San Jose State University

presents

Design of A BNN Inference Engine Using Quantization And Retraining Technique

Under the Guidance of

Dr. Chang Choo

By

Niveditha Shiva Murthy
murthyniveditas@gmail.com

Tushar Tarihalkar
tushartarihalkar@gmail.com

ABSTRACT

Deep Neural Networks (DNNs) have gained wide-spread importance for many Artificial Intelligence(AI) applications including Speech Recognition, Robotics, and Computer Vision. The demand for smart/intelligent mobile devices, the deterring computational cost, energy consumption and memory requirements in the case of deep neural networks based models are increasing with its wide-spread real-time and non-real time applications. Hence, the objective of this project focused on developing more accurate and efficient inference schemes by quantization and retraining the model, whose network layers are pruned and consumes less resources by the process of Adaptation. A quantized inference framework that is efficiently able to implement mixed-arithmetic and that handles zero-points is proposed. For this project, the quantization was applied on the ResNet model with weights quantized from 32-bit Tensor to a 1-bit representation and feature maps from 32-bit to 8-bit representation using the Binary Neural Network(BNN) approach.

The approach included saving the accuracy and optimizing the energy efficiency of a relatively smaller model size using the concept of dataflow scheduling, i.e., to implement an energy-efficient scheduling of the data for the Deep Neural Networks. To make the designed AlexNet model adapt to a specified target,

resource, or energy budget, NetAdapt was applied to the pre-trained network(AlexNet), which improved the overall latency and accuracy and these results have been illustrated in following chapters. Thus the accuracy of the AlexNet+NetAdapt model (after 13 iterations of Network Adaptation and Pruning) was increased by 28% from that of the AlexNet model with the number of Floating Point Operations per second(FLOPS) reduced from 710133440.0 to 454945680.0. By the NetAdapt implementation the output feature map size was also reduced from [64 192 384 256 256 4096 4096 10] to [48 160 336 184 224 1952 4096 10]. All these included a mechanism of decreasing the deep neural network size to a relatively smaller size, by the use of Network Pruning. The proposed quantized inference scheme helps improve the efficiency by restoring the model accuracy to the close levels as that of the native one with smaller model size. More improvements in the model can be achieved by further increasing the number of iterations and training with many more different image datasets. The future scope of this project is that it can be applied to many real time applications such as Autonomous driving systems where a simple model, with less number of FLOPs and memory access becomes very crucial.

KEYWORDS – Quantization, inference framework, training, efficiency, deep neural networks, mixed-arithmetic, Row Stationary Dataflow.

TABLE OF CONTENTS

LIST OF FIGURES

	PKU-Autonomous driving dataset.
Figure 32	Training and test results for the ResNet model using CIFAR 10 dataset.
Figure 33(a)	LeNet Model Summary along with kernel size, stride and padding details.
Figure 33(b)	Accuracy v/s Epoch for ResNet model using CIFAR-10 dataset.
Figure 34	LeNet Model Summary along with kernel size, stride and padding details.
Figure 35(a)	Training and test results for LeNet.
Figure 35(b)	Training loss v/s test loss for LeNet.
Figure 35(c)	Training accuracy v/s test accuracy for LeNet.
Figure 36(a).	Quantized ResNet without a defined weight threshold.
Figure 36(b)	Quantized ResNet with a defined weight threshold of 0.1.
Figure 37(a)	Training Loss for each batch of 100 in epoch 1.
Figure 37(b)	Training Loss for each batch of 100 in epoch 2.

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The demanding changes and advancements in the field of machine learning, deep neural networks, and computer vision are increasingly more over the past few decades, with more and more research works. At the same time, there has been a demand for a model with the more efficient, less computational cost for memory access and minimal model size. The most evolving neural networks among all in recent days are the Convolutional Neural Networks (CNNs). There are a number of pre-trained models which are based on CNNs and are handy for working or learning an algorithm. Pre-trained models are used as a reference to improve the already existing model instead of building a model from scratch. A number of pre-trained models already exist and for the project purpose, we have considered a few of them such as ResNet, AlexNet, MobileNet, LeNet, and EfficientNet. These pre-trained models are a good framework to start with for working on Deep Neural Networks based models. DNNs are a requisite component of artificial intelligence for providing a near-human or superhuman efficiency and accuracy on some of the most famous computer-vision tasks like image processing, classification, segmentation, and object detection.

The DNNs are usually heavy and dense hence always run on GPUs or very rarely on CPUs. Tesla P100-PCIE-16GB GPU was used (from Google Colaboratory) for this project. The dataset fed to the heavy DNN algorithms consumes more energy and computation cost even when running on GPUs. Especially, these become more challenging when the DNNs are applied to real-time applications where the input dataset keeps flooding to the model and the computation cost also increases adversely affecting memory of the device. Apart from the incoming dataset, there will be a huge amount of weights and feature-map computed at each layer of the model which needs to be

accommodated in the memory in an efficient way so that the other important variables and factors defined for the model do not vanish from the storage. There is always a need for organized data-flow scheduling for memory management issues. DNNs based on Artificial Intelligence(AI) applications are also computationally costly especially when implemented on a platform that is constrained by resources. Solving all the above-mentioned issues should not deteriorate the performance of the base model. Hence, finding an efficient algorithm for memory storage of weights/feature-map, pruning the network, and at the same time maintaining a decent accuracy of the model with a very little loss is more important and a challenging task.

1.2 OBJECTIVE

The objective of the project was to develop a small-sized inference model i.e., a pruned network with less complexity, by quantization, net-adaptation, and retraining in order to reduce the computation cost and maintain a better efficiency and performance. Also, to make sure that the model was capable of adapting to the available resource constraint with a lower amount of latency at each layer of the network that was designed. Also, to avoid the issues of more computational cost, an efficient method of data-flow scheduling and optimization of the network size was implemented on the DNN model. To understand the difference of behavior of the model on different datasets, multiple datasets were considered for implementation purposes. Also, to understand the performance of different pre-trained DNN models, few of them were trained and a comparison study was made based on their accuracy/loss factors.

1.3 DATASET DESCRIPTION

For the purpose of developing a better and efficient model, two different datasets were made use of. The first dataset is the most famous and widely used one for Machine Learning and Neural Network research, CIFAR-10 (Canadian Institute For Advanced Research) dataset. The second dataset that was used for the project is PKU-Autonomous Driving. A detailed description and Visualization of these two datasets are illustrated further in this chapter.

1.3.1 CIFAR-10

The CIFAR-10 dataset is a pinnacle for most of all the CNN learning algorithms and Machine Learning Engineering for image-recognition. The neural network models and algorithms often learn from recognizing the images and this particular dataset is a full set of color images that have been particularly helpful for the computer to learn how to recognize the objects. CIFAR-10 images are often low-resolution, which is useful for the researchers to implement different algorithms in a short duration and also to understand their behavior in a limited amount of time. This particular dataset consists of 60,000 color images of size 32x32 consisting of one of 10 object classes (multi-label classification) with 6000 images per class. The 10 object classes represent different images of airplanes, automobiles, birds, cats, deers, dogs, frogs, horses, ships, and trucks. Below figure 1(a) shows an illustration of the first 20 color images available in the training dataset and their corresponding classes/labels. The CIFAR-10 dataset was divided into 50000 color images of 32x32 for the training of the Deep Neural Networks while the remaining 10000 were considered as the test data. The actual size of the train set and test set was also verified and the corresponding results are shown in figure 1(b).

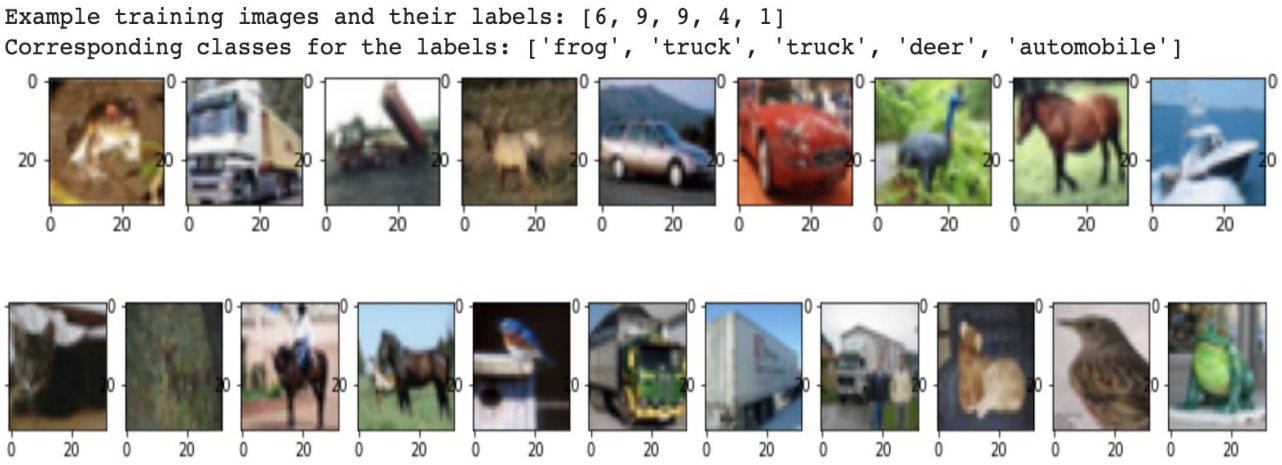
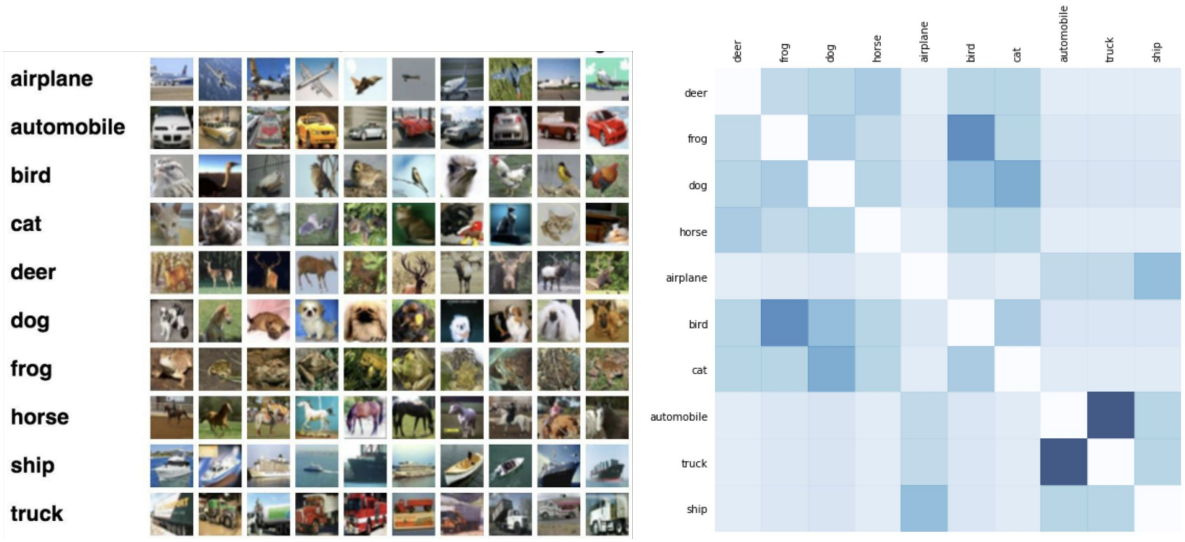


Figure 1:(a) Illustration of the first few images in the training dataset along with their labels.

```
Shape of training data:
(50000, 32, 32, 3)
(50000, 1)
Shape of test data:
(10000, 32, 32, 3)
(10000, 1)
```

Figure 1:(b) Shape of training dataset and test dataset.

Semantic Network for CIFAR-10: The dataset can be represented using the semantic network framework, which helps to find a relation between various concepts in a network. This form of representation also serves as a knowledge-based representation or a graph. For this representation, there is no need for the actual dataset but just the knowledge of different classes is often more helpful to represent a semantic network. Further, the Semantic distance between different categories of classes is quantified and represented in the form of a comparison matrix. The comparison matrix helps us understand the drawbacks of the dataset. Figure 2(a) shows the sample of available class objects and figure 2(b) represents the comparison matrix for each class object. Higher the intensity of blue color for an object means higher the similarity. Hence, conclude that the similarity among the objects belonging to either of the branches such as living things or artifacts has large similarities to one another when compared to those in the opposite branch.



(a) (b)

Figure 2: (a)Available set of classes/labels in the CIFAR-10 dataset, (b) Comparison matrix for each class object.

1.3.2 PKU-AUTONOMOUS DRIVING

Autonomous Driving Library(RAL) and Peking University have been working to remove the space created due to loopholes in the autonomous cars by providing around 60,000 3-dimensional car instances from 5,277 real-world images that are labeled, based on an industry-grade based CAD car model. The dataset consists of pictures of the streets, which are captured from the roof of a car. The information of pose is in a string format for each of the input images of the car. The formatted string is in the following format:

```
model type, YAW, pitch, roll, x, y, z
```

Apart from the previously mentioned details, the dataset also includes further information and a detailed description of each of these files is illustrated in table 1 below. The train CSV file has a file dimension of 4262x2, train_images, and test_images are two zip files consisting of 4262 input train images required for training the model along with 2021images for testing the model.

File_name.type_of_file	Description
Train.csv	information regarding images utilized for training of the model
Train_images.zip	a zip file of the images for training
Train_masks.zip	mask for training images (binary mask).
Test_images.zip	set of images for training
Test_masks.zip	mask for test images (binary mask).
Car_model.zip	Unmasked cars in the training/ test images for pose estimation and more.
Camera.zip	Camera intrinsic parameters
Sample_submission.csv	a sample file for correct formatting of submission

Table1 Data Set Description

The train.csv file has a string format as above mentioned. Figure 3(a) below shows the first few of the header information of the input train image and (b) represents one of its respective images.

	ImageId	PredictionString
0	ID_8a6e65317	16 0.254839 -2.57534 -3.10256 7.96539 3.20066 ...
1	ID_337ddc495	66 0.163988 0.192169 -3.12112 -3.17424 6.55331...
2	ID_a381bf4d0	43 0.162877 0.00519276 -3.02676 2.1876 3.53427...
3	ID_7c4a3e0aa	43 0.126957 -3.04442 -3.10883 -14.738 24.6389 ...
4	ID_8b510fad6	37 0.16017 0.00862796 -3.0887 -3.04548 3.4977 ...

(a)

(b)

Figure 3: (a)initial information in train_image dataset, (b) train_image sample display.

The dataset includes different types of car models each having a different number of counts. A

bar graph of all the available car models is plotted and is shown in figure 4 and gives a better idea of different available input images. This would indeed help training a model that has learned many different scenarios making it to be more useful for real-time applications of traffic.

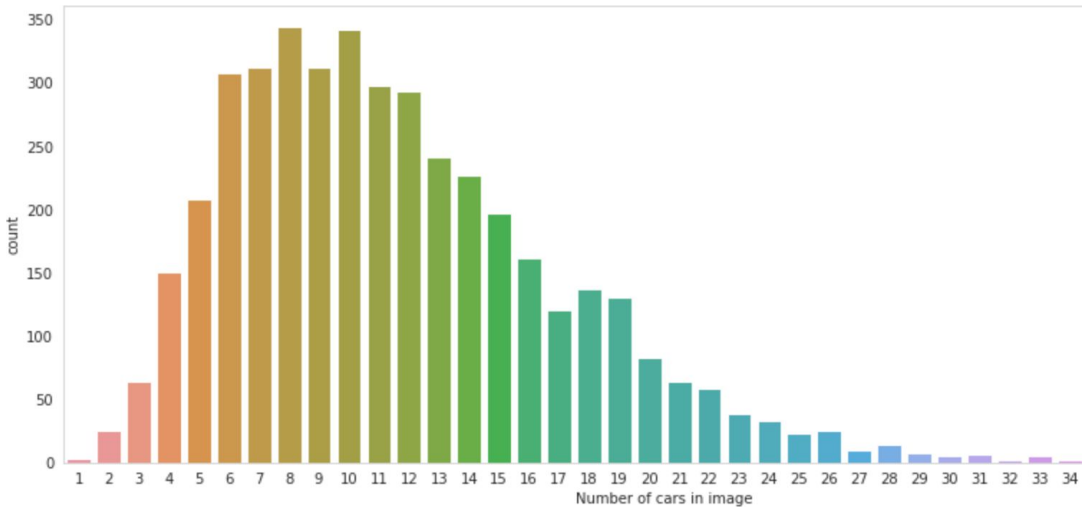
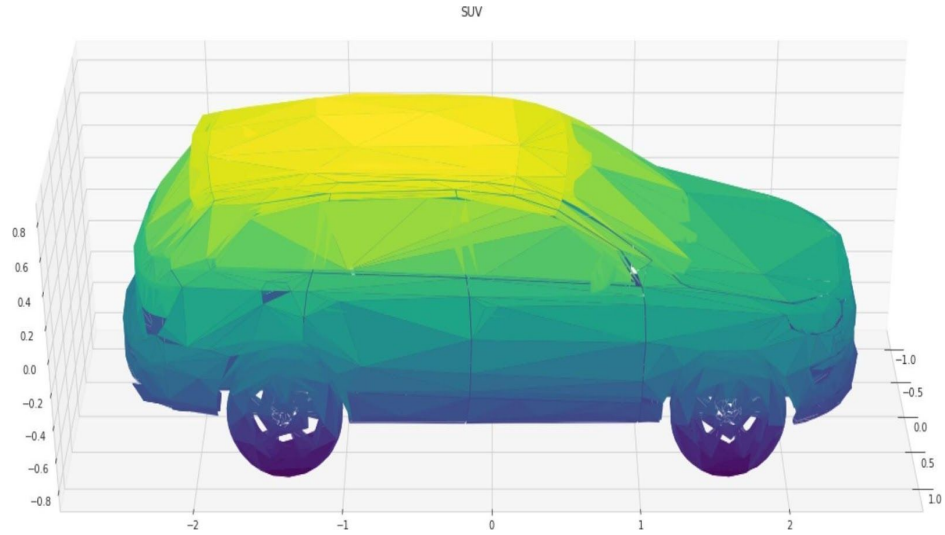


Figure 4: Distribution of types of model cars in data.

A visualization in the form of 3dimension of a car model i.e., Audi-Q7-SUV from a different angle is displayed from the samples available from the car_models zip file. The same is illustrated in the below figure 5.



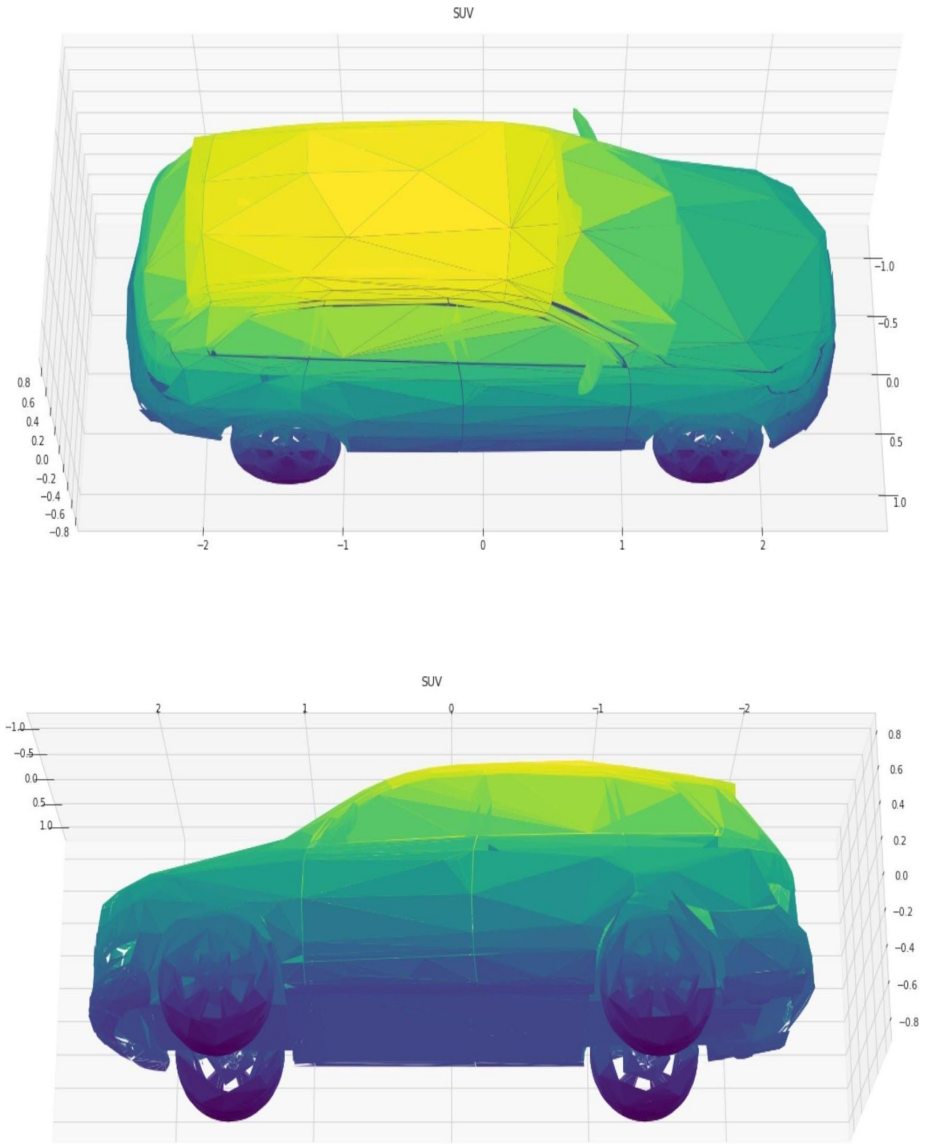


Figure 5: An Audi-Q7-SUV car model 3D representation.

The 3D movement of any object in a given medium can be defined in terms of three important factors, such as pitch, roll, and yaw. Pitch refers to as “Rotation around the side-to-side axis”, Rolls is the “Rotation around the front-to-back axis” and Yaw is defined as “Rotation around the vertical axis”. As illustrated before these factors are formatted as a prediction string and can be plotted as histogram shown in figure 6 below.

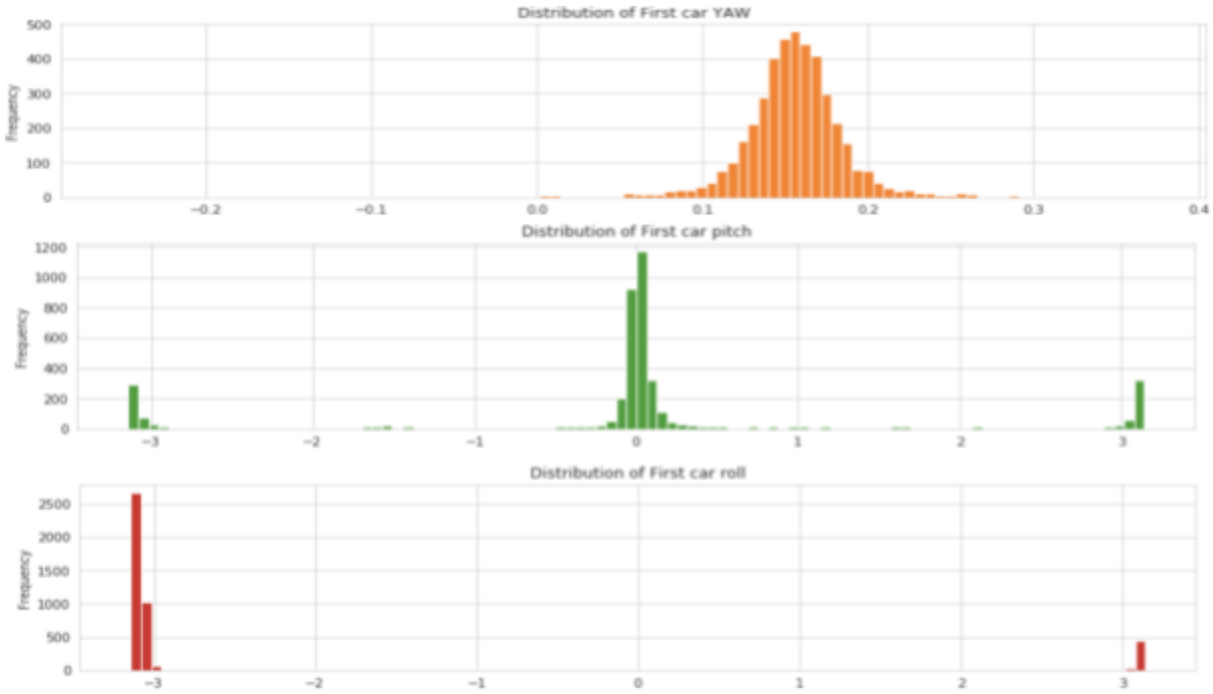


Figure 6: 6 degrees of freedom of a model in the training dataset.

Along with values of pitch, roll, and yaw in the prediction string, the coordinate values of the 3-dimension is defined in the train CSV file. The distribution of this x, y, and z values of one of the car models is also illustrated using the histogram in figure 7.

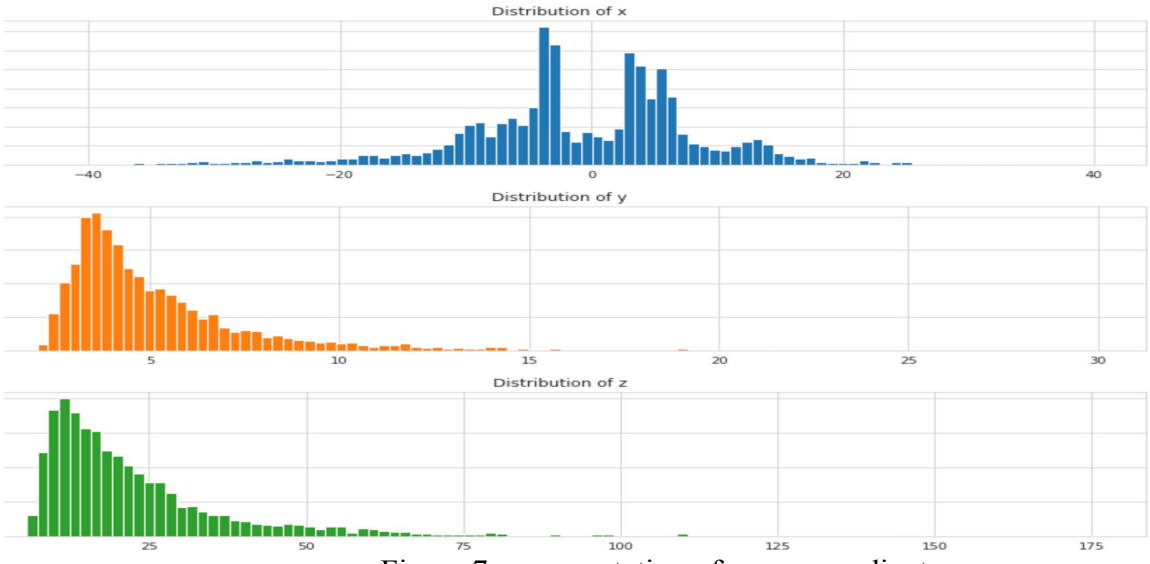


Figure 7: representation of x, y,z coordinates

The 3D coordinates x, y, and z are plotted as a 3D space using the plot library, by considering their respective pitch values as illustrated in below figure 8. In the plot, we can observe the coordinate values of each pitch value and learn their range of values.

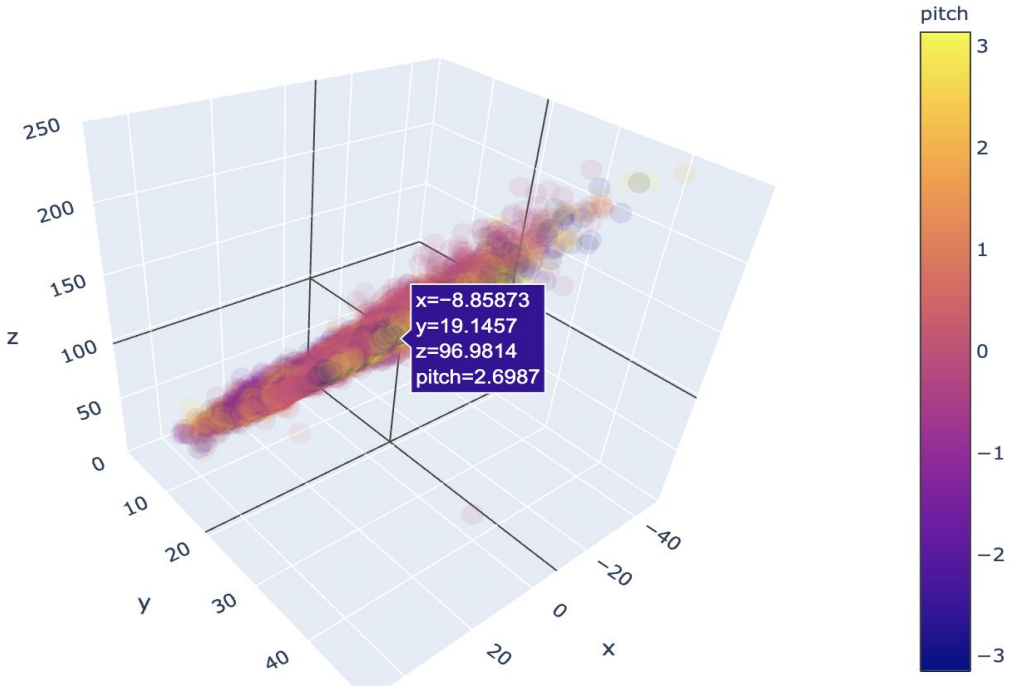


Figure 8: Pitch values of the 3D scatter plot.

CHAPTER 2

LITERATURE SURVEY

The following section presents a brief overview of each of the research papers, journals, and supporting implementations that were considered for the project. Each of them helps and gives a better understanding of the algorithms and also certain important steps to be considered for a successful implementation.

2.1 RESEARCH

Kunyuan Du, Ya Zhang, and Haibing Guan proposed **"From Quantized DNNs to Quantifiable DNNs".**

The above-mentioned paper explains efficient concepts and algorithms for implementing possible bit quantizations to a pre-trained network such as AlexNet and ResNet. The possible bits for quantization include 1-bit, 2-bit, 3-bit, 4-bit, and 32-bit. The quantized DNNs also change both the weight kernel and activation functions into a discrete spatial form. Hence, helps to decrease memory storage issues to a huge extent.[18]

Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun proposed **"Deep Residual Learning for Image Recognition".**

The above-mentioned paper helps focus on developing a better pre-trained base model that considers the residual learning framework in order to ease the process of training the model that is deeper in terms of a number of layers. The layers are reformulated based on a residual function (reference from input layers), instead of learning from an unreferenced function. A base model for

the pre-trained network such as Resent was used for the project.[19]

Tien-Ju Yang1, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze1, and Hartwig Adam proposed **"NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications".**

The above-specified paper was considered for the implementation of NetAdapt, which helps a pre-trained neural network to adapt to a particular constrained resource budget while helping to improve efficiency. The methodology follows the simplification of the existing Neural network layers and applies network pruning. Also, involves the calculation of latency at each network layer and building up a latency look-up table accordingly which later is used for applying net-adapt.[20]

Yu-Hsin Chen, Joel Emer, and Vivienne Sze proposed **"Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks".**

The above-mentioned research paper proposes a novel method for minimizing the data movement which indeed helps decrease the energy consumed on the spatial architecture. This method is referenced as Row Stationery (RS) dataflow, which reuses the local filter weights and feature map by implementing the activations, on a high-dimensional based convolutional network and at the same time reducing the data movement across the sum accumulations that are partially used.[9]

Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis proposed **"TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators".**

The above-specified paper describes the efficient method of implementing the complex DNNs using both intra-layer parallelism, every tile process on the available single layer and inter-layer pipelining, all the layers are executed in a pipelined manner across the tiles. The overall workflow introduces optimization of dataflows to overcome the shortcomings of already prevailing methods for parallel dataflow on DNNs. This method also affects the optimization by increasing the DNN size and complexity.[21]

Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis proposed **"TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory".**

The above research paper was used for the effective understanding of algorithms and research work conducted on hardware architecture as well as scheduling and computational partitioning for a scalable DNN accelerator with the help of 3-dimensional memory. Introduces a hybrid partitioning method that improves parallelism of DNN computations over many accelerators.[22]

CHAPTER 3

DESCRIPTION OF ARCHITECTURE AND ALGORITHM

3.1 INITIAL STEPS AND IMAGE PRE-PROCESSING

The section describes the initial procedure that was followed to obtain the previously mentioned dataset and its pre-processing inorder to obtain an input image of similar properties such as size which will be further fed to each of the designed CNN models.

3.1.1 IMAGE PRE-PROCESSING FOR CIFAR-10 DATASET

The initial step is to download the available open-source CIFAR-10 dataset from the official website. Then to assign the datasets into training and test samples. Next, to understand the details of the image and its properties check for the image dimensions and then normalize the input train and test data to a float32 value, divided by 255. The purpose of normalization is to make the computation easy and suitable for the activation function. To be more precise, always the values of the input images range from 0-255, given any activation function, when a back-propagation algorithm is implemented in order to optimize the network, the larger values of input could raise the problem of exploding gradients or vanishing gradients which is not favorable for developing a best Neural Network model. The below figure 9 shows an example of loading the data and normalizing its input values.

```
# Load the CIFAR10 data.
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Input image dimensions.
input_shape = x_train.shape[1:]

# Normalize data.
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255
```

Figure 9: Code snippet for loading CIFAR-10 Dataset and its Normalization.

3.1.2 IMAGE PRE-PROCESSING FOR PKU-AUTONOMOUS DRIVING DATASET

Initially, start with the dataset download from the Kaggle API into the working environment i.e., Google Colaboratory with runtime GPU from Tesla. The dataset is unzipped and visualized using python libraries. Next, the image has to be pre-processed before using them as an input to the designed model. Pre-processing of images majorly involves resizing the images to a uniform size by techniques such as concatenation. Also, the images are normalized to avoid exploding or vanishing of gradients.

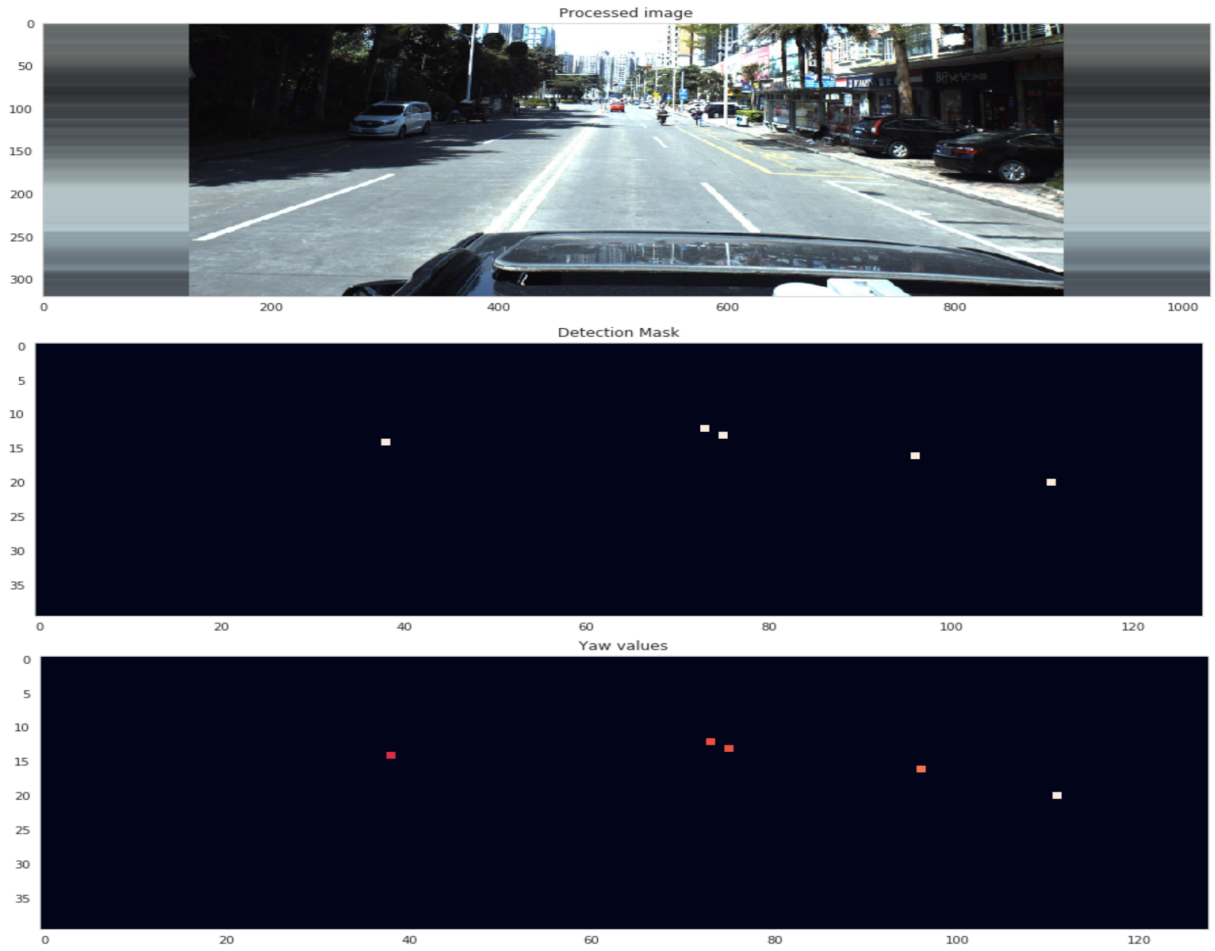


Figure 10: Preprocessed image, Detection mask, and Yaw values of the first input in the training set.

In the above figure 10, the outcome of the preprocessing of images has been illustrated. Also, to understand the dimension of input images input_image.shape was used as shown below.

```
img.shape (320, 1024, 3) std: 0.29819912
mask.shape (40, 128) std: 0.031234737
regr.shape (40, 128, 7) std: 0.013211883
```

3.2 MODEL DESCRIPTION, ALGORITHM, AND ARCHITECTURE

3.2.1 BASELINE MODEL

3.2.1.1 Convolutional Neural Network

The growth of artificial intelligence has been enormous over the past decade, especially with the number of improvements and research work on a particular neural network architecture - Convolutional Neural Network(CNN). For the purpose of the project multiple CNN based architectures were implemented to learn their behavior with the available dataset. All these follow the Convolutional Neural Network(CNN) architecture. Hence before stepping into each of the different CNN networks, first a brief description of Convolutional Neural Network(CNN) is discussed ahead.

Convolutional Neural Network(CNN) is one of the Deep Neural Network-based learning algorithms that includes an input image, weights, and biases at each layer of the network. Often CNNs are referred to as “Shift Invariant or Space Invariant Artificial Neural Networks” because of possessing the translation invariance properties as well as the shared-weights concept. CNN’s are a version of multilayer perceptrons that are regularized. This Neural Network was inspired by the connectivity pattern that exists among the neurons of the human brain. As the name suggests the network implements convolutional linear operation. The architecture includes an input layer, hidden layers such as multiple convolutional layers, pooling layers, normalization layers, fully-connected layers, and finally has an output layer. The basic architecture of CNN is shown below in figure 11.

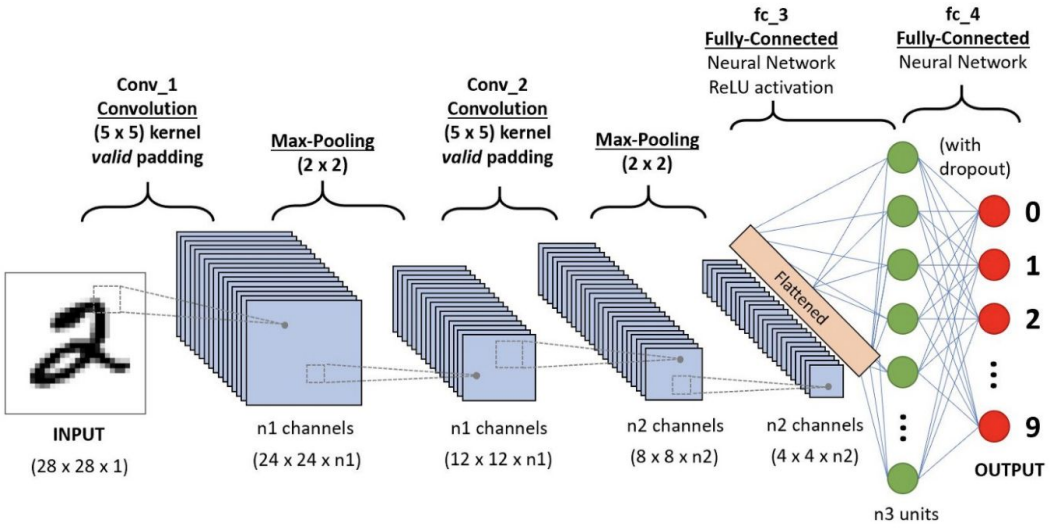


Figure 11: Convolutional Neural Network(CNN) Architecture.

3.2.1.2 EfficientNet

One of the already existing CNN is the EfficientNet architecture which performs well in the aspects of scaling and also increases accuracy. EfficientNet-b0 has demonstrated that the selection baseline model scaling creates a difference in the overall result. EfficientNet when trained with a multiple object based architecture, helps in optimization of accuracy and the number of Floating Point Operations(FLOPS) same as that of the MnasNet.accuracy and FLOPS (Floating Point Operation). The Below table illustrates the architecture for EfficientNet-b0.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

Table 3.2.1.2 EfficientNet – b0 Architecture

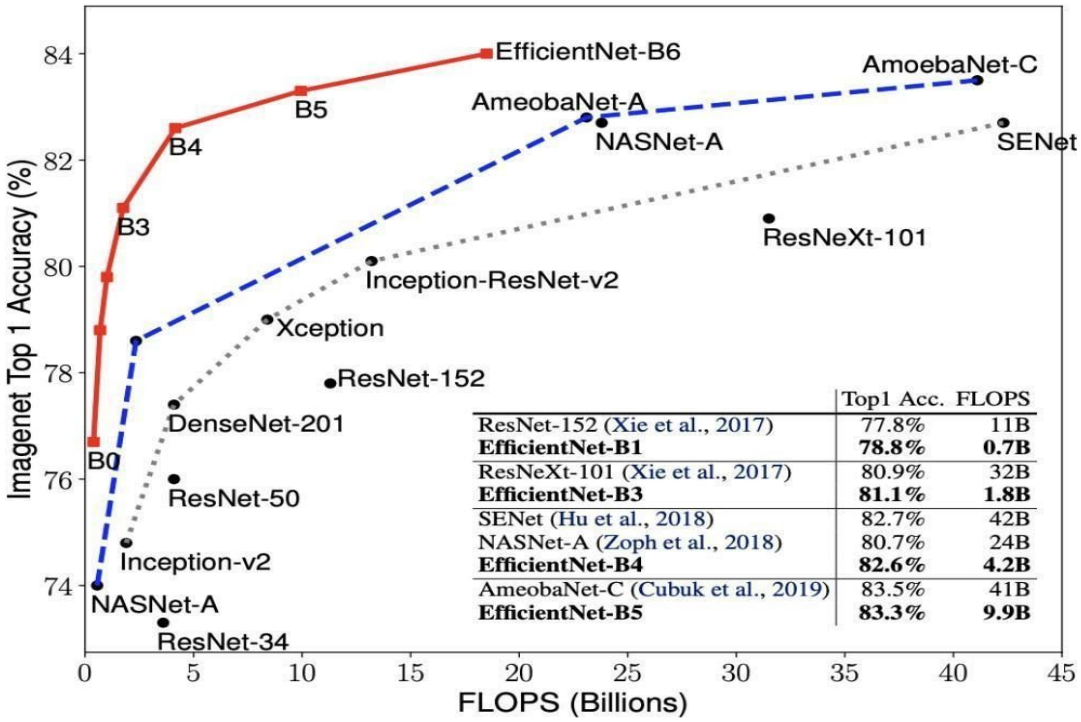


Figure 12: Performance graph of several pre-trained neural networks

By using a pre-trained network such as EfficientNet – b0 weights can be obtained. For the project EfficientNet-b0 was considered as a baseline model for a pre-trained network as this architecture proves to perform well in terms of accuracy. The figure 12 above illustrates a performance graph of many available CNN networks.

As mentioned earlier, EfficientNet being the baseline model, a modified version was developed i.e., CenterNet. The modifications prove to help improve the prediction of the center of objects known as heatmaps, which is a graphical form of data that makes use of a set of color-coding to represent the different pixel values in an input data/image. CenterNet is based on an end-to-end differentiable approach, to be precise a center-point method is replaced with the bounding box

method. Thus, provides an easy method for predicting the key-point which indeed leads to the detection of the center point of the bounding box while neglecting all the nearby objects that are of no interest. 4 double_convolutional layers along with a max-pooling layer is added to the baseline model, which results in the CenterNet model. The double_convolutional layer is defined by the following layer and activation functions: (Conv => BatchNormalization => ReLU) * 2

Figure 13 below illustrates the prediction of heat maps obtained from training the input images and can be concluded that the model has been trained/learnt well since the predictions and accuracy scores are high.

Figure 13: Heatmap of the train_images obtained using the baseline model

3.3 PRE-TRAINED MODELS AND THEIR ARCHITECTURE

3.3.1 RESNET MODEL (RESNET-50)

ResNet (Residual Network) is a Deep neural network, which is a major development in the field of Neural Networks. The fundamental breakthrough for a ResNet model is its compatibility with networks to work with more than 150 layers with less complexity and produce efficient DNN for a large dataset. The utilization of a ResNet for computation of large datasets can be done by its subsidiary network ResNet-50 which then led to the development of a VGG network with 19 convolutional layers and Inception (GoogleNet) with 22 layers which further helped in developing the ResNet to achieve 152 convolutional layers for transfer learning.

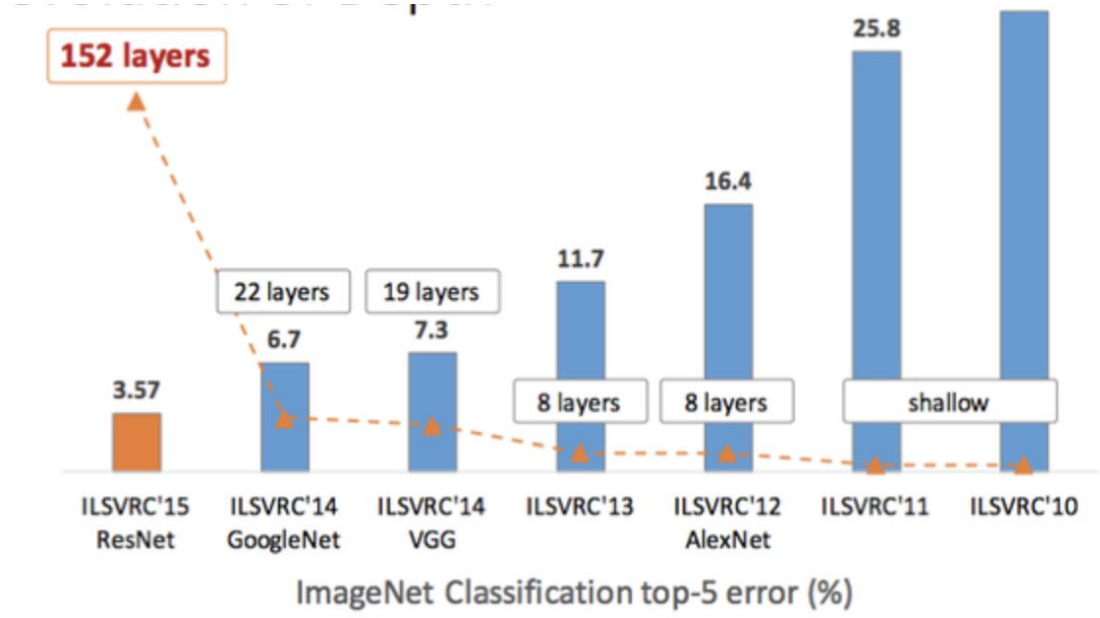


Figure 14: Revolution of Depths

Increasing the number of layers/stacking of multiple convolutional layers does not conceive the idea of a Deep neural network as the deep networks are tedious to train n test due to vanishing gradient problem occurring to the calculation of a gradient using backpropagation to its earlier layer and repeated multiplication causing the gradient to diminish eventually to a very small scale. Resulting in the saturation of the model's performance and eventually degrading rapidly. The

strength of ResNet for application in producing an efficient neural network for autonomous vehicles to determine the 6 degrees of freedom is held firm by its property of skin connection (the strength of Resnet). This process includes a stacking layer of convolutional layers with the input being fed forward to the next layer to avoid the vanishing gradient, this process is called Skin Connection of the Resnet Model.

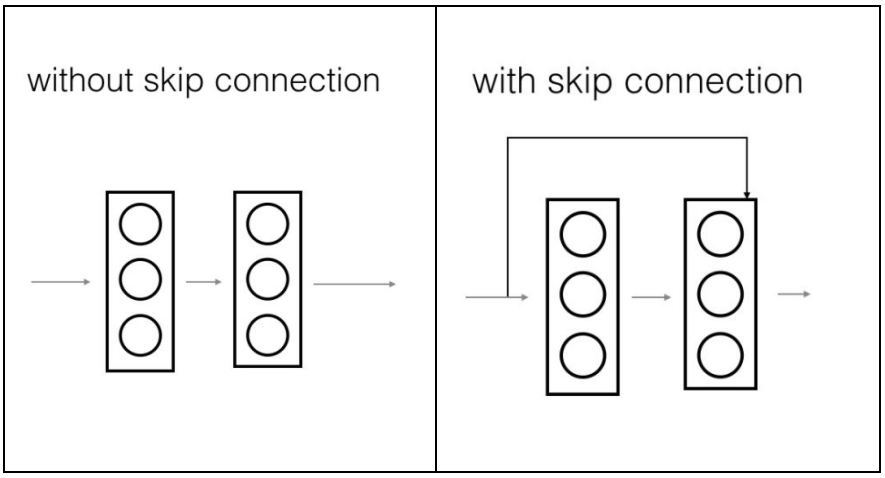


Figure 15: Types of skin connection

In a Resnet model, adding or multiplying two values (data points) can be done if the are of the same size. An important feature of these convolutional layers is to initially convert each data point in the given dataset into fragments of similar data type and size (matrices) for a skin connection to work over a deep convolutional neural network.

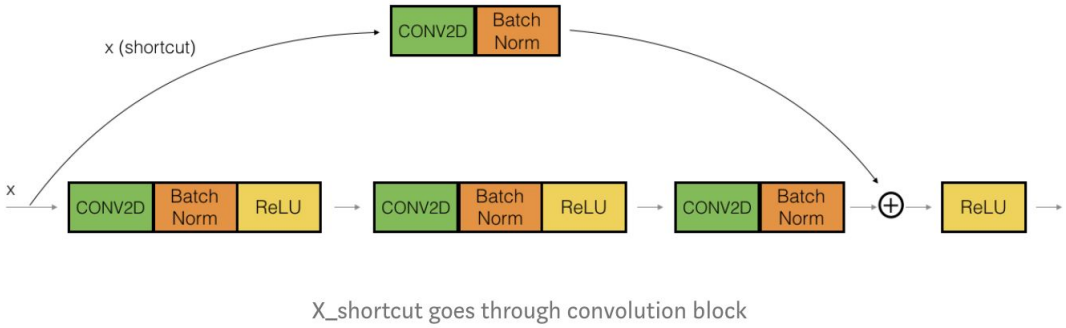


Figure 16: Skin connection over deep neural network

In general, the utilization of a residual network is to apply over a model with multiple stacks of convolutional neural networks who learn network features as they move deeper into the network not by learning from the layers but from the residue left by them after completion of each neural network. Residues can be defined as the subtraction of features learned from the input to the layer.[27][28]

Quantizing the ResNet model:

Utilizing a Resnet model as a baseline model to pre-train the dataset allows the network to be then quantized by a resnet (neural network) to be converted to a minimum of 50 layers deep neural network. This model can be utilized to classify the dataset of different labels and categories to get an accuracy of (Top-1 accuracy %)75.9% and reduce training and test losses between a range of 32-51. This model can be converted/altered to utilize a centernet baseline model to be built over a resnet model for training, validation, and test to reduce the loss and improve efficiency.[29]

After training the ResNet-50 model for quantization of the dataset the parameters for the weights and activation functions were tweaked following the labeled training data. Instead of pessimistically choosing quantization parameters to represent all the numbers observed including the global minimum and maximum, we choose quantization parameters that minimize L2 quantization errors with respect to the collected histogram. We use symmetric quantization (float value 0.0f is quantized to 0) for weights to minimize saturation when 16-bit accumulation is used (this is a workaround for the fact that 16-bit accumulation is needed for high performance in current x86 processors). Then the operators in the trained Resnet50 model were converted into the quantized version of operators using these quantization parameters except for Softmax operators.

Each of these quantized operators has two arguments: Y_scale and Y_zero_point, which are the quantization parameters for the output tensor of the operator. The DUNLOP engine is used in the quantized operators to utilize and handle the quantized arithmetics.[29] ResNet-50 architecture and flow has been added in the APPENDIX B.[30]

3.3.2. ALEXNET

AlexNet was named after its first author Alex Krizhevsky. the mention of Alex/met first appeared in a 2012 paper for image classification which was able to achieve a top-5 error rate among its top five predictions of 15.3%. AlexNet was a winning network at ILSVRC 2012 provided a solution for image classification consisting of over a thousand images classes and the output in the form of vector numbers. the output vector follows the fundamental rule of probability stating that the total probability should be equal to one, therefore the total of all the outputs in the output vector is equal to one.[23]

The AlexNet model can take input of RGB image data of size 256x256, therefore all images in the training dataset and test dataset must be scaled up or scaled down to a size of 256x256 pixels. to train the network, if the training dataset consists of images of size greater than 256x256, the smaller dimension is reshaped to a and cropped to receive a resulting 256x256 pixel data image.

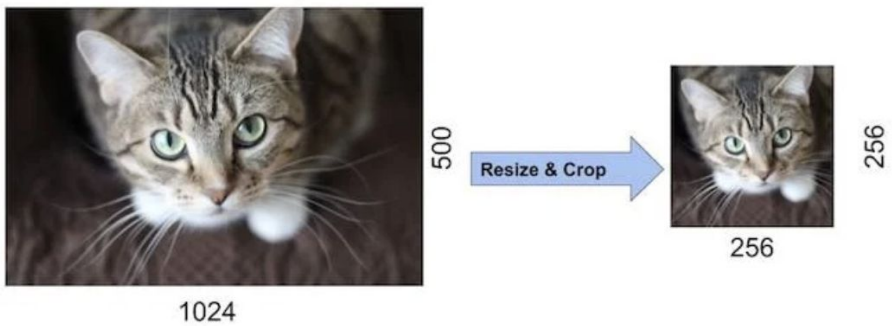


Figure 17: AlexNet Image Compression

The initial images for the first layer of the DNN consist of images to be scaled to a size of 256x256 but the images sent to the next layer were cropped down to 227x247.

3.3.2.1 AlexNet Architecture

AlexNet model totally consists of 8 convolutional layers which are internally segmented as 5 convolutional layers and 3 fully connected layers. Every convolutional layer is specially designed to extract a specific feature from the input to the layer. For example, before the images are used as data input to the first layer, and if the image belongs to a grayscale image, the dataset is converted to an RGB image with the combination of multiple ranges of grayscale from 0-255 where 0 represents complete black and 255 represents while. with a combination of multiple grayscales on the pixel, the image is converted to an RGB image which includes distortion.

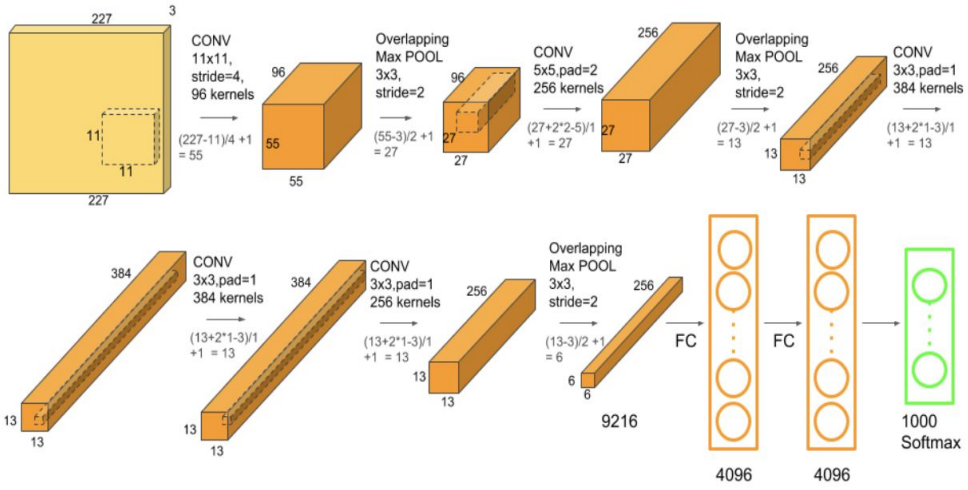


Figure 18: AlexNet Architecture

In the above figure18, the first convolutional layer consists of 96 kernels of size 11x11x3 which

reduces the data size from 227x227 to 55x55. the width and height of the kernel are usually the same but the depth depends on the number of channels. The second convolutional layer consists of max-pooling layers while the third, fourth, and fifth layers are connected directly to each other. The firth layer consists of another max-pooling layer whose output is fed to a fully connected layer that is fed to a softmax classifier with multiple classes for classification.

3.3.3 LeNet

LeNet is the base for multilayer convolutional neural networks for application to image classification for small and large sets of data like hand-written digit recognition.The input for the first layer of LeNet is a 32x32 image which passes through the first convolutional layer with a convolutional filter kernel of size 6 due to which the dimension of the input is convolved to 28x28x6, moving further reducing the length and width of the image but consequently increasing the depth.[24]

3.3.3.1 LeNet Architecture

The LeNet architecture is considered as the beginner's model for image classification as the architecture of a LeNet model is small and light-weighted which allows it to process on a CPU. Although being the small size and lightweight model, LeNet is capable of running heavy/large image datasets on it for image classification for e.g. Cifar dataset. The LeNet is a type of multilayer classification consisting of two convolutional layers and two pooling (average layers) placed alternately to each other followed by a hidden layer and an output layer with softmax classification which is fully connected to the previously hidden layer and the final pooling layer.

Figure 19 below illustrates the LeNet architecture and its layer details.

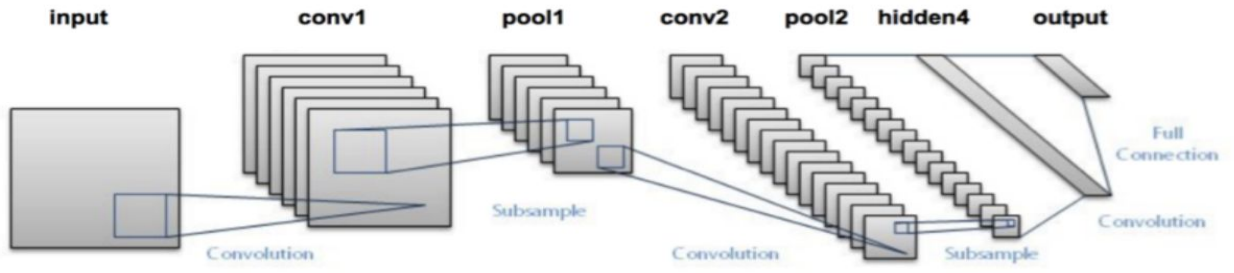


Figure 19: LeNet Architecture

Layer		Feature Map	Size	Kernel Size	Stride	Activation
Input	Image	1	32x32	-	-	-
1	Convolution	6	28x28	5x5	1	tanh
2	Average Pooling	6	14x14	2x2	2	tanh
3	Convolution	16	10x10	5x5	1	tanh
4	Average Pooling	16	5x5	2x2	2	tanh
5	Convolution	120	1x1	5x5	1	tanh
6	FC	-	84	-	-	tanh
Output	FC	-	10	-	-	softmax

Figure 20: Layers in a LeNet DNN.

CHAPTER 4

DESIGN AND IMPLEMENTATION

4.1 PRE-TRAINED CNN NETWORKS

To have a better decision and understanding of the pre-trained model, few of the available pre-trained CNN architectures were implemented. Some of them include AlexNet, ResNet, LeNet, and a combination, ResNet based on CenterNet. All the above mentioned CNN architectures have proven to be best neural network models especially for image classification and computer vision-based datasets. Since the 2 datasets that were considered to be images these CNN architectures have been handy for implementation. Initially, each of the models was trained with an epoch value of 20, defined with the criterion to be cross-entropy loss function and the optimizer defined as Adam optimizer. The initial learning rate was fixed to a value of 0.001. A comparative result and conclusions will be illustrated later in the upcoming chapters. The main goal of this comparative study was to decide on the base model on which further new customizations such as quantization, network pruning, and dataflow scheduling have to be applied.

4.2 QUANTIZATION

Quantization plays a major role for this project. Usually a CNN such as AlexNet, ResNet and many others consumes a huge amount of memory space, and is not favourable especially when the proposed Network has dense layers and is complex. A typical ResNet-101 takes around 170MB[19] of storage space whereas in the case of AlexNet around 250MB[25]. Apart from storage consumption these dense networks also utilize billions of FLOPs for each image during the

inference stage when the run time type is a GPU. Many techniques have been proposed to overcome this issue and one of the most crucial one that has a better impact is “Quantization”. The Quantization approach retains the topology of the network by reducing the size of the model by converting the parameters to a low-precision. For the implementation purpose the weights which originally were a 32-bit Floating point were quantized to a 1-bit and feature maps were reduced to a 8-bit low precision representation. Also, to stabilize the training process to a huge extent a weight threshold of 0.1 was defined and a hysteresis loop for these 1-bit weights is illustrated in the below figure 20.

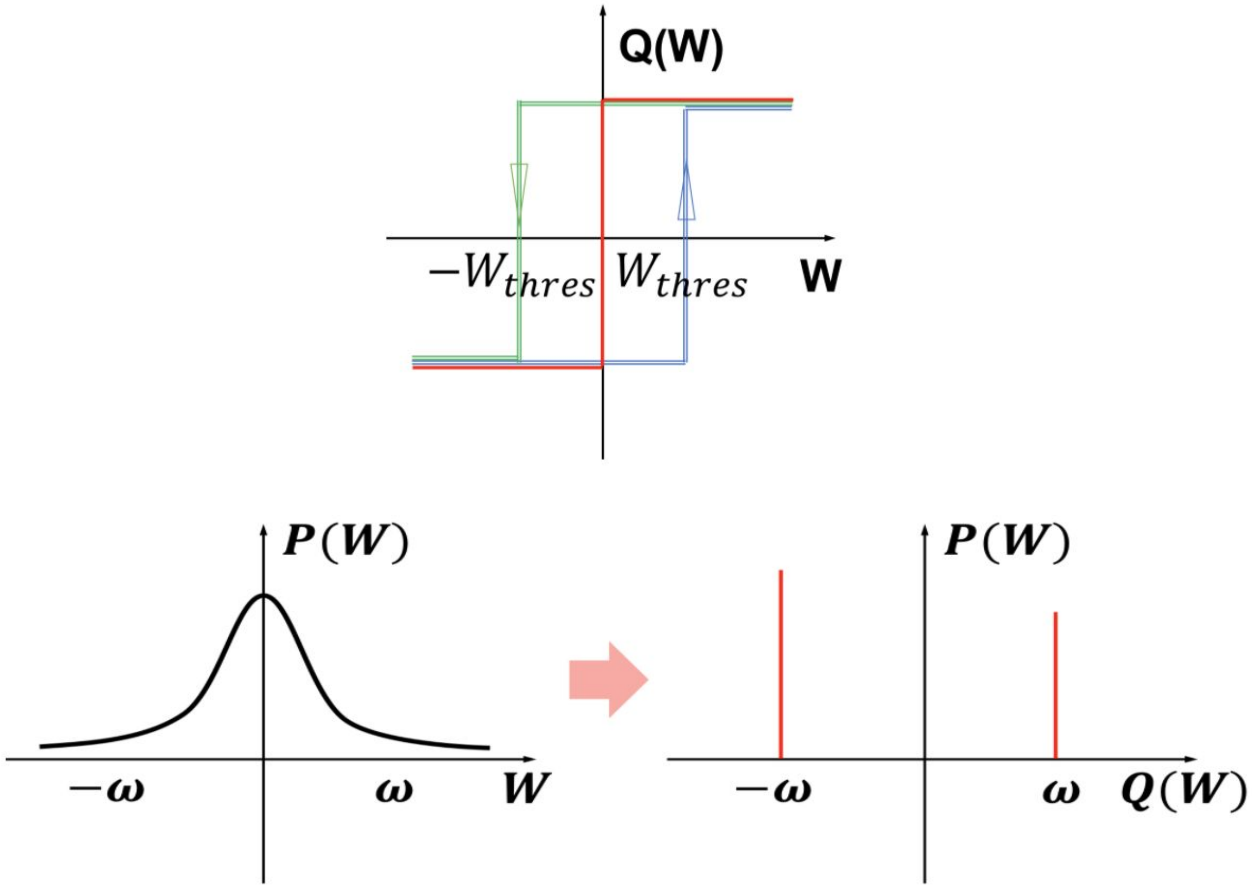


Figure 21: (a) (top)Illustration of a hysteresis loop for the 1-bit weights with a threshold of 0.1, (b)(left to right bottom) Full-precision of weight distribution, converted 1-bit quantized weight distribution.

Binary Neural Network (BNN) is one of the important approaches for quantization methods. Using BNN, the weights and activation functions are expressed in a 1-bit form ranging between (-w, +w) instead of a full-precision, thus utilizing a memory space of only 1-bit for each variable resulting in a small binarized model of 1/32.[26] These weight optimizations are applied when the customized SGD optimizer function is called from the train class. The figure 21 gives an overall idea of the algorithm for a defined variable $x \in R_n$, the binary value x_b is determined by the sign. Here B(x) is the binary function, with $||x||_1/n$ is the scaling factor used to maintain a value range.

$$\textbf{Forward: } x_b = B(x) = \frac{\|x\|_1}{n}\text{sign}(x),$$

$$\textbf{Backward: } \frac{\partial B}{\partial x} \approx \mathbb{I}\{|x| < 1\}.$$

Figure 22: Methodology for Binarization implementation.

The above proposed Quantization was applied to ResNet, the corresponding results and discussions have been explained in the upcoming chapter. Figure 22 below illustrates the model summary with the details of each layer and its respective kernel size, stride and padding.

```
cuda
=> Building model...
ResNet_Cifar(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (layer1): Sequential(
    (0): BasicBlock(
      (binact1): BinActLayer()
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (binact2): BinActLayer()
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (binact1): BinActLayer()
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (binact2): BinActLayer()
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (2): BasicBlock(
      (binact1): BinActLayer()
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (binact2): BinActLayer()
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (binact1): BinActLayer()
      (conv1): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (binact2): BinActLayer()
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(16, 32, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
```

Figure 23: (a) Quantized ResNet Model Summary.

```
  (1): BasicBlock(
    (binact1): BinActLayer()
    (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (binact2): BinActLayer()
    (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (2): BasicBlock(
    (binact1): BinActLayer()
    (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (binact2): BinActLayer()
    (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(layer3): Sequential(
  (0): BasicBlock(
    (binact1): BinActLayer()
    (conv1): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (binact2): BinActLayer()
    (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)
    (downsample): Sequential(
      (0): Conv2d(32, 64, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (binact1): BinActLayer()
    (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)
    (relu): ReLU(inplace=True)
    (binact2): BinActLayer()
    (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)
  )
  (2): BasicBlock(
    (binact1): BinActLayer()
    (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)
    (relu): ReLU(inplace=True)
    (binact2): BinActLayer()
    (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)
  )
)
(avgpool): AvgPool2d(kernel_size=8, stride=1, padding=0)
```

Figure 23: (b) Quantized ResNet Model Summary.

4.3 NETADAPT APPLIED ON ALEXNET

DNNs used for Artificial Intelligence applications are often constrained to the available resources. Most of the proposed work in order to improve the efficiency of DNNs are based on "indirect metrics", i.e., the weights and Multiply-Accumulate Operations duplicates with the increase in the amount of resource consumed for a pre-defined network. However, the approach of indirect metrics does not provide a precise approximation value in the case of dealing with energy consumption and latency for real time applications. The NetAdapt follows a "Direct metrics" approach inside the optimization network. These metrics are evaluated from the empirical values obtained from the target resource. First the pre-trained CNN architecture in this case AlexNet is trained in a normal way and the trained model is saved as model.pth. As a next step, the latency look-up table(LUT) is built, which involves measuring latency for each individual layer in the AlexNet as already defined. Finally, we apply NetAdapt to the network with the following flow of steps: Selecting the number of filters, where a decision is made on the number of filters to be used for each layer from the measurements of the empirical values. Next being, to select the type of filter to be used and finally to fine tune the network from end-to-end. The algorithm flow and pseudo code for NetAdapt is illustrated in the figure 23 (a) & (b) below.

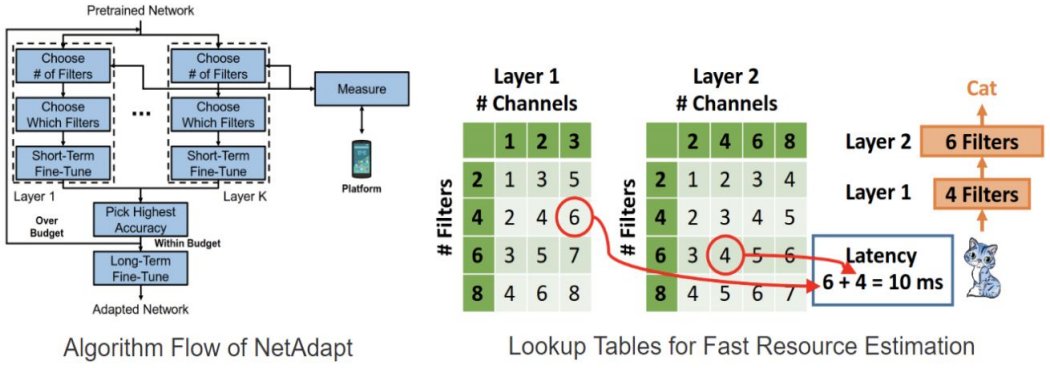


Figure 24: (a)NetAdapt algorithm flow and its implementation

Algorithm 1: NetAdapt

Input: Pretrained Network: Net_0 (with K CONV and FC layers), Resource Budget: Bud, Resource Reduction Schedule: ΔR_i

Output: Adapted Network Meeting the Resource Budget: $\hat{Net}$

```
i = 0;
Res_i = TakeEmpiricalMeasurement(Net_i);
while Res_i > Bud do
    Con = Res_i - ΔR_i;
    for k from 1 to K do
        /* TakeEmpiricalMeasurement is also called inside
           ChooseNumFilters for choosing the correct number of filters
           that satisfies the constraint (i.e., current budget). */
        N_Filt_k, Res_Simp_k = ChooseNumFilters(Net_i, k, Con);
        Net_Simp_k = ChooseWhichFilters(Net_i, k, N_Filt_k);
        Net_Simp_k = ShortTermFineTune(Net_Simp_k);
    Net_{i+1}, Res_{i+1} = PickHighestAccuracy(Net_Simp_:, Res_Simp_:);
    i = i + 1;
N̂et = LongTermFineTune(Net_i);
return N̂et;
```

Figure 24: (b) Pseudocode for NetAdapt.

The most important stage of NetAdapt is making use of "Fast Resource Consumption Estimation"[20] which eliminates the tedious task of calculating the empirical values at a faster rate by making use of the pre-built LUT for a defined network and resource availability. A table calculated based on network is not considered because of its exponential characteristics. Hence, a layer-wise LUT is considered for empirical values. For the project a LUT was obtained for the layers in the AlexNet model (saved as lut_alexnet.pkl) and the latency calculated for the first feature has been illustrated in the figure 24 below.

```
([('features.0',
   {'groups': 1,
    'input_feature_map_size': [1, 3, 224, 224],
    'is_depthwise': False,
    'kernel_size': (11, 11),
    'latency': {(3, 8): 0.0019575567245483398,
     (3, 16): 0.002051778793334961,
     (3, 24): 0.002144847869873047,
     (3, 32): 0.0022347893714904786,
     (3, 40): 0.0033642854690551756,
     (3, 48): 0.003461637496948242,
     (3, 56): 0.0035645837783813477,
     (3, 64): 0.0036860408782958985},
    'layer_type_str': 'Conv2d',
    'num_in_channels': 3,
    'num_out_channels': 64,
    'padding': (2, 2),
    'stride': (4, 4)}),
```

Figure 25: Snippet of Latency calculated for first feature.

CHAPTER 5

SIMULATION RESULTS AND VERIFICATION

5.1 Training And Testing Results For Different CNN Architectures

5.1.1 AlexNet - Results

The CIFAR-10 dataset was trained using AlexNet model with the following hyperparameters: Number of Epochs - 20, Learning rate - 0.001 with Adam optimizer, Loss - Cross Entropy function, lr_scheduler - MultiStepLR, Device - GPU. The model summary, training and test results for the AlexNet model using the CIFAR-10 dataset have been illustrated in figure 26 & 27 below. The accuracy and loss plots for train vs test are illustrated in figure 28 & 29.

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=1024, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

Figure 26: AlexNet Model Summary along with kernel size, stride and padding details.

```
===> epoch: 0/20
train:
 782/782 [==============================>]  Step: 22ms | Tot: 22s482ms | Loss: 1.8757 | Acc: 26.558% (13279/50000)
(1466.7729167938232, 0.26558)
test:
 157/157 [==============================>]  Step: 12ms | Tot: 2s506ms | Loss: 1.6056 | Acc: 38.070% (3807/10000)

===> epoch: 1/20
train:
 782/782 [==============================>]  Step: 23ms | Tot: 22s365ms | Loss: 1.4868 | Acc: 43.676% (21838/50000)
(1162.703111410141, 0.43676)
test:
 157/157 [==============================>]  Step: 12ms | Tot: 2s444ms | Loss: 1.3562 | Acc: 49.640% (4964/10000)

===> epoch: 2/20
train:
 782/782 [==============================>]  Step: 21ms | Tot: 22s430ms | Loss: 1.3229 | Acc: 51.244% (25622/50000)
(1034.4793426394463, 0.51244)
test:
 157/157 [==============================>]  Step: 9ms | Tot: 2s484ms | Loss: 1.2592 | Acc: 54.300% (5430/10000)

===> epoch: 3/20
train:
 782/782 [==============================>]  Step: 21ms | Tot: 22s356ms | Loss: 1.2170 | Acc: 55.898% (27949/50000)
(951.6871206760406, 0.55898)
test:
 157/157 [==============================>]  Step: 7ms | Tot: 2s462ms | Loss: 1.1741 | Acc: 57.180% (5718/10000)

===> epoch: 4/20
train:
 782/782 [==============================>]  Step: 18ms | Tot: 22s382ms | Loss: 1.1358 | Acc: 59.094% (29547/50000)
(888.1891688704491, 0.59094)
test:
 157/157 [==============================>]  Step: 10ms | Tot: 2s501ms | Loss: 1.0949 | Acc: 59.910% (5991/10000)

===> epoch: 5/20
train:
 782/782 [==============================>]  Step: 20ms | Tot: 22s240ms | Loss: 1.0751 | Acc: 61.284% (30642/50000)
(840.6984245181084, 0.61284)
test:
 157/157 [==============================>]  Step: 10ms | Tot: 2s416ms | Loss: 1.0315 | Acc: 63.030% (6303/10000)

===> epoch: 6/20
train:
 782/782 [==============================>]  Step: 21ms | Tot: 22s152ms | Loss: 1.0212 | Acc: 63.412% (31706/50000)
(798.5762397050858, 0.63412)
test:
 157/157 [==============================>]  Step: 8ms | Tot: 2s434ms | Loss: 1.0596 | Acc: 61.580% (6158/10000)
```

Figure 27 : Training and test results for AlexNet.

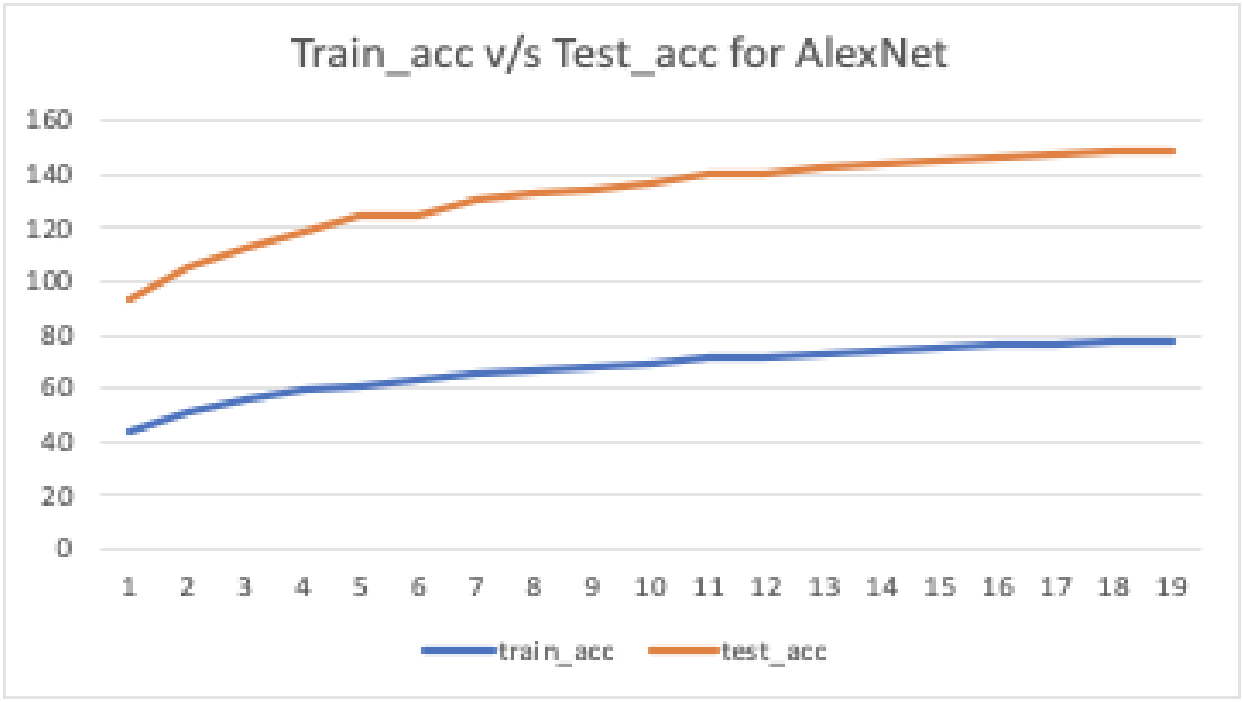


Figure 28 : Training vs Test Accuracy for AlexNet.

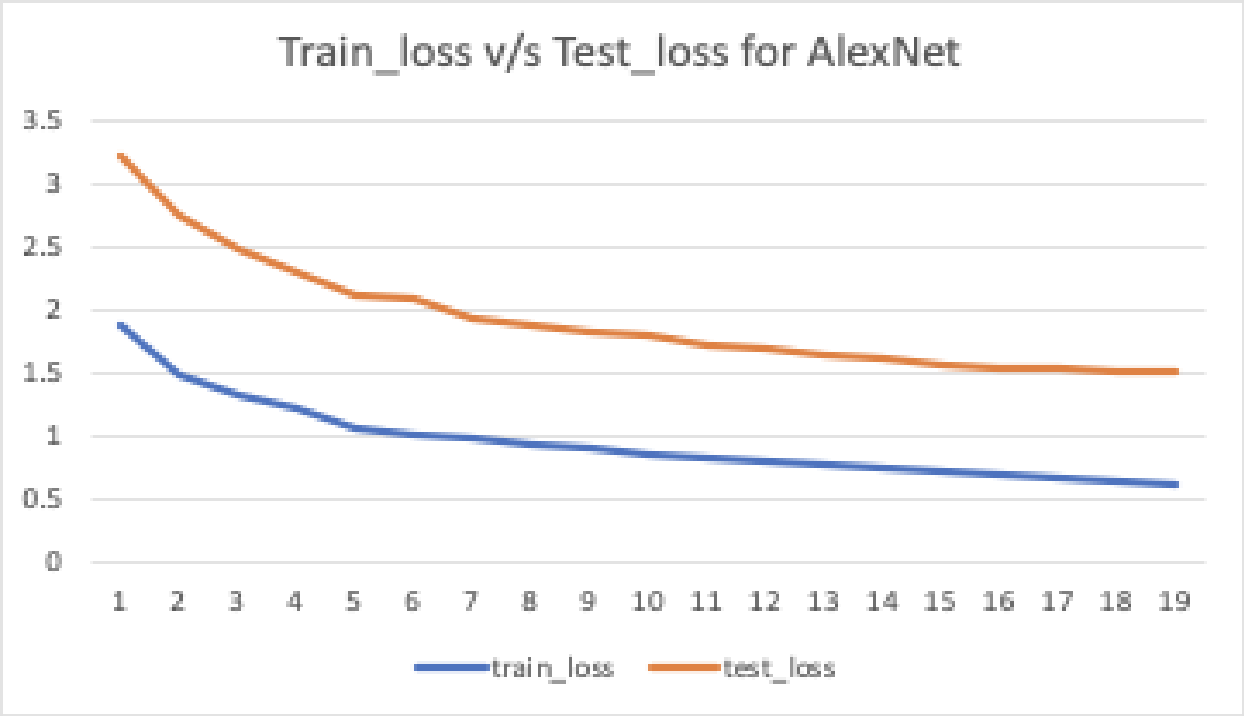


Figure 29 : Training vs Test Loss for AlexNet.

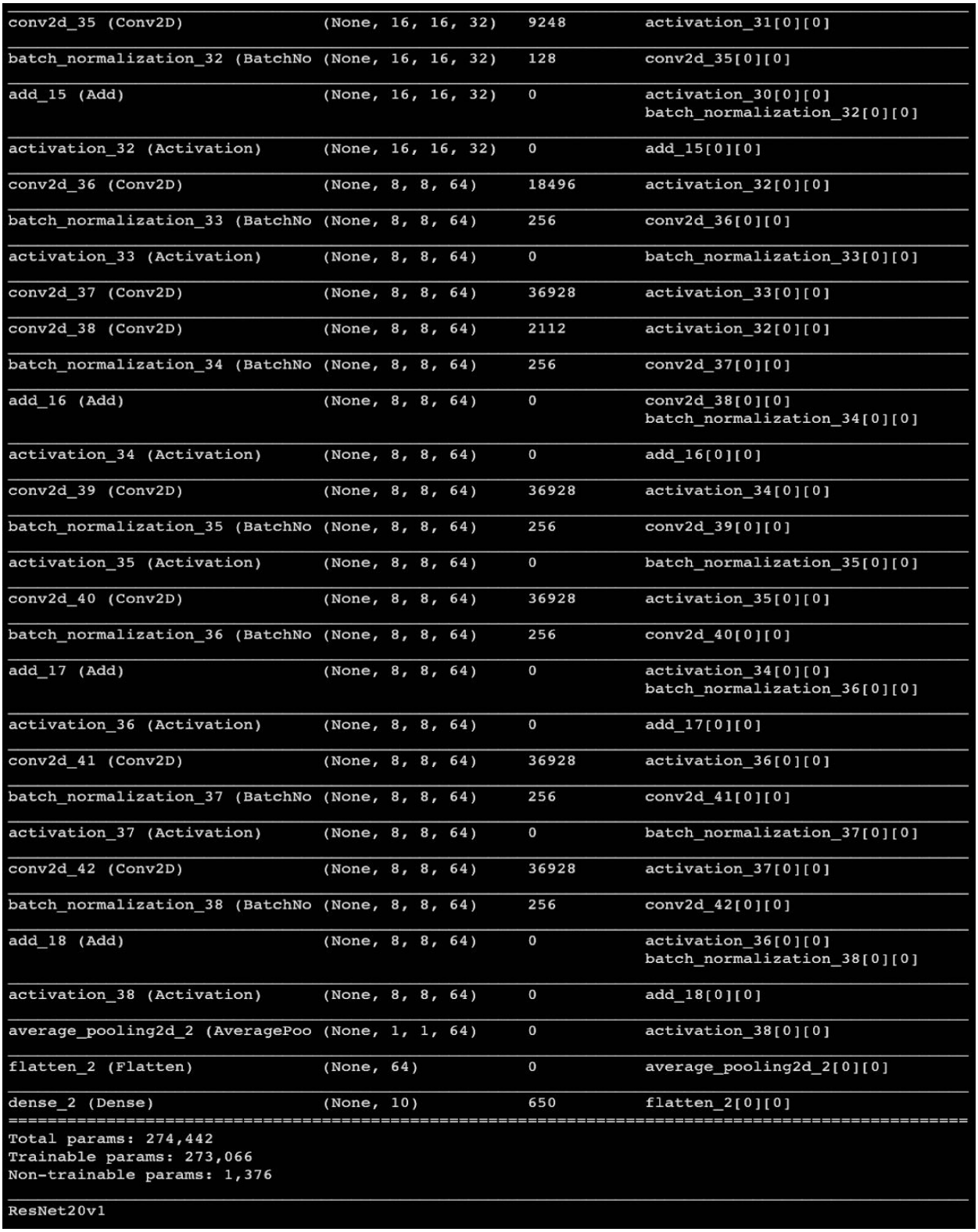
5.1.2 ResNet - Results

The CIFAR-10 dataset was trained using ResNet model with the following hyperparameters: Number of Epochs - 20, Learning rate - 0.001 with Adam optimizer, Loss - Cross Entropy function, Device - GPU.The model summary, training and test results for the ResNet model using the CIFAR-10 & PKU-Autonomous Driving dataset have been illustrated in figures 28, 29 & 30 below.

Learning rate: 0.001
Model: "model_2"

Layer (type)	Output Shape	Param #	Connected to
input_2 (InputLayer)	(None, 32, 32, 3)	0	
conv2d_22 (Conv2D)	(None, 32, 32, 16)	448	input_2[0][0]
batch_normalization_20 (BatchNo	(None, 32, 32, 16)	64	conv2d_22[0][0]
activation_20 (Activation)	(None, 32, 32, 16)	0	batch_normalization_20[0][0]
conv2d_23 (Conv2D)	(None, 32, 32, 16)	2320	activation_20[0][0]
batch_normalization_21 (BatchNo	(None, 32, 32, 16)	64	conv2d_23[0][0]
activation_21 (Activation)	(None, 32, 32, 16)	0	batch_normalization_21[0][0]
conv2d_24 (Conv2D)	(None, 32, 32, 16)	2320	activation_21[0][0]
batch_normalization_22 (BatchNo	(None, 32, 32, 16)	64	conv2d_24[0][0]
add_10 (Add)	(None, 32, 32, 16)	0	activation_20[0][0] batch_normalization_22[0][0]
activation_22 (Activation)	(None, 32, 32, 16)	0	add_10[0][0]
conv2d_25 (Conv2D)	(None, 32, 32, 16)	2320	activation_22[0][0]
batch_normalization_23 (BatchNo	(None, 32, 32, 16)	64	conv2d_25[0][0]
activation_23 (Activation)	(None, 32, 32, 16)	0	batch_normalization_23[0][0]
conv2d_26 (Conv2D)	(None, 32, 32, 16)	2320	activation_23[0][0]
batch_normalization_24 (BatchNo	(None, 32, 32, 16)	64	conv2d_26[0][0]
add_11 (Add)	(None, 32, 32, 16)	0	activation_22[0][0] batch_normalization_24[0][0]
activation_24 (Activation)	(None, 32, 32, 16)	0	add_11[0][0]
conv2d_27 (Conv2D)	(None, 32, 32, 16)	2320	activation_24[0][0]
batch_normalization_25 (BatchNo	(None, 32, 32, 16)	64	conv2d_27[0][0]
activation_25 (Activation)	(None, 32, 32, 16)	0	batch_normalization_25[0][0]
conv2d_28 (Conv2D)	(None, 32, 32, 16)	2320	activation_25[0][0]
batch_normalization_26 (BatchNo	(None, 32, 32, 16)	64	conv2d_28[0][0]
add_12 (Add)	(None, 32, 32, 16)	0	activation_24[0][0] batch_normalization_26[0][0]
activation_26 (Activation)	(None, 32, 32, 16)	0	add_12[0][0]
conv2d_29 (Conv2D)	(None, 16, 16, 32)	4640	activation_26[0][0]
batch_normalization_27 (BatchNo	(None, 16, 16, 32)	128	conv2d_29[0][0]
activation_27 (Activation)	(None, 16, 16, 32)	0	batch_normalization_27[0][0]
conv2d_30 (Conv2D)	(None, 16, 16, 32)	9248	activation_27[0][0]
conv2d_31 (Conv2D)	(None, 16, 16, 32)	544	activation_26[0][0]

Figure 30 : ResNet Model Summary along with kernel size, stride and padding details.



conv2d_35 (Conv2D)	(None, 16, 16, 32)	9248	activation_31[0][0]
batch_normalization_32 (BatchNo	(None, 16, 16, 32)	128	conv2d_35[0][0]
add_15 (Add)	(None, 16, 16, 32)	0	activation_30[0][0] batch_normalization_32[0][0]
activation_32 (Activation)	(None, 16, 16, 32)	0	add_15[0][0]
conv2d_36 (Conv2D)	(None, 8, 8, 64)	18496	activation_32[0][0]
batch_normalization_33 (BatchNo	(None, 8, 8, 64)	256	conv2d_36[0][0]
activation_33 (Activation)	(None, 8, 8, 64)	0	batch_normalization_33[0][0]
conv2d_37 (Conv2D)	(None, 8, 8, 64)	36928	activation_33[0][0]
conv2d_38 (Conv2D)	(None, 8, 8, 64)	2112	activation_32[0][0]
batch_normalization_34 (BatchNo	(None, 8, 8, 64)	256	conv2d_37[0][0]
add_16 (Add)	(None, 8, 8, 64)	0	conv2d_38[0][0] batch_normalization_34[0][0]
activation_34 (Activation)	(None, 8, 8, 64)	0	add_16[0][0]
conv2d_39 (Conv2D)	(None, 8, 8, 64)	36928	activation_34[0][0]
batch_normalization_35 (BatchNo	(None, 8, 8, 64)	256	conv2d_39[0][0]
activation_35 (Activation)	(None, 8, 8, 64)	0	batch_normalization_35[0][0]
conv2d_40 (Conv2D)	(None, 8, 8, 64)	36928	activation_35[0][0]
batch_normalization_36 (BatchNo	(None, 8, 8, 64)	256	conv2d_40[0][0]
add_17 (Add)	(None, 8, 8, 64)	0	activation_34[0][0] batch_normalization_36[0][0]
activation_36 (Activation)	(None, 8, 8, 64)	0	add_17[0][0]
conv2d_41 (Conv2D)	(None, 8, 8, 64)	36928	activation_36[0][0]
batch_normalization_37 (BatchNo	(None, 8, 8, 64)	256	conv2d_41[0][0]
activation_37 (Activation)	(None, 8, 8, 64)	0	batch_normalization_37[0][0]
conv2d_42 (Conv2D)	(None, 8, 8, 64)	36928	activation_37[0][0]
batch_normalization_38 (BatchNo	(None, 8, 8, 64)	256	conv2d_42[0][0]
add_18 (Add)	(None, 8, 8, 64)	0	activation_36[0][0] batch_normalization_38[0][0]
activation_38 (Activation)	(None, 8, 8, 64)	0	add_18[0][0]
average_pooling2d_2 (AveragePoo	(None, 1, 1, 64)	0	activation_38[0][0]
flatten_2 (Flatten)	(None, 64)	0	average_pooling2d_2[0][0]
dense_2 (Dense)	(None, 10)	650	flatten_2[0][0]

Total params: 274,442
Trainable params: 273,066
Non-trainable params: 1,376

ResNet20v1

Figure 31 : ResNet Model Summary along with kernel size, stride and padding details.

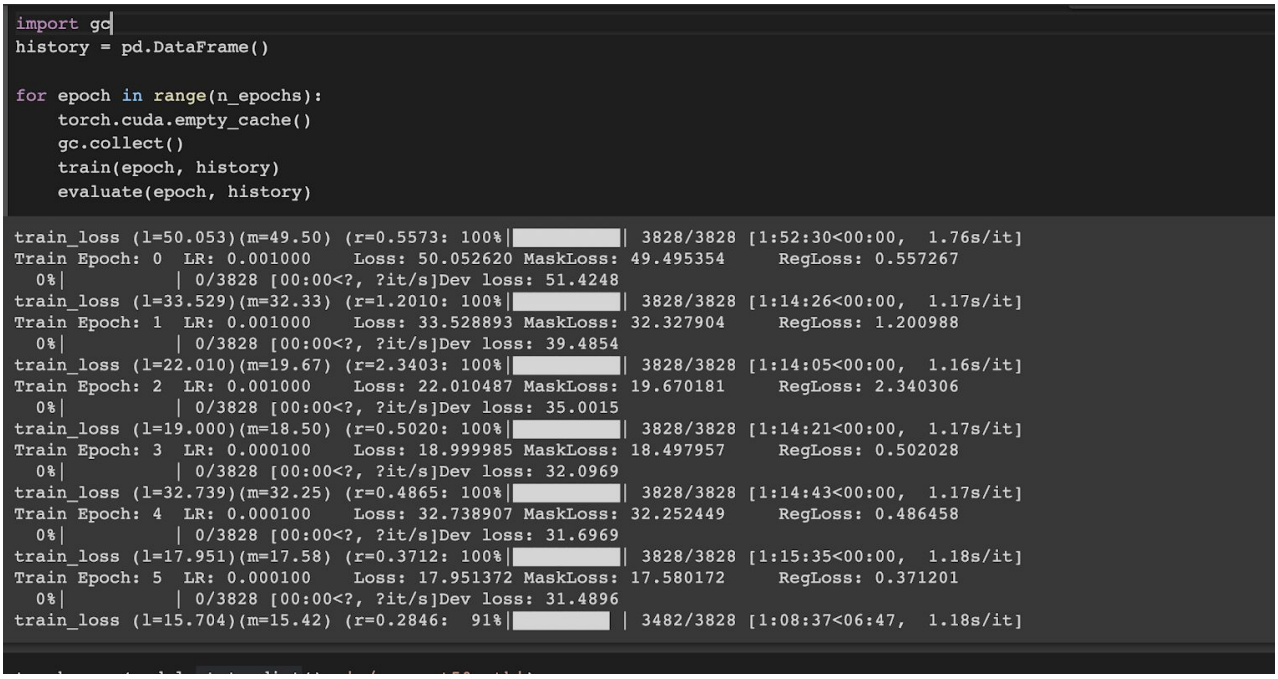


Figure 32 : Training results for ResNet model using PKU-Autonomous driving dataset.

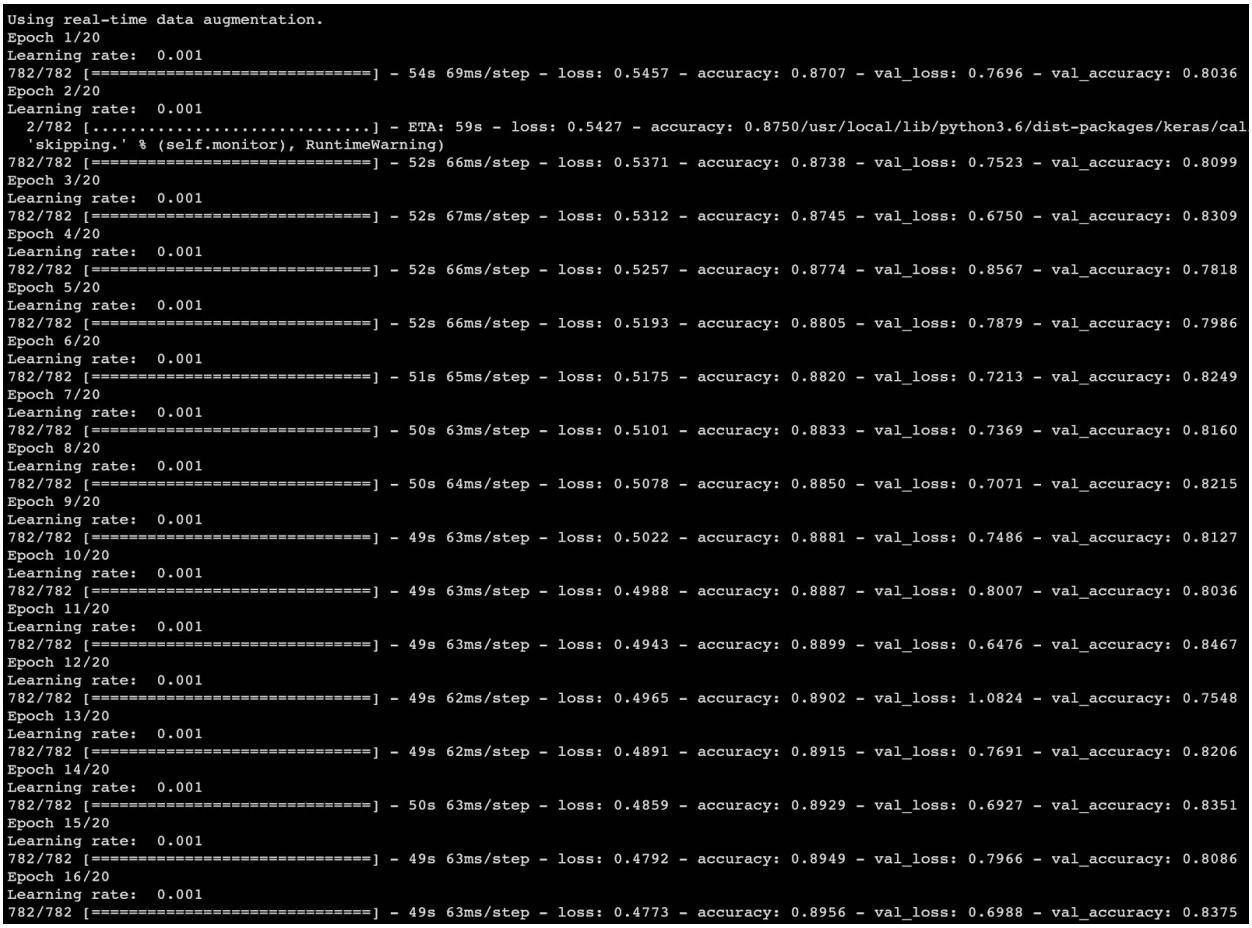


Figure 33(a) : Training and test results for ResNet model using CIFAR 10 dataset.

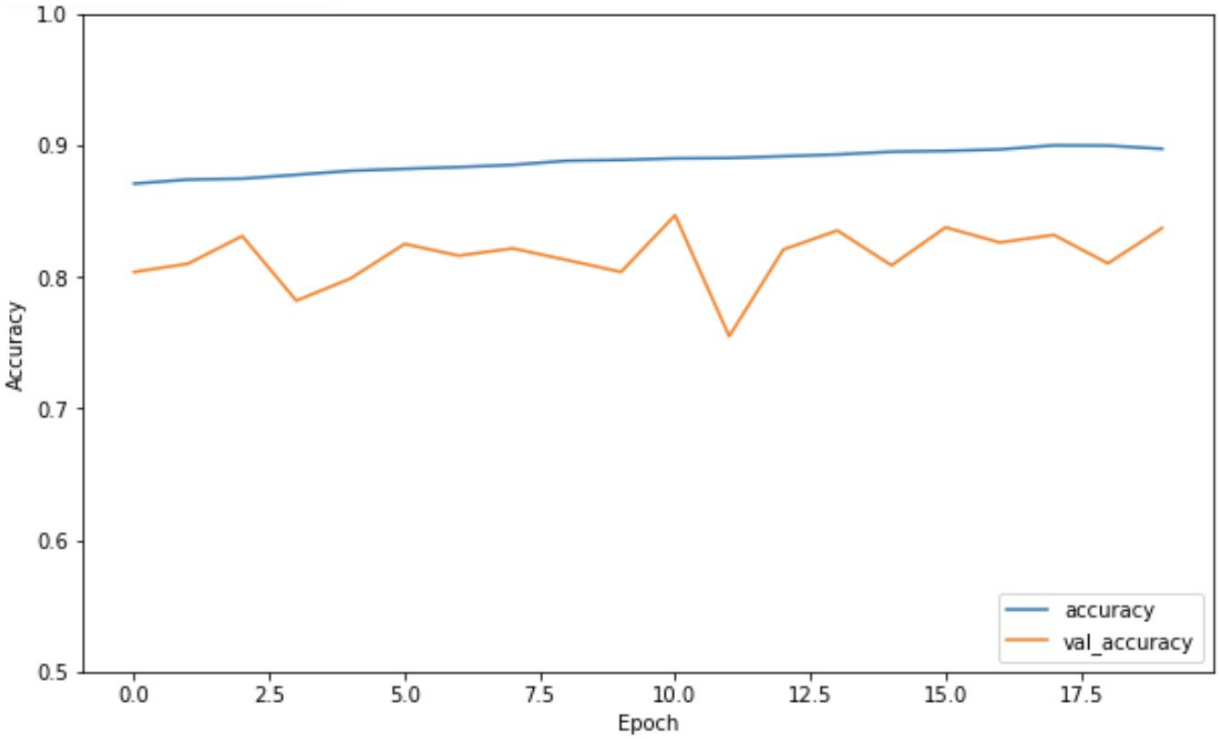


Figure 33 (b) : Accuracy v/s Epoch for ResNet model using CIFAR-10 dataset.

5.1.3 LeNet - Results

The CIFAR-10 dataset was trained using LeNet model with the following hyperparameters: Number of Epochs - 20, Learning rate - 0.001 with Adam optimizer, Loss - Cross Entropy function, lr_scheduler - MultiStepLR, Device - GPU. The model summary, training and test results for the LeNet model using the CIFAR-10 dataset have been illustrated in figures 34 & 35 below.

```
LeNet(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

Figure 34 : LeNet Model Summary along with kernel size, stride and padding details.

```
===> epoch: 0/20
train:
 782/782 [==============================>]  Step: 17ms | Tot: 18s668ms | Loss: 1.8336 | Acc: 31.226% (15613/50000)
(1433.8574421405792, 0.31226)
test:
 157/157 [==============================>]  Step: 13ms | Tot: 3s231ms | Loss: 1.7453 | Acc: 35.720% (3572/10000)

===> epoch: 1/20
train:
 782/782 [==============================>]  Step: 20ms | Tot: 18s743ms | Loss: 1.7051 | Acc: 36.872% (18436/50000)
(1333.382658958435, 0.36872)
test:
 157/157 [==============================>]  Step: 16ms | Tot: 3s256ms | Loss: 1.6528 | Acc: 38.700% (3870/10000)

===> epoch: 2/20
train:
 782/782 [==============================>]  Step: 19ms | Tot: 18s806ms | Loss: 1.6030 | Acc: 41.212% (20606/50000)
(1253.5423909425735, 0.41212)
test:
 157/157 [==============================>]  Step: 10ms | Tot: 3s275ms | Loss: 1.5931 | Acc: 41.370% (4137/10000)

===> epoch: 3/20
train:
 782/782 [==============================>]  Step: 19ms | Tot: 18s844ms | Loss: 1.5474 | Acc: 43.704% (21852/50000)
(1210.0334109067917, 0.43704)
test:
 157/157 [==============================>]  Step: 9ms | Tot: 3s256ms | Loss: 1.5043 | Acc: 44.760% (4476/10000)

===> epoch: 4/20
train:
 782/782 [==============================>]  Step: 19ms | Tot: 18s839ms | Loss: 1.5106 | Acc: 45.000% (22500/50000)
(1181.2867991924286, 0.45)
test:
 157/157 [==============================>]  Step: 11ms | Tot: 3s170ms | Loss: 1.5013 | Acc: 45.550% (4555/10000)

===> epoch: 5/20
train:
 782/782 [==============================>]  Step: 20ms | Tot: 18s613ms | Loss: 1.4797 | Acc: 46.010% (23005/50000)
(1157.0909713506699, 0.4601)
test:
 157/157 [==============================>]  Step: 10ms | Tot: 3s265ms | Loss: 1.4577 | Acc: 46.550% (4655/10000)

===> epoch: 6/20
train:
 782/782 [==============================>]  Step: 18ms | Tot: 18s831ms | Loss: 1.4569 | Acc: 47.154% (23577/50000)
(1139.2800818681717, 0.47154)
test:
 157/157 [==============================>]  Step: 13ms | Tot: 3s218ms | Loss: 1.4335 | Acc: 47.600% (4760/10000)
```

Figure 35 (a): Training and test results for LeNet.

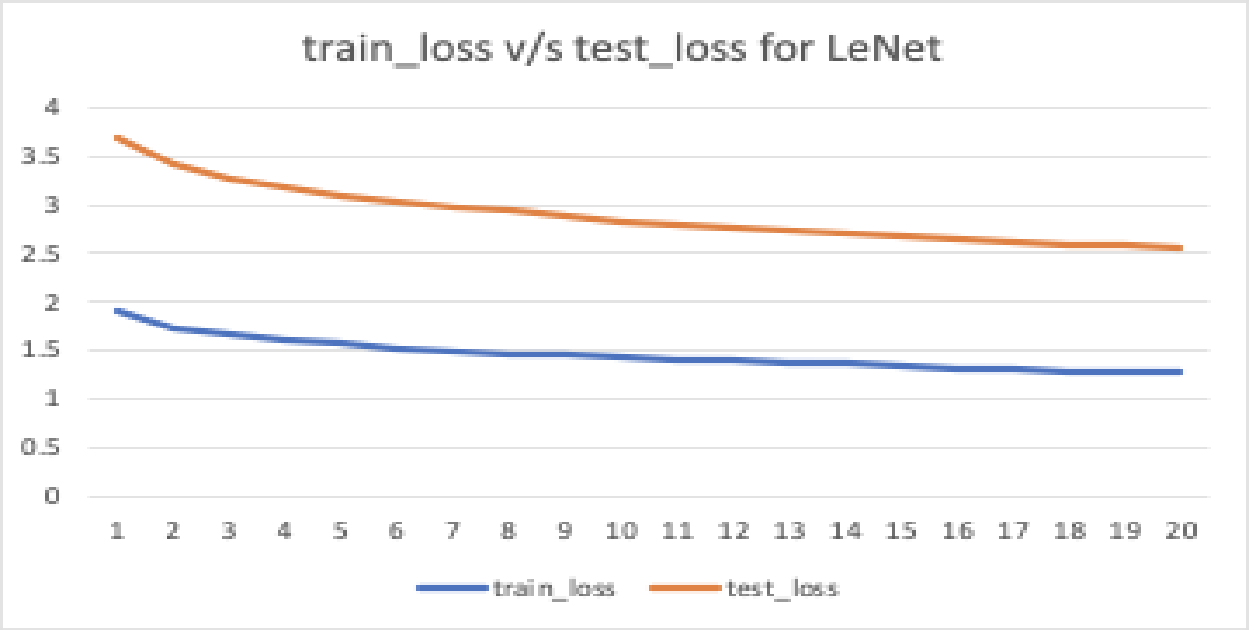


Figure 35(b) : training loss v/s test loss for LeNet

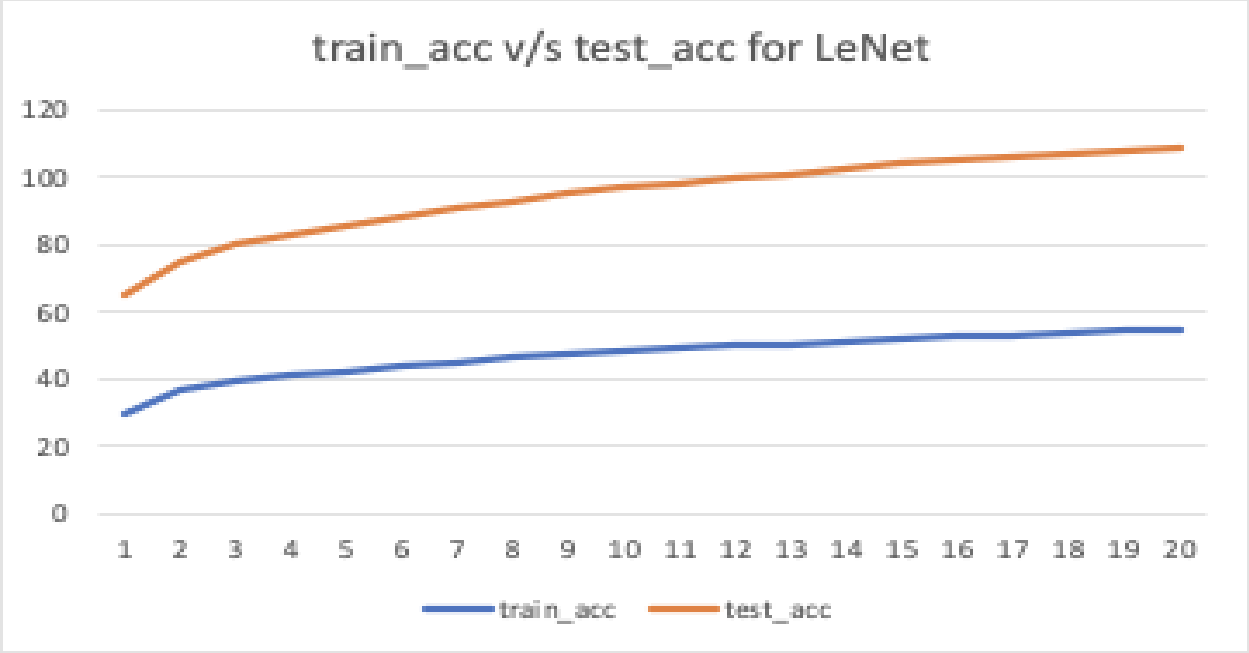


Figure 35(c) :training accuracy v/s test accuracy for LeNet

5.2 Results for Quantized ResNets

The CIFAR-10 dataset was trained using Quantized ResNet model with the following hyperparameters: Number of Epochs - 20, Learning rate - 0.1 with Stochastic Gradient Descent (SGD) optimizer, Loss - Cross Entropy function, batch_size =128, momentum = 0.9, weight_decay = 1e-4, Device - GPU. The model summary, training and test results for the Quantized ResNet model using the CIFAR-10 dataset have been illustrated in figures 36 (a) (without weight threshold) & 36 (b) (with weight threshold = 0.1) below. The plot for loss for each epoch is also shown in figures 37 (a), (b) & (c).

```
Epoch: [0][0/391]       Time 0.304 (0.304)      Data 0.144 (0.144)      Loss 2.4892 (2.4892)    Prec 12.500% (12.500%)
Epoch: [0][100/391]     Time 0.123 (0.126)      Data 0.002 (0.004)      Loss 2.5274 (2.4950)    Prec 9.375% (9.731%)
Epoch: [0][200/391]     Time 0.124 (0.125)      Data 0.002 (0.003)      Loss 2.5132 (2.4943)    Prec 7.031% (9.569%)
Epoch: [0][300/391]     Time 0.122 (0.125)      Data 0.002 (0.003)      Loss 2.4921 (2.4957)    Prec 7.812% (9.627%)
Testing on model of full precision
Test: [0/100]   Time 0.327 (0.327)      Loss 2.3745 (2.3745)    Prec 12.000% (12.000%)
 * Prec 9.710%
Epoch: [1][0/391]       Time 0.317 (0.317)      Data 0.152 (0.152)      Loss 2.5036 (2.5036)    Prec 10.156% (10.156%)
Epoch: [1][100/391]     Time 0.120 (0.123)      Data 0.002 (0.004)      Loss 2.4975 (2.5040)    Prec 7.812% (9.352%)
Epoch: [0][0/391]       Time 0.318 (0.318)      Data 0.143 (0.143)      Loss 2.4825 (2.4825)    Prec 11.719% (11.719%)
Epoch: [0][100/391]     Time 0.123 (0.126)      Data 0.002 (0.003)      Loss 2.4096 (2.4977)    Prec 14.844% (9.653%)
Epoch: [0][200/391]     Time 0.126 (0.125)      Data 0.002 (0.003)      Loss 2.5185 (2.4947)    Prec 11.719% (9.768%)
Epoch: [0][300/391]     Time 0.126 (0.125)      Data 0.002 (0.003)      Loss 2.4753 (2.4957)    Prec 7.812% (9.619%)
Testing on model of full precision
Test: [0/100]   Time 0.248 (0.248)      Loss 2.4107 (2.4107)    Prec 14.000% (14.000%)
 * Prec 9.310%
Epoch: [1][0/391]       Time 0.330 (0.330)      Data 0.172 (0.172)      Loss 2.4744 (2.4744)    Prec 10.156% (10.156%)
Epoch: [1][100/391]     Time 0.123 (0.126)      Data 0.002 (0.004)      Loss 2.5467 (2.4991)    Prec 8.594% (9.568%)
Epoch: [1][200/391]     Time 0.123 (0.125)      Data 0.002 (0.003)      Loss 2.4810 (2.4975)    Prec 10.156% (9.558%)
Epoch: [1][300/391]     Time 0.127 (0.125)      Data 0.002 (0.003)      Loss 2.4344 (2.5000)    Prec 7.031% (9.391%)
Testing on model of full precision
Test: [0/100]   Time 0.234 (0.234)      Loss 2.4134 (2.4134)    Prec 13.000% (13.000%)
 * Prec 9.530%
Epoch: [2][0/391]       Time 0.337 (0.337)      Data 0.158 (0.158)      Loss 2.5400 (2.5400)    Prec 8.594% (8.594%)
Epoch: [2][100/391]     Time 0.133 (0.126)      Data 0.003 (0.004)      Loss 2.5889 (2.4982)    Prec 5.469% (9.421%)
Epoch: [2][200/391]     Time 0.133 (0.125)      Data 0.002 (0.003)      Loss 2.5253 (2.4967)    Prec 8.594% (9.429%)
Epoch: [2][300/391]     Time 0.123 (0.124)      Data 0.002 (0.003)      Loss 2.6575 (2.4989)    Prec 3.906% (9.375%)
Testing on model of full precision
Test: [0/100]   Time 0.239 (0.239)      Loss 2.4025 (2.4025)    Prec 13.000% (13.000%)
 * Prec 9.540%
Epoch: [3][0/391]       Time 0.318 (0.318)      Data 0.156 (0.156)      Loss 2.4863 (2.4863)    Prec 10.938% (10.938%)
Epoch: [3][100/391]     Time 0.122 (0.125)      Data 0.002 (0.003)      Loss 2.5195 (2.5017)    Prec 10.938% (9.344%)
Epoch: [3][200/391]     Time 0.123 (0.125)      Data 0.002 (0.003)      Loss 2.4485 (2.4972)    Prec 11.719% (9.628%)
Epoch: [3][300/391]     Time 0.125 (0.124)      Data 0.002 (0.002)      Loss 2.5071 (2.4983)    Prec 10.938% (9.699%)
Testing on model of full precision
Test: [0/100]   Time 0.248 (0.248)      Loss 2.4279 (2.4279)    Prec 13.000% (13.000%)
```

Figure 36: (a) Quantized ResNet without a defined weight threshold

```
Epoch: [0][0/391]       Time 0.317 (0.317)      Data 0.141 (0.141)      Loss 2.5411 (2.5411)    Prec 10.156% (10.156%)
Epoch: [0][100/391]     Time 0.130 (0.151)      Data 0.002 (0.004)      Loss 2.4714 (2.5263)    Prec 9.375% (9.530%)
Epoch: [0][200/391]     Time 0.127 (0.138)      Data 0.002 (0.003)      Loss 2.4369 (2.5191)    Prec 15.625% (9.942%)
Epoch: [0][300/391]     Time 0.124 (0.134)      Data 0.002 (0.003)      Loss 2.4759 (2.5188)    Prec 12.500% (9.923%)
Testing on model of full precision
Test: [0/100]   Time 0.244 (0.244)      Loss 2.4805 (2.4805)    Prec 13.000% (13.000%)
 * Prec 10.000%
Epoch: [1][0/391]       Time 0.331 (0.331)      Data 0.186 (0.186)      Loss 2.4469 (2.4469)    Prec 13.281% (13.281%)
Epoch: [1][100/391]     Time 0.123 (0.127)      Data 0.002 (0.004)      Loss 2.5484 (2.5209)    Prec 11.719% (9.677%)
Epoch: [1][200/391]     Time 0.123 (0.126)      Data 0.002 (0.003)      Loss 2.5587 (2.5213)    Prec 10.156% (9.834%)
Epoch: [1][300/391]     Time 0.125 (0.125)      Data 0.002 (0.003)      Loss 2.3847 (2.5188)    Prec 16.406% (9.967%)
Testing on model of full precision
Test: [0/100]   Time 0.242 (0.242)      Loss 2.5028 (2.5028)    Prec 13.000% (13.000%)
 * Prec 10.000%
Epoch: [2][0/391]       Time 0.323 (0.323)      Data 0.169 (0.169)      Loss 2.4997 (2.4997)    Prec 11.719% (11.719%)
Epoch: [2][100/391]     Time 0.125 (0.127)      Data 0.002 (0.004)      Loss 2.6100 (2.5149)    Prec 5.469% (10.087%)
Epoch: [2][200/391]     Time 0.130 (0.126)      Data 0.002 (0.003)      Loss 2.4767 (2.5182)    Prec 7.812% (10.009%)
Epoch: [2][300/391]     Time 0.124 (0.126)      Data 0.002 (0.003)      Loss 2.4994 (2.5173)    Prec 9.375% (10.006%)
Testing on model of full precision
Test: [0/100]   Time 0.250 (0.250)      Loss 2.5004 (2.5004)    Prec 13.000% (13.000%)
 * Prec 10.000%
Epoch: [3][0/391]       Time 0.328 (0.328)      Data 0.179 (0.179)      Loss 2.5133 (2.5133)    Prec 10.156% (10.156%)
Epoch: [3][100/391]     Time 0.123 (0.128)      Data 0.002 (0.004)      Loss 2.5517 (2.5253)    Prec 7.812% (9.785%)
Epoch: [3][200/391]     Time 0.125 (0.126)      Data 0.002 (0.003)      Loss 2.3914 (2.5205)    Prec 12.500% (9.884%)
Epoch: [3][300/391]     Time 0.123 (0.126)      Data 0.002 (0.003)      Loss 2.4949 (2.5188)    Prec 12.500% (9.917%)
Testing on model of full precision
Test: [0/100]   Time 0.247 (0.247)      Loss 2.4823 (2.4823)    Prec 13.000% (13.000%)
```

Figure 36: (b) Quantized ResNet with a defined weight threshold of 0.1.

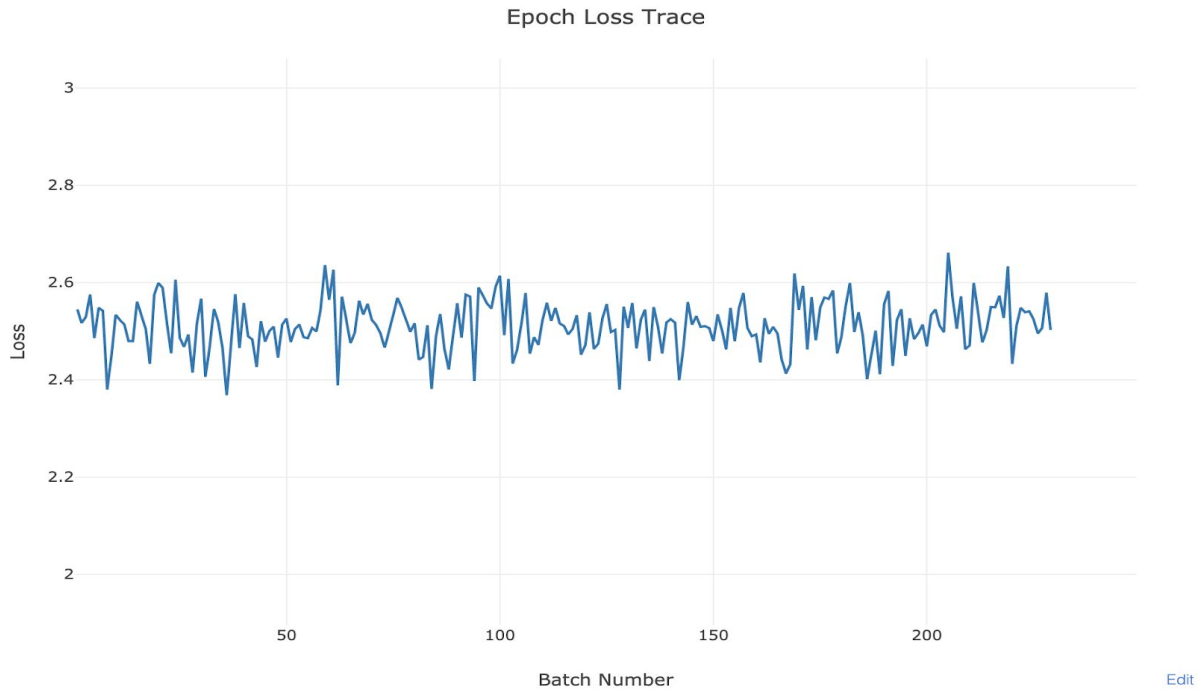


Figure 37(a): Training Loss for each batch of 100 in epoch 1

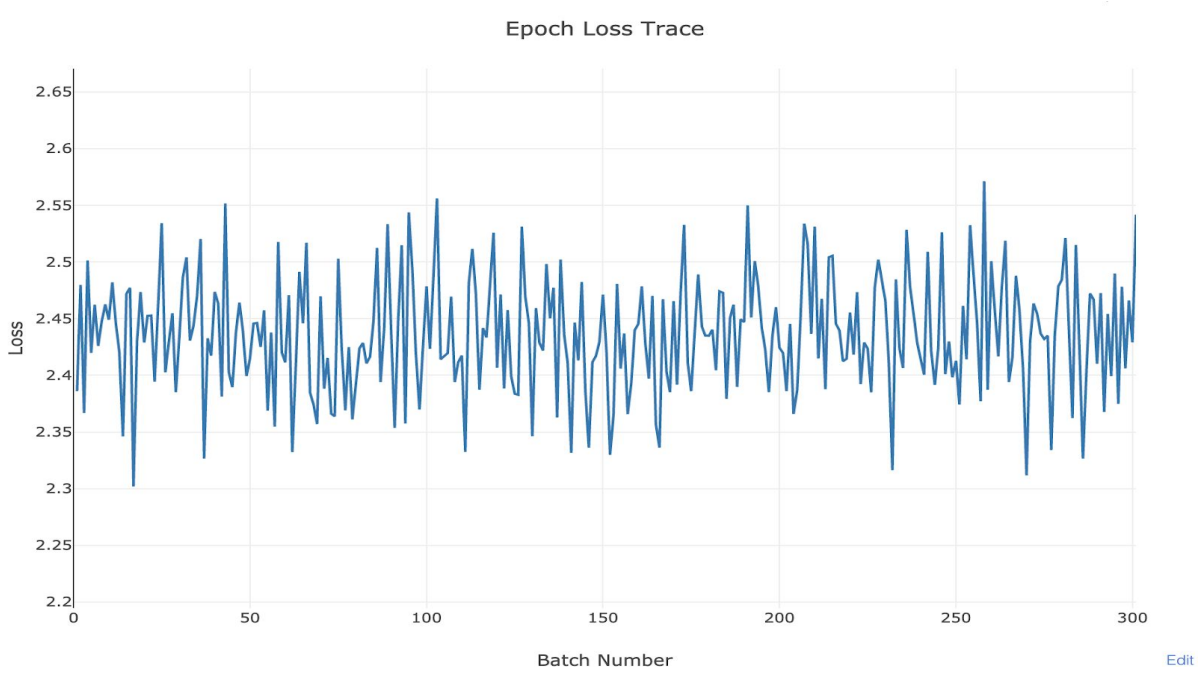


Figure 37(b): Training Loss for each batch of 100 in epoch 2

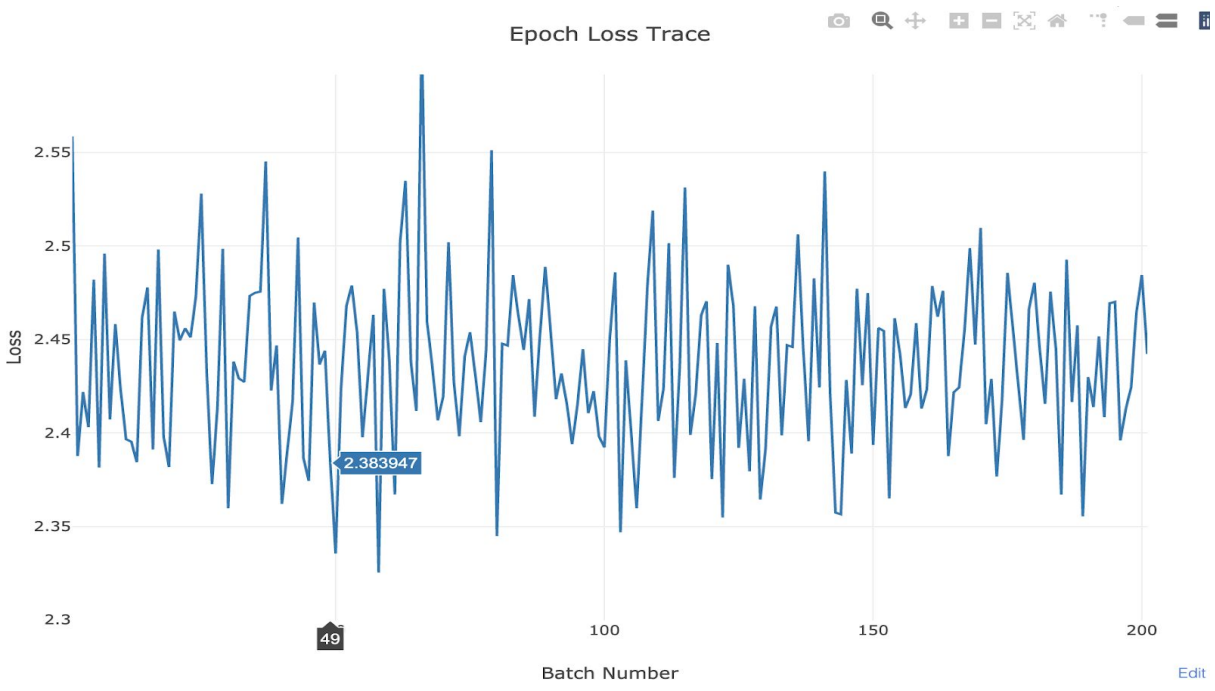


Figure 37(c): Training Loss for each batch of 100 in epoch 3

5.3 Results for NetAdapt Applied on AlexNet

The CIFAR-10 dataset was trained using NetAdapt applied on AlexNet model with the following hyperparameters: Number of Epochs - 20, Learning rate - 0.001 with MultiStepLR optimizer, Loss - Cross Entropy function, batch_size =128, momentum = 0.9, weight_decay = 1e-4, Device - GPU.The model summary, training and test results along with plots for the NetAdapt+AlexNet model using the CIFAR-10 dataset have been illustrated in figures 38 & 39 below.

```
Iteration,Accuracy,Resource,Block,Source Model
0,98.982,710133440.0,None,model.pth.tar,64 192 384 256 256 4096 4096 10
1,98.942,673355240.0,0,model_1_Gp_4/prune-by-mac/worker/iter_1_block_0_model.pth.tar,56 192 384 256 256 4096 4096 10
2,98.968,656176616.0,4,model_1_Gp_4/prune-by-mac/worker/iter_2_block_4_model.pth.tar,56 192 384 256 224 4096 4096 10
3,98.904,630501992.0,1,model_1_Gp_4/prune-by-mac/worker/iter_3_block_1_model.pth.tar,56 176 384 256 224 4096 4096 10
4,98.944,614732264.0,2,model_1_Gp_4/prune-by-mac/worker/iter_4_block_2_model.pth.tar,56 176 360 256 224 4096 4096 10
5,98.892,599556584.0,5,model_1_Gp_4/prune-by-mac/worker/iter_5_block_5_model.pth.tar,56 176 360 256 224 2848 4096 10
6,98.962,578238248.0,3,model_1_Gp_4/prune-by-mac/worker/iter_6_block_3_model.pth.tar,56 176 360 232 224 2848 4096 10
7,98.878,563344616.0,2,model_1_Gp_4/prune-by-mac/worker/iter_7_block_2_model.pth.tar,56 176 336 232 224 2848 4096 10
8,98.696,538838120.0,1,model_1_Gp_4/prune-by-mac/worker/iter_8_block_1_model.pth.tar,56 160 336 232 224 2848 4096 10
9,98.68599999999999,525209960.0,3,model_1_Gp_4/prune-by-mac/worker/iter_9_block_3_model.pth.tar,56 160 336 216 224 2848 4096 10
10,98.514,493097360.0,0,model_1_Gp_4/prune-by-mac/worker/iter_10_block_0_model.pth.tar,48 160 336 216 224 2848 4096 10
11,98.546,479469200.0,3,model_1_Gp_4/prune-by-mac/worker/iter_11_block_3_model.pth.tar,48 160 336 200 224 2848 4096 10
12,98.512,465841040.0,3,model_1_Gp_4/prune-by-mac/worker/iter_12_block_3_model.pth.tar,48 160 336 184 224 2848 4096 10
13,98.494,454945680.0,5,model_1_Gp_4/prune-by-mac/worker/iter_13_block_5_model.pth.tar,48 160 336 184 224 1952 4096 10
```

Figure 38(a) : History obtained for 13 models and along with its accuracy & resource consumption

Figure 38(b): Train loss v/s accuracy for each epoch

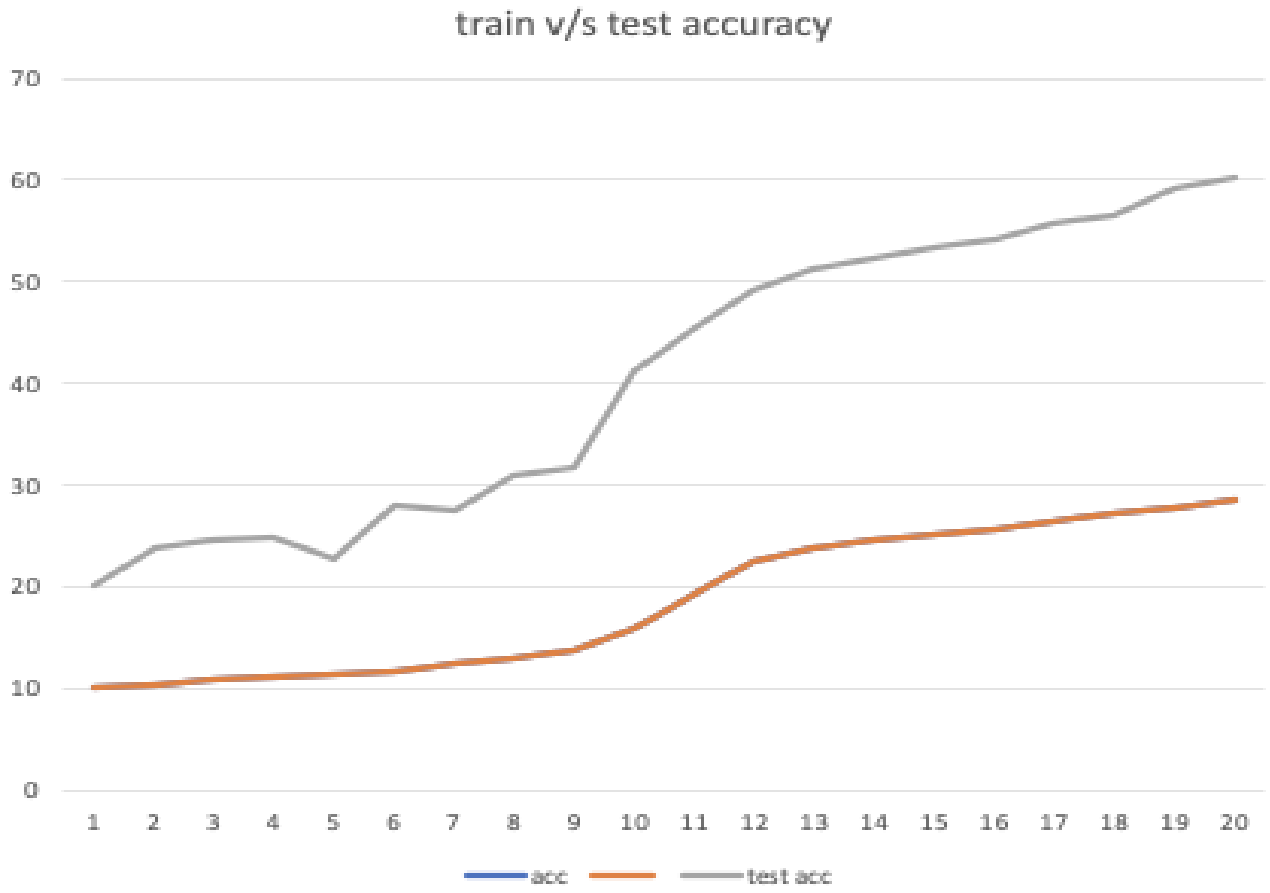


Figure 38(c): Train v/s Test accuracy for each epoch

```
Percent: [##########] 98.73% (78/79) ESA:     0.00s, acc: 37.50
Test accuracy: 31.74% (time =    15.52s)
================================================================
Best accuracy: 31.74
```

Figure 39(a): Evaluation Result for NetAdapt+AlexNet model

```
No checkpoint found at 'model_1_Gp_4prune-by-macmaster/iter_13_best_model.pth.tar'
Epoch [1/20]
==========================================================================
Percent: [##########] 99.74% (390/391) ESA:     0.00s, loss: 0.33, acc: 17.50%
Finish epoch 1: time =    85.30s, loss = 0.49, acc = 10.07%
==========================================================================
Percent: [##########] 98.73% (78/79) ESA:     0.00s, acc: 6.25
Test accuracy: 10.09% (time =    16.03s)
==========================================================================
/usr/local/lib/python3.6/dist-packages/torch/serialization.py:402: UserWarning: Cou
  "type " + obj.__name__ + ". It won't be checked "

Epoch [2/20]
==========================================================================
Percent: [##########] 99.74% (390/391) ESA:     0.00s, loss: 0.33, acc: 13.75%
Finish epoch 2: time =    86.05s, loss = 0.33, acc = 10.44%
==========================================================================
Percent: [##########] 98.73% (78/79) ESA:     0.00s, acc: 18.75
Test accuracy: 13.56% (time =    15.95s)
==========================================================================

Epoch [3/20]
==========================================================================
Percent: [##########] 99.74% (390/391) ESA:     0.00s, loss: 0.33, acc: 12.50%
Finish epoch 3: time =    85.79s, loss = 0.33, acc = 10.98%
==========================================================================
Percent: [##########] 98.73% (78/79) ESA:     0.00s, acc: 12.50
Test accuracy: 13.62% (time =    16.01s)
==========================================================================

Epoch [4/20]
==========================================================================
Percent: [##########] 99.74% (390/391) ESA:     0.00s, loss: 0.33, acc: 11.25%
Finish epoch 4: time =    82.53s, loss = 0.33, acc = 11.27%
==========================================================================
Percent: [##########] 98.73% (78/79) ESA:     0.00s, acc: 6.25
Test accuracy: 11.66% (time =    15.15s)
==========================================================================

Epoch [5/20]
==========================================================================
Percent: [##########] 99.74% (390/391) ESA:     0.00s, loss: 0.33, acc: 11.25%
Finish epoch 5: time =    81.26s, loss = 0.33, acc = 11.55%
==========================================================================
Percent: [##########] 98.73% (78/79) ESA:     0.00s, acc: 6.25
Test accuracy: 11.53% (time =    15.27s)
==========================================================================

Epoch [6/20]
==========================================================================
Percent: [##########] 99.74% (390/391) ESA:     0.00s, loss: 0.33, acc: 17.50%
Finish epoch 6: time =    80.86s, loss = 0.33, acc = 11.77%
==========================================================================
```

Figure 39(b): Training result for first 6 epochs of NetAdapt+AlexNet model

CHAPTER 6

CONCLUSION AND FUTURE SCOPE

The overall objective of the project was to design a quantized model with less complexity in terms of the number of layers. Initially, for this project two datasets were considered namely, CIFAR-10 and PKU-Autonomous Driving dataset. These datasets were visualized and normalized to a defined uniform image size by the process of image pre-processing. As a second step, learnt the behavior of ResNet, AlexNet and LeNet for different datasets by training and testing the three models with CIFAR-10 and PKU-Autonomous Driving dataset. Then, after a careful analysis of the results obtained from the three models, ResNet and AlexNet was used as the base model for Quantization and Netadapt respectively. Furthermore, the ResNet and AlexNet model was expected to perform with more accuracy whereas at the same time with less energy/resource consumption and computational cost,i.e., data movement across the memory. The Quantization of ResNet was implemented by Binary Neural Networks(BNNs) for reducing the weights from full-precision to 1-bit representation and the feature maps were reduced from 32-bit to 8-bit representation values. The overall loss was reduced significantly when compared to the ResNet base model. The plots for Quantized ResNet were plotted live using Visdom Library, a contribution by Facebook, to plot, visualize and manage large datasets lively while they are being trained and tested. As a next step, the AlexNet base model's layer latency was calculated and a corresponding Look-up table(LUT) was generated. NetAdaptat was applied to this already trained AlexNet model and its results were analysed. Thus the accuracy of the AlexNet+NetAdapt model (after 13 iterations of Network Adaptation and Pruning) was increased by 28% from that of the AlexNet model with the number of Floating Point Operations per second(FLOPS) reduced from

710133440 to 454945680. These resources were reduced gradually by each number of iterations which are evident from the model results discussed in chapter 5. By the NetAdapt implementation the output feature map size was also reduced from [64 192 384 256 256 4096 4096 10] to [48 160 336 184 224 1952 4096 10].

```
Percent: [##########] 98.73% (78/79) ESA:     0.00s, acc: 37.50
Test accuracy: 31.74% (time =    15.52s)
```

Figure 40: Evaluation Result for NetAdapt+AlexNet model

The figure 40 above illustrates the test result of NetAdapt+AlexNet model, and it is evident that out of 79 test images from CIFAR-10 dataset 78 were correctly predicted with a prediction accuracy of 98.73% in 15.52 seconds. The future scope of this project is that it can be applied to many real time applications such as Autonomous driving systems where a simple model and memory management becomes very crucial.

REFERENCES:

[1] Volume 52 Issue 2, May 2019, Department of Computer and Information Science and Engineering, 40, 2019-05-31 (yyyy-mm-dd), China Scholarship Council Engineering and Physical Sciences Research Council Horizon 2020, Intel Corporation, Lee Foundation, Royal Academy of Engineering ACM New York, NY, USA, ISSN: 0360-0300 EISSN: 1557-7341

[2] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. CoRR, abs/1504.04788, 2015.

[3] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie E. Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineectual-neuron-free Deep Neural Network Computing. In ACM SIGARCH Computer Architecture News.

[4] Vin De Silva and Lek-Heng Lim. 2006. Tensor Rank and the Ill-posedness of the Best Low-rank Approximation Problem. SIAM Journal on Matrix Analysis and Applications 30, 3 (2006).

[5] V. Sze, Y. Chen, T. Yang and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," in Proceedings of the IEEE , vol. 105, no. 12, pp. 2295-2329, Dec. 2017. doi: 10.1109/JPROC.2017.2761740

URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8114708&isnumber=81147 2

[6] W. Choi, K. Choi and J. Park, "Low Cost Convolutional Neural Network Accelerator Based on Bi-Directional Filtering and Bit-Width Reduction," in IEEE Access , vol. 6, pp. 14734-14746, 2018.doi: 10.1109/ACCESS.2018.2816019

URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8316848&isnumber=8274 85

[7] Z. Du et al. "ShiDianNao: Shifting vision processing closer to the sensor" Proc. 42nd Annu. Int. Symp. Comput. Archit. pp. 92-104 2015.

[8] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. CoRR, abs/1510.00149, 2, 2015.

[9] Y.-H. Chen T. Krishna J. Emer V. Sze "Eyeris: An energy-efficient reconfigurable accelerator for deep convolutional neural networks" IEEE J. Solid-State Circuits vol. 51 pp. 127-138 Jan. 2017.

[10] J. Sim J.-S. Park M. Kim D. Bae Y. Choi L.-S. Kim "A 1.42TOPS/W deep convolutional neural network recognition processor for intelligent IoE systems" IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers pp. 264-265 Jan./Feb. 2016.

[11] S. Kim, P. Howe, T. Moreau, A. Alaghi, L. Ceze and V. S. Sathe, "Energy-Efficient Neural Network Acceleration in the Presence of Bit-Level Memory Errors," in IEEE Transactions on Circuits and Systems I: Regular Papers , vol. 65, no. 12, pp. 4285-4298, Dec. 2018.doi: 10.1109/TCSI.2018.2839613,

URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8374828&isnumber=85110 2

[12] Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. Yodann: An ultra-low power convolutional neural network accelerator based on binary weights. In VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on, pages 236–241. IEEE, 2016. Dzmitry

Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Proceedings of the International Conference on Learning Representations (ICLR), 2015, http://www.jmlr.org/papers/volume18/16-456/16-456.pdf

[13] Yaman Umuroglu , Magnus Jahre, Towards efficient quantized neural network inference on mobile devices: work-in-progress, Proceedings of the 2017 International Conference on Compilers, Architectures and Synthesis for Embedded Systems Companion, p.1-2, October 15-20, 2017, Seoul, Republic of Korea

[14] Proceeding FPGA '17 Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays Pages 65-74 Monterey, California, USA — February 22 - 24, 2017 ACM New York, NY, USA ©2017 table of contents ISBN: 978-1-4503-4354-1 doi>10.1145/3020078.3021744

[15] Y. Gong, L. Liu, M. Yang, and L. Bourdev. Compressing deep convolutional networks using vector quantization. arXiv preprint arXiv:1412.6115, 2014.

[16] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen. Incremental network quantization: Towards lossless cnns with low precision weights. arXiv preprint arXiv:1702.03044, 2017.

[17] Raghuraman Krishnamoorthi raghuramank@google.com, Quantizing deep convolutional networks for efficient inference: A whitepaper, June 2018 URL: https://arxiv.org/pdf/1806.08342.pdf

[18] Kunyuan Du, Ya Zhang, and Haibing Guan, “From Quantized DNNs to Quantizable DNNs”, 11 April 2020.

URL:https://arxiv.org/pdf/2004.05284.pdf

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun propsed "Deep Residual Learning for Image Recognition", Microsoft Research, 10 December 2015.

URL:https://arxiv.org/pdf/1512.03385.pdf

[20] Tien-Ju Yang1, Andrew Howard , Bo Chen , Xiao Zhang , Alec Go , Mark Sandler, Vivienne Sze1 , and Hartwig Adam proposed "NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications", EVCC 2018 paper.

URL:http://openaccess.thecvf.com/content_ECCV_2018/papers/Tien-Ju_Yang_NetAdapt_Platform-Aware_Neural_ECCV_2018_paper.pdf

[21] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis proposed "TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators", 2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA. ACM, New York, NY, USA.

URL:https://dl.acm.org/doi/10.1145/3297858.3304014

[22] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis proposed "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory", Architectural Support for Programming Languages and Operating Systems (ASPLOS '17), China.

URL:https://dl.acm.org/doi/pdf/10.1145/3037697.3037702

[23] "Understanding AlexNet"

URL: **https://www.learnopencv.com/understanding-alexnet/**

[24] "LeNet-5 – A Classic CNN Architecture"

URL: **https://engmrk.com/lenet-5-a-classic-cnn-architecture/**

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," neural information processing systems, vol. 141, no. 5, pp. 1097–1105, 2012.

[26] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to + 1 or-1," arXiv preprint arXiv:1602.0283.

[27] "ResNet-50"

URL:https://www.quora.com/What-is-the-deep-neural-network-known-as-%E2%80%9CResNet-50%E2%80%9D

[28] "ResNet-50 Architecture and Quantization"

URL: Paper on Resnet for our dataset applications: "https://arxiv.org/pdf/1512.03385.pdf"

[29] " Caffe2 Models"

URL: https://github.com/caffe2/models/tree/master/resnet50_quantized

[30] "ResNet-50 Flow Chart"

URL: http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006

Appendix A

USER GUIDE

Download the folder EE297B_Project_Nivedita_Tushar from the drive link.

Google Colab Settings

- Open the colab environment from the Google apps or just by Google search.
- From the Menu bar, **File** option - > select **Upload Notebook** -> pop up menu appears and select the .ipynb file that you are interested in implementing.
- From the Menu bar, **Runtime** option -> select **change runtime type** -> in the drop down menu select **GPU** as **Hardware Accelerator**.

Implementation Procedure

- To understand the Visualization of datasets and behaviour of different CNN architectures such as ResNet, AlexNet and LeNet open the files CIFAR10_Visualization.ipynb, CIFAR10_ResNet.ipynb, CIFAR10_LeNet.ipynb and CIFAR10_AlexNet.ipynb using the Google Colaboratory. Follow the procedure to get the colab settings as mentioned above or from the README file of the Zip file. Once the colab is set from the Menu bar, **Runtime** option -> select **Run all option**. This should automatically start the training and testing of the model once it reaches that cell from the beginning.
- To implement Quatized ResNet, follow the colab settings to open the file - CIFAR10_Quantized_Resnet.ipynb. Here we have used Visdom library to obtain live plots while training and testing of the dataset. The second cell is the one which creates a

server that gets integrated with the Colab in order to plot the results. Run the cells one after the other after executing the second cell a http link will be generated with a format such as https://(server_name).ngrok.io.

- Extract the value in the server_name field and place them in the server_value field of the next cell such as `get_ipython().system_raw('ssh -o ServerAliveInterval=60 -o StrictHostKeyChecking=no -R server_value:80:localhost:6006 serveo.net &')`
- Similarly, replace the value in the next cell as well. `cfg = {"server": "server_name.ngrok.io",` Next run all the rest of the cells and open the generated link to observe the training loss plotted for each epoch.

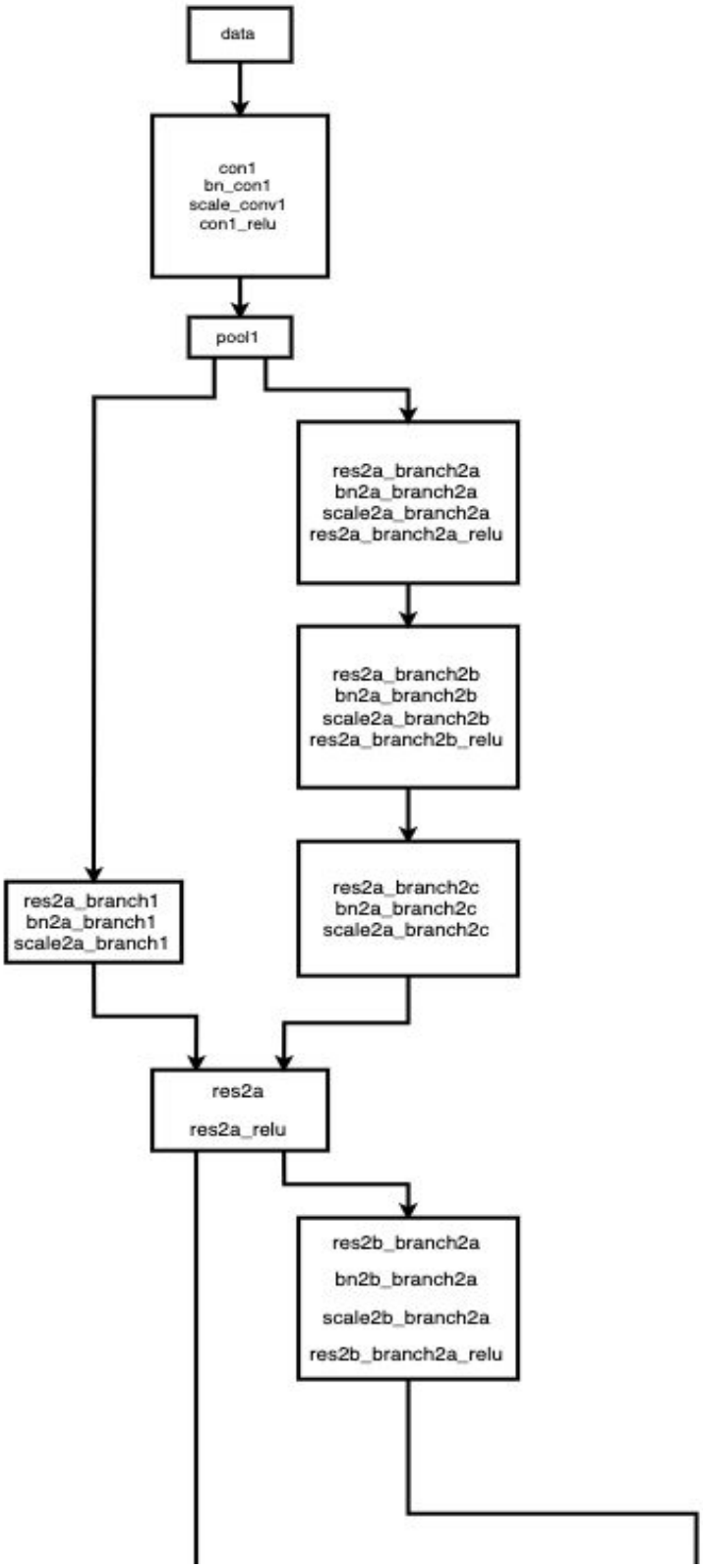
- To implement NetAdapt on AlexNet follow the colab settings to open the file - CIFAR10_AlexNet+NetAdapt.ipynb. Run all the cells and the corresponding functions can be imported from the zip file provided and make sure to change the directory for the path accordingly.
- Also we have a .pth extension of every model that was trained that can also be used as a pre-trained model for the implementation. This can be found in the folder models of the Zip attachment.

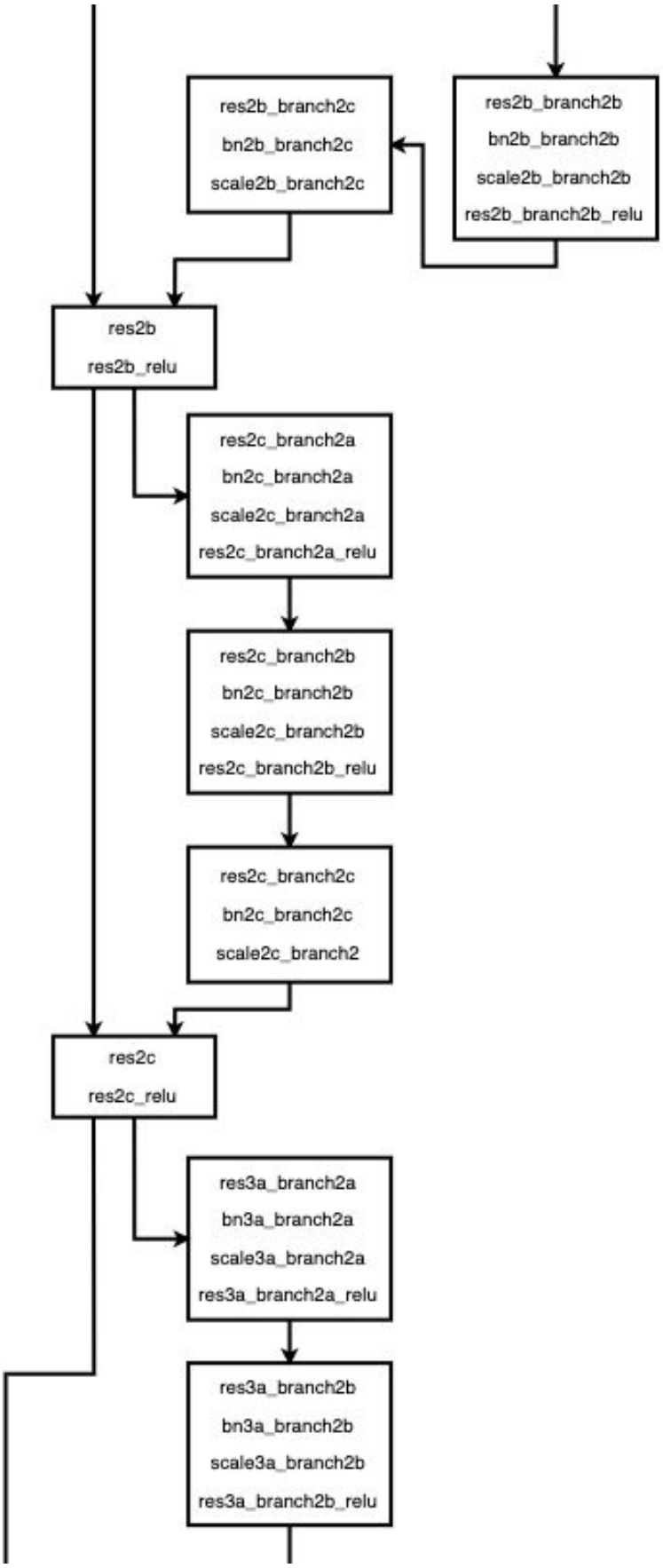
List of codings

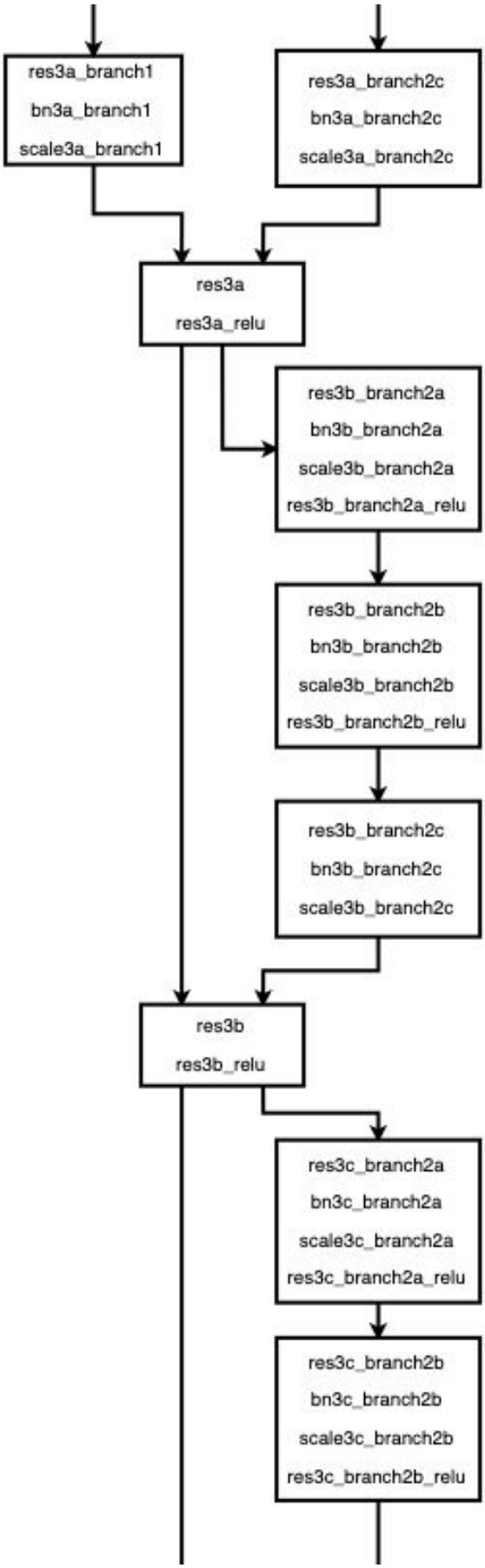
The Google Colab includes the following files for code execution:

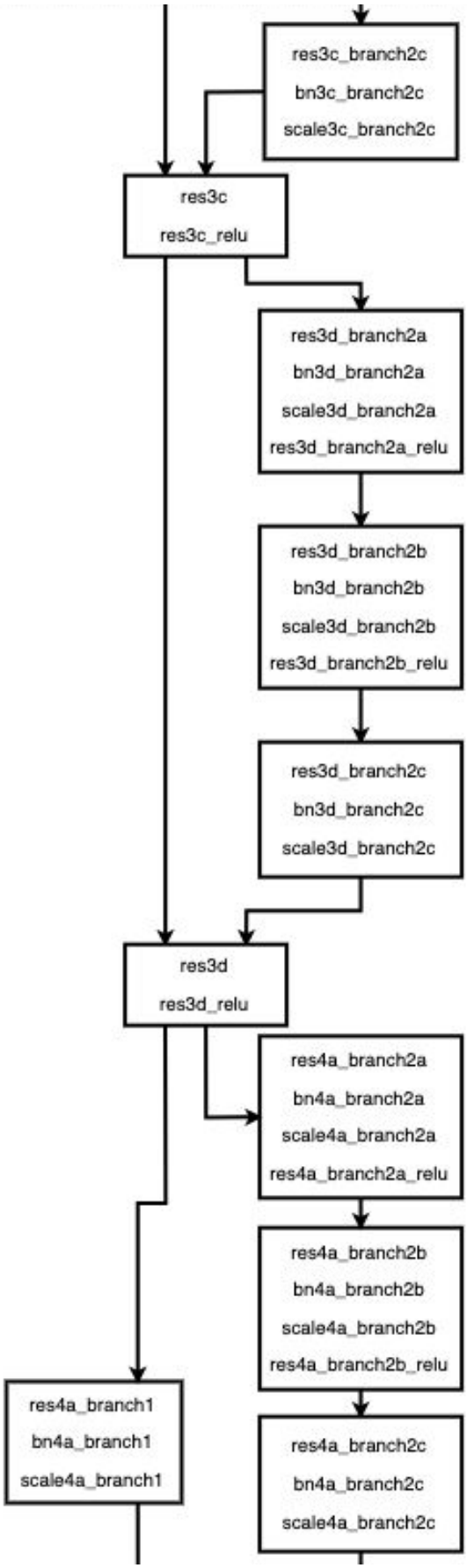
1. CIFAR10_Visualization.ipynb - Visualization of CIFAR-10 dataset
2. CIFAR10_AlexNet.ipynb - To train and test the AlexNet model using CIFAR10 dataset.
3. CIFAR10_ResNet.ipynb - To train and test the ResNet model using CIFAR10 dataset.
4. PKU_Autonomous_driving_ResNet.ipynb - to train and test ResNet models for pku_autonomous driving dataset for resNet50- architecture.
5. CIFAR10_LeNet.ipynb - To train and test the LeNet model using CIFAR10 dataset.
6. CIFAR10_Quantized_Resnet.ipynb - Quantization applied on ResNet with live plots using Visdom library.
7. CIFAR10_AlexNet+NetAdapt.ipynb - NetAdapt applied on AlexNet.

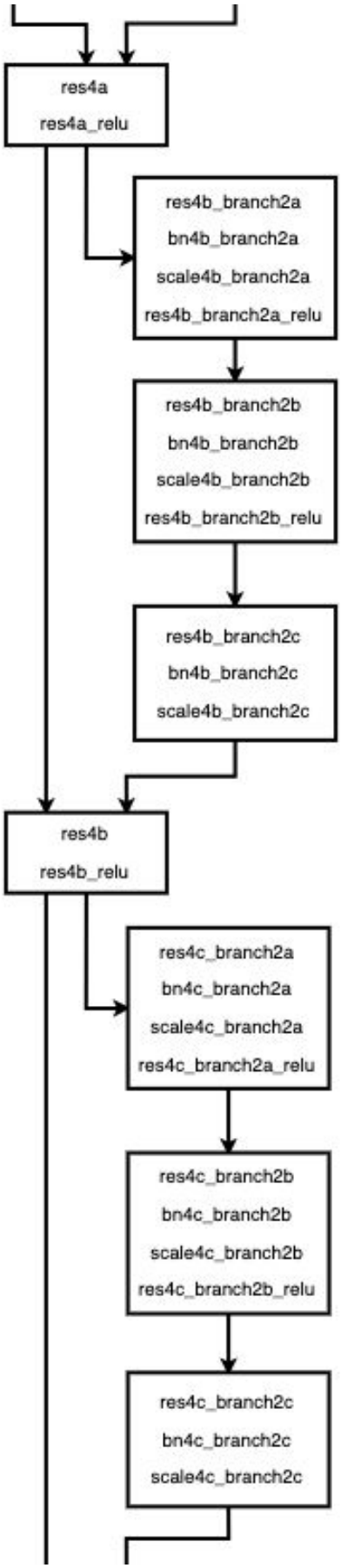
Flowchart of ResNet model

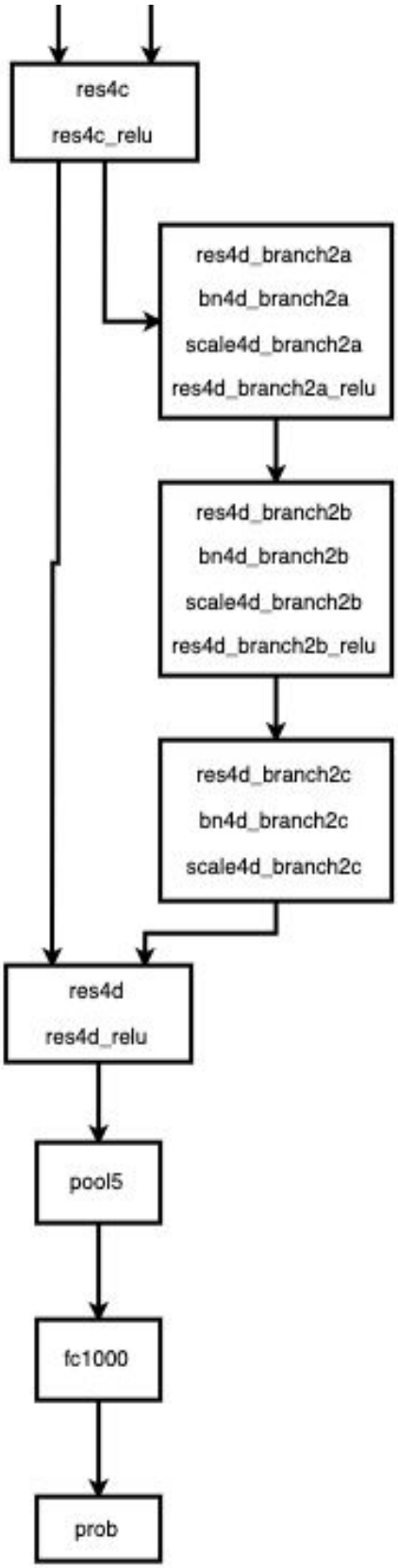



res2b_branch2c
bn2b_branch2c
scale2b_branch2c
res2b_branch2b
bn2b_branch2b
scale2b_branch2b
res2b_branch2b_relu
res2b
res2b_relu
res2c_branch2a
bn2c_branch2a
scale2c_branch2a
res2c_branch2a_relu
res2c_branch2b
bn2c_branch2b
scale2c_branch2b
res2c_branch2b_relu
res2c_branch2c
bn2c_branch2c
scale2c_branch2
res2c
res2c_relu
res3a_branch2a
bn3a_branch2a
scale3a_branch2a
res3a_branch2a_relu
res3a_branch2b
bn3a_branch2b
scale3a_branch2b
res3a_branch2b_relu


res3a_branch1
bn3a_branch1
scale3a_branch1
res3a_branch2c
bn3a_branch2c
scale3a_branch2c
res3a
res3a_relu
res3b_branch2a
bn3b_branch2a
scale3b_branch2a
res3b_branch2a_relu
res3b_branch2b
bn3b_branch2b
scale3b_branch2b
res3b_branch2b_relu
res3b_branch2c
bn3b_branch2c
scale3b_branch2c
res3b
res3b_relu
res3c_branch2a
bn3c_branch2a
scale3c_branch2a
res3c_branch2a_relu
res3c_branch2b
bn3c_branch2b
scale3c_branch2b
res3c_branch2b_relu


res3c_branch2c
bn3c_branch2c
scale3c_branch2c
res3c
res3c_relu
res3d_branch2a
bn3d_branch2a
scale3d_branch2a
res3d_branch2a_relu
res3d_branch2b
bn3d_branch2b
scale3d_branch2b
res3d_branch2b_relu
res3d_branch2c
bn3d_branch2c
scale3d_branch2c
res3d
res3d_relu
res4a_branch2a
bn4a_branch2a
scale4a_branch2a
res4a_branch2a_relu
res4a_branch2b
bn4a_branch2b
scale4a_branch2b
res4a_branch2b_relu
res4a_branch1
bn4a_branch1
scale4a_branch1
res4a_branch2c
bn4a_branch2c
scale4a_branch2c


res4a
res4a_relu
res4b_branch2a
bn4b_branch2a
scale4b_branch2a
res4b_branch2a_relu
res4b_branch2b
bn4b_branch2b
scale4b_branch2b
res4b_branch2b_relu
res4b_branch2c
bn4b_branch2c
scale4b_branch2c
res4b
res4b_relu
res4c_branch2a
bn4c_branch2a
scale4c_branch2a
res4c_branch2a_relu
res4c_branch2b
bn4c_branch2b
scale4c_branch2b
res4c_branch2b_relu
res4c_branch2c
bn4c_branch2c
scale4c_branch2c


res4c
res4c_relu
res4d_branch2a
bn4d_branch2a
scale4d_branch2a
res4d_branch2a_relu
res4d_branch2b
bn4d_branch2b
scale4d_branch2b
res4d_branch2b_relu
res4d_branch2c
bn4d_branch2c
scale4d_branch2c
res4d
res4d_relu
pool5
fc1000
prob