Introduction to Deep Learning Neural Networks

Shivaleela Ishwarappa Hubli
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14226 shivalee@buffalo.edu

October 19, 2020

Abstract

The task of extracting feature from a given image for classification has become vital in many fields including medicine etc. Numerous machine learning and deep learning models have been proposed and implemented towards the task. The architectures may vary in terms of their depth, width, composition of layers, normalization and regularization techniques used, optimizer used and so on. Hence it is very important that we analyze the models on aspects that effect the performance of the model like the ability of the model to recognize, memory usage, inference time and computational cost. We analyse the performance of models on metrics like accuracy, precision and recall. In this project, we analyze the performance and architecture changes that can yield better performance at the task of image classification by three such models called VGG Net 16, Res Net 18 and Inception Net V2 on CIFAR 100 dataset.

1 Introduction

The task of extracting features from a given image for classification has become vital in many fields including medicine etc. Numerous machine learning and deep learning models have been proposed and implemented towards the task. The most common one being the convolutional neural networks (CNN). But simple CNN has few drawbacks such as they do not encode the position and orientation of object and lack the ability to be spatially invariant to the input data [1]. Hence more complex architectures have emerged such as VGG network, Residual network and Inception network. With the explosion of new architectures every day, a necessity to evaluate these networks emerge in order to choose the best model for the given task.

2 Dataset

The CIFAR-100 dataset consists of 60000 32x32 colour images in 100 classes [2]. Each class contains 500 training images and 100 testing images. The 100 classes in the CIFAR-100 are grouped into 20 super classes. Each image comes with a class it belongs to called "fine" label and a super class it belongs to called "coarse" label. Here is the list of classes in the CIFAR-100:

3 Networks

In this section we will see the original architectures of VGG Net 16, Res Net 18 and Inception Net V2 used in the project.

3.1 VGGNet 16

The VGG network architecture was introduced by Karen Simonyan and Andrew Zisserman in their 2014 paper, Very Deep Convolutional Networks for Large Scale Image Recognition [4]. It is a simple network that uses only 3x3 convolutional layers stacked on top of each other in increasing depth. The volume is reduced

Superclass	Classes
aquatic mammals	beaver, dolphin, otter, seal, whale
fish	aquarium fish, flatfish, ray, shark, trout
flowers	orchids, poppies, roses, sunflowers, tulips
food containers	bottles, bowls, cans, cups, plates
fruit and vegetables	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices	clock, computer keyboard, lamp, telephone, television
household furniture	bed, chair, couch, table, wardrobe
insects	bee, beetle, butterfly, caterpillar, cockroach
large carnivores	bear, leopard, lion, tiger, wolf
large man-made outdoor things	bridge, castle, house, road, skyscraper
large natural outdoor scenes	cloud, forest, mountain, plain, sea
large omnivores and herbivores	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates	crab, lobster, snail, spider, worm
people	baby, boy, girl, man, woman
reptiles	crocodile, dinosaur, lizard, snake, turtle
small mammals	hamster, mouse, rabbit, shrew, squirrel
trees	maple, oak, palm, pine, willow
vehicles 1	bicycle, bus, motorcycle, pickup truck, train
vehicles 2	lawn-mower, rocket, streetcar, tank, tractor

by using max pooling layers. Two fully-connected layers, each with 4,096 nodes are then followed by a softmax classifier. Due to its depth and number of fully-connected nodes, VGG is over 533MB for VGG16. VGG is still used in many applications today. The original network proposed in the network was for 1000 class classification hence the final layer has 1000 nodes.

3.1.1 **Architecture**

In the original paper, the images were pre-processed by subtracting the mean RGB value from each pixel computed on training set. The model consists of a stack of convolutional layers with a receptive field of 3x3 with a stride of 1 pixel. The spatial resolution is preserved by using padding of 1 pixel. The volume of the data is reduced by using max-pooling layers followed by every convolutional layer. Max-pooling is performed over a 2x2 window with stride 2. [4]. This is followed by 2 fully connected layers, each with 4096 nodes and a softmax classifier consisting of a fully connected layer with 1000 nodes and softmax activation function.

The hidden layers use ReLU activation function. The model does not use any normalization techniques as explained in the paper that it did not provide any performance enhancements instead increased the memory usage. The paper used different configurations for every described network from 11 weight layers to 19 weight layers and those configurations can be seen in Fig. 1. The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as “conv⟨receptive field size⟩-⟨number of channels⟩”. The width of convolutional layers is small, starting from 64 in the first layer and then increases by a factor of 2 after each max-pooling layer, until it reaches 512.

One of the important aspects of VGG Net is that here only small receptive fields of 3x3 are used for convolution, this reduces the number of parameters of the model hence reducing the complexity and in turn reducing the computational cost. Also, the decision function is made discriminate using ReLU activation

The network was trained and tested ILSVRC-2012 dataset with evaluation parameters as top-1 error and top-5 error. The network outperformed previous generation of architectures. The network exploited the benefits of depth for classification for better accuracy in visual representations.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224×224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 1: VGG Net configurations.

3.2 **ResNet 18**

The ResNet architecture was introduced by Kaiming He in the 2014 paper, Deep Residual Learning for Image Recognition [5]. ResNet is a very deep neural network that can be trained using residual modules as shown in Fig. 2 and gradient descent but it differs from most residual architectures such as AlexNet and OverFeat in a way that ResNet uses micro-architecture modules also called as network-in-network architectures. Micro-architecture refers to using a set of building blocks to build the network. The building blocks along with other convolutional layers and pooling layers constitute to a macro-architecture.

3.2.1 **Architecture**

The 2015 paper implements ResNet50 with 50 weight layers in Keras core. Even though ResNet is much deeper than VGG16 and VGG19, the model size is actually substantially smaller due to the usage of global average pooling rather than fully-connected layers — this reduces the model size down to 102MB for ResNet50 [3]. As we already know deeper neural networks are difficult to train. The layers learn the residual function with respect to layer inputs instead of an unreferenced function. The paper also provides proof to show that it is easy to optimize a residual network and by increasing the depth, the accuracy can be improved. The paper used different configurations for every described network from 18 weight layers to 152 weight layers and those configurations can be seen in Fig. 3.

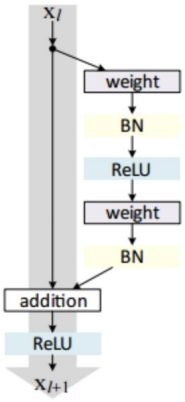

Figure 2: The residual learning: a building block [5].

ResNet is mainly inspired by VGG. The convolutional layers mostly have 3×3 filters and follow two simple design rules: (i) for the same output feature map size, the layers have the same number of filters; and (ii) if the feature map size is halved, the number of filters is doubled so as to preserve the time complexity per layer. The down sampling is directly by convolutional layers that have a stride of 2. The network ends with a global average pooling layer and a 1000-way fully-connected layer with softmax. The ResNet has fewer filters and lower complexity than VGG networks. For the above architecture, a shortcut connection is added to each pair of 3×3 filters. The shortcut performs identity mapping, with extra zero entries padded for increasing dimensions. This option introduces no additional parameter.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figure 3: ResNet configurations.

In this project, ResNet 18 is implemented. As shown in Fig. 3, the 18-layer network consists of 5 convolutions blocks. The first basic block consists of a 7x7 convolutional layer with 2 strids followed by max-pooling layer of 3x3 receptive field with stride 2. The following 4 convolutional blocks consists of 2 blocks of each reception block and an identity block of same number of filters and convolutional layers. Each of those has 2 convolutional layers followed by max-pooling layer and then the original data is added to the output of second max-pooling layer in order to retain the original data which constitutes for the residual block. These are followed by fully connected layer with 100 nodes with softmax activation and average pooling for classification.

The network was trained and tested ImageNet dataset with evaluation of residual networks with depth up to 152 layers which 8 times deeper than VGG but with lower complexity. An ensemble of these residual nets achieves 3.5% error on the ImageNet test set. Similar to VGG, the network exploited the benefits of depth for classification for better accuracy in visual representations with 28% relative improvement on the COCO object detection dataset.

3.3 Inception v2

The Inception architecture was introduced by Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan,
Vincent Vanhoucke, Andrew Rabinovich in the 2014 paper, Going Deeper with Convolutions [6]. The inception module act as a "multi-level feature extractor" by computing 1×1, 3×3, and 5×5 convolutions within the same module of the network and the output of these filters are then stacked along the channel dimension and before being fed into the next layer in the network. The original architecture was called GoogLeNet, but subsequent manifestations have simply been called Inception vN where N refers to the version number put out by Google. The weights for Inception V3 are smaller than both VGG and ResNet, coming the right in at 96MB.

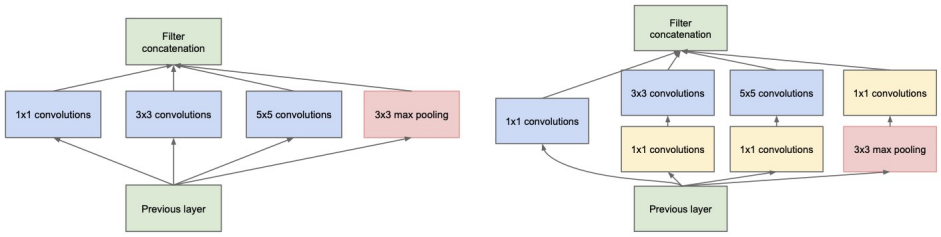

Figure 4: Inception module, naive version on the left and with dimension reduction on

3.3.1 Architecture

The main principle of this architecture is the improved utilization of the computing resources inside the network. This was achieved by a carefully crafted design that allows for increasing the depth and width of the network while keeping the computational budget constant. To optimize quality, the architectural decisions were based on the Hebbian principle and the intuition of multi-scale processing. That is a layer-by layer construction in which one should analyze the correlation statistics of the last layer and cluster them into groups of units with high correlation. In order to avoid patch alignment issues, current version of the architecture is restricted to filter sizes 1×1, 3×3 and 5×5, however this decision was based more on convenience rather than necessity. It also means that the suggested architecture is a combination of all those layers with their output filter banks concatenated into a single output vector forming the input of the next stage. Additionally, since pooling operations have been essential for the success in current state of the art convolutional networks, it suggests that adding an alternative parallel pooling path in each such stage should have additional beneficial effect, too.

Inception modules as shown in the Fig. 4(left) are stacked on top of each other, their output correlation statistics are bound to vary: as features of higher abstraction are captured by higher layers, their spatial concentration is expected to decrease suggesting that the ratio of 3×3 and 5×5 convolutions should increase as we move to higher layers. Further judiciously applying dimension reductions and projections wherever the computational requirements would increase too much. This is based on the success of embedding: even low dimensional embedding might contain a lot of information about a relatively large image patch as shown in Fig 4(right). In general, an Inception network is a network consisting of modules of the above type stacked upon each other, with occasional max-pooling layers with stride 2 to halve the resolution of the grid.

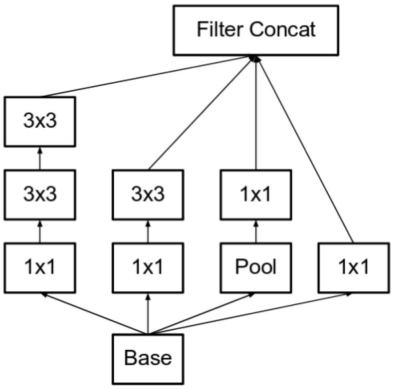

Figure 5: Inception module A

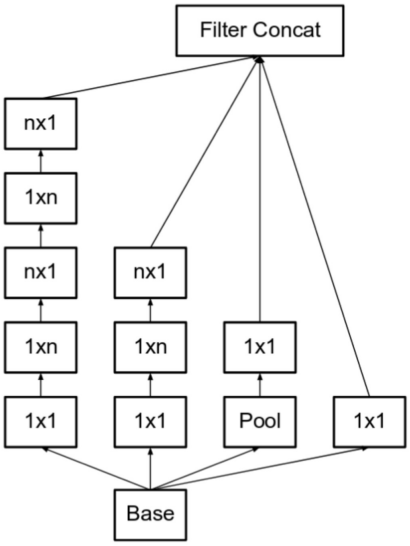

Figure 6: Inception module B

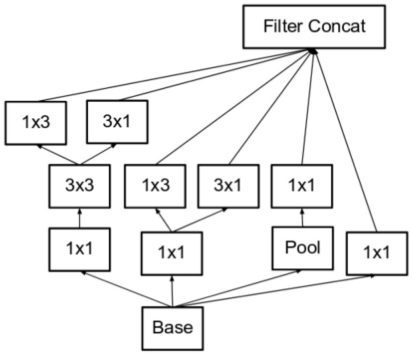

Figure 7: Inception module C

type	patch size/stride or remarks	input size
conv	3×3/2	299×299×3
conv	3×3/1	149×149×32
conv padded	3×3/1	147×147×32
pool	3×3/2	147×147×64
conv	3×3/1	73×73×64
conv	3×3/2	71×71×80
conv	3×3/1	35×35×192
3×Inception	As in figure 5	35×35×288
5×Inception	As in figure 6	17×17×768
2×Inception	As in figure 7	8×8×1280
pool	8 × 8	8 × 8 × 2048
linear	logits	1 × 1 × 2048
softmax	classifier	1 × 1 × 1000

Figure 8: Inception V2

The network was trained using the DistBelief distributed machine learning system using modest amount of model and data-parallelism. The training used asynchronous stochastic gradient descent with 0.9 momentum, fixed learning rate schedule. Polyak averaging was used to create the final model used at inference time. The results provided the proof that approximating the expected optimal sparse structure by readily available dense building blocks is a viable method for improving neural networks for computer vision. The main advantage of this method is a significant quality gain at a modest increase of computational requirements compared to shallower and less wide networks. Fig 8. shows the inception V2 architecture.

4 Experimentation

4.1 VGGNet 16

The code is referenced from github repository [8].

4.1.1 Optimizer - SGD

No regularization

1. Original architecture with a learning rate=0.1 - Accuracy: 0.241900 Precision: 0.262950 Recall: 0.241900
2. Removed the last 3 512 conv2d layers and the 2 dense layers have only 512 nodes with learning rate of 0.01 - Accuracy: 0.315700 Precision: 0.336981 Recall: 0.315700
3. Same as 2 with every conv2d layer parameters reduced by half - Accuracy: 0.295000 Precision: 0.316370 Recall: 0.295000
4. Same as 2 with 'elu' activation function for conv2d layers and dense layers - Accuracy: 0.401600 Precision: 0.403368 Recall: 0.401600
5. Same as 4 with learning rate decay of 1e-6 - Accuracy: 0.343400 Precision: 0.359040 Recall: 0.343400
6. Same as 2 with LeakyRelu activation layer after conv2d layers - Accuracy: 0.295800 Precision: 0.320377 Recall: 0.295800. If node is inactive it sends 0 which causes issue while calculating gradient. Leaky ReLUs allow a small, positive gradient when the unit is not active [7].

7. Same as 4 but with data augmentation using data_fit Accuracy: 0.508600 Precision: 0.533312 Recall: 0.508600

From the above experimentation we can see that the best accuracy is obtained by using an architecture obtained by removing the last 3 512 conv2d layers and the 2 dense layers have only 512 nodes with learning rate of 0.01 with elu as activation function also the data augmentation on training data is used.

Batch normalization

For comparison when the same architecture that gave the highest accuracy for no regularization was subjected to batch normalization for every convolutional layer we obtained - Accuracy: 0.462100 Precision: 0.534580 Recall: 0.462100

Dropout

For comparison when the same architecture that gave the highest accuracy for no regularization was subjected to Drop Out of 0.2, we obtained - Accuracy: 0.490900 Precision: 0.517390 Recall: 0.490900

4.1.2 Optimizer - ADAM

No regularization

1. Original architecture with learning rate=0.1 - Accuracy: 0.422500 Precision: 0.446721 Recall: 0.422500
2. Architecture obtained by removing the last 3 512 conv2d layers and the 2 dense layers have only 512 nodes with learning rate of 0.01 with elu as activation function also the data augmentation on training data is used Accuracy: 0.477100 Precision: 0.510851 Recall: 0.477100
3. Same as 2 with gradient clipping of clipnorm=0.1 - Accuracy: 0.468600 Precision: 0.501904 Recall: 0.468600
4. Same as 2 with gradient clipping of clipnorm=0.5 - Accuracy: 0.480200 Precision: 0.505121 Recall: 0.480200
5. Same as 2 with gradient clipping of clipnorm=1.0 - Accuracy: 0.482800 Precision: 0.513714 Recall: 0.482800

For the comparison the same architecture of VGG is used that gave the highest accuracy for SGD optimizer and from the above experimentation we can see that a gradient clipnorm = 1.0 gave the highest accuracy.

Batch normalization

For comparison when the same architecture that gave the highest accuracy for no regularization was subjected to batch normalization for every convolutional layer we obtained - Accuracy: 0.487500 Precision: 0.521984 Recall: 0.487500

Dropout

For comparison when the same architecture that gave the highest accuracy for no regularization was subjected to Drop Out of 0.2, we obtained - Accuracy: 0.441900 Precision: 0.478217 Recall: 0.441900

4.2 ResNet 16

The code is referenced from github repositories [27] [29] [31] [32].

4.2.1 Optimizer - SGD

No regularization

1. Original architecture with learning rate=0.1 - yields a loss=nan signifying exploding gradient problem

2. Original architecture with learning rate=0.01 loss=nan signifying exploding gradient problem
3. Original architecture with learning rate=0.001 - Accuracy: 0.321700 Precision: 0.360789 Recall: 0.321700
4. Same as 3 with learning decay = 1e-6 - Accuracy: 0.321200 Precision: 0.385822 Recall: 0.321200
5. Same as 3 with 'elu' activation function for conv2d layers and dense layers - Accuracy: 0.317100 Precision: 0.362616 Recall: 0.317100
6. Same as 5 with learning rate decay of 1e-6 - Accuracy: 0.321900 Precision: 0.352532 Recall: 0.321900
7. Same as 3 but with data augmentation using data_fit_generator - Accuracy: 0.389600 Precision: 0.432452 Recall: 0.389600
8. Same as 5 but with data augmentation using data_fit_generator - Accuracy: 0.398100 Precision: 0.432255 Recall: 0.398100
9. Same as 6 but with data augmentation using data_fit_generator - Accuracy: 0.396500 Precision: 0.418651 Recall: 0.396500

From the above experimentation we can see that the best accuracy is obtained by using an architecture with learning rate of 0.001 with elu as activation function also the data augmentation on training data is used.

Batch normalization

For comparison when the same architecture that gave the highest accuracy for no regularization was subjected to batch normalization for every convolutional layer we obtained - Accuracy: 0.365700 Precision: 0.366236 Recall: 0.365700.

Dropout

For comparison when the same architecture that gave the highest accuracy for no regularization was subjected to Drop Out of 0.2, we obtained - Accuracy: 0.423600 Precision: 0.439352 Recall: 0.423600

4.2.2 Optimizer - ADAM

No regularization

1. Same as 3 in no regularization with gradient clipping of clipnorm=1.0 Accuracy: 0.253800 Precision: 0.264663 Recall: 0.253800
2. Same as 3 in no regularization with gradient clipping - Accuracy: 0.255000 Precision: 0.263088 Recall: 0.255000
3. Architecture with Elu activation function, learning rate=0.0001, with data augmentation and no gradient clipping - Accuracy: 0.422000 Precision: 0.442704 Recall: 0.422000

From the above experimentation we can see that the best accuracy is obtained by using an architecture with learning rate of 0.0001 with elu as activation function also the data augmentation on training data is used.

Batch normalization

For comparison when the same architecture that gave the highest accuracy for no regularization was subjected to batch normalization for every convolutional layer we obtained - 0.439500 Precision: 0.468948 Recall: 0.439500

Dropout
For comparison when the same architecture that gave the highest accuracy for no regularization was subjected to Drop Out of 0.2, we obtained - Accuracy: 0.436400 Precision: 0.444581 Recall: 0.436400

4.3 Inception v2

4.3.1 Optimizer - SGD

No regularization: After various experimentation, for the original architecture of Inception V2, with learning rate = 0.01, we obtained - Accuracy: 0.255900 Precision: 0.282296 Recall: 0.255900

Batch normalization: For comparison when the same architecture that gave the highest accuracy for no regularization was subjected to batch normalization for every convolutional layer we obtained - Accuracy: 0.259300 Precision: 0.262751 Recall: 0.259300

Dropout: For comparison when the same architecture that gave the highest accuracy for no regularization was subjected to Drop Out of 0.2, we obtained - Accuracy: 0.243000 Precision: 0.277225 Recall: 0.243000

4.3.2 Optimizer - ADAM

No regularization: After various experimentation, for the original architecture of Inception V2, with learning rate = 0.01, we obtained - Accuracy: 0.213400 Precision: 0.221329 Recall: 0.213400

Batch normalization: For comparison when the same architecture that gave the highest accuracy for no regularization was subjected to batch normalization for every convolutional layer we obtained - Accuracy: 0.210400 Precision:0.219738 Recall: 0.210400

Dropout: For comparison when the same architecture that gave the highest accuracy for no regularization was subjected to Drop Out of 0.2, we obtained - Accuracy: 0.222300 Precision: 0.232447 Recall: 0.222300

5 Learning points

1. Difference between model.fit and model.fit_generator: In model.fit the entire training set can fit into RAM and there is no data augmentation going on but whereas in model.fit_generator real-world data sets are often too large to fit into memory and needs data augmentation to avoid over fitting and increase the ability of our model to generalize [9].
2. **Early stopping callbacks** is implemented in all the experiments. It is used to stop learning when the monitored metric has stopped improving. It has parameters like monitor(Quantity to be monitored), min_delta(Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement.) and patience(Number of epochs with no improvement after which training will be stopped) [10].
3. **Model Checkpoint** for saving weights after training the model which facilitates to use the saved weights to test later point of time [11] and the implementation was referred from [12]. The model checkpoint has parameters such as monitor(quantity to be monitored), save_best_only(if true only saves the best model and parameters and can also be used to save model at regular intervals), save_weights_only(if true only saves the weights of the model, otherwise the entire model along with parameters are saved), mode(can be minimum or maximum based on the quantity being monitored for e.g. if we are monitoring validation loss, then the mode would be min since we would want a minimum loss for the model).

4. For plotting loss and accuracy graphs, the history after model training is saved. The history consists of various model parameters and evaluated metrics such as accuracy, loss, validation accuracy and validation loss and [13] was referred for plotting accuracy and loss of training and validation data for all experiments.

5. **Batch normalization** to train models faster [14]. Batch normalization is a regularization technique that reduces over fitting of the model. Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1. The placement of batch normalization also effects the model performance. It is advised to place the activation layer after the batch normalization layer [15].

6. **Dropout** is also a regularization technique that helps to reduce over fitting problem. Dropout is a technique where randomly selected neurons are ignored during training. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass. [16]. The implementation and usage of dropout is explained in [17].

7. For this experiment the evaluation metrics used are accuracy, precision and recall.

    - **Precision** talks about how precise/accurate your model is out of those predicted positive, how many of them are actual positive and is given by TP/(TP+FP) [18].
    - **Recall** calculates how many of the Actual Positives our model capture through labeling it as Positive and is given by TP/(TP+FN) [18].
    - **Accuracy** is fraction of predictions our model got right and is given by (TP+TN)/(TP+TN+FP+FN). Where TP is true positive, TN is true negative, FP is false positive and FN is false negative. [19] The usage of sklearns to evaluate these matrices is referred from [20].

8. **Activation functions** are mathematical equations that determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated ("fired") or not, based on whether each neuron's input is relevant for the model's prediction. Activation functions also help normalize the output of each neuron to a range between 1 and 0 or between -1 and 1[21]. The syntax to use the activation functions can be referred in [22].

    - **Leaky ReLU** makes the model a lot sparser, the training process tends to be impacted only by the features in your dataset that actually contribute to the model's decision power [7].
    - **ReLU** is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance [23].
    - **Elu** is a function that tend to converge cost to zero faster and produce more accurate results. ELU is very similiar to RELU except negative inputs. They are both in identity function form for nonnegative inputs. On the other hand, ELU becomes smooth slowly until its output equal to alpha whereas RELU sharply smooth [24]. This provided better performance during the experimentation hence elu has been used for a major number of experimentations.

9. **Data augmentation** is a technique to artificially create new training data from existing training data. This is done by applying domain-specific techniques to examples from the training data that create new and different training examples [25]. For this project data augmentation Image_data_generator module is used.

10. A common and relatively easy solution to the exploding gradients problem is to change the derivative of the error before propagating it backward through the network and using it to update the weights.

Gradient clipping involves forcing the gradient values (element-wise) to a specific minimum or maximum value if the gradient exceeded an expected range [26].

** To run the code for evaluation, follow these steps:

1. Open the notebook in Google Colab.
2. Upload the corresponding weights into Google Colab.
3. Run the first cells up till the model compile.
4. Followed by running the testing cell mentioned in every notebook.

** Citations

All the citations regarding the code sources are mentioned in the report.

6 Results

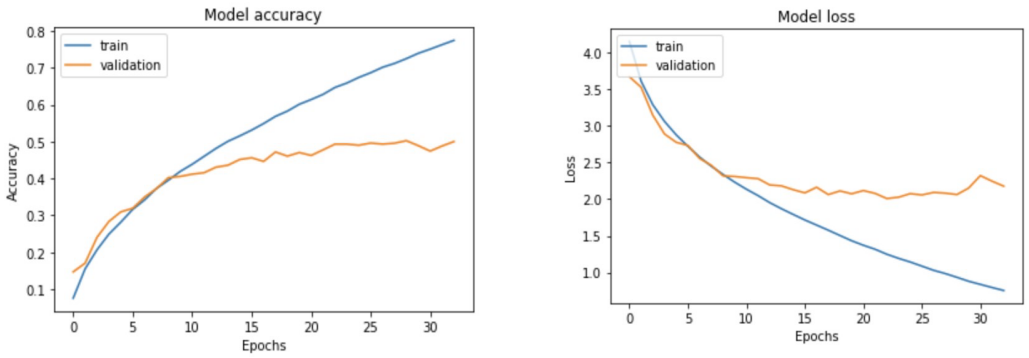

Fig 9: Accuracy and loss of VGGNet 16 with SGD no regularization

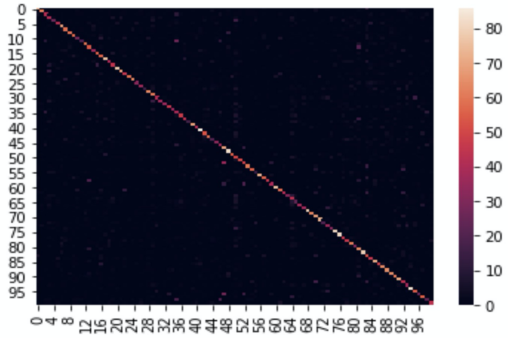

Figure 10: Heat map of VGGNet 16 with SGD optimizer and no regularization

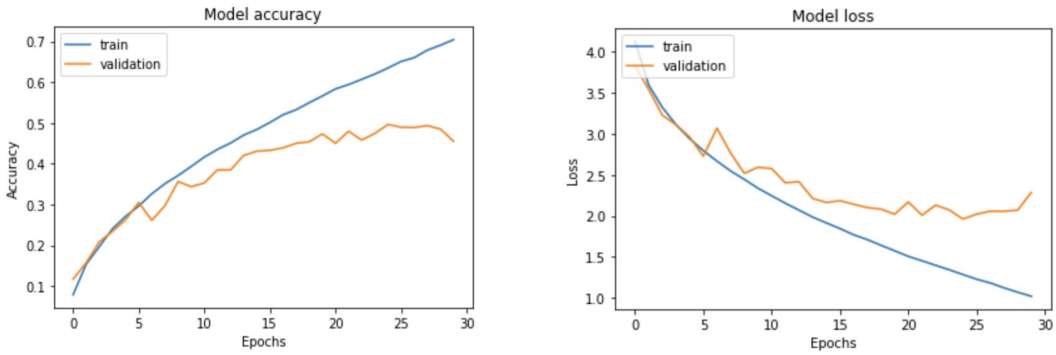

Figure 11: Accuracy and loss of VGGNet 16 with SGD optimizer and batch normalization

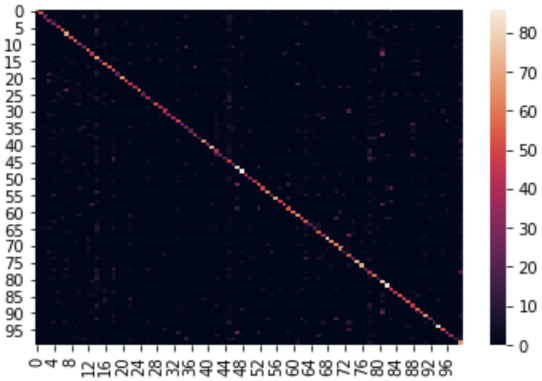

Figure 12: Heat map of VGGNet 16 with SGD optimizer and batch normalization

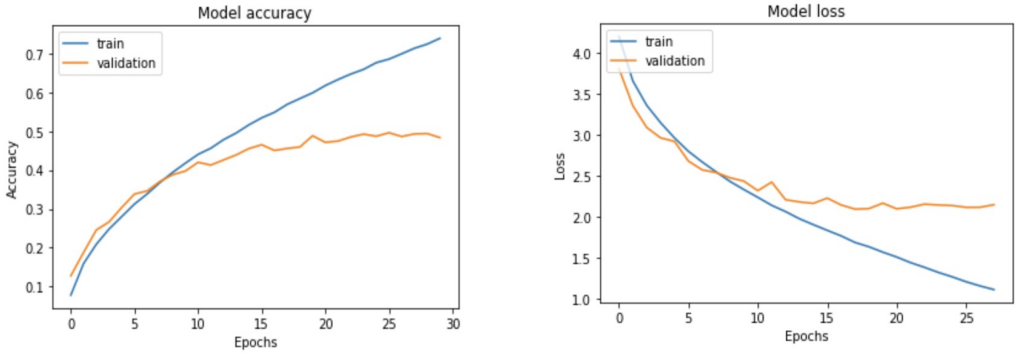

Figure 13: Accuracy and loss of VGGNet 16 with SGD optimizer and dropout

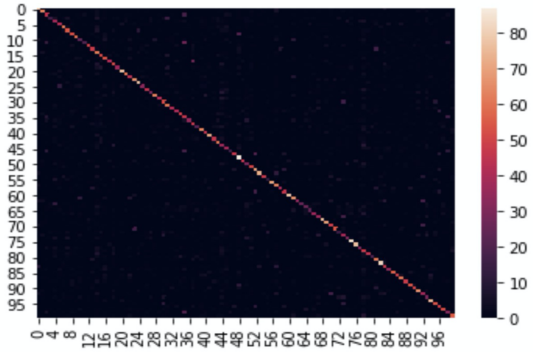

Figure 14: Heat map of VGGNet 16 with SGD optimizer and dropout

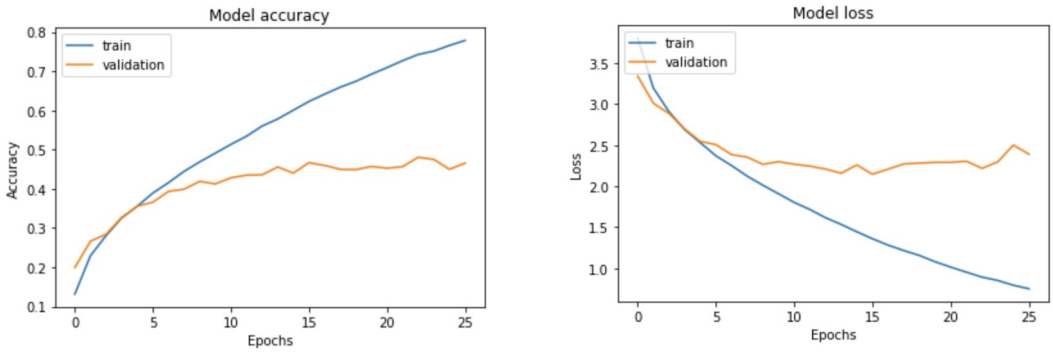

Figure 15: Accuracy and loss of VGGNet 16 with ADAM optimizer and no regularization

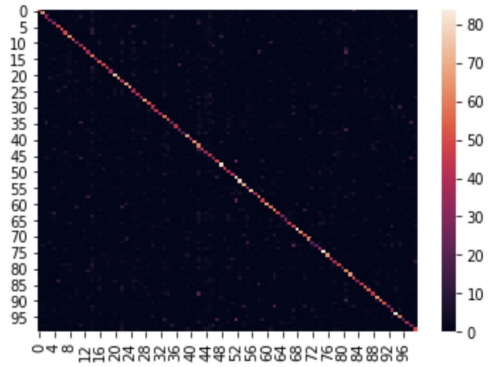

Figure 16: Heat map of VGGNet 16 with ADAM optimizer and no regularization

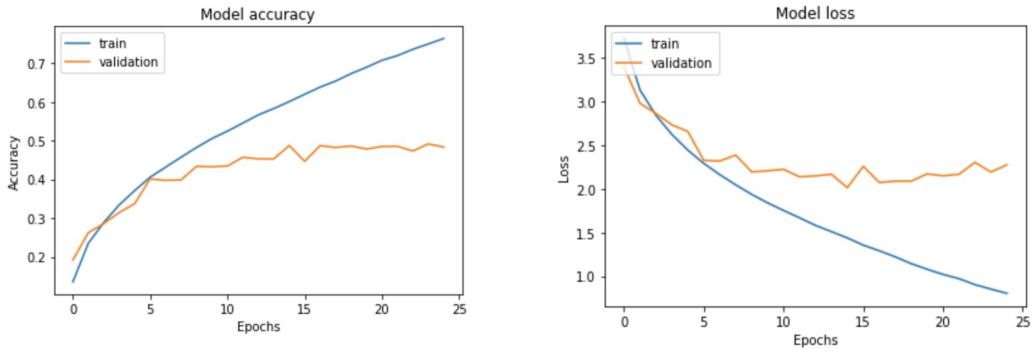

Figure 17: Accuracy and loss of VGGNet 16 with ADAM optimizer and batch normalization

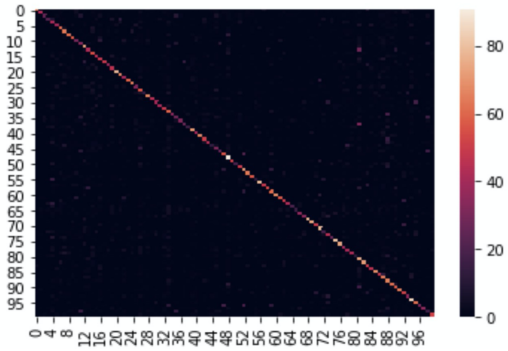

Figure 18: Heat map of VGGNet 16 with ADAM optimizer and batch normalization

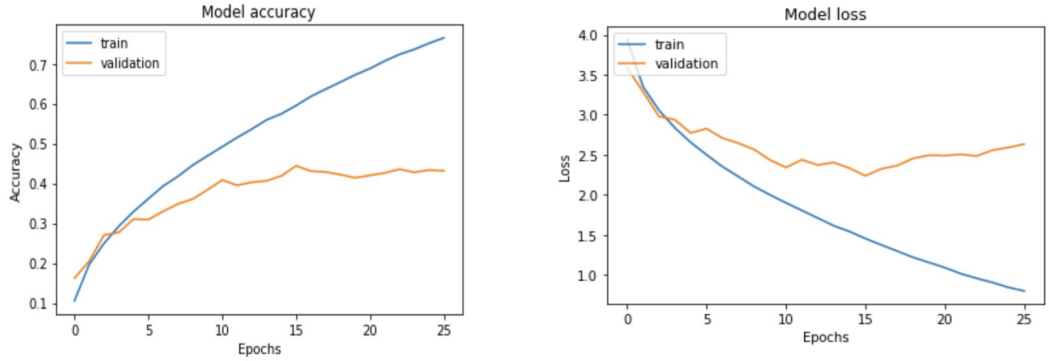

Figure 19: Accuracy and loss of VGGNet 16 with ADAM optimizer and dropout

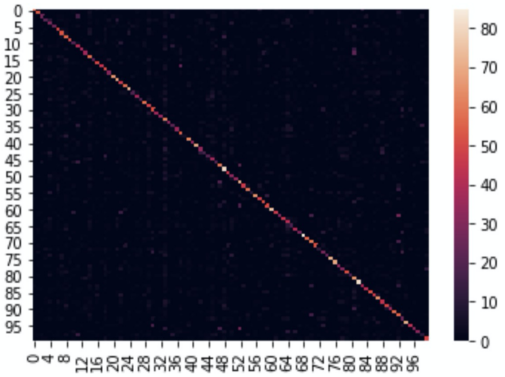

Figure 20: Heat map of VGGNet 16 with ADAM optimizer and dropout

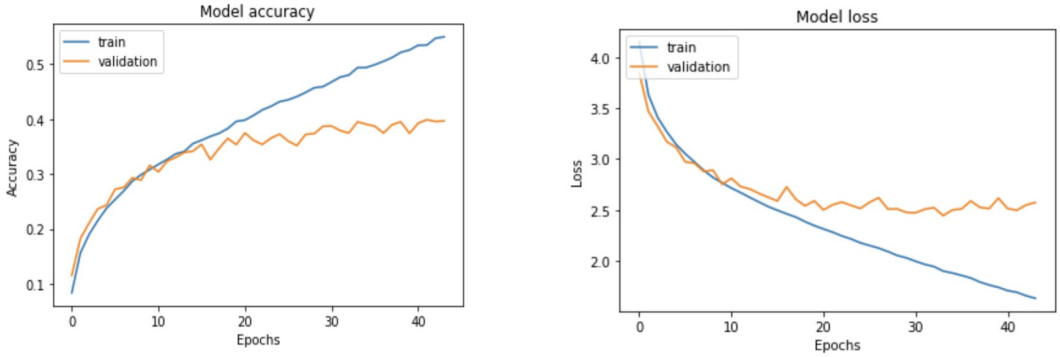

Figure 21: Accuracy and loss of ResNet 16 with SGD optimizer and no regularization

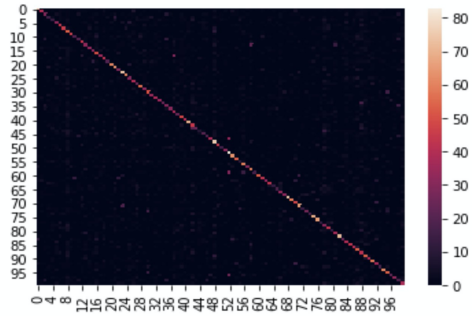

Figure 22: Heat map of ResNet 16 with SGD optimizer and no regularization

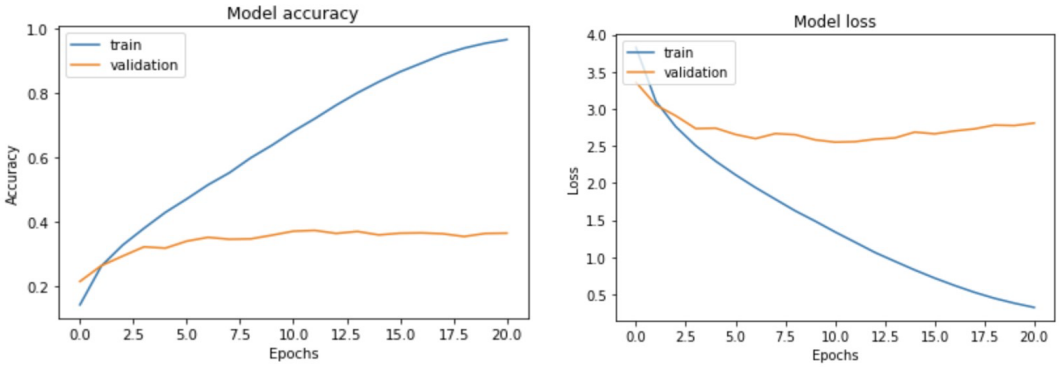

Figure 23: Accuracy and loss of ResNet 16 with SGD optimizer and batch normalization

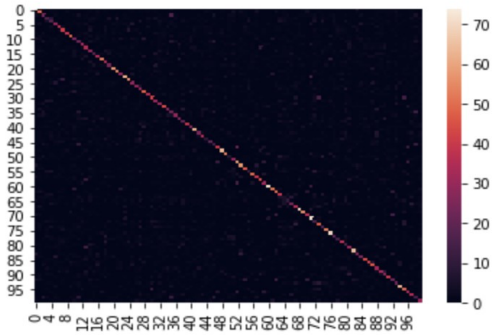

Figure 24: Heat map of ResNet 16 with SGD optimizer and batch normalization

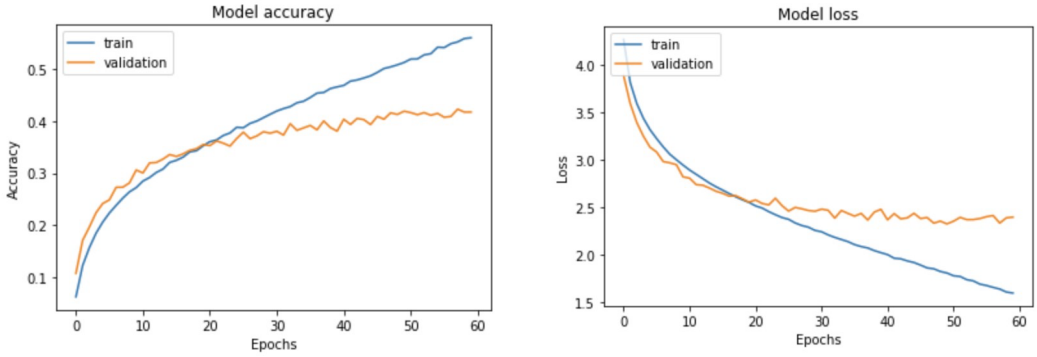

Figure 25: Accuracy and plot of ResNet 16 with SGD optimizer and dropout

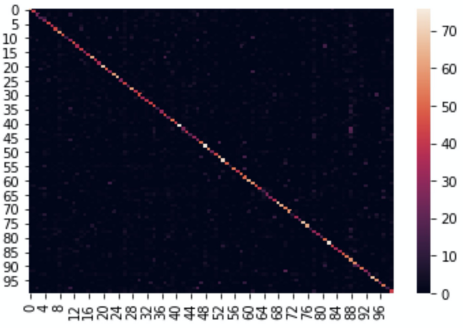

Figure 26: Heat map of ResNet 16 with SGD optimizer and dropout

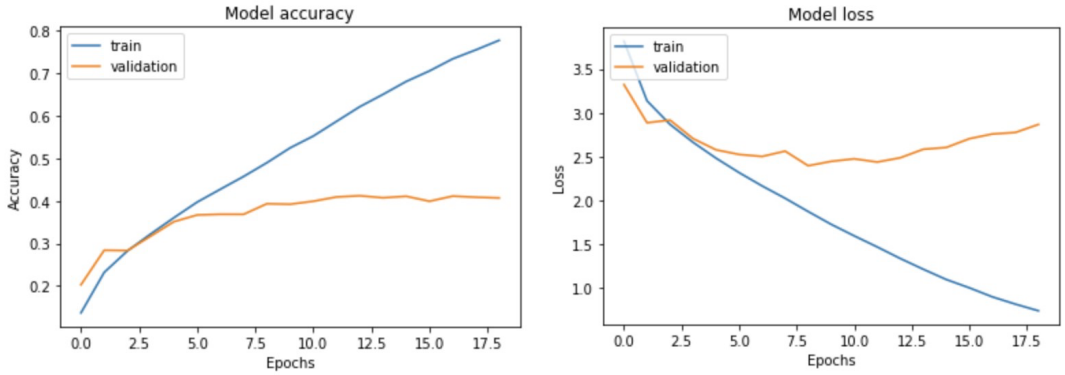

Figure 27: Accuracy and loss of ResNet 16 with ADAM optimizer and no regularization

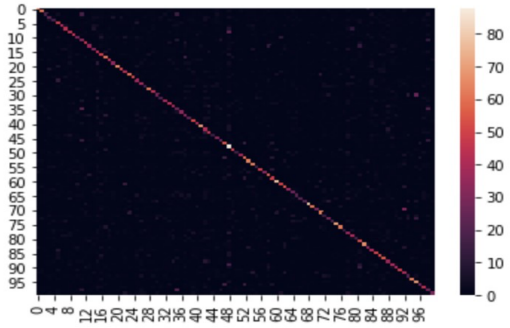

Figure 28: Heat map of ResNet 16 with ADAM optimizer and no regularization

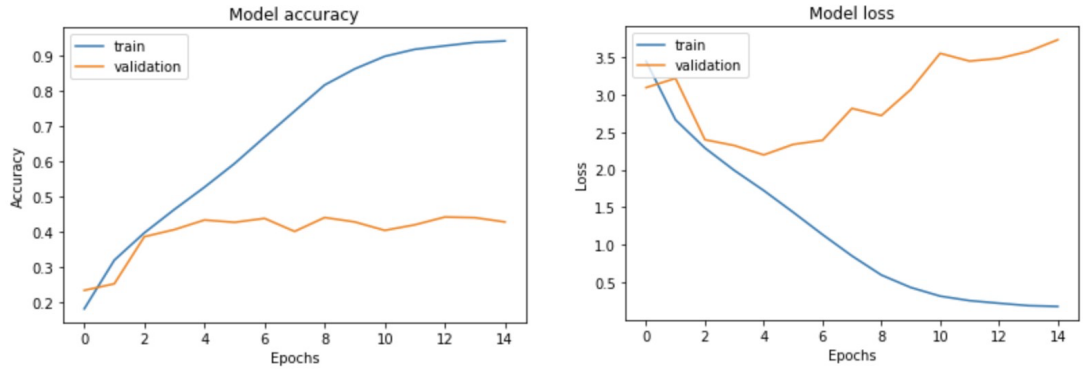

Figure 29: Accuracy and of ResNet 16 with ADAM optimizer and batch normalization

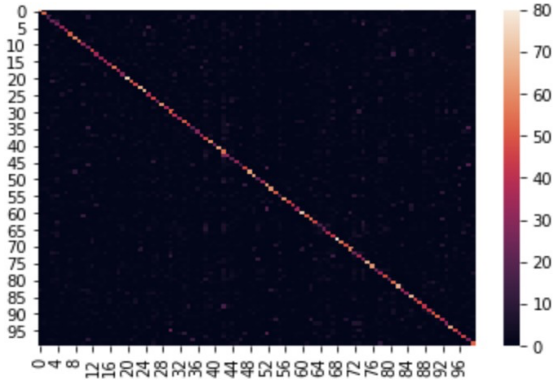

Figure 30: Heat map of ResNet 16 with ADAM optimizer and batch normalization

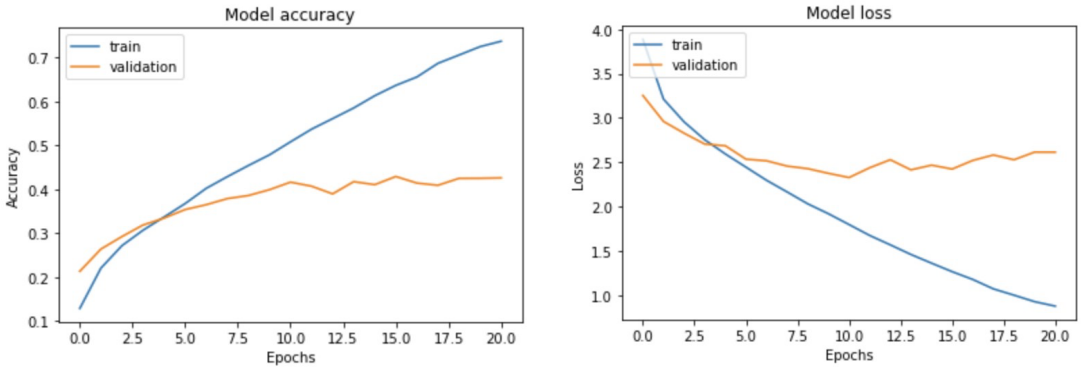

Figure 31: Accuracy and loss of ResNet 16 with ADAM optimizer and dropout

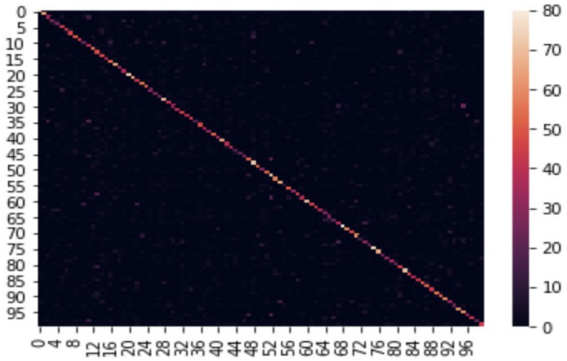

Figure 32: Heat map of ResNet 16 with ADAM optimizer and dropout

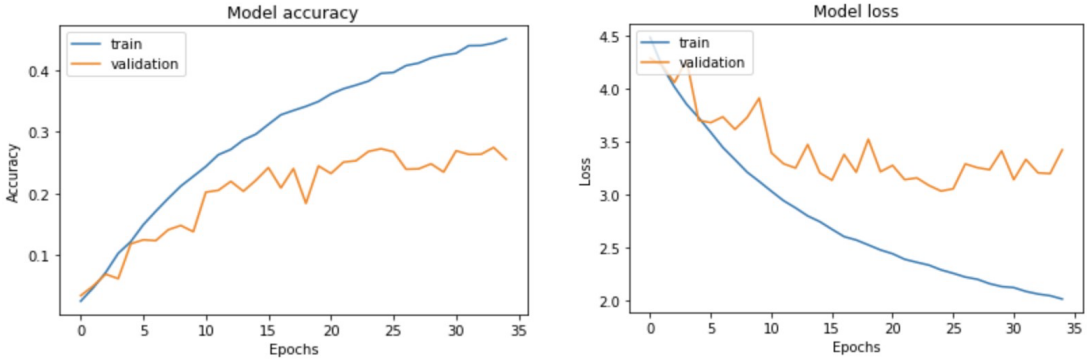

Figure 33: Accuracy and loss of Inception V2 with SGD optimizer and no regularization

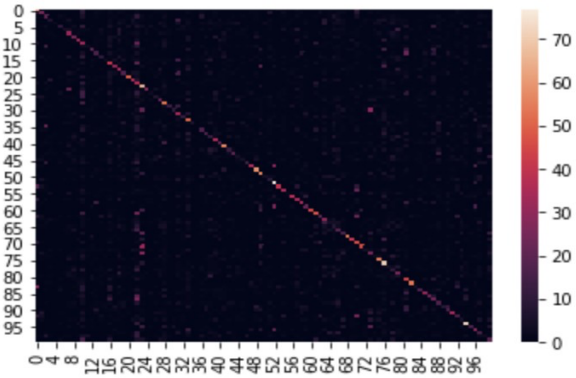

Figure 34: Heat map of Inception V2 with SGD optimizer and no regularization

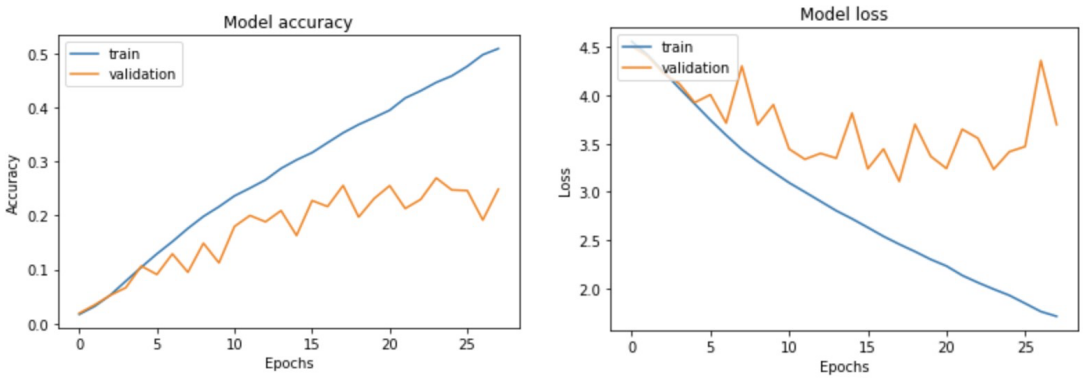

Figure 35: Accuracy and loss of Inception V2 with SGD optimizer and batch normalization

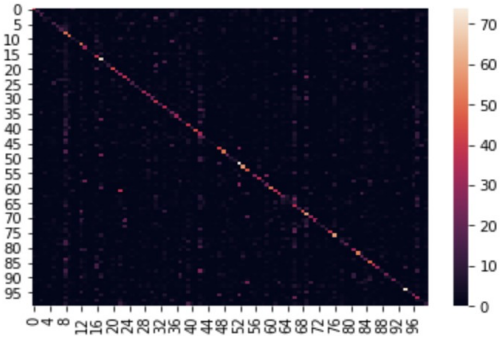

Figure 36: Heat map of Inception V2 with SGD optimizer and batch normalization

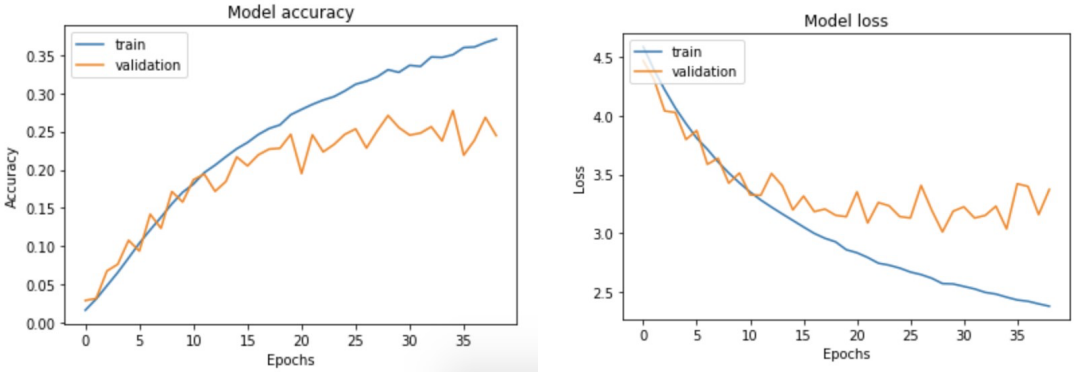

Figure 37: Accuracy and loss of Inception V2 with SGD optimizer and dropout

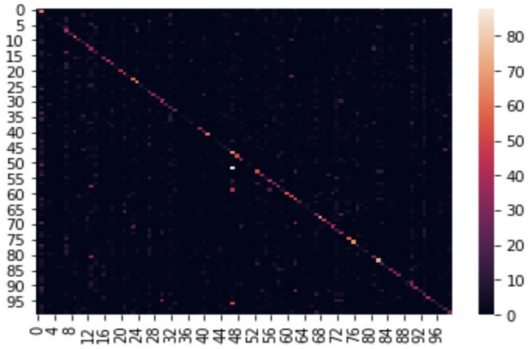

Figure 38: Heat map of Inception V2 with SGD optimizer and dropout

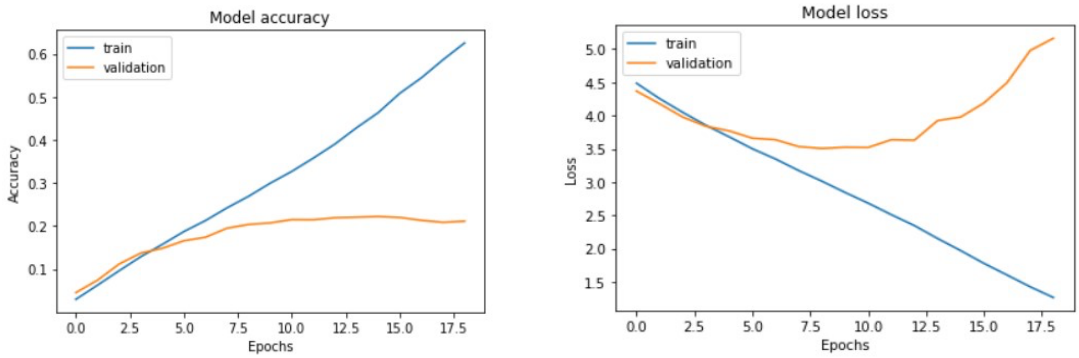

Figure 39: Accuracy and loss of Inception V2 with ADAM optimizer and no regularization

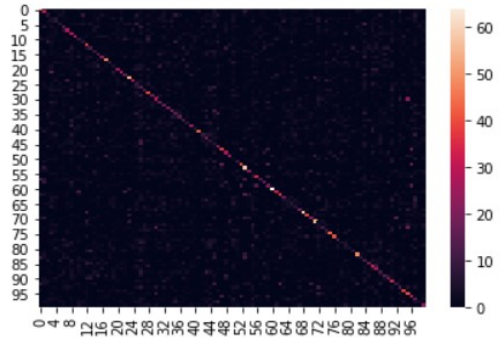

Figure 40: Heat map of Inception V2 with ADAM optimizer and no regularization

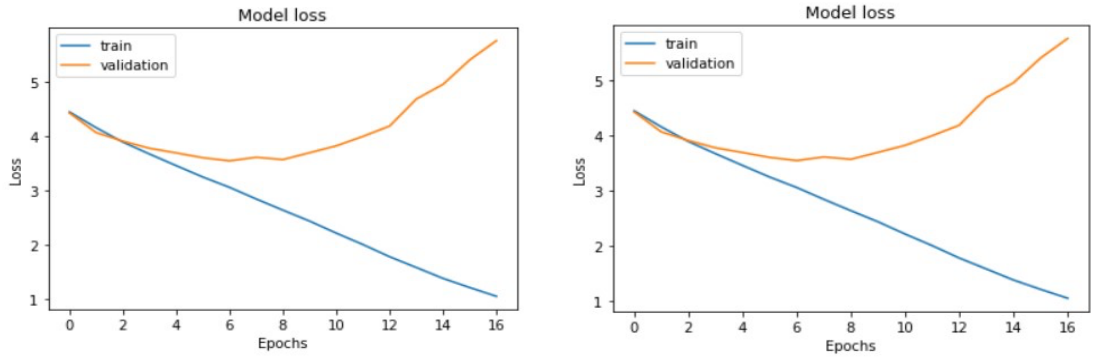

Figure 41: Accuracy and loss of Inception V2 with ADAM optimizer and batch normalization

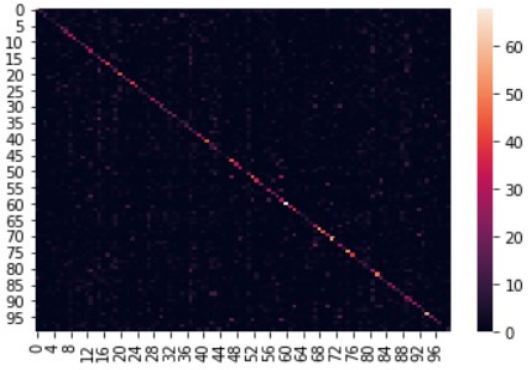

Figure 42: Heat map of Inception V2 with ADAM optimizer and batch normalization

	Architecture	VGG 16			ResNet 18			Inception v2		
Optimizer	Score	Accuracy	Precision	Recall	Accuracy	Precision	Recall	Accuracy	Precision	Recall
	Setting									
SGD	No regularization	50.86	53.33	50.86	39.81	43.22	39.81	25.59	28.22	25.59
	Batch normalization	46.21	53.45	46.21	36.57	36.62	36.57	25.93	26.27	25.93
	Drop out	49.09	51.73	49.09	42.36	43.93	42.36	24.30	27.72	24.30
ADAM	No regularization	48.28	51.37	48.28	42.22	44.27	42.22	21.34	22.13	21.34
	Batch normalization	48.75	52.19	48.75	43.95	46.89	43.95	21.04	21.97	21.04
	Drop out	44.19	47.82	44.19	43.64	44.45	43.64	22.23	23.24	22.23

7 Conclusion

VGGNet 16

From the above experimentation we can see that for VGGNet 16, with SGD optimizer, no regularization provides better accuracy of 50.86%, though this contradicts the concept of regularization that indeed reduces overfitting. This might be because of the huge model size or the model architecture against which the testing is done may not be the optimum architecture. For further analysis, the architecture of model with batch normalization or dropout with different parameters and varied architecture can be tested. And as for the ADAM optimizer, batch normalization technique resulted with the highest accuracy of 48.28% with gradient clipping.

ResNet 18

From the above experimentation we can see that for ResNet 18, with SGD optimizer with drop out regularization provides better accuracy of 42.36%. And as for the ADAM optimizer, batch normalization technique resulted in the highest accuracy of 48.28% with gradient clipping. And as for the ADAM optimizer, batch normalization technique resulted with the highest accuracy of 43.95% with gradient clipping.

Inception V2

From the above experimentation we can see that for Inception V2, with SGD optimizer with batch normalization provides better accuracy of 25.93%. And as for the ADAM optimizer, drop out gave the best accuracy of 22.23%.

Overall the networks have many parameters and hyper parameters that can be considered while experimenting further. Also, since these networks are quite large, for most of the experiments, the data was overfit. Hence smaller network configurations can be utilized to enhance the performance of networks. This project provided a good understanding of VGG 16, ResNet 18 and Inception V2 networks.

References

[1] https://towardsdatascience.com/image-feature-extraction-traditional-and-deeplearning-techniques-ccc059195d04

[2] https://www.cs.toronto.edu/ kriz/cifar.html

[3] https://www.pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inceptionxception-keras/

[4] Karen Simonyan and Andrew Zisserman, "Very Deep Convolutional Networks for Large Scale Image Recognition", *ICLR 2015*.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, "Deep Residual Learning for Image Recognition" *1512.03385v1 cs.CV 2015*.

[6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, "Going Deeper with Convolutions", *1409.4842 cs.CV 2014*

[7] https://www.machinecurve.com/index.php/2019/11/12/using-leaky-relu-withkeras/

[8] https://github.com/geifmany/cifar-vgg/blob/master/cifar100vgg

[9] https://www.pyimagesearch.com/2018/12/24/how-to-use-keras-fit-andfit_generator-a-hands-on-tutorial/

[10] https://keras.io/api/callbacks/early_stopping/

[11] //keras.io/api/callbacks/model_checkpoint/

[12] //towardsdatascience.com/checkpointing-deep-learning-models-in-kerasa652570b8de

[13] https://machinelearningmastery.com/display-deep-learning-model-traininghistory-in-keras/

[14] https://keras.io/api/layers/normalization_layers/batch_normalization/

[15] https://www.dlology.com/blog/one-simple-trick-to-train-keras-model-fasterwith-batch-normalization/

[16] https://keras.io/api/layers/regularization_layers/dropout/

[17] https://machinelearningmastery.com/dropout-regularization-deep-learningmodels-keras/

[18] https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

[19] https://developers.google.com/machine-learning/crashcourse/classification/accuracy

[20] https://machinelearningmastery.com/how-to-calculate-precision-recall-f1-andmore-for-deep-learning-models/

[21] https://missinglink.ai/guides/neural-network-concepts/7-types-neural-networkactivation-functions-right/

[22] https://keras.io/api/layers/activations/

[23] https://machinelearningmastery.com/rectified-linear-activation-function-fordeep-learning-neural-networks/

[24] https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html

[25] https://github.com/geifmany/cifar-vgg/blob/master/cifar100vgg.py

[26] https://machinelearningmastery.com/how-to-avoid-exploding-gradients-inneural-networks-with-gradient-clipping/

[27] https://github.com/raghakot/keras-resnet

[28] https://zhenye-na.github.io/2018/10/07/pytorch-resnet-cifar100.html

[29] https://github.com/priya-dwivedi/Deep-Learning/blob/master/resnet_keras/Residual_Networks_yourself.ipynb

[30] https://neurohive.io/en/popular-networks/resnet/

[31] https://github.com/fchollet/deep-learning-models/blob/master/resnet50.py

[32] https://github.com/raghakot/keras-resnet/blob/master/resnet.py