

Abstract

Inference engine for CNN (convolutional neural network) requires a significant amount of computation and memory space on its hardware platform. The choice of number representation format used in CNN design can have impact on model accuracy and performance of the embedded device. Hence the main objective of this project is to design CNN models using fixed and floating point numbers and compare the change in accuracy, power usage, and memory utilization using these number representation formats, to show that the floating point representation is more efficient than fixed point. The results are demonstrated on popular CNN architectures like AlexNet, ResNet, VGGNet etc.

In this project, a simple convolutional neural network is designed to run on ARM processors. This CNN uses small datasets with random images as inputs to the model which are pre-trained. To measure the performance metrics the number formats are changed from fixed point to IEEE754 single and half-precision floating point. The simulations are done on Jupyter notebook and Colab. The simulation results show that the neural network performance increases as the format is changed from fixed to floating point. We also show that CNN can be trained with half-precision incurring no degradation or even improvement in the performance. The accuracy improvement from fixed to half-precision is seen to be by 0.65X on average making the inference to embedded devices more compatible. The dataset considered is based on image recognition and the CNN model used is sequential. Future work includes use of various datasets and parallel network layers to furthermore improve the performance.

TABLE OF CONTENTS

1. Introduction.	5
1.1 Problem Definition.	6
1.2 Objective.	7
2. Literature Survey	10
2.1 Research	10
2.2 Convolutional Neural Networks	13
2.3 MNIST	14
3. Architecture and Algorithm	16
4. Design and Implementation of CNN	20
5. Simulation.	28
6. Discussion/Analysis.	31
7. Conclusion	50
8. References	52
9. Appendix	57

List of Figures

Figure 1	Basic CNN model
Figure 2	MNIST digit dataset
Figure 3	LeNet model
Figure 4	Basic layout of AlexNet architecture.
Figure 5	VGGNet model
Figure 6	Identity Block of ResNet
Figure 7	ResNet skip connection
Figure 8	Fixed point number format
Figure 9	floating point number format
Figure 10	Research on run time on single and half precision floating point number
Figure 11	Basic CNN model for MNIST
Figure 12	LeNet CNN model
Figure 13	AlexNet CNN model
Figure 14	VGGNet CNN model
Figure 15	Pre-processing of MNIST dataset
Figure 16	Data Augmentation
Figure 17	Importing Wandb
Figure 18	ARMv8-A processor (ARM64)
Figure 19	Processor specification from Android
Figure 20	Model Summary of CNN model
Figure 21	Training of the model

Figure 22	wandb simulation
Figure 23	weights and Biases website
Figure 24	MNIST data model loss for fixed point
Figure 25	MNIST data model accuracy for fixed point
Figure 26	MNIST data model loss for fixed point(ResNet)
Figure 27	MNIST data model accuracy for fixed point(ResNet)
Figure 28	MNIST data model loss for fixed point LeNet
Figure 29	MNIST data model accuracy for fixed point LeNet
Figure 30	MNIST data model loss for fixed point VGG
Figure 31	MNIST data model accuracy for fixed point VGG
Figure 32	MNIST data model loss for fixed point AlexNet
Figure 33	MNIST data model accuracy for fixed point AlexNet
Figure 34	MNIST data model loss for 16 bit(on the left) and 32 bit(on the right) point
Figure 35	MNIST data model accuracy for 16 bit(on the left) and 32 bit(on the right) point
Figure 36	MNIST data model accuracy for 32 bit(on the left) and 16 bit(on the right) point ResNet
Figure 37	MNIST data model accuracy for 16 bit(on the left) and 32 bit(on the right) point ResNet
Figure 38	MNIST data model loss for 16 bit(on the left) and 32 bit(on the right) point LeNet
Figure 39	MNIST data model accuracy for 16 bit(on the left) and 32 bit(on the right) point LeNet
Figure 40	MNIST data model accuracy for 16 bit(on the left) and 32 bit(on the right) point VGG

Figure 41	MNIST data model loss for 16 bit(on the left) and 32 bit(on the right) point VGG
Figure 42	MNIST data model accuracy for 32 bit(on the left) and 16 bit(on the right) point AlexNet
Figure 43	MNIST data model loss for 32 bit(on the left) and 16 bit(on the right) point AlexNet
Figure 44	MNIST data model memory usage
Figure 45	MNIST data model CPU usage
Figure 46	MNIST data model memory usage AlexNet
Figure 47	MNIST data model process CPU thread use on weights & Biases
Figure 48	MNIST data model process GPU usage on Colab on weights & Biases AlexNet
Figure 49	MNIST data model process system and CPU usage on weights & Biases LeNet
Figure 50	MNIST data model process GPU usage on Colab on weights & Biases LeNet
Figure 51	MNIST data model process system and CPU usage on weights & Biases VGG
Figure 52	MNIST data model process GPU usage on Colab on weights & Biases VGG
Figure 53	MNIST data model process system and CPU usage on weights & Biases ResNet
Figure 54	MNIST ResNet process memory in use ResNet
Figure 55	MNIST data model process GPU usage on Colab on weights & Biases ResNet
Figure 56	MNIST data model process GPU usage on Colab on weights & Biases fixed
Figure 57	MNIST data model process CPU usage on Colab on weights & Biases
Figure 58	MNIST data model process CPU usage on Colab on weights & Biases for all half precision model
Figure 59	MNIST data model process usage on Colab on weights & Biases for all half precision model
Figure 60	MNIST data model GPU usage for all half precision on Weights & Biases
Figure 61	MNIST data model process CPU usage on Colab on weights & Biases for all single precision model
Figure 62	MNIST data model process usage on Colab on weights & Biases for all single precision model

Figure 63	MNIST data model GPU usage for all single precision on Weights & Biases
-----------	---

1.Introduction

Convolutional Neural Networks (CNNs) is a category of the neural network which has shown state of art performances in many fields. The need to use CNN is due to its hidden layers that automatically represent data just from training. The availability of a large amount of improvement and data in the hardware processing unit makes the use of CNN in research and other fields so enormous. To such a deep learning neural network the scope of improvement is never limited. Either with the use of a different activation function or loss function, optimization of the parameter, representation of data or regularization the ways of improving the performance are in abundance.[7][10]

CNNs are so far one of the best techniques for learning image and the content of those images and have also shown results in state of art on segmentation, detection, and other tasks. The need to build a better and efficient architecture for CNN is the most researched work in every known big company be it Google, Apple, Intel, etc.[7][10]

Convolutional neural networks have made significant attention in every field. Applications such as object detection, speech recognition, self-driving car, image recognition are some of the few achievements which are in use due to CNN. The use of CNNs has made its advancement in large scale image classification field and also in pattern recognition tasks. Graphics processing units (GPU) offers both training and inference using deep convolutional neural network cloud on its platform.[7][6][10]

CNN inference design runs by the process of pooling, normalizing, scaling and convolution. These stages have numerous addition and multiplication of the weights and bias. The need for a numeric representation that can display the result at the highest precision is highly needed. The use of hardware is depended on the type of representation of values, which can result in higher efficiency or may give a vague result. In CNN the process of feeding input through a pre-trained network to receive a CNN outcome is called inference.[7][10]

The significance of CNN has been observed in the field of pattern recognition, large scale image classification. Though graphic processing unit (GPU) provides a platform to perform training and inference on CNN using cloud-based servers. However, the energy dissipation in embedded systems makes their use inefficient. Hence the need for an accelerator to deploy pre-trained CNN models.[6]

CNN inference has a limit to its deployment on embedded devices due to the need for high computation and memory requirements. As a solution to this problem, the use of low precision fixed-point number representation was used for activation and weights. As there are variation in weights across the different layers and networks within the same network the use of fixed-point alone doesn't solve the problem.[1]

The number representation format of the input data contributes to the efficiency of the learning algorithm of CNN. A good data representation can increase the performance and bad representation can lower the performance. The current research is focused on building a representation format to draw the raw data features into the algorithm thereby improving the overall accuracy.[10]

The famous Intel Pentium bug in 1994, has led many major companies to investigate the use of fixed-point representation and the drawback of implementing it. Though the fixed point is more compatible with the hardware and are more efficient but comes with an increase in complexity of the system and need for higher memory storage requirement. Intel, AMD, and IBM have researched FP hardware verification after the loss from the buy which costs \$475M.[7][9]

The process which uses the IEEE standard floating-point format has far more resources compared to the fixed point, which simplifies the bit width and the number representation leading to potential resources conservation in the inference stage. Using floating-point format, the multiplication and addition can be carried out more efficiently with the use of one fixed point operand and one floating point resulting in a fixed-point result.[1]

The work focuses on using IEEE754 floating point representation and fixed-point representation on popular CNN architectures like LeNet, AlexNet, VGG, and ResNet and demonstrates the increase in efficiency using floating point. MNIST digit recognition dataset is considered for this experiment. The results will indicate the reduced memory usage and power consumption of the hardware accelerator using floating point representation.

1.1 Problem Definition

The computation and memory usage on inferencing to a convolutional neural network is significantly high, making CNN undesirable for deployment on embedded devices. The low efficiency of CNN on hardware is predominantly due to the use of low precision fixed point representation. The fixed-point representation is an ideal choice for any hardware system due to low power consumption. The usage of fixed point/integer format for arithmetic's on hardware is simpler at register transfer level. As the design of a CNN model gets complex with the use of fixed-point representation the trade-off is between low power consumption and accuracy.[1]

The implementation of the fixed-point representation though easier due to higher compatibility on hardware. Fixed point in a complex algorithm makes bug detection almost impossible. The 1994 Intel Pentium bug involving fixed point division costs \$475M to debug. Many semiconductor industries, including AMD, Intel, IBM, and several institutions carried out research in debugging and finding better fixed-point hardware verifications. Though the bug was resolved, the cost of debugging enlightened to find a better numeric representation format.[7][9]

In CNN the layers are cascaded till the output layer, using fixed point reduces the accuracy of the entire network. The precision required for each network is different. The inference on CPU with varying bit width using fixed point deteriorate the efficiency. The search for finding another format for numeric

representation put floating point into a new light. The floating point has bit width which is segregated as sign, exponent and mantissa. The mantissa controls the precision and range is controlled by exponent. The use of floating point is complex to design and verification can be challenging but bug detection is relatively easier due to split in bit width.[1]

The search for hardware compatible and performance efficient numeric form led to exploring different representations. Since the fixed point/integer is hardware friendly and floating point provides the best precision, custom precision design has been under research. The use of IEEE754 32/16 floating point bit is widely under considering over fixed point. A combination of fixed point and IEEE floating point is being explored. The need for an energy efficient and high performance is evaluated through various custom designed precisions and through implementation of these on various CNN models is under research. This project presents few such approach.

1.2 Objective

The demand for efficient computation of neural network is increasing. In this project, we implement fixed point representation on popular CNN model and investigate the output of precision and memory usage. The IEEE754 floating point representation is implemented on inference for accuracy and efficiency and compare the outcomes. Various CNN architectures like VGG, AlexNet, LeNet etc. are implemented using both the representation and the results are further compared. The research conducted so far are considered and the results are graphically represented.

The use of fixed point on inference is reevaluated and custom design precisions experimented by various researchers are implemented on the CNN model for comparison. Traditional CNN models are trained using IEEE754 single precision bit representation for better precision. However, the need for such high precision is not required by every CNN model on inference. The current study on number representation shows use of low precision floating point representation. The use of fixed-point quantization with floating point computation is considered to be

memory efficient in storing activation and weights, enhancing results on hardware platform. [30]

Based on recent research the use of substantial low precision is used to train CNN and employing suitable rounding methods making the format energy efficient. The studies show the loss of precision due to lower precision IEEE754 floating point format is minimal and the performance can still be attained equivalent to the traditional format. The choices in the project are evaluated layer by layer with careful watch over the performance on GPU/CPU. The mathematical capabilities of a network are improved with the change in number representation format from fixed to low precision floating point, improving the accuracy and speed up in training and inference. [1]

The objective of this project is to experiment with the CNN model on a simpler dataset like MNIST and the results are evaluated with various number representation formats. These models are trained on popular CNN types and the compatibility of a format on each of the types of CNN is observed. The change in accuracy of the training and the power consumption as well as memory usage is observed when used on GPU/CPU and the inference results on an ARM processor are evaluated for better hardware compatibility.

2. Literature Survey

In this chapter, the prior and current research done in the field of CNN is listed out, along with the different approaches in implementing various numeric representations and the way the overall efficiency differs in each.

2.1 Research:

Gysel et al. (Gysel, 2016) proposed representing both CNN activation function and weights using mini floats i.e... shorter bit-width floating point number. Gysel considers floating-point representation to be lesser hardware efficient than fixed-point numbers and focuses on research in fixed-point quantization. Gupta et al. (Gupta et al., 2015) present the effects of various fixed-point adjusting plans on the exactness. Gupta et al. (Gupta et al., 2015) exhibit network training with half precision fixed point weights utilizing a stochastic rounding scheme for higher accuracy.[1]

Judd et al. (Judd et al., 2015) showed that the required data precision varies with different layers and different networks in the same network. Lin et al. (Lin et al., 2015) present a fixed-point quantization to distinguish the data precision for all layers of the network. Gysel et al. (Gysel et al., 2016) present Ristretto for fixed-point quantization and retraining of CNN's dependent on Caffe (Jia et al., 2014).[1]

FPGAs have been cutting their specialty in profound taking in applications from full precision accelerators exploiting Intel's Arria 10 hard floating-point units to its logic design exploiting the core network at low precision.[5]

Contrasting with IEEE 754 32-bit floating-point, quantizing weight and activation function on lower bits (for example quarter precision bits and lower) brings about numerous funds relying upon the hardware accelerators agent being overutilized. Savings include memory footprint for bandwidth, weights and activation functions to write and read from memory and lower powers in hardware and routing required resources.[5]

Pre-trained neural network weights in floating-point numbers and quantization implementation of activations in fixed point reduced the memory need and hardware efficiency. The use of floating-point representation of weights has better range, precision, and accuracy in comparison to fixed-point representation of the number of bits and its implementation in state of art CNN's i.e. AlexNet (Krizhevsky et al., 2012), VGG-16 (Simonyan and Zisserman, 2014), GoogLeNet (Szegedy et al., 2015) and SqueezeNet (Iandola et al., 2016).[1]

The intel white paper shows the use of an 8-bit floating point for multiplication and 16 bits for addition instead of using single-precision floating-point numbers. This way the memory bandwidth and reducing precision move in memory hierarchy easier. Also the lesser use of silicon and power as used in the Intel Xeon Scalable platform.[7]

Xilinx solved the intel bug using a floating-point arithmetic hardware unit (FPU). But the implementation of this is easier on system Verilog platform rather than C/C++ due to the no availability of set and clock option for synchronization and needs additional implementation requirement when used RTL design. The use of a half-precision floating point is given higher preference than single precision.[9]

The work of **Courbariaux et al., 2014** demonstrates training of CNN with fixed point and dynamic fixed point and compared with floating point to prove fixed point weights being a better choice for training. The work shows 8-bit number are sufficient for training on network with 10 sets of data. Though this worked well for a smaller set of data, the challenge was to implement on larger dataset. The use of a term quantized neural network (QNN) to train on ImageNet of 1000 class with low precision format was worked on. This idea was further implemented on AlexNet

model to achieve 73.67% accuracy and was optimized on GoogleNet. The approach

of trained ternary quantization (TTQ) using ternary weights at forward pass to remove multiplication factors was implemented. This method used full precision model for training and reducing weights to 2 bits and fine tuning the network as the results approach.[20][21]

Courbariaux et al further studied the approach of using low precision fixed point for forward propagation to preserve the CNN inference accuracy. Jacob et al. used 8-bit quantization on CNN models like MobileNet to reduce the inference latency on ARM CPU by 50% and drop of 1.88pp accuracy on COCO dataset. With this inventory, many researches were based on low precisions fixed point was implemented on CNN and RNN inference FPGA designs to achieve better throughputs than floating point with accuracy drop as a tradeoff. With constant quantization resolution the suboptimal efficiency on bandwidth was attained on the entire network though different ranges can be seen on different layers.[20][21]

Furthermore, hardware accelerators for CNN were explored for the deployment of CNN on embedded devices and obtain low power consumption performance (Qiu et al., 2016; Shin et al., 2017). The reduction in precision of the data was the standard approach implemented to make CNN energy efficient.[1]

Zhou et al. (Zhou et al., 2016) worked on techniques to improve overall suboptimal accuracies on the results by re-training the AlexNet with 2-bit activation, 6-bit gradient and 1-bit weights. Deng et al., 2015 proposed quantization of weights at runtime which is fetched randomly to reduce the bandwidth of memory by storing (32-bit)single precision weights and quantized half precision floating point weights.[1]

With the floating-point numeric representation being explored on CNN for recent hardware development, the idea of reducing the precision for calculations is under research. The use of single precision floating point on a learning algorithm using 8-bit data is considered a waste of space. The approach of using half-precision format is shown to attain similar accuracy without scarifying the overall

performance. The recent work of NVIDIA is to custom develop logics for their hardware to implement MAC operation inside CNN. This reduced floating point precision is being exploited on their hardware feature called tensor core which has shown to increase GPU capabilities and has speeded up the applications.[31]

Researchers have demonstrated deep learning training with 16-bit multipliers and inference with 8-bit multipliers or less of numerical precision accumulated to 32-bits with minimal to no loss in accuracy across various models.

Vanhoucke, et al. (2011) worked with 32-bit floating point for the input layer of CNN for speech recognition on CPU by using weights of 8-bit and bias at first layer and quantized activation. Hwang, et al. (2014) trained using backpropagation for MNIST dataset using high precision weights and feed forward to backpropagation being a simple network of quantized weights resulting in minimal performance loss. Micikevicius, et al. (2017) used half precision floating point multiplier and accumulator of 32-bit floating point and weights updated for AlexNet and other CNN architecture resulted in minimal loss in accuracy. Baidu researchers (2017) used fixed precision of 8 bit (1 sign bit, 3 bit fractional and 4 bit integer) with series of quantization to show no loss to minimal at reduced precision.[7]

2.2 Convolutional Neural Network (CNN)

A multilayer perceptron (MLP) is a part of artificial neural network with feedforward feature. This MLP cannot process image as the weights become unmanageable for large images. The spatial information is lost when image is flattened by MLP. This disadvantage led to the immense use of convolutional neural network. Convolutional neural network (CNN/ ConvNet) is a deep neural network class whose applications are highly used in the field of image recognition, object detection etc. CNN consists of input, hidden and output layer like MLP. The hidden layers are series of convolutional layers which convolve with dot product or multiplication. The most commonly used activation function is RELU which is followed by series of pooling layer. The CNNs are fully connected and are normalized all in the hidden layer and the results are obtained in output layer. The

filters in CNN allows weight sharing and parameter sharing so that the filters can focus on specific pattern. [23]

CNN takes an image as an input and features/ weights are assigned to the image. This input image is further trained with series of pooling, drop out and activation function using kernel for convolving. This trained image is compared with the test image for recognition and other applications. The pre-processing of CNN is lesser than the other algorithm.[25][26][29]

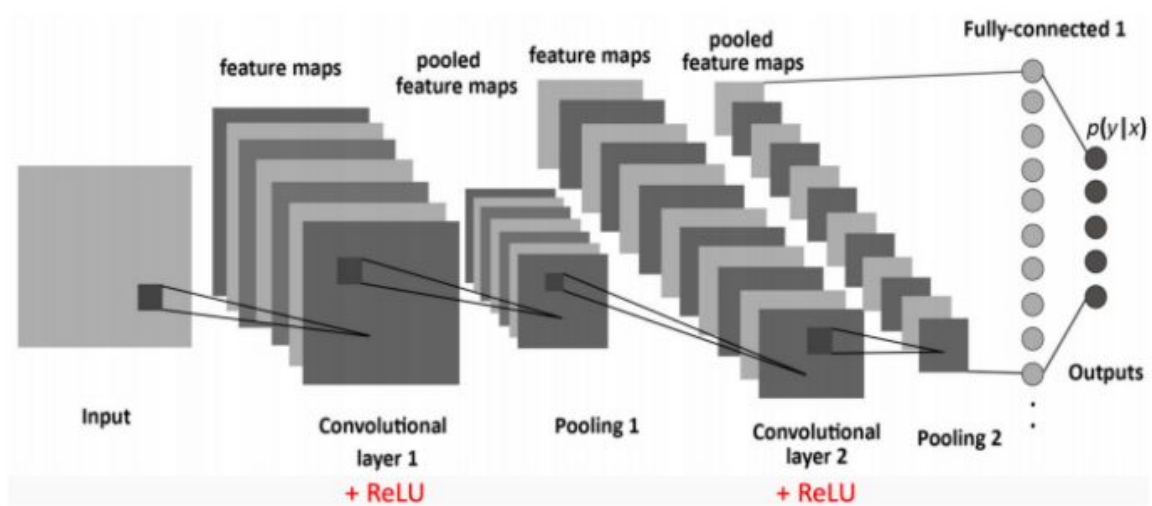


Figure 1: Basic CNN model

2.3 MNIST

The MNIST dataset (Modified National Institute of Standards and Technology) is a large database of handwritten digit recognition used for training the various images. MNIST is the most commonly used dataset used for training and testing machine learning. This dataset is created using NIST's dataset and re-mixing the samples. [27]

The MNIST dataset has 60,000 training images and 10,000 testing images. The training set and test set's half the samples are collected from NIST's training and

testing database. The NIST original black and white images are normalized to 0/1 to fit in 20X20 pixel setting ratio. The images are further greyscale and added with 28x28 image size. The pre-processing of the images are relatively easier with this dataset. [27]

This dataset has four files one for training labels and one for training images and the other two are for testing labels and testing images. The labels for the files are further added while pre-processing the training and test dataset as required.[27]

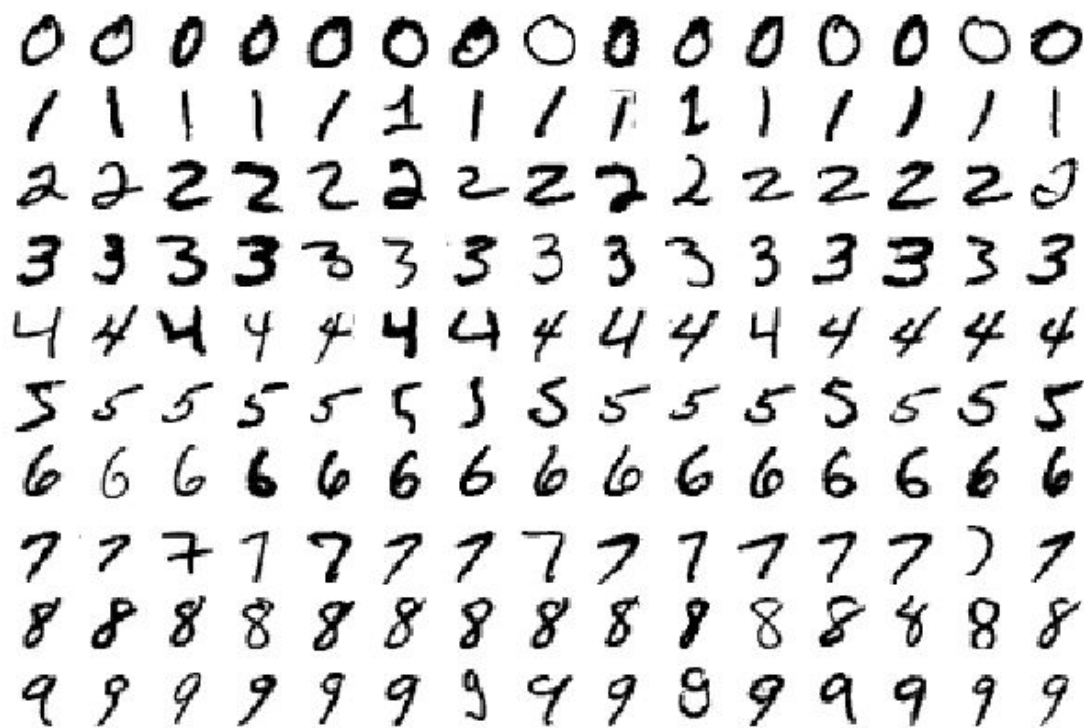


Figure 2: MNIST digit dataset

3. Architecture & Algorithm

CNN architectures are alternating layers of convolution and pooling with fully connected layers at the end. Average pooling layers are sometimes replaced with average pooling. Often batch normalizing and dropouts are added in the hidden layers for optimized results. This placement of the layers are fundamental in developing new CNN architectures. In this chapter few such architectures are explained. [27][28]

CNN provided best techniques for learning image content and have shown results for image recognition, object detection and segmentation. Companies like Google, Microsoft are on constant hunt of finding new architectures of CNN.

3.1 LeNet

LeNet the pioneer convolutional network proposed by LeCuN in 1998. LeNet was the first to show state of art performance on hand digit recognition. It was used by several banks to recognise hand written check numbers. This type of CNN can classify digits even with small rotation and variation of position scale applied. Like CNN LeNet is a feed forward neural network with five alternating layers of convolution and pooling and two fully connected layers in hidden layer. In 2000 when GPU wasn't common for training of network and CPU speed was slower which made use of MLP tiresome, LeNet implementation helped in understanding pixel correlation in an image and the distribution structure of an image.[28][29]

This made parameters in convolutional an effective way to extract features in pool of parameters. The training of images was better understood with each pixel being

considered as input feature and helping to avoid correlation among the neighbouring pixels. LeNet became the first to automatically learn the features and reduce the computation and number of parameters required for training. [28][29]

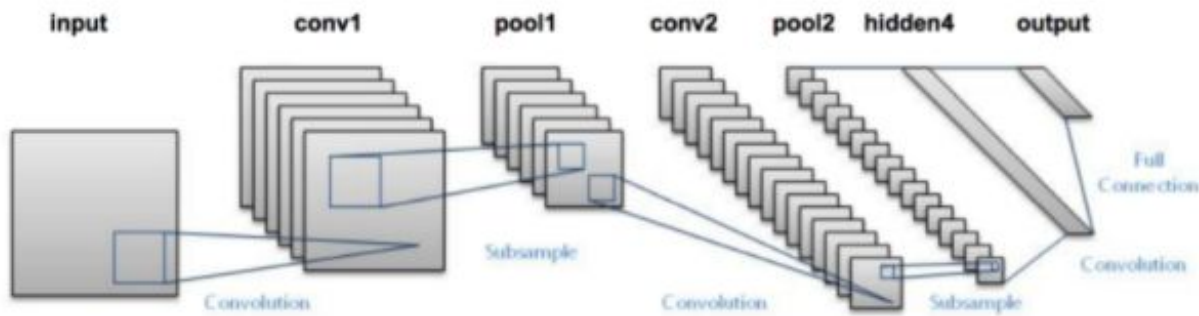


Figure 3: LeNet model

3.2 AlexNet

LeNet being the pioneer for deep CNN but was limited to just hand digit recognition and use of LeNet on all classes of images wasn't as expected. In 2012 AlexNet significantly outperformed the results of LeNet for image classification and image recognition. AlexNet is the first deep convolutional neural network architectures were proposed by Krizhevsky et al., enhancing the learning of CNN by implementing numerous optimizations and by applying parameters. [28][29]

Due to hardware limitations in learning a deep CNN architecture, AlexNet was trained for small size, but to get better results out of AlexNet parallel training was performed on two NVIDIA GTX 580 GPU. The feature extraction stages of AlexNet is more than LeNet making AlexNet's application more diverse. To overcome the drawback of overfitting the network, Krizhevsky et al., implemented the idea of Hinton of random skip of training units which made the model more robust. [28][29]

The AlexNet has 11x11 convolutions, 5x5max pooling, 3x3dropout and data augmentations are added. The use of RELU as activation function after every

convolutional and fully connected layer was to avoid the problem of vanishing gradients. [28][29]

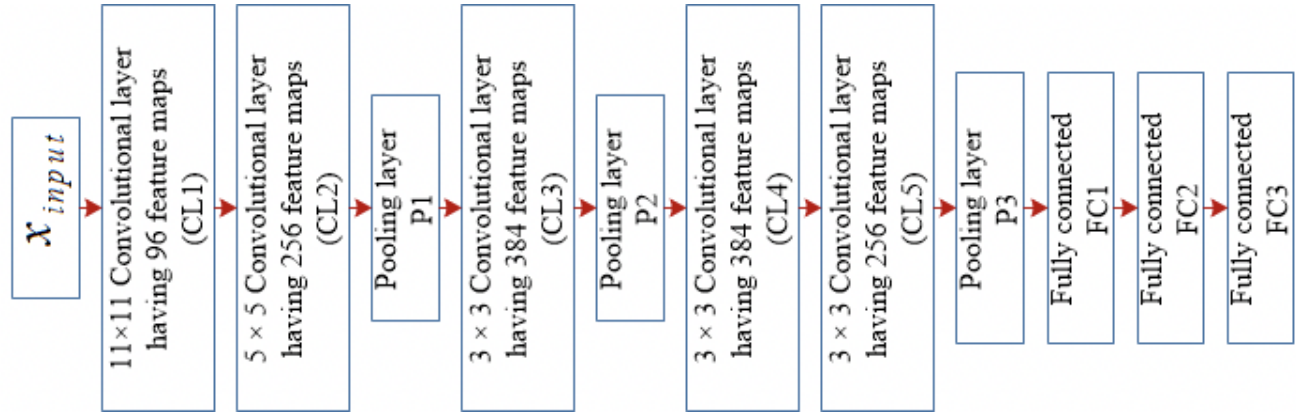


Figure 4: Basic layout of AlexNet architecture.

3.3 VGGNet

The runner up at the ILSVRC 2014 competition was the architecture proposed by Simonyan et al. for image recognition. The VGG proposed by Simonyan et al. was of 19 layers in compared to AlexNet. VGG has 11×11 convolutions and 5×5 filters with an additional stack of 3×3 filters. Since the trend of using small size filters emerged, VGG was implemented to use 1×1 convolution between convolutional layers resulting in a linear combination of resultant feature maps. Max pooling layer is added after convolutional and padding is implemented to maintain spatial resolution. VGG shows good results in localization and image recognition. [28][29]

The limitations of VGG lies in computational cost. The training of this architecture took 2-3 weeks on GPU. The use of VGG is in demand for extraction features of images. However, even for small size filters, VGG consists of around 140 million parameters challenging its computational use. [28][29]

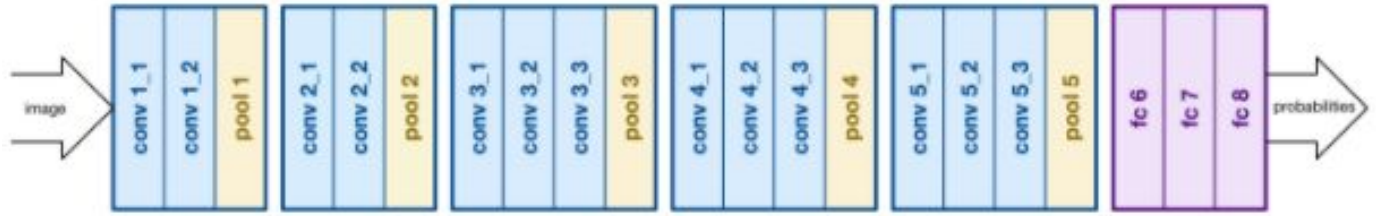


Figure 5: VGGNet model

3.4 ResNet

Residual neural network (ResNet) was proposed by He et al., with the concept of skip connections and batch normalization features. ResNet started a new approach of residual learning in CNN and devised an efficient training technique of deep Nets. This technique trained a network of 152 layers and won 2015-ILSVRC competition. ResNet was 20 times deeper than AlexNet and 8times deeper than VGG and also shows lower complexity among the other architectures of CNN. [28][29]

He et al., showed ResNet with 50/101/152 layers to have less error for image classification in empirical format. ResNet is highly recognized in its use in COCO dataset. ResNet achieves top-5 error rate of 3.57% beating human level performance. ResNet uses convolution and identity blocks in the network. [28][29]

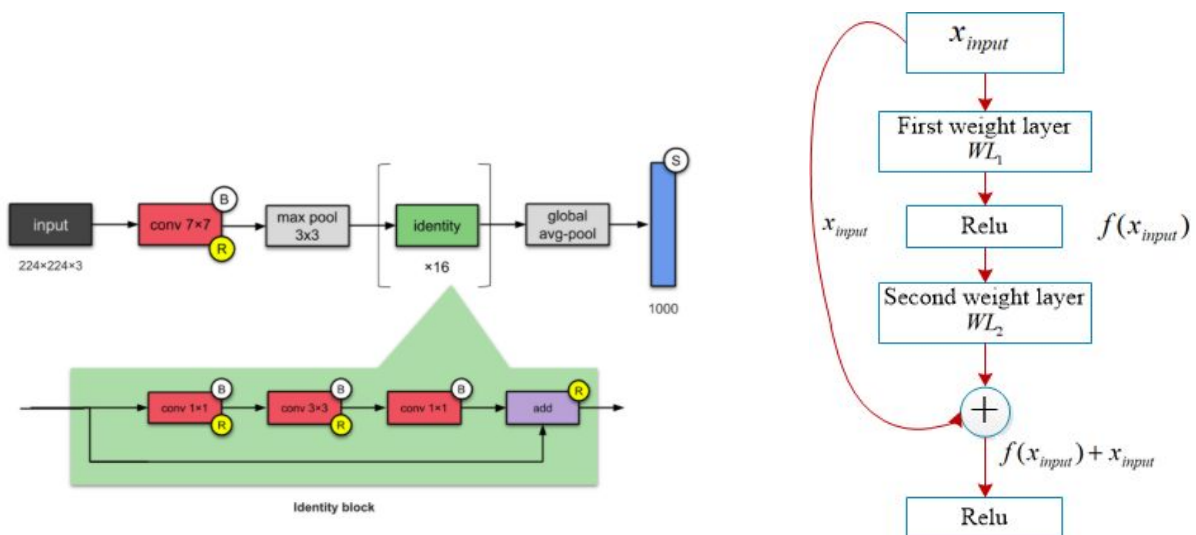


Figure 6: Identity Block of ResNet

Figure 7: ResNet skip connection

4. Design And Implementation

The hardware for integer/ fixed point is simple to design and more efficient at register transfer level but on CNN layer the accuracy reduces. As CNN layers are cascaded, with each layer the fixed point precision reduces leading to low performance. The bug detection using fixed point representation becomes very time consuming. Due to limitations in computation and high memory requirement led to being incompatible choice for embedded device. Due to this the need for new number format brought floating point representation as an alternative solution. Though floating point is complex to implement and has higher power consumption the overall performance of the network provides best results on CNN models. Precision requirement of a neural network cannot be generalized. Fixed point requires more precision even for small size. [1]

The project also proposes the use of varies IEEE floating-point number representation and shows the various efficiency ratio obtained by them. The network can be trained by using 32 bit, 16 bit and 8-bit floating point for all the arrays in the matrix, weights, activation function, errors, and gradients. Through the scaling, the partial products of multiplication can be accumulated to the desired bit-level which can drastically improve the efficiency in terms of energy, hardware, computational and also memory bandwidth. The use of cortex platform will be used in the implementation of these results. These different cases can be tried on MNIST to get a clear distinction. Also this scheme of representation will be implemented on different CNN models and the accuracy of each will be compared.[1][9][32].

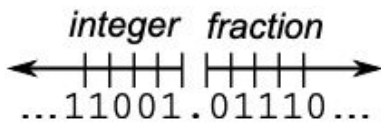


Figure 8: Fixed point number format

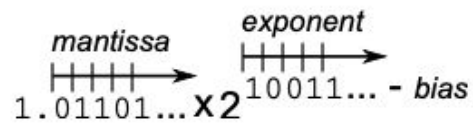


Figure 9: floating point number format

Through the implementation of various forms of IEEE754 floating point representation, analysis of performance, efficiency, power and memory of various floating point precisions are compared in various CNN models. The use of IEEE754 single precision floating point is preferred format on CNN models to achieve maximum accuracy on training dataset. But the use of 32 bit is likely needed for every network. The use of such high precision consumes more memory and time in training. The idea of using lower precision like IEEE754 half precision floating point is considered in this project as an optimal choice. The advantages of using lower precision are better usage of cache and bottleneck situation of bandwidth can be reduced. Since use of high precision means more run time hence high power and area consumption, the use of reduced precision can also help in reducing these facts with minimal to none accuracy loss.[31]

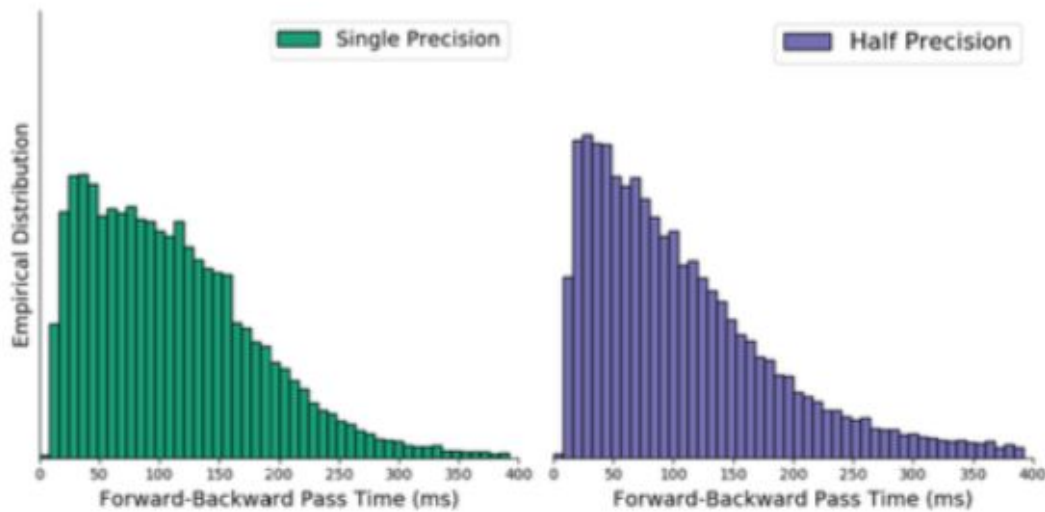


Figure 10: Research on run time on single and half precision floating point number

This figure shows the average run time of both the precision and is that the implementation is faster in half precision floating point than the single precision floating point. The run times are highly concentrated in the left portion of half

precision than single but few exceptions can be expected for different network as the half precision is observed to have thicker tail towards the right. [31]

4.1 CNN Implementation For MNIST Data

The convolutional neural network used in this project has a combination of convolutional layers, max-pooling layers, drop out, dense layers and so on. The parameters needed for a CNN model is shown below. The activation function used RELU and SoftMax. This image shows the basic training model of CNN for MNIST dataset. Each model is tested using fixed point and IEEE754 floating point (both 32 bit and 16 bit).

```
model = Sequential() # helps in adding layer by layer hence the use of "add"
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape, data_format="channels_last"))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25)) # avoids overfitting by dropping out few neurons for next layer
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.optimizers.Adadelta(), metrics=['accuracy'])
```

Figure 11: Basic CNN model for MNIST

The CNN model for different CNN architecture are different as the layers for each type of CNN is different based on number of convolutions performed, use of max pooling layer and the kernel size and identity block. Here we show the model for each CNN architecture:

4.1.1 LeNet:

```
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=(5,5), padding='same', activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPool2D(strides=2))
model.add(Conv2D(filters=48, kernel_size=(5,5), padding='valid', activation='relu'))
model.add(MaxPool2D(strides=2))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(84, activation='relu'))
model.add(Dense(10, activation='softmax'))
```


Figure 12: LeNet CNN model

The LeNet model consists of 2 convolutional layer and max pool layer with three dense layers with reduced filter size and the last dense layer being number of classes in the dataset (for MNIST =10)

4.1.2 AlexNet:

```
def AlexNet_Model():
    inputs = keras.Input(shape=(28, 28, 1))

    #first convolution layer:
    conv1 = keras.layers.Conv2D(filters=64, kernel_size=(3, 3), strides=1, padding='SAME', activation= tf.nn.relu)(inputs)
    pool1 = keras.layers.MaxPool2D(pool_size=(2, 2), padding='SAME')(conv1)
    norm1 = tf.nn.local_response_normalization(pool1, depth_radius=4, bias= 1.0, alpha= 0.001/ 9.0, beta=0.75)
    drop1 = keras.layers.Dropout(0.8)(norm1)

    #second convolution layer:
    conv2 = keras.layers.Conv2D(filters=128, kernel_size=(3, 3), strides=1, padding='SAME', activation= tf.nn.relu)(drop1)
    pool2 = keras.layers.MaxPool2D(pool_size=(2, 2), padding='SAME')(conv2)
    norm2 = tf.nn.local_response_normalization(pool2, depth_radius=4, bias= 1.0, alpha= 0.001/ 9.0, beta=0.75)
    drop2 = keras.layers.Dropout(0.8)(norm2)

    #third convolution layer:
    conv3 = keras.layers.Conv2D(filters=256, kernel_size=(3, 3), strides=1, padding='SAME', activation= tf.nn.relu)(drop2)
    pool3 = keras.layers.MaxPool2D(pool_size=(2, 2), padding='SAME')(conv3)
    norm3 = tf.nn.local_response_normalization(pool3, depth_radius=4, bias= 1.0, alpha= 0.001/ 9.0, beta=0.75)
    drop3 = keras.layers.Dropout(0.8)(norm3)

    #fully connected layer:
    flat = keras.layers.Flatten()(drop3)
    dense1 = keras.layers.Dense(units=1024, activation=tf.nn.relu)(flat)
    dense2 = keras.layers.Dense(units=1024, activation=tf.nn.relu)(dense1)

    logits = keras.layers.Dense(units=10)(dense2)
    return keras.Model(inputs=inputs, outputs=logits)

model = AlexNet_Model()
model.summary()
optimizer = tf.optimizers.Adam(learning_rate=learning_rate)
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=[ 'accuracy' ])
```

Figure 13: AlexNet CNN model

The AlexNet requires use of 3 convolutional layers and fully connected layer with max pooling, normalization and dropout at each layer and finally dense layers.

4.1.3 VGGNet:

```
cnn = models.Sequential()

cnn.add(conv.ZeroPadding2D((1,1), input_shape=(28, 28,1),))
cnn.add(conv.Convolution2D(nb_filters_1, nb_conv, nb_conv, activation="relu"))
cnn.add(conv.ZeroPadding2D((1, 1)))
cnn.add(conv.Convolution2D(nb_filters_1, nb_conv, nb_conv, activation="relu"))
cnn.add(conv.MaxPooling2D(strides=(2,2)))

cnn.add(conv.ZeroPadding2D((1, 1)))
cnn.add(conv.Convolution2D(nb_filters_2, nb_conv, nb_conv, activation="relu"))
cnn.add(conv.ZeroPadding2D((1, 1)))
cnn.add(conv.Convolution2D(nb_filters_2, nb_conv, nb_conv, activation="relu"))
cnn.add(conv.MaxPooling2D(strides=(2,2)))

cnn.add(conv.ZeroPadding2D((1, 1)))
cnn.add(conv.Convolution2D(nb_filters_3, nb_conv, nb_conv, activation="relu"))
cnn.add(conv.ZeroPadding2D((1, 1)))
cnn.add(conv.Convolution2D(nb_filters_3, nb_conv, nb_conv, activation="relu"))
cnn.add(conv.ZeroPadding2D((1, 1)))
cnn.add(conv.Convolution2D(nb_filters_3, nb_conv, nb_conv, activation="relu"))
cnn.add(conv.ZeroPadding2D((1, 1)))
cnn.add(conv.Convolution2D(nb_filters_3, nb_conv, nb_conv, activation="relu"))
cnn.add(conv.MaxPooling2D(strides=(2,2)))

cnn.add(core.Flatten())
cnn.add(core.Dropout(0.2))
cnn.add(core.Dense(1024, activation="relu"))
cnn.add(core.Dense(10, activation="softmax"))

cnn.summary()
cnn.compile(loss="categorical_crossentropy", optimizer="adadelata", metrics=["accuracy"])
```

Figure 14: VGGNet CNN model

4.2 Pre-Processing Of The Data

The MNIST dataset is pre-processed in each of the CNN model before being trained. The pre-processing of the data helps in better understanding of the data while training the model. Here, binary labels are added to the testing and training label and the desired number representation of the model is provided and further normalized between the value 0 and 1.[33]

```

# drop training label
labels = train['label'].values
train.drop('label', axis=1, inplace=True)
# reshape
images = train.values
images = np.array([np.reshape(i, (28, 28)) for i in images])
images = np.array([i.flatten() for i in images])

label_binarizer = LabelBinarizer() # to convert numerical variables to categorical variables for multi-class classification
labels = label_binarizer.fit_transform(labels)

# splitting
x_train, x_test, y_train, y_test = train_test_split(images, labels, test_size=0.3, random_state=101)

# input image dimensions
input_shape = (28,28,1)

# normalizing training and test data
x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255

x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)

z_train = kutils.to_categorical(x_train[:, 0])
nb_classes = z_train.shape[1]

```

Figure 15: Pre-processing of MNIST dataset

4.3 Data Augmentation

Data augmentation is a technique to create new training data artificially using existing data. Data Augmentation is used to enhance performance and better generalization of the model. New training examples are created for the same data. The use of Image data augmentation is done here for LeNet model as shown below for better fit of the model.[34]

```

# Data Augmentation
datagen = ImageDataGenerator(
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    zoom_range=0.1)
datagen.fit(X_dev)

```

Figure 16: Data Augmentation

4.4 Visualization

Python tensor board provides use of a website called “weights & Biases” when used after the model training and testing, stores the hyperparameters of the CNN model in its cloud. This wandb stores the memory usage, thread utilization, GPU/CPU time and also power consumption for graphical visualization and comparison amount the models. This can be imported using the following instruction.[35]

```
! pip install virtualenv

!pip install --upgrade wandb

import wandb

wandb.init(sync_tensorboard=True)
```

Figure 17: Importing Wandb

4.5 Processor Specification

Advanced RISC machine (ARM) is a family of RISC family of computer processors. ARM holdings develop architectures like system on chip(SOC) and system on modules(SOM) that integrate memory, interface etc. ARM also known for their implementation of instruction set on core designs to many companies. ARM have developed series of processors and one of them is used in implementation of CNN design in this project.[36]

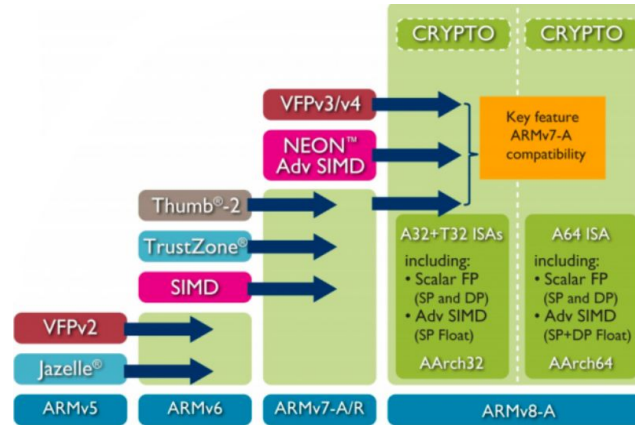


Figure 18: ARMv8-A processor (ARM64)

ARMv-8-A series are used in CPU supporting 64 bit architecture also known as AArch64/ ARM64. Their previous series ARMv7-A were of 32 bit architecture, with the 8th series the implementation is expanded to 64 bit instruction set and yet devices using 7 series are still compatible. These CPU are used on Apple, Samsung, Cortex-A54 cores and many other devices. With the new 8th series the IEEE754 are more compliant and support even double precision floating point format.[36]

For this project the ARM processors used is a CPU of ARM64 architecture type on an android phone to run the CNN model. The CPU version of AArch64 is 64 bit processing with more computing power and battery friendly with 8 core processors. This android CPU is used to run the CNN model using Jupyter Notebook. For future scope the ARM SDK kit can be implemented to run the CNN model.[37]

Droid Info	
Device	System
Memory	C
PROCESSOR	
CPU Architecture	AArch64 Processor rev 13 (aarch64)
Board	sdm845
Chipset	Qualcomm Technologies, Inc SDM845
Cores	8
Clock Speed	1766 MHz - 2803 MHz
Instruction Sets	arm64-v8a
CPU Features	fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp
CPU Governor	schedutil
Kernel Version	4.9.179-perf+
Kernel Architecture	aarch64

Figure 19: Processor specification obtained from android

5. Simulation

The simulation is performed using Google Colab/Jupyter Notebook which runs and trains the MNIST dataset. The Colab provided two runtime options GPU and TPU. Depending on the applications being run on Colab the runtime tool can be changed. The hardware accelerators help in faster training of the CNN models. The Jupyter notebook utilizes localhost which is prompted by the terminal of the system. The training on Jupyter utilises the use of CPU of the system making it slower to run in comparison to Google Colab.

The MNIST dataset is available on either TensorFlow or Keras library. This dataset contains four files, one csv of train images, one of train labels and each of these for test set as well. The training images are 60,000 and test of 10,000. As the datasets are uploaded to the notebook and training of the model is performed the model summary details are obtained.

Model: "sequential_3"

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 28, 28, 32)	832
max_pooling2d_5 (MaxPooling2D)	(None, 14, 14, 32)	0
conv2d_6 (Conv2D)	(None, 10, 10, 48)	38448
max_pooling2d_6 (MaxPooling2D)	(None, 5, 5, 48)	0
flatten_3 (Flatten)	(None, 1200)	0
dense_7 (Dense)	(None, 256)	307456
dense_8 (Dense)	(None, 84)	21588
dense_9 (Dense)	(None, 10)	850
Total params: 369,174		
Trainable params: 369,174		
Non-trainable params: 0		

Figure 20: Model Summary of CNN model

As the model is ready for training, based on the batch size and epochs provided the model training will be performed with the live update of accuracy and loss predictions provided as desired in the model.

```
Epoch 1/20
582/582 [=====] - 20s 34ms/step - loss: 0.3819 - accuracy: 0.8821 - val_loss: 0.1081 - val_accuracy: 0.9617
Epoch 2/20
6/582 [.....] - ETA: 15s - loss: 0.1953 - accuracy: 0.9417/usr/local/lib/python3.6/dist-packages/keras/callbacks/callbacks.py:1042
(self.monitor, ', '.join(list(logs.keys()))), RuntimeWarning
582/582 [=====] - 20s 34ms/step - loss: 0.1200 - accuracy: 0.9640 - val_loss: 0.0543 - val_accuracy: 0.9833
Epoch 3/20
582/582 [=====] - 20s 35ms/step - loss: 0.0834 - accuracy: 0.9736 - val_loss: 0.0321 - val_accuracy: 0.9894
Epoch 4/20
582/582 [=====] - 20s 34ms/step - loss: 0.0677 - accuracy: 0.9795 - val_loss: 0.0310 - val_accuracy: 0.9928
Epoch 5/20
582/582 [=====] - 20s 34ms/step - loss: 0.0576 - accuracy: 0.9822 - val_loss: 0.0263 - val_accuracy: 0.9889
Epoch 6/20
582/582 [=====] - 20s 34ms/step - loss: 0.0516 - accuracy: 0.9844 - val_loss: 0.0219 - val_accuracy: 0.9933
Epoch 7/20
582/582 [=====] - 20s 34ms/step - loss: 0.0451 - accuracy: 0.9857 - val_loss: 0.0177 - val_accuracy: 0.9939
Epoch 8/20
582/582 [=====] - 20s 34ms/step - loss: 0.0393 - accuracy: 0.9874 - val_loss: 0.0274 - val_accuracy: 0.9900
Epoch 9/20
582/582 [=====] - 19s 33ms/step - loss: 0.0373 - accuracy: 0.9883 - val_loss: 0.0208 - val_accuracy: 0.9939
Epoch 10/20
582/582 [=====] - 20s 34ms/step - loss: 0.0359 - accuracy: 0.9885 - val_loss: 0.0245 - val_accuracy: 0.9911
Epoch 11/20
582/582 [=====] - 20s 34ms/step - loss: 0.0336 - accuracy: 0.9892 - val_loss: 0.0224 - val_accuracy: 0.9928
Epoch 12/20
582/582 [=====] - 19s 33ms/step - loss: 0.0320 - accuracy: 0.9897 - val_loss: 0.0174 - val_accuracy: 0.9939
Epoch 13/20
582/582 [=====] - 20s 34ms/step - loss: 0.0278 - accuracy: 0.9915 - val_loss: 0.0182 - val_accuracy: 0.9944
Epoch 14/20
582/582 [=====] - 19s 33ms/step - loss: 0.0283 - accuracy: 0.9912 - val_loss: 0.0163 - val_accuracy: 0.9944
Epoch 15/20
582/582 [=====] - 20s 34ms/step - loss: 0.0271 - accuracy: 0.9912 - val_loss: 0.0172 - val_accuracy: 0.9956
Epoch 16/20
582/582 [=====] - 19s 33ms/step - loss: 0.0250 - accuracy: 0.9923 - val_loss: 0.0261 - val_accuracy: 0.9906
Epoch 17/20
582/582 [=====] - 19s 33ms/step - loss: 0.0234 - accuracy: 0.9924 - val_loss: 0.0216 - val_accuracy: 0.9922
Epoch 18/20
582/582 [=====] - 19s 33ms/step - loss: 0.0233 - accuracy: 0.9926 - val_loss: 0.0207 - val_accuracy: 0.9928
Epoch 19/20
```

Figure 21: Training of the model

The simulation results are obtained in the end and the plots of loss v/s accuracy of training and test data are plotted based on the results obtained. For the visualisation of memory and power etc, weights & Biases are imported in the end of the model. The wandb library is from Tensor board which links the results to a website where graphical visualisation the parameters and results can be stored after each run and also used for comparison among various models.[35]

```

Requirement already satisfied: virtualenv in /usr/local/lib/python3.6/dist-packages (20.0.17)
Requirement already satisfied: six<2,>=1.9.0 in /usr/local/lib/python3.6/dist-packages (from virtualenv) (1.12.0)
Requirement already satisfied: distlib<1,>=0.3.0 in /usr/local/lib/python3.6/dist-packages (from virtualenv) (0.3.0)
Requirement already satisfied: appdirs<2,>=1.4.3 in /usr/local/lib/python3.6/dist-packages (from virtualenv) (1.4.3)
Requirement already satisfied: importlib-resources<2,>=1.0; python_version < "3.7" in /usr/local/lib/python3.6/dist-packages (from virtualenv) (1.4.0)
Requirement already satisfied: importlib-metadata<2,>=0.12; python_version < "3.8" in /usr/local/lib/python3.6/dist-packages (from virtualenv) (1.6.0)
Requirement already satisfied: filelock<4,>=3.0.0 in /usr/local/lib/python3.6/dist-packages (from virtualenv) (3.0.12)
Requirement already satisfied: zipp>=0.4; python_version < "3.8" in /usr/local/lib/python3.6/dist-packages (from importlib-resources<2,>=1.0; python_version < "3.7" in /usr/local/lib/python3.6/dist-packages) (0.8.3)
Requirement already satisfied, skipping upgrade: GitPython>=1.0.0 in /usr/local/lib/python3.6/dist-packages (from wandb) (3.1.1)
Requirement already satisfied, skipping upgrade: gql==0.2.0 in /usr/local/lib/python3.6/dist-packages (from wandb) (0.2.0)
Requirement already satisfied, skipping upgrade: six>=1.10.0 in /usr/local/lib/python3.6/dist-packages (from wandb) (1.12.0)
Requirement already satisfied, skipping upgrade: configparser>=3.8.1 in /usr/local/lib/python3.6/dist-packages (from wandb) (5.0.0)
Requirement already satisfied, skipping upgrade: psutil>=5.0.0 in /usr/local/lib/python3.6/dist-packages (from wandb) (5.4.8)
Requirement already satisfied, skipping upgrade: watchdog>=0.8.3 in /usr/local/lib/python3.6/dist-packages (from wandb) (0.10.2)
Requirement already satisfied, skipping upgrade: sentry-sdk>=0.4.0 in /usr/local/lib/python3.6/dist-packages (from wandb) (0.14.3)
Requirement already satisfied, skipping upgrade: requests>=2.0.0 in /usr/local/lib/python3.6/dist-packages (from wandb) (2.21.0)
Requirement already satisfied, skipping upgrade: python-dateutil>=2.6.1 in /usr/local/lib/python3.6/dist-packages (from wandb) (2.8.1)
Requirement already satisfied, skipping upgrade: shortuuid>=0.5.0 in /usr/local/lib/python3.6/dist-packages (from wandb) (1.0.1)
Requirement already satisfied, skipping upgrade: nvidia-ml-py3>=7.352.0 in /usr/local/lib/python3.6/dist-packages (from wandb) (7.352.0)
Requirement already satisfied, skipping upgrade: subprocess32>=3.5.3 in /usr/local/lib/python3.6/dist-packages (from wandb) (3.5.4)
Requirement already satisfied, skipping upgrade: Click>=7.0 in /usr/local/lib/python3.6/dist-packages (from wandb) (7.1.1)
Requirement already satisfied, skipping upgrade: PyYAML>=3.10 in /usr/local/lib/python3.6/dist-packages (from wandb) (3.13)
Requirement already satisfied, skipping upgrade: docker-pycreds>=0.4.0 in /usr/local/lib/python3.6/dist-packages (from wandb) (0.4.0)
Requirement already satisfied, skipping upgrade: gitdb<5,>=4.0.1 in /usr/local/lib/python3.6/dist-packages (from GitPython>=1.0.0->wandb) (4.0.4)
Requirement already satisfied, skipping upgrade: graphql-core<2,>=0.5.0 in /usr/local/lib/python3.6/dist-packages (from gql==0.2.0->wandb) (1.1)
Requirement already satisfied, skipping upgrade: promise<3,>=2.0 in /usr/local/lib/python3.6/dist-packages (from gql==0.2.0->wandb) (2.3)
Requirement already satisfied, skipping upgrade: pathtools>=0.1.1 in /usr/local/lib/python3.6/dist-packages (from watchdog>=0.8.3->wandb) (0.1.2)
Requirement already satisfied, skipping upgrade: urllib3>=1.10.0 in /usr/local/lib/python3.6/dist-packages (from sentry-sdk>=0.4.0->wandb) (1.24.3)
Requirement already satisfied, skipping upgrade: certifi in /usr/local/lib/python3.6/dist-packages (from sentry-sdk>=0.4.0->wandb) (2020.4.5.1)
Requirement already satisfied, skipping upgrade: chardet<3.1.0,>=3.0.2 in /usr/local/lib/python3.6/dist-packages (from requests>=2.0.0->wandb) (3.0.4)
Requirement already satisfied, skipping upgrade: idna<2.9,>=2.5 in /usr/local/lib/python3.6/dist-packages (from requests>=2.0.0->wandb) (2.8)
Requirement already satisfied, skipping upgrade: smmap<4,>=3.0.1 in /usr/local/lib/python3.6/dist-packages (from gitdb<5,>=4.0.1->GitPython>=1.0.0->wandb) (3.0.1)
Logging results to Weights & Biases \(Documentation\).
Project page: https://app.wandb.ai/shrusti/uncategorized
Run page: https://app.wandb.ai/shrusti/uncategorized/runs/25q436so
W&B Run: https://app.wandb.ai/shrusti/uncategorized/runs/25q436so

```

Figure 22: wandb simulation

As the last three hyperlinks seen in the figure above can be clicked on for viewing the graphical results.[35]

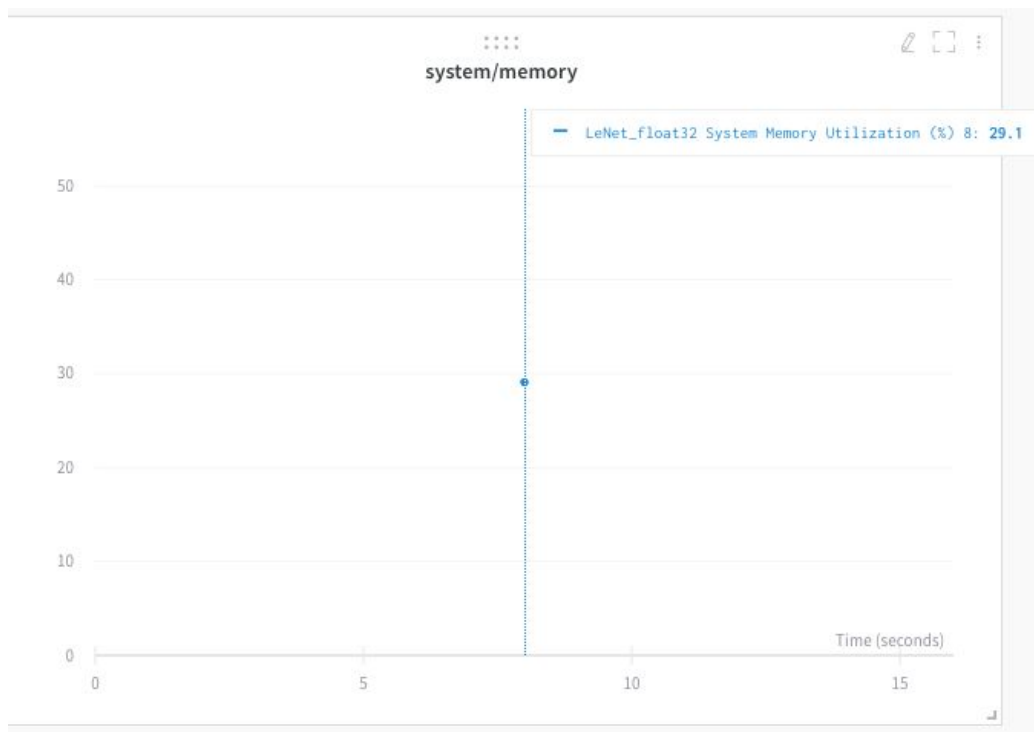


Figure 23: weights and Biases website

6. Discussion/Analysis

Aiming for implementation on embedded devices enhances the significance of the CNN applications for image recognition, object detection etc. The choice of number representation can lead to reduction in performance of the entire system. The hardware implementations of such a model can be highly area and memory consuming based on the choice of number representation. The use of fixed point is always considered as safe choice due to their hardware compatibility, but as the design got complex the need for alternative format switched preference to floating point.

In this project the results obtained used fixed point and floating point for the same CNN models are evaluated. The change in the accuracy, loss and also power and memory consumption of each CNN model based on the numeric format is shown. Further the 32 bit and 16 bit IEEE754 floating point precision effect on each model is presented.

The performance metrics are based on:

1. Accuracy
2. Loss (Mean Squared Error)
3. Memory consumption of each model
4. Power consumption of each model

The next few pages show the graphs for accuracy and loss of the validation and training of each CNN architecture with fixed point, single precision and half precision data. The graph will show the change in numeric representation format's impact on each CNN architecture and prove that change in precision can improve in the accuracy, loss, power consumption and memory consumption of CNNs.[35]

6.1 Fixed point data

MNIST data training for 20 epochs on each CNN architecture are shown below. The accuracy obtained is 43.3

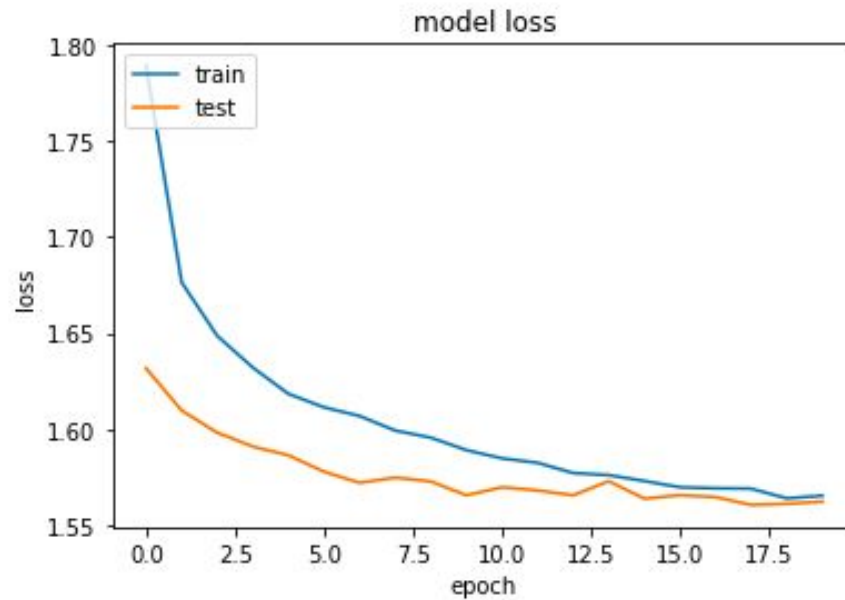


Figure 24: MNIST data model loss for fixed point

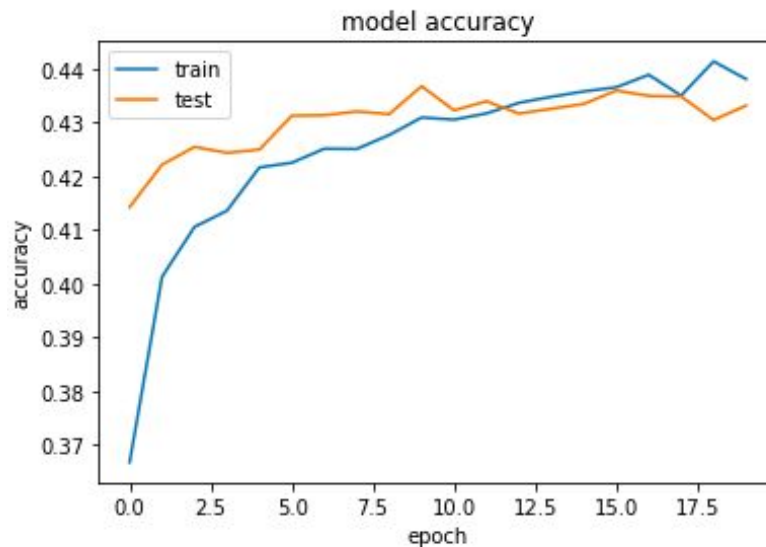


Figure 25: MNIST data model accuracy for fixed point

For ResNet model training of 20 epochs. The accuracy was 98.88

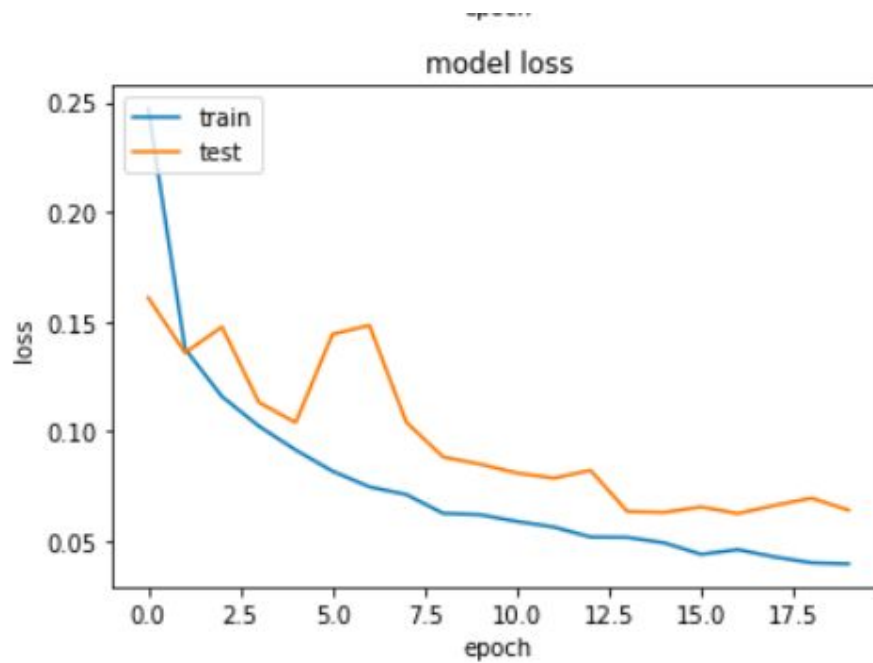


Figure 26: MNIST data model loss for fixed point

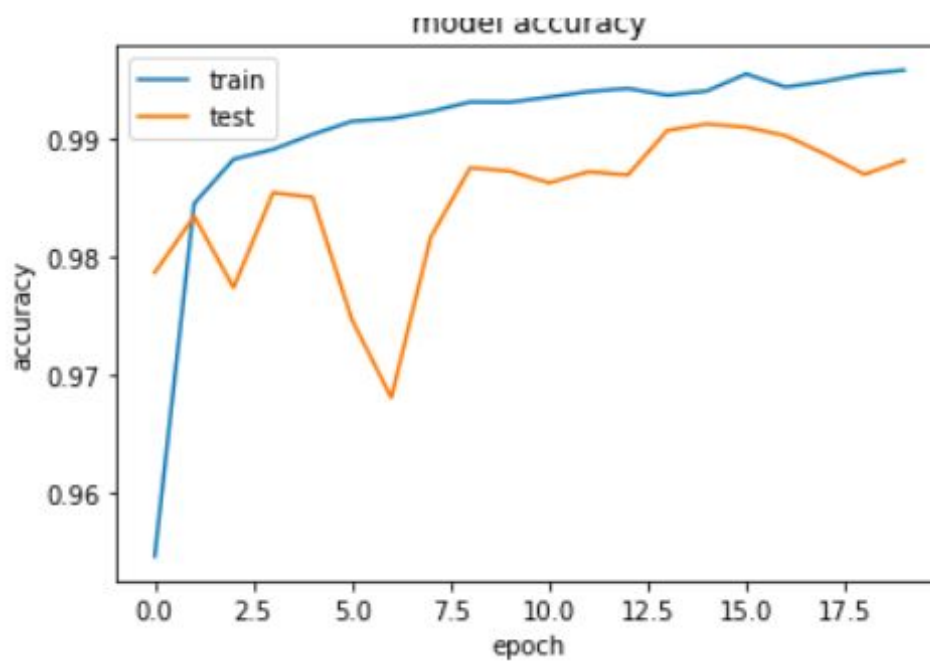


Figure 27: MNIST data model accuracy for fixed point

For LeNet model for 20 epochs. The accuracy is 99.2.

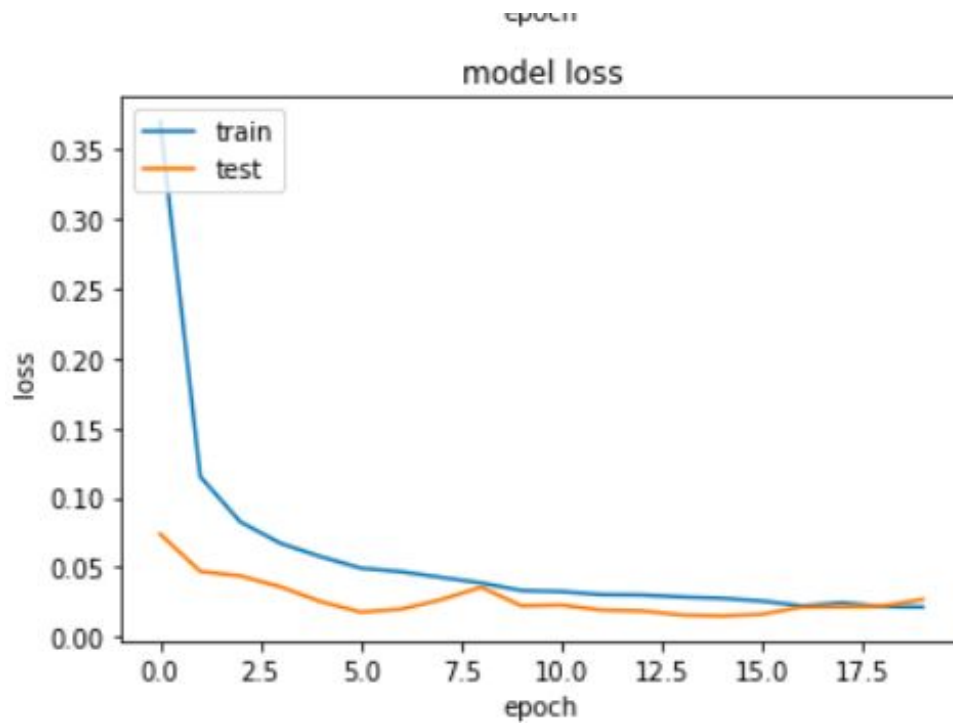


Figure 28: MNIST data model loss for fixed point

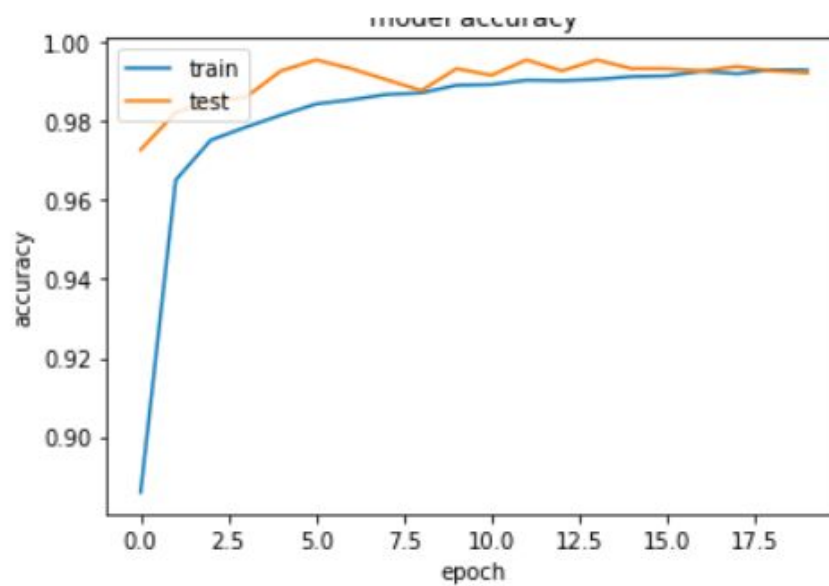


Figure 29: MNIST data model accuracy for fixed point

For VGGNet with 20 epochs, the accuracy obtained is 99.38

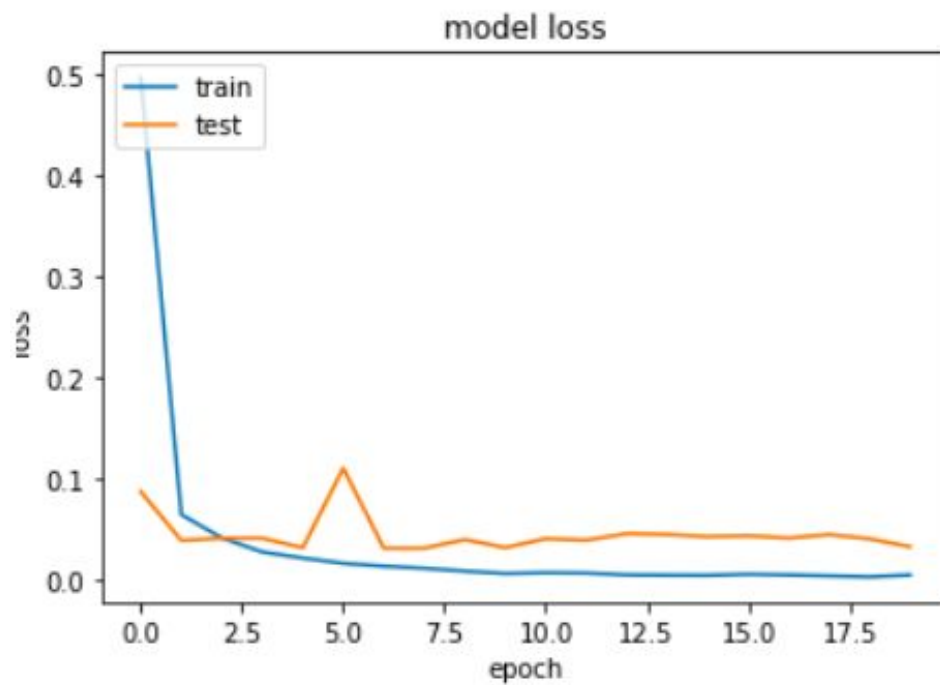


Figure 30: MNIST data model loss for fixed point

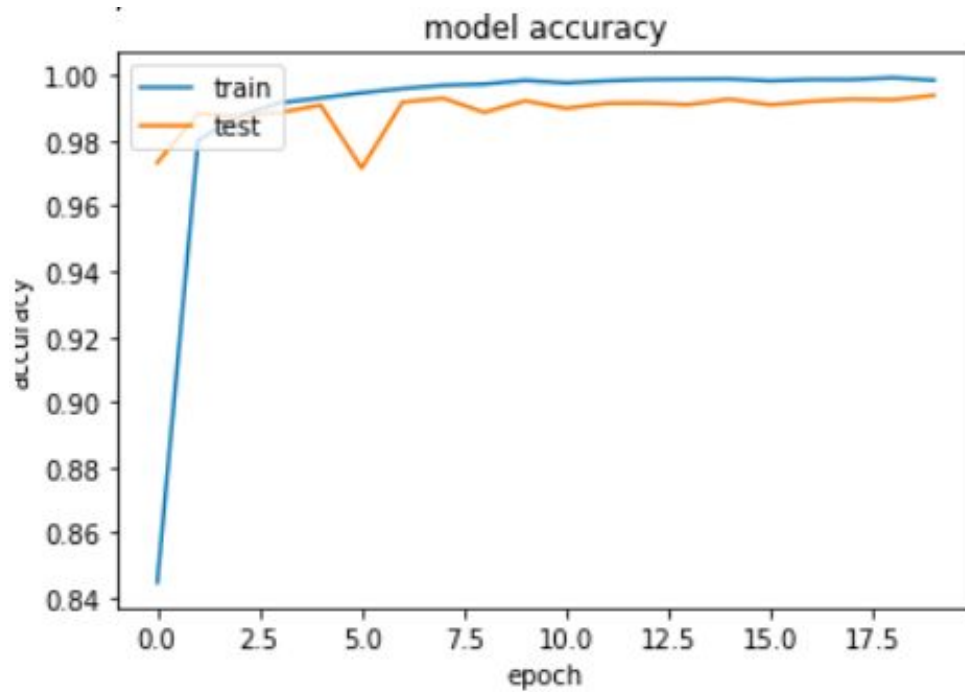


Figure 31: MNIST data model accuracy for fixed point

For AlexNet model with 20 epochs, the accuracy obtained is 98.4

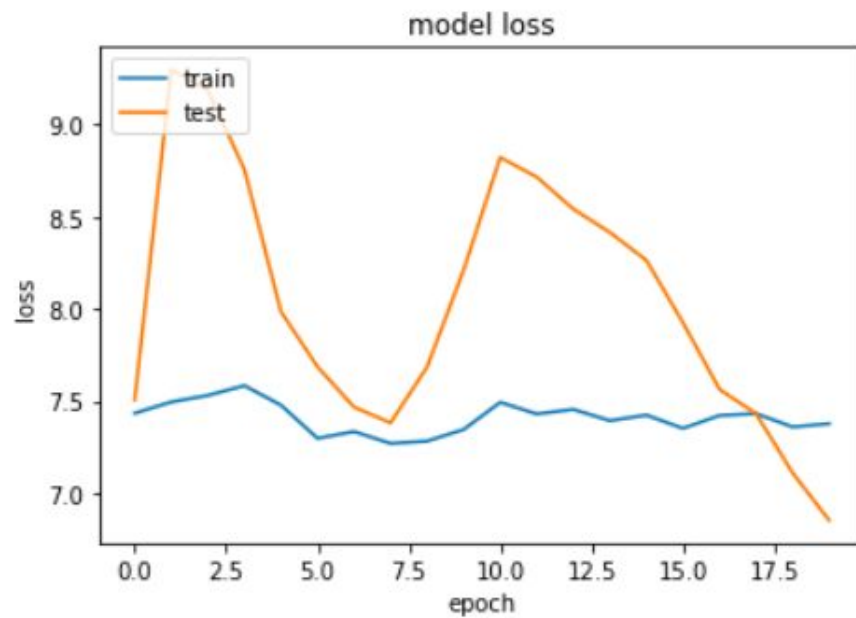


Figure 32: MNIST data model loss for fixed point

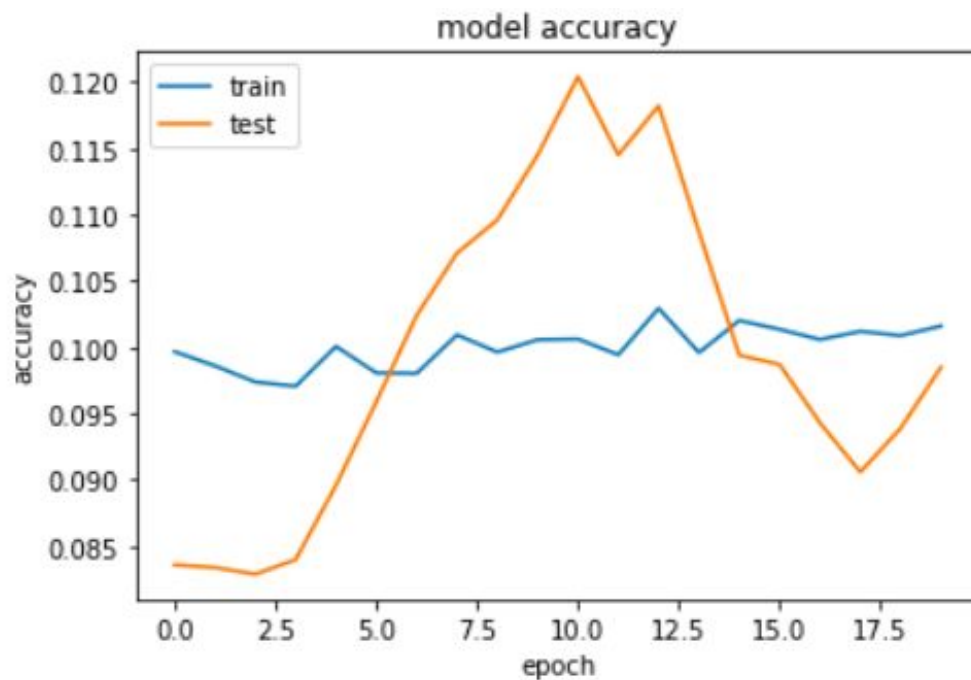


Figure 33: MNIST data model accuracy for fixed point

6.2 Floating point

The following graphs will present the CNN architectures accuracy and loss using IEEE754 single precision and half precision. Following graphs are for MNIST data for 20 epochs .

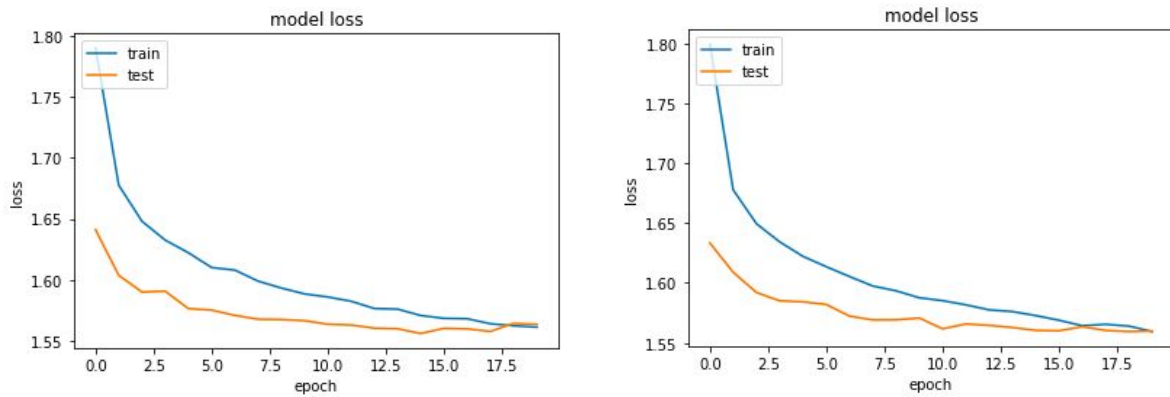


Figure 34: MNIST data model loss for 16 bit(on the left) and 32 bit(on the right) point

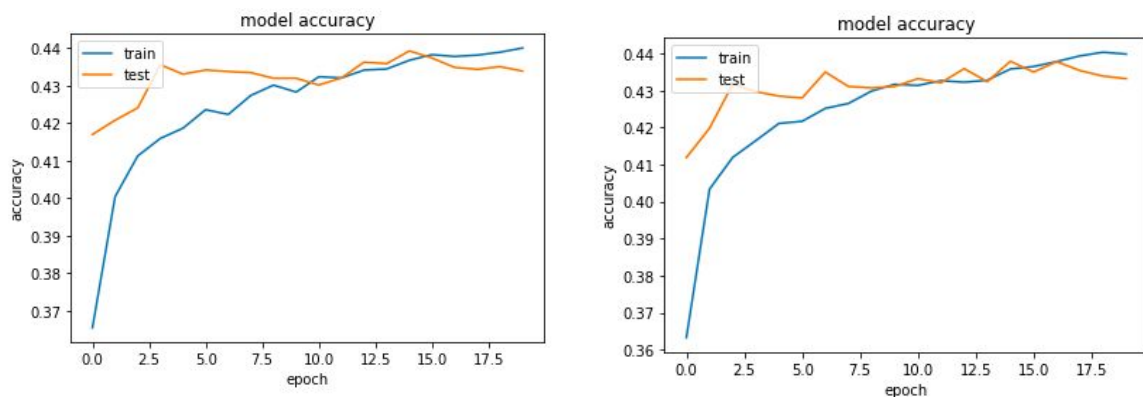


Figure 35: MNIST data model accuracy for 16 bit(on the left) and 32 bit(on the right) point

For ResNet model of 20 epochs

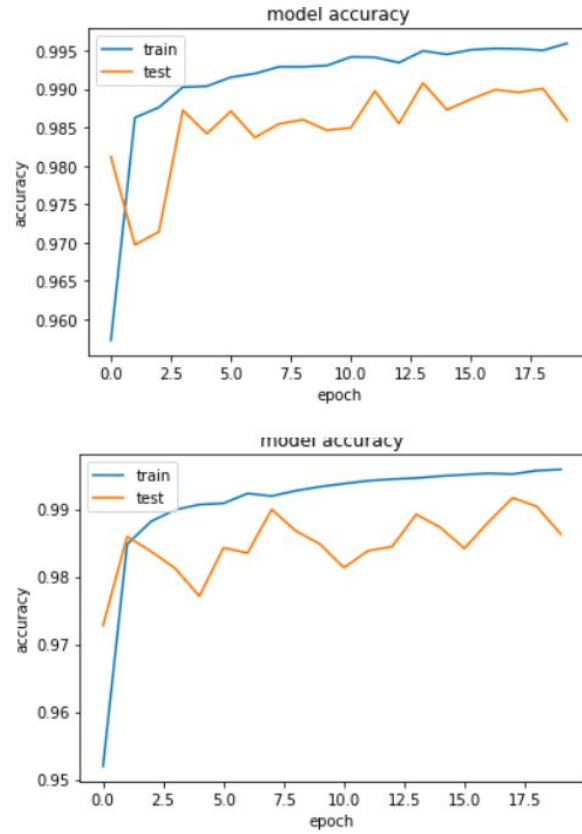


Figure 36: MNIST data model accuracy for 32 bit(on the left) and 16 bit(on the right) point

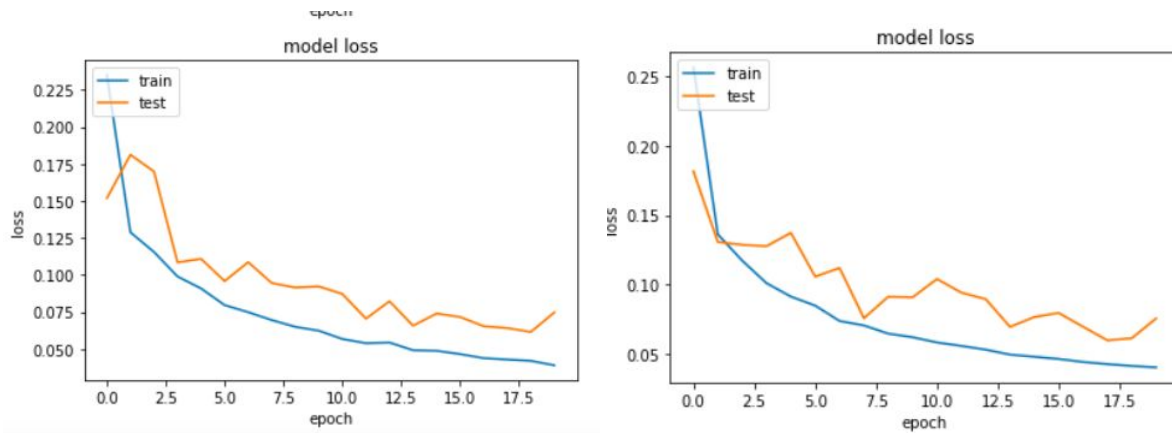


Figure 37: MNIST data model accuracy for 16 bit(on the left) and 32 bit(on the right) point

For LeNet model of 20 epochs

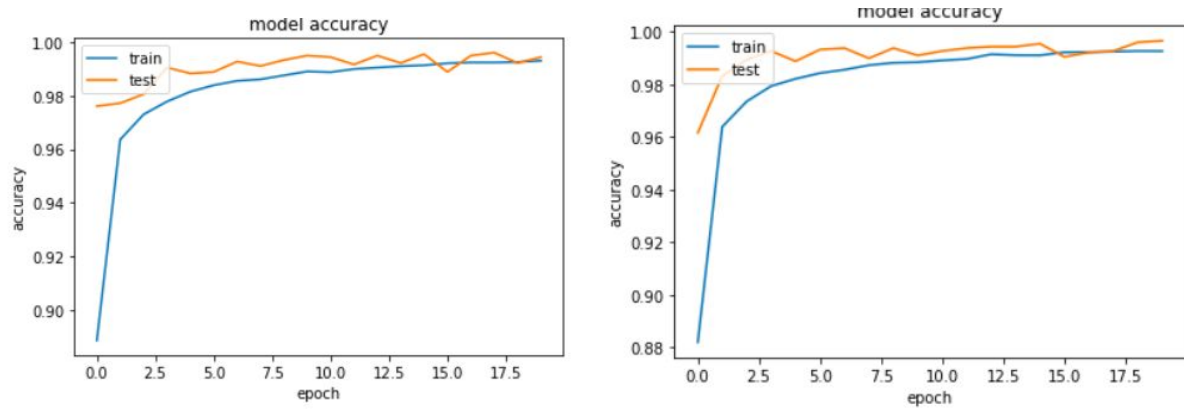


Figure 38: MNIST data model loss for 16 bit(on the left) and 32 bit(on the right) point

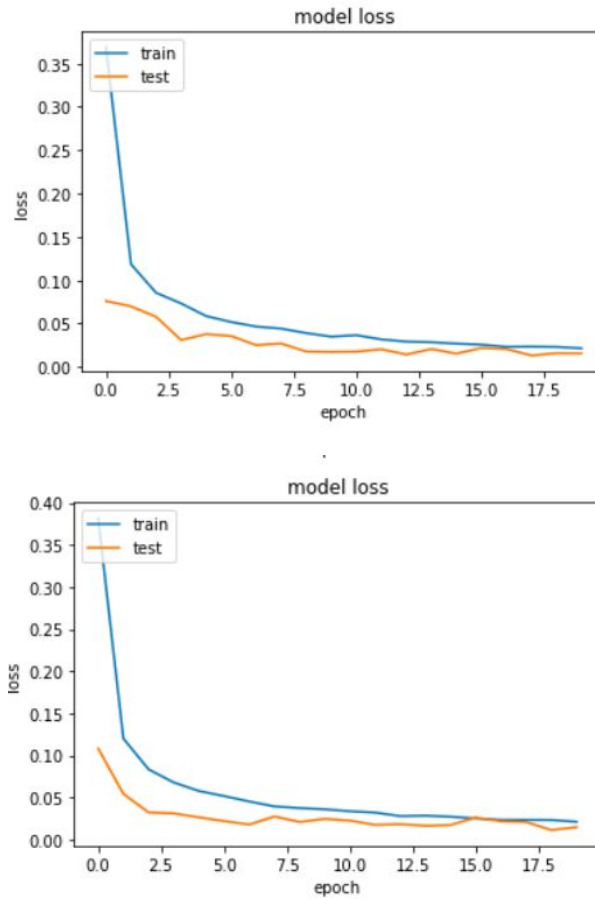


Figure 39: MNIST data model accuracy for 16 bit(on the left) and 32 bit(on the right) point

For VGGNet model of 20 epochs

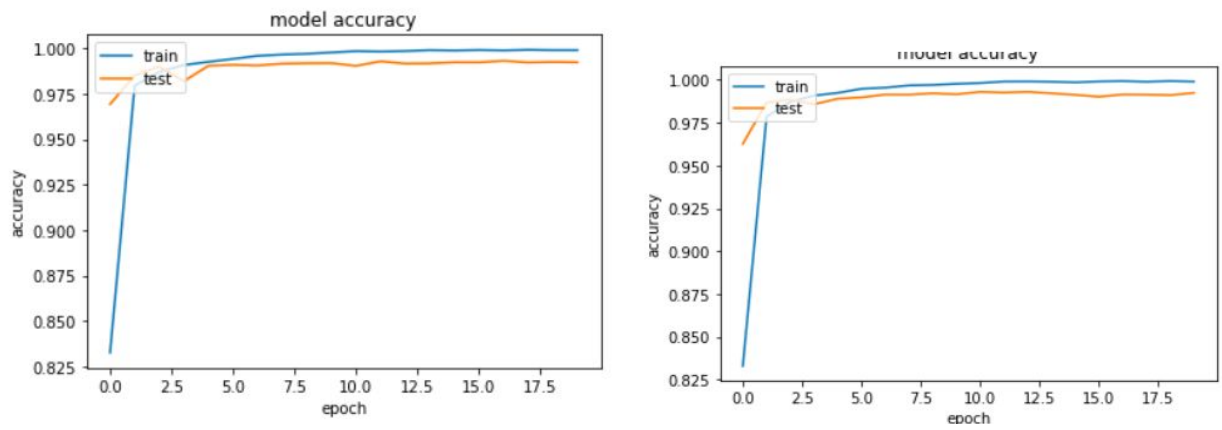


Figure 40: MNIST data model accuracy for 16 bit(on the left) and 32 bit(on the right) point

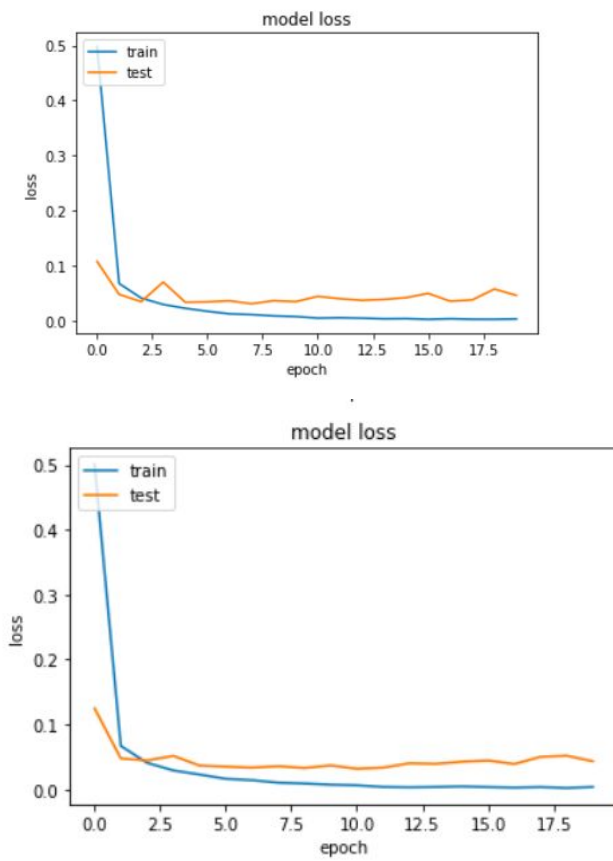


Figure 41: MNIST data model loss for 16 bit(on the left) and 32 bit(on the right) point

For AlexNet model of 20 epochs



Figure 42: MNIST data model accuracy for 32 bit(on the left) and 16 bit(on the right) point

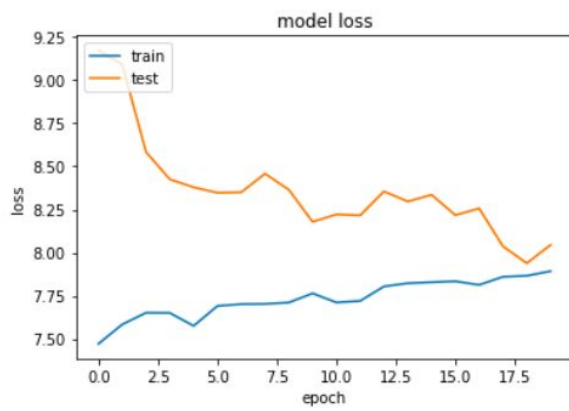
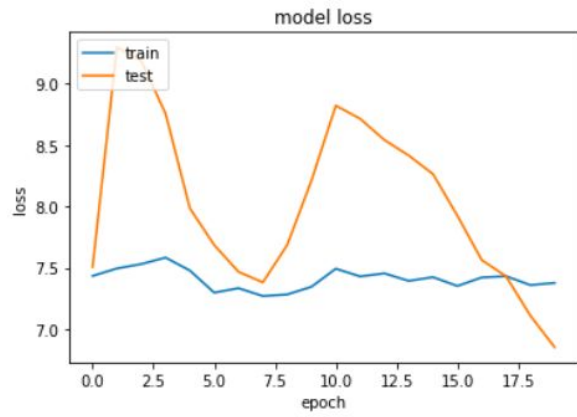


Figure 43: MNIST data model loss for 32 bit(on the left) and 16 bit(on the right) point

As the model accuracy and loss of each CNNs are presented above using both fixed and floating point, here each model will be compared in terms of memory usage, system utilization and power usage for fixed, half precision and single precision floating point. [35]

For MNIST CNN model:

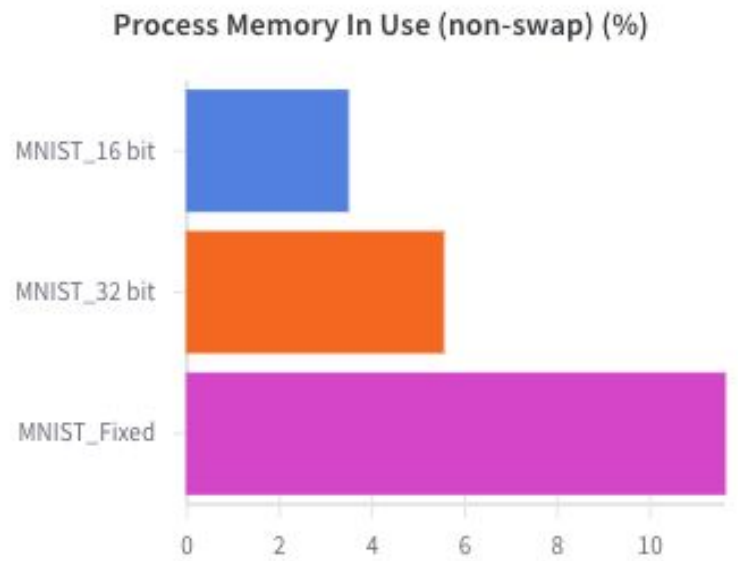


Figure 44: MNIST data model memory usage

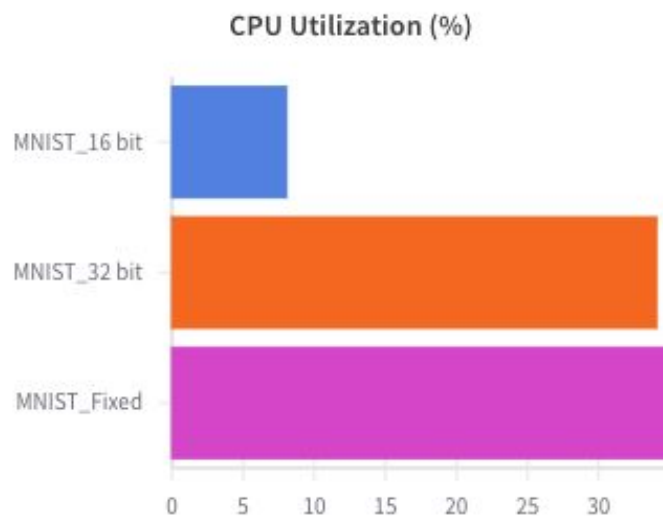


Figure 45: MNIST data model CPU usage

For AlexNet model the results were unexpected for few things.

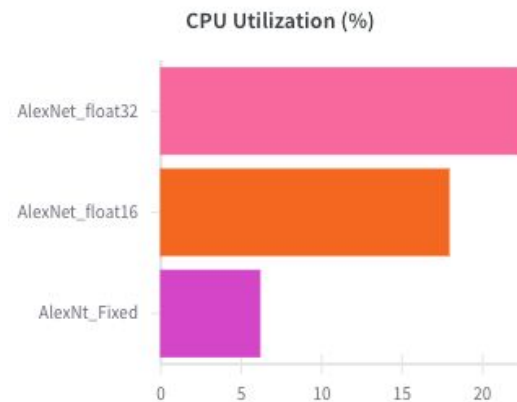


Figure 46: MNIST data model CPU usage

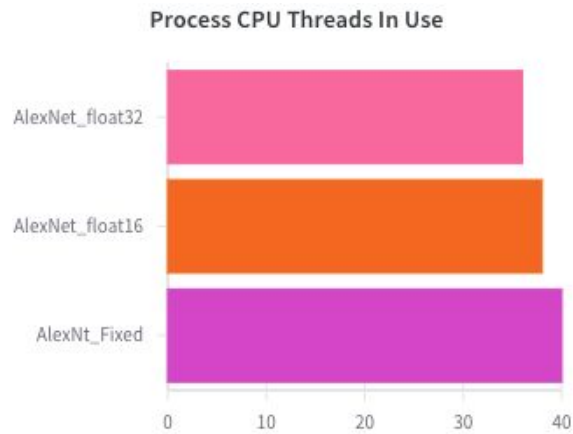


Figure 47: MNIST data model process CPU thread use on weights & Biases

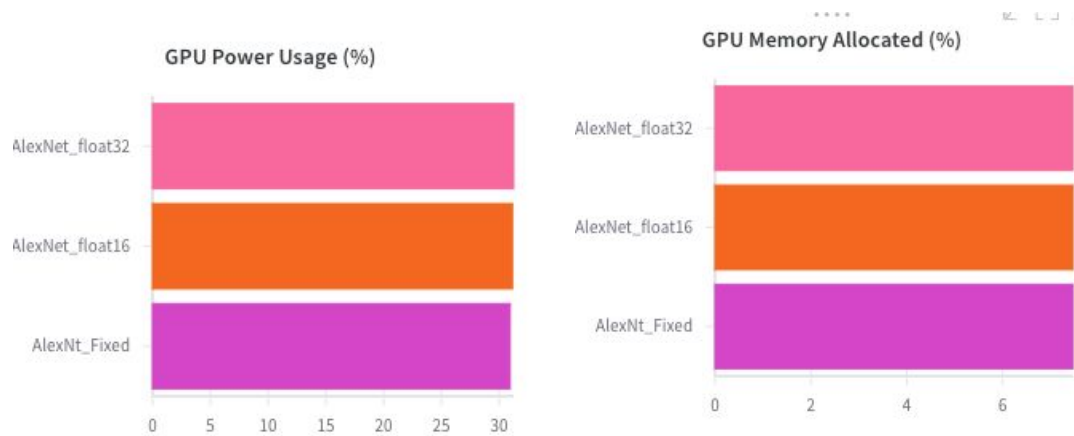


Figure 48: MNIST data model process GPU usage on Colab on weights & Biases

For LeNet model the results were unexpected for few things.

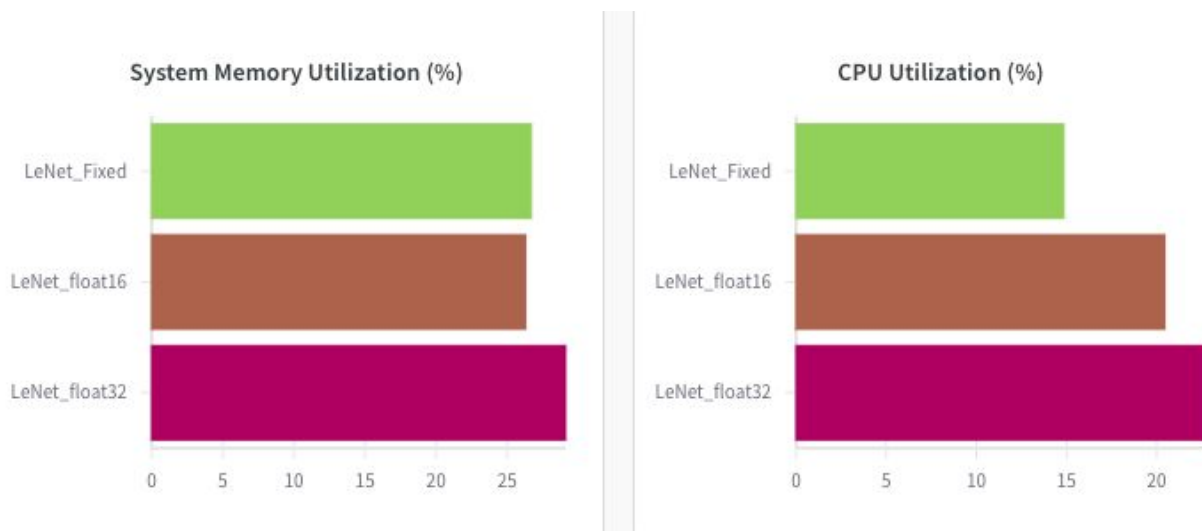


Figure 49: MNIST data model process system and CPU usage on weights & Biases



Figure 50: MNIST data model process GPU usage on Colab on weights & Biases

For VGGNet model the results are as shown below

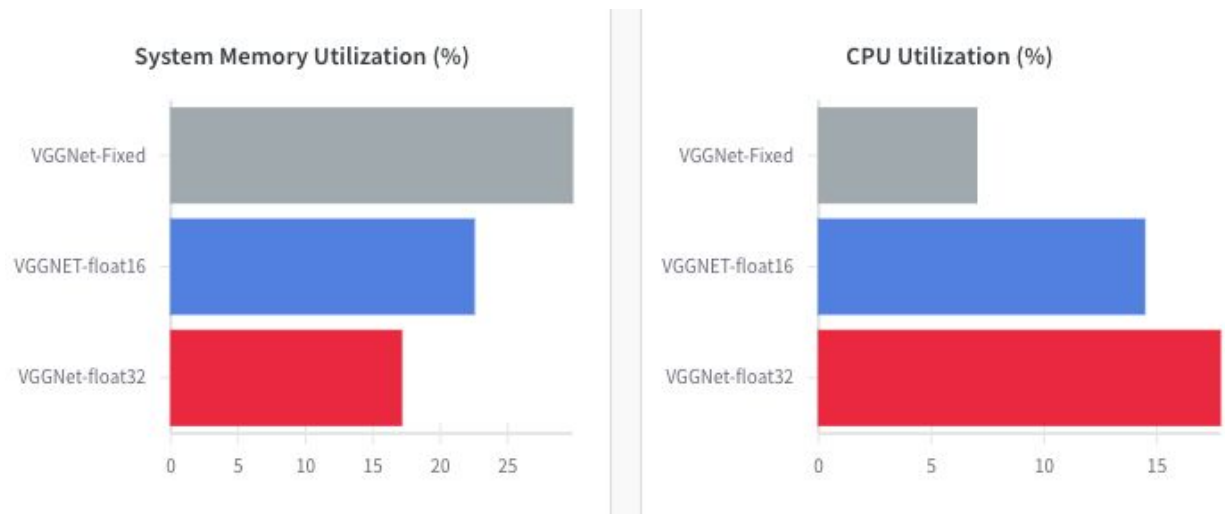


Figure 51: MNIST data model process system and CPU usage on weights & Biases

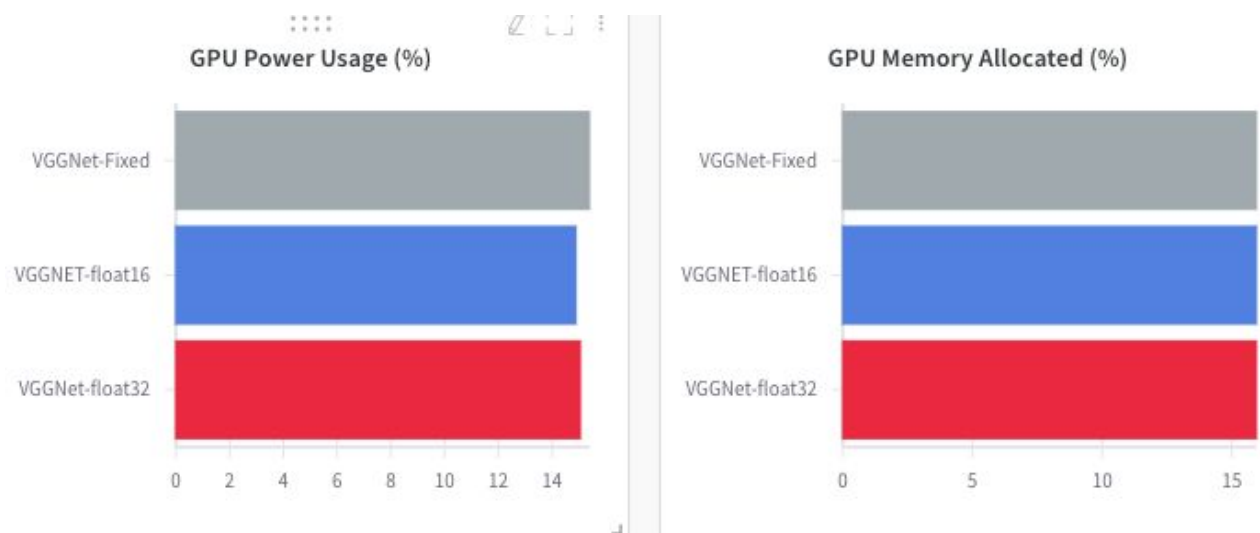


Figure 52: MNIST data model process GPU usage on Colab on weights & Biases

For ResNet model the results are as shown below

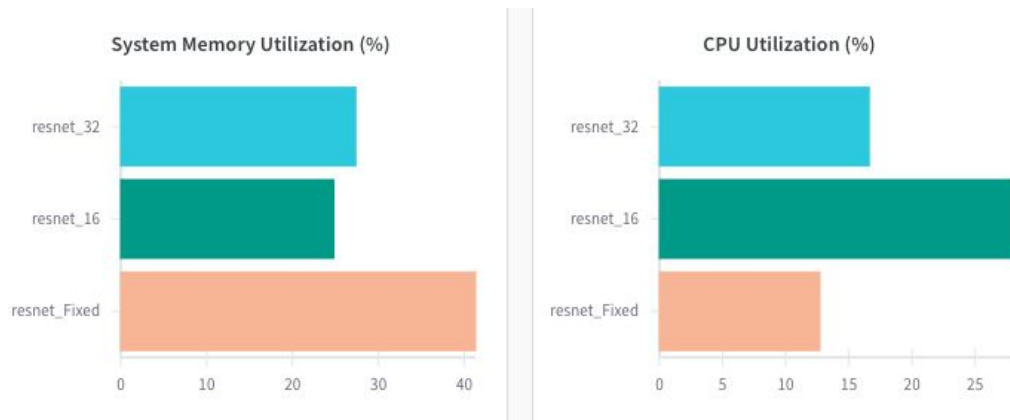


Figure 53: MNIST data model process system and CPU usage on weights & Biases

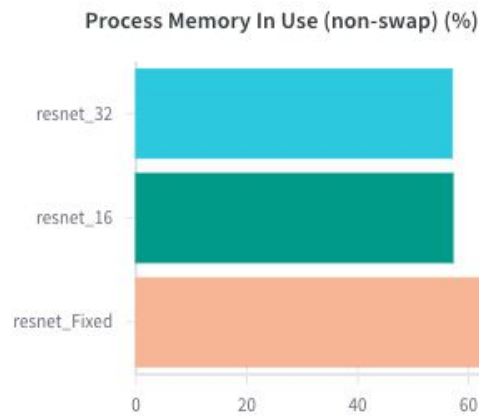


Figure 54: MNIST ResNet process memory in use

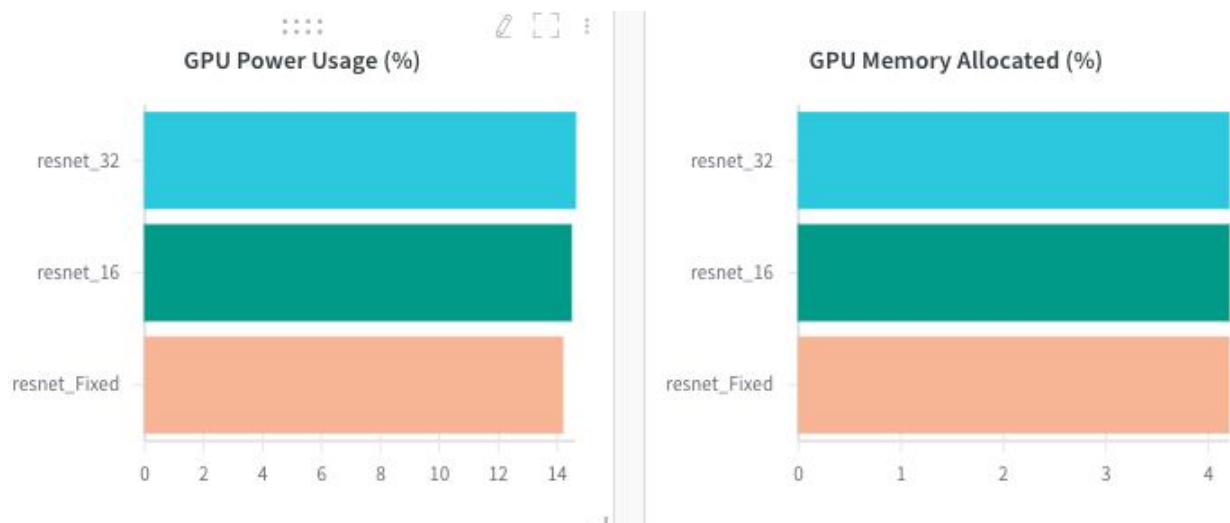


Figure 55: MNIST data model process GPU usage on Colab on weights & Biases

As seen above the usage of memory, power and CPU/GPU of each CNN architecture are different due to the CNN model. Also, we can observe not all the models depict the lower power usage for fixed or higher memory usage for float. Some CNNs also represent similar usage for even reduced precision. The overall charts for the above presented CNN are as shown.[35]

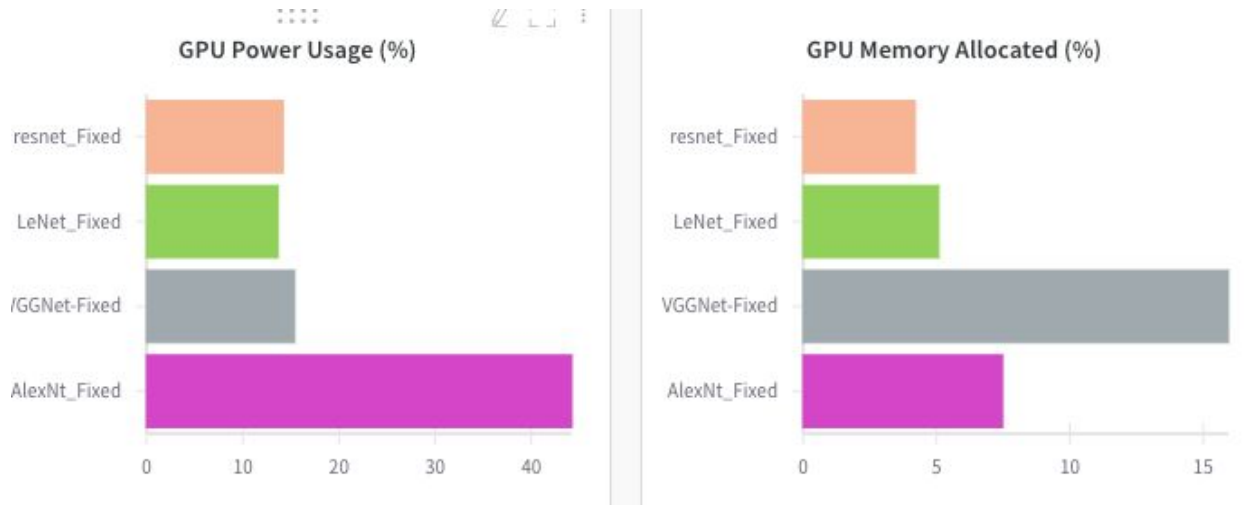


Figure 56: MNIST data model process GPU usage on Colab on weights & Biases



Figure 57: MNIST data model process CPU usage on Colab on weights & Biases

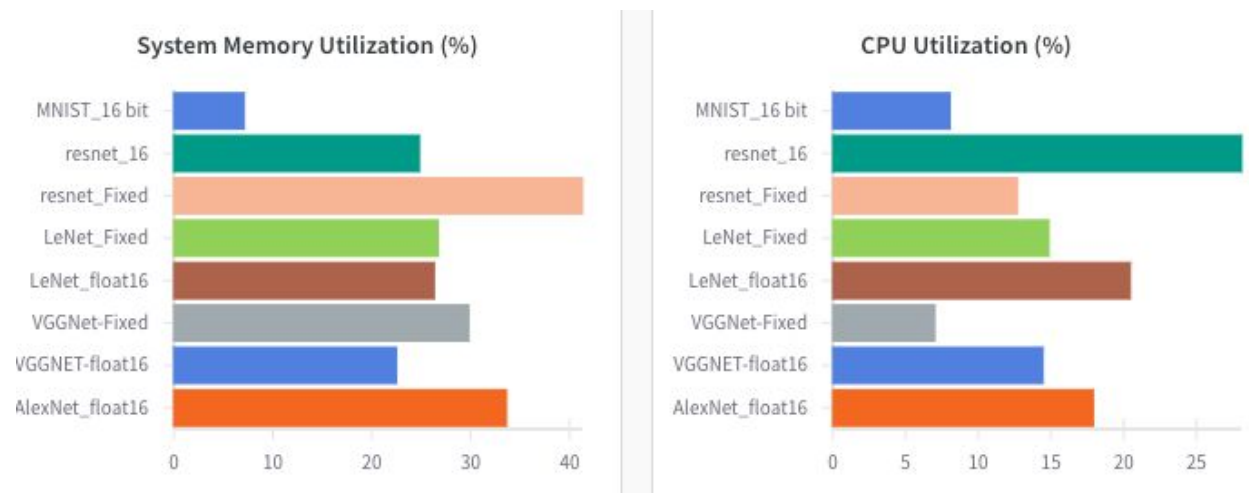


Figure 58: MNIST data model process CPU usage on Colab on weights & Biases for all half precision model

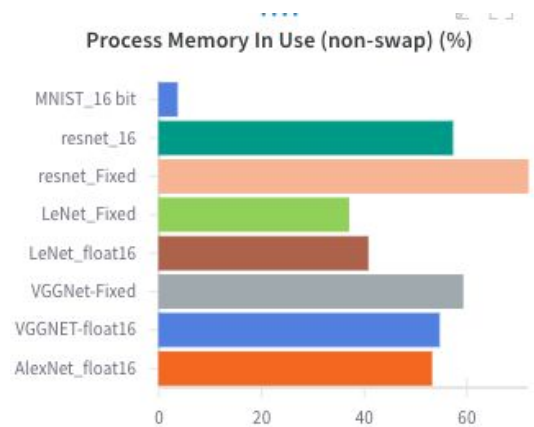


Figure 59: MNIST data model process usage on Colab on weights & Biases for all half precision model

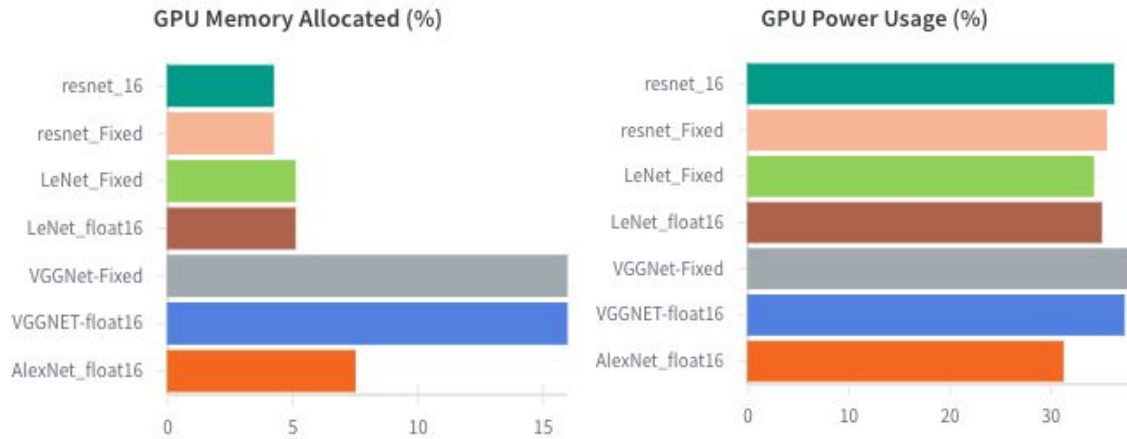


Figure 60: MNIST data model GPU usage for all half precision on Weights & Biases

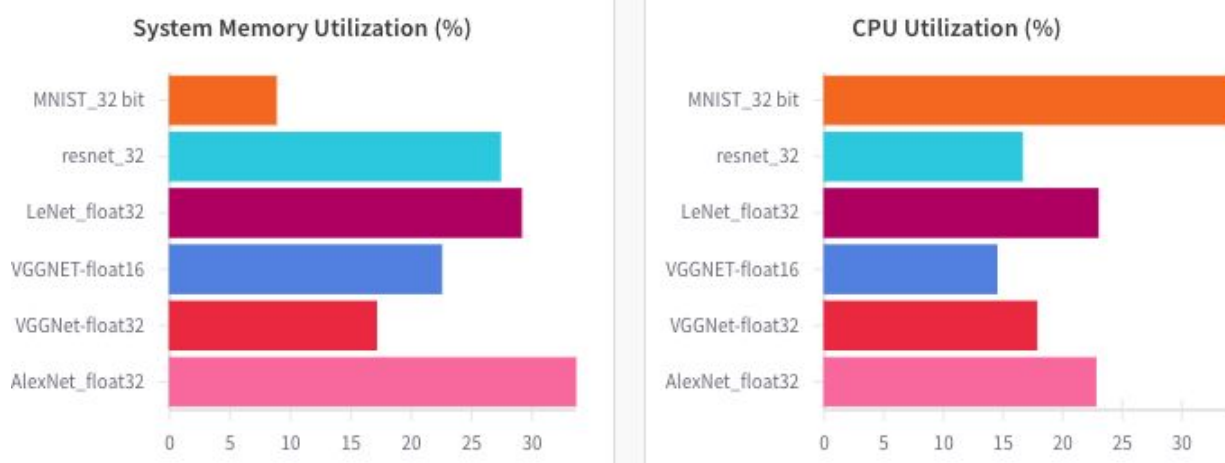


Figure 61: MNIST data model process CPU usage on Colab on weights & Biases for all single precision model

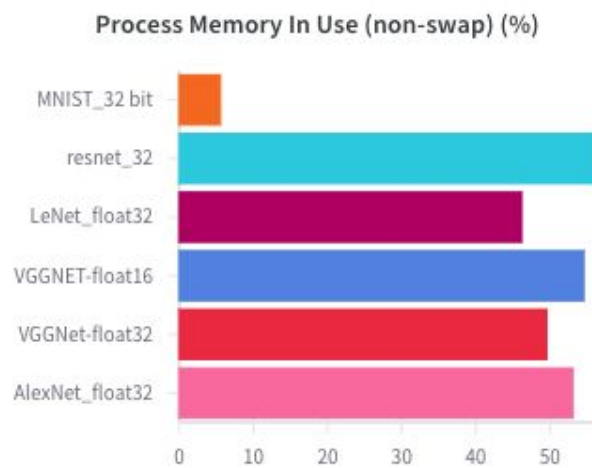


Figure 62: MNIST data model process usage on Colab on weights & Biases for all single precision model

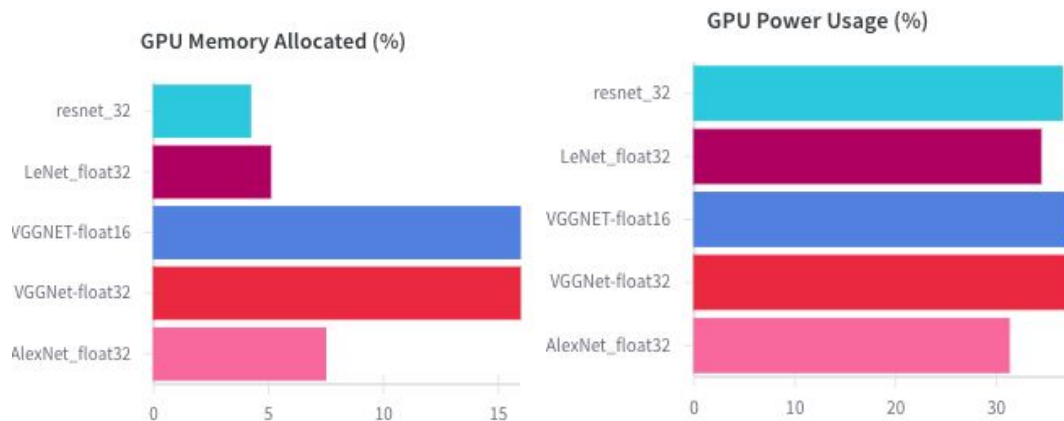


Figure 63: MNIST data model GPU usage for all single precision on Weights & Biases

7. Conclusion

This project proposed the change in the design of CNN models by using floating point number representation to improve the performance of the network. The fundamental goal was to inference CNN models to hardware friendly embedded systems. The various CNN architectures were implemented to notice the change in accuracy among each CNN type with a different numeric format. The performance of each CNN model was demonstrated on ARM processor and GPU.

In this project, a simple convolutional neural network is designed using a small scale image recognition dataset. The CNN inferences on an embedded platform for easier analysis. The CNN model accuracy and loss are obtained at the end of the training and using weights and biases library in python the power usage, CPU/GPU utilization, and memory usage of the design for each number format is visualized. This project compares the performance efficiency of the CNN model based on the use of fixed or floating point format in training and testing the model. In the table below, the change in accuracy using these formats can be seen on different CNN types.

CNN Architecture	Fixed point	32 bit floating point	16 bit floating point
Pure CNN	43.32	43.32	43.39
AlexNet	98	98.4	98.6
LeNet	99.2	99.6	99.4
VGGNet	99.38	99.2	99.2
ResNet	98.8	98.58	98.76

The table above shows the performance of each CNN type to the number format in terms of accuracy. As observed the change in format from fixed to 32-bit floating point increased the accuracy by 0.7X on average. This project also showed that by reducing the floating point precision the degradation of the performance is minimal or as observed in the table the increase in accuracy can be observed by 0.25X on average.

In this project, the CNN model was designed using the sequential model with a linear stack in layers and the training dataset considered was small scale image classification type. In the future, the use of large scale datasets and parallel network layers can be analysed for performance improvement. The training of layers can be replaced with RNN to obtain better results. Future research can be of different number formats to achieve better inference on embedded devices.

8. References

[1] Deep Convolutional Neural Network Inference with Floating-point Weights and Fixed-point Activations: <https://arxiv.org/abs/1703.03073>

[2] Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks Urs Köster , Tristan J. Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William H. Constable, Ögüz H. Elibol, Scott Gray†, Stewart Hall†, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, Naveen Rao

[3] Training Deep Neural Networks with 8-bit Floating-Point Numbers
Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen and Kailash Gopalakrishnan

IBM T. J. Watson Research Center
Yorktown Heights, NY 10598, USA
{nwang, choir, dan brand, cchen, kailash}@us.ibm.com

[4] Quantizing deep convolutional networks for efficient inference: A whitepaper
Raghuraman Krishnamoorthi raghuramank@google.com

[5] Exploration of Low Numeric Precision Deep Learning Inference Using Intel® FPGAs

Philip Colangelo Intel Corporation, San Jose, CA, USA

philip.colangelo@intel.com

Asit Mishra, Intel Corporation Hillsboro, OR USA asit.k.mishra@intel.com

Nasibeh Nasiri Intel Corporation San Jose, CA, USA nasibeh.nasiri@intel.com

Martin Margala University of Massachusetts Lowell Lowell, MA, USA

Martin_Margala@uml.edu

Eriko Nurvitadhi Intel Corporation Hillsboro, OR, USA

eriko.nurvitadhi@intel.com

Kevin Nealis Intel Corporation San Jose, CA USA kevin.nealis@intel.com

[6] Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi

[7] Intel white paper: Lower Numerical Precision Deep Learning Inference and Training Andres Rodriguez, Eden Segal, Etay Meiri, Evarist Fomenko, Young Jim Kim, Haihao Shen, and Barukh Ziv
January 2018

[8] FloatSD: A New Weight Representation and Associated Update Method for Efficient Convolutional Neural Network Training Po-Chen Lin, Mu-Kai Sun, Chuking Kung, and Tzi-Dar Chiueh, Fellow, IEEE

[9] FORMAL VERIFICATION OF FLOATING-POINT HARDWARE WITH ASSERTION-BASED VIP

Ravi Ram¹, Adam Elkins¹, Adnan Pratama¹ – Xilinx Inc.

Sasa Stamenkovic², Sven Beyer³, Sergio Marchese³ – OneSpin Solutions

¹Xilinx Inc., ravi.ram@xilinx.com

2100 Logic Drive, San Jose, CA 95124-3400, USA, +14088792763

²OneSpin Solutions, sasa.stamenkovic@onespin.com

4820 Hardwood Road, #250, San Jose, CA, 95124, USA, +14087341900
3OneSpin Solutions, {firstname.lastname}@onespin.com Nymphenburger Strasse,
20a, 80335, Munich, Germany, +4989990130

[10] A Survey of the Recent Architectures of Deep Convolutional Neural Networks

Asifullah Khan^{1, 2*}, Anabia Sohail^{1, 2}, Umme Zahoora¹, and Aqsa Saeed Qureshi¹
¹ Pattern Recognition Lab, DCIS, PIEAS, Nilore, Islamabad 45650, Pakistan.
² Deep Learning Lab, Center for Mathematical Sciences, PIEAS, Nilore, Islamabad 45650, Pakistan
asif@pieas.edu.pk

[11] O. Chapelle, “Support vector machines for image classification,” Stage deuxième année magistère d’informatique l’École Norm. Supérieure Lyon, vol. 10, no. 5, pp. 1055–1064, 1998.

[12] D. G. Lowe, “Object recognition from local scale-invariant features,” Proc. Seventh IEEE Int. Conf. Comput. Vis., pp. 1150–1157 vol.2, 1999.

[13] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, “Speeded-Up Robust Features (SURF),” Comput. Vis. Image Underst., vol. 110, no. 3, pp. 346–359, 2008.

[14] N. Dalal and W. Triggs, “Histograms of Oriented Gradients for Human Detection,” 2005 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. CVPR05, vol. 1, no. 3, pp. 886– 893, 2004.

[15] T. Ojala, M. Pietikäinen, and D. Harwood, “A comparative study of texture measures with classification based on featured distributions,” Pattern Recognit., vol. 29, no. 1, pp. 51–59, 1996.

[16] Semiconductor engineering

<https://semiengineering.com/artificial-intelligence-chips-must-get-the-floating-point-math-right/>

[17] OneSpin <https://www.onespin.com/fpu/>

[18] Cavigelli, Lukas, Gschwend, David, Mayer, Christoph, Willi, Samuel, Muheim, Beat, and Benini, Luca. Origami: A convolutional network accelerator. In Proceedings of the 25th edition on Great Lakes Symposium on VLSI, pp. 199–204. ACM, 2015.

[19] Chen, Yu-Hsin, Krishna, Tushar, Emer, Joel S, and Sze, Vivienne. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE Journal of Solid-State Circuits, 2016.

[20] Courbariaux, Matthieu and Bengio, Yoshua. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. arXiv:1602.02830, 2016.

[21] Courbariaux, Matthieu, David, Jean-Pierre, and Bengio, Yoshua. Training deep neural networks with low precision multiplications. arXiv preprint arXiv:1412.7024, 2014.

[22] Deng, Zhaoxia, Xu, Cong, Cai, Qiong, and Faraboschi, Paolo. Reduced-precision memory value approximation for deep learning. 2015.

[23] Multi-layer perceptron and CNN:
<https://mc.ai/multilayer-perceptron-mlp-vs-convolutional-neural-network-in-deep-learning/>

[24] CNN:
<https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac>

[25] CNN Medium:
<https://medium.com/nybles/a-brief-guide-to-convolutional-neural-network-cnn-642f47e88ed4>

[26] CNN Wikipedia: https://en.wikipedia.org/wiki/Convolutional_neural_network

- [27] MNIST dataset Github: <https://github.com/datapythonista/mnist>
- [28] CNN architectures Medium :
<https://medium.com/analytics-vidhya/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5>
- [29] CNN architectures:
<https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d#e4b1>
- [30] CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs
- [31] Fine-Grained Exploitation of Mixed Precision for Faster CNN Training
- [32] Rethinking Numerical Representations For Deep Neural Networks
- [33] Pre Processing Python: https://en.wikipedia.org/wiki/Data_pre-processing
- [34] Data-Augmentation:
<https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/>
- [35] Weights & Biases:
<https://app.wandb.ai/shrusti/uncategorized?workspace=user-shrusti>
- [36] ARM website: https://en.wikipedia.org/wiki/ARM_architecture#ARMv8-A
- [37] Android CPU for Arm:
<https://android.gadgethacks.com/how-to/android-basics-see-what-kind-processor-you-have-arm-arm64-x86-0168051/>

9. Appendix

9.1 Research work

This chapter will be a beginner's guide to under the project in depth and know the course of project done in steps. This project is to find the performance analysis comparison between CNN designs using fixed point and IEEE754 floating point representation and conclude which format will make the deployment of the CNN inference easier on the embedded devices or more say ARM tools. This project has intense references from previous or on-going research work and many research papers have been referred which can be found in the reference section of this report.

The image classification dataset MNIST is used to implement on various CNN architecture.

9.2 The Library and Code details

To start a simple MNIST CNN code is implemented to verify the change in accuracy from fixed to single and half precision. TensorFlow backend with Keras library is used throughout all the CNN models. The MNIST dataset model is obtained either from standard library or the CSV files are retrieved from google drive. Each dataset is labialised for easier access and the reshaping and normalisation of the training and test set is done to get results in desired datatype. The model training is done for multiple CNN architecture. The reference codes can be found at the end of this chapter.

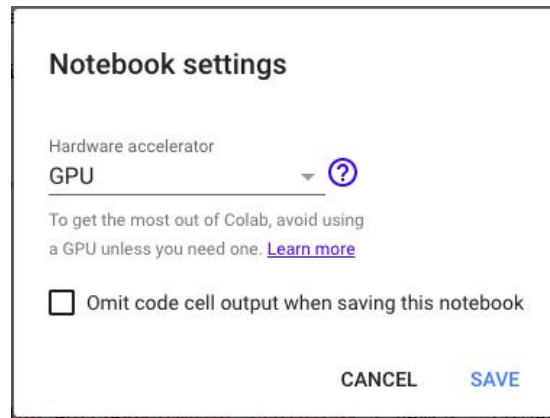
9.3 Performance Analysis Tool

The use of Wandb library is used to check the performance of the CNN model on parameters of CPU utilization, power usage, memory utilization, process thread usage and many others can be found using this library. The library calls a function after the model is trained which provides link to the website to view all the runs.

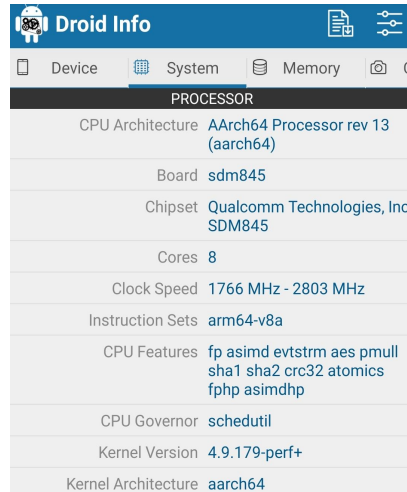
Logging results to [Weights & Biases \(Documentation\)](#).
Project page: <https://app.wandb.ai/shrusti/uncategorized>
Run page: <https://app.wandb.ai/shrusti/uncategorized/runs/25q436so>
W&B Run: <https://app.wandb.ai/shrusti/uncategorized/runs/25q436so>

9.4 Processor settings

The idea is to make CNN design inference on ARM processors easier and to check use of which number representation format makes it easier. To implement this project Google Colab with GPU runtime hardware accelerator is used during initial stage to make sure the code is running smoothly as GPU runs the model faster than CPU.



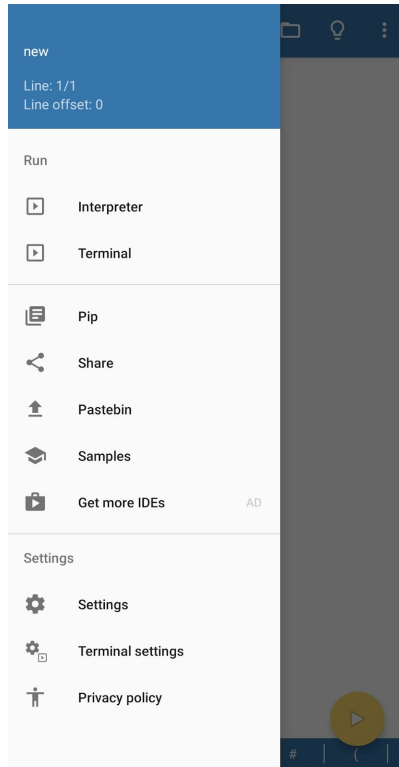
For the further use the Android CPU was used to implement the CNN model on ARM processor. The phone used in this project runs on ARM64 bit architecture type CPU with 64 bit instruction set. The CPU details of any device can be obtained using droid app.



The screenshot shows the 'Droid Info' application interface. At the top, there's a blue header with the app's name and an Android robot icon. Below the header is a navigation bar with tabs for 'Device', 'System', 'Memory', and 'Camera'. The 'System' tab is currently selected. The main content area displays a list of system specifications under the heading 'PROCESSOR'. The specifications include CPU Architecture, Board, Chipset, Cores, Clock Speed, Instruction Sets, CPU Features, CPU Governor, Kernel Version, and Kernel Architecture.

PROCESSOR	
CPU Architecture	AArch64 Processor rev 13 (aarch64)
Board	sdm845
Chipset	Qualcomm Technologies, Inc SDM845
Cores	8
Clock Speed	1766 MHz - 2803 MHz
Instruction Sets	arm64-v8a
CPU Features	fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp
CPU Governor	schedutil
Kernel Version	4.9.179-perf+
Kernel Architecture	aarch64

Python 3 was installed on this device to run the CNN code. In this app the terminal can be used to install all the libraries used and Jupyter notebook can be installed and called by using terminal to run the python editor. A run through the features of the app is will make installation and setting easier. Jupyter notebook is used to run the python code in the same usual way it is done on a PC.



9.5 The Future Scope

The work done so far uses CNN model of various architecture to run either on fixed or floating point dataset on a model. The understanding of the standard python library can be beneficial to manually use the desired weights and other parameters on each layer to obtain the change in precision with each numeric format in each layer and improve the overall performance by making changes at layer level. And a combination of fixed and floating point number format can be implemented on weights and activation function to make the deployment more easier.

The use of an ARM processor can be implemented by either running the code on ARM software development kit or any other tool to implement this CNN model on an ARM tool. ARM website have many ways to inference the CNN model on embedded device, a thorough study can guide in implementing the design on one such software or inference tool.

armDeveloper

IP PRODUCTS TOOLS AND SOFTWARE ARCHITECTURES SOLUTIONS COMMUNITY SUPPORT DOCUMENTATION DOWNLOADS

Home | IP Products | Processors | Machine Learning | Arm NN

Arm NN

Overview Processors DesignStart Graphics and Multimedia System IP Physical IP Security IP Subsystem

Arm NN

Software Developer Kit (SDK)

Arm NN is an inference engine for CPUs, GPUs and NPUs. It bridges the gap between existing NN frameworks and the underlying IP. It enables efficient translation of existing neural network frameworks, such as TensorFlow and Caffe, allowing them to run efficiently, without modification, across Arm Cortex-A CPUs, Arm Mali GPUs and Arm Ethos NPUs.

Arm NN is free of charge.

Download Arm NN SDK

participants_95....csv Show All

All the models implemented in this project are sequential for easier feed forward network flow and by implementing it in parallel way the understanding of impact of numeric representation in each layer can be evaluated and changes for optimised and improved performance can be implemented. The future scope of this project is enormous as deployment of CNN model with better performance on embedded device is under research by many companies.

9.6 Code

FOR MNIST CNN

```
!pip install keras-tqdm
#%tensorflow_version 2.x
!pip install tensorflow-probability
!pip install --upgrade tensorflow
```

```

!pip install tensorflow==1.15
!pip install tensorflow.compat.v1

import tensorflow as tf
print(tf.__version__)
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras.utils.vis_utils import plot_model
#!pip install tf-nightly

from keras_tqdm import TQDMNotebookCallback
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow.compat.v1 as tf
#tf.disable_v1_behavior()

# Loading the data
import tensorflow as tf
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

import matplotlib.pyplot as plt
%matplotlib inline
# You may select anything up to 60,000
image_index = 5908

print(y_train[image_index])
plt.imshow(x_train[image_index], cmap='Greys')

from keras import backend as K

# input image dimensions
from decimal import Decimal
import decimal as D

img_rows, img_cols = 28, 28
batch_size = 128
num_classes = 10
epochs = 15
# the data, split between train and test sets
if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols) # 1 of 60K, grayscale value, 28,28
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

```

```

x_train= x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

x_test.shape
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras.utils.vis_utils import plot_model

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential() # helps in adding layer by layer hence the use of "add"
model.add(Conv2D(32, kernel_size=(3, 3),
activation='relu',input_shape=input_shape,data_format="channels_last"))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))# avoids overfitting by dropping out few neurons for next layer
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.optimizers.Adadelta(),
metrics=['accuracy'])

model.summary()

# start train # fit is the keyword used to train
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=20,
verbose=0,validation_data=(x_test, y_test), callbacks=[TQDMNotebookCallback()])

# save model
history.history['accuracy']
# evaluate and print test accuracy
score = model.evaluate(x_test, y_test, verbose=0)
print("Accuracy on test set: ",score[1])

plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

```

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
# Install virtualenv system-wide
! pip install virtualenv
!pip install --upgrade wandb
import wandb
wandb.init(sync_tensorboard=True)
```

FOR CNN AlexNet

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import mnist
```

```
import numpy as np
```

```
from keras.models import Model
from keras.layers import Input, Dense
```

```
learning_rate = 0.001
epochs=20
batch_size=64
display_step=20
(train_images,train_labels), (test_images,test_labels) = mnist.load_data()
```

```
def load_data():
    (train_images,train_labels), (test_images,test_labels) = mnist.load_data()
    train_images = train_images.astype(np.object)
    test_images = test_images.astype(np.object)
    train_images = np.expand_dims(train_images, axis=-1)
    test_images = np.expand_dims(test_images, axis=-1)
    train_labels = to_categorical(train_labels,10)
    test_labels = to_categorical(test_labels,10)
    return train_images, train_labels, test_images, test_labels

#train_dataset = tf.data.Dataset.from_tensor_slices((train_images,
train_labels)).shuffle(buffer_size=1000000).batch(batch_size)
#test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels)).batch(batch_size)
```



```

train_dataset = tf.data.Dataset.from_tensor_slices((train_images,
train_labels)).shuffle(buffer_size=1000000).batch(batch_size)
test_dataset = tf.data.Dataset.from_tensor_slices((test_images, test_labels)).batch(batch_size)

```

```

from sklearn.preprocessing import LabelBinarizer
label_binarizer = LabelBinarizer() # to convert numerical variables to categorical variables for multi-class
classification
train_labels = label_binarizer.fit_transform(train_labels)

test_labels = label_binarizer.fit_transform(test_labels)

```

```

def AlexNet_Model():
    inputs = keras.Input(shape=(28, 28, 1))
    #first convolution layer:
    conv1 = keras.layers.Conv2D(filters=64, kernel_size=(3, 3), strides=1, padding='SAME', activation=
tf.nn.relu)(inputs)
    pool1 = keras.layers.MaxPool2D(pool_size=(2, 2), padding='SAME')(conv1)
    norm1 = tf.nn.local_response_normalization(pool1, depth_radius=4, bias= 1.0, alpha= 0.001/ 9.0, beta=0.75)
    drop1 = keras.layers.Dropout(0.8)(norm1)
    #second convolution layer:
    conv2 = keras.layers.Conv2D(filters=128, kernel_size=(3, 3), strides=1, padding='SAME', activation=
tf.nn.relu)(drop1)
    pool2 = keras.layers.MaxPool2D(pool_size=(2, 2), padding='SAME')(conv2)
    norm2 = tf.nn.local_response_normalization(pool2, depth_radius=4, bias= 1.0, alpha= 0.001/ 9.0, beta=0.75)
    drop2 = keras.layers.Dropout(0.8)(norm2)
    #third convolution layer:
    conv3 = keras.layers.Conv2D(filters=256, kernel_size=(3, 3), strides=1, padding='SAME', activation=
tf.nn.relu)(drop2)
    pool3 = keras.layers.MaxPool2D(pool_size=(2, 2), padding='SAME')(conv3)
    norm3 = tf.nn.local_response_normalization(pool3, depth_radius=4, bias= 1.0, alpha= 0.001/ 9.0, beta=0.75)
    drop3 = keras.layers.Dropout(0.8)(norm3)
    #fully connected layer:
    flat = keras.layers.Flatten()(drop3)
    dense1 = keras.layers.Dense(units=1024, activation=tf.nn.relu)(flat)
    dense2 = keras.layers.Dense(units=1024, activation=tf.nn.relu)(dense1)

    logits = keras.layers.Dense(units=10)(dense2)
    return keras.Model(inputs=inputs, outputs=logits)

```

```

model = AlexNet_Model()
model.summary()
optimizer = tf.optimizers.Adam(learning_rate=learning_rate)
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

hist= model.fit(train_images, train_labels,
               batch_size=batch_size,
               epochs=20,
               verbose=1,
               validation_data=(test_images, test_labels))
score = model.evaluate(test_images, test_labels, verbose=0)
print("Test loss:", score[0])

```

```

print('Test accuracy:', score[1])
import matplotlib.pyplot as plt
plt.plot(hist.history['accuracy'])
plt.plot(hist.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

plt.plot(hist.history['loss'])
plt.plot(hist.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

! pip install virtualenv
!pip install --upgrade wandb
import wandb
wandb.init(sync_tensorboard=True)

```