_558_class_06_3_resnet.ipynb