EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer science

Evaluation of Quantization Methods for Neural Networks

Master Thesis Report

This report was made in accordance with the TU/e Code of Scientific Conduct for the Master thesis

Author

Junaid Mundichipparakkal (1553771)
j.j.mundichipparakkal@student.tue.nl

Supervisors

dr. ir. W. P. A. J. (Wil) Michiels
ir. F. D. (Frits) Schalij
dr. ir. S. C. (Sibylle) Hess

March 29, 2023

Abstract

Neural networks are widely used in machine learning applications. However, their deployment in resource-constrained devices is challenging due to their demanding computational and memory requirements. Quantization is a popular method that achieves computational savings by data compression but may come with a cost of drop in accuracy. In recent years, several methods have been proposed for effective neural network quantization. This thesis evaluates and characterizes state-of-the-art quantization methods applied to image classification and object detection tasks. For both tasks, we conducted a comprehensive study of how quantization methods supported by frameworks such as *TensorFlow Lite*, *SQuant* and *QKeras* perform for different bit precisions. Model architectures *MobileNet*, *ResNet50* and *InceptionV3* were used for image classification, where as *Efficientdet-Lite0*, *SSD_MobileNet*, *CenterNet* and *RetinaNet* were used for object detection. Quantization to 16 and 8-bit precision using *TensorFlow Lite* achieved a size reduction of $\sim$ 75% and $\sim$ 85%, respectively for both the image classification and object detection models. For both the 16 and 8-bit precisions, image classification models perform at $<$ 1% change in accuracy and object detection models perform at $<$ 2% change in mean average precision(mAP). Quantization of the image classification models to 32, 16 and 8-bit precision using *QKeras* performed very poorly. i.e.,$>$ 95% change in accuracy for the models we used in this work. Quantization of the image classification models to 8, 6 and 4-bit precision using *SQuant* performed with a change in the accuracy of $<$ 1%, $<$ 5% and $<$ 25%, respectively. From the models selected for this research, all the image classification models were successfully quantized, while for object detection, only *SSD_MobileNet* was quantized without any errors. While most of the state-of-the-art quantization frameworks used in this work are supported well for image classification, the frameworks we used for object detection may require further development to support the quantization of the models we used in this work.

Acknowledgements

I would like to start by thanking my supervisor dr. ir. Wil Michiels for his excellent guidance and for making my thesis internship an informative and exciting time. Secondly, I would like to thank NXP Semiconductors for granting me this opportunity and all the people I met there for being so welcoming and always ready to help me. Special shout-out to ir. Frits Schalij for always being there to support me be it advices, ideas and even coding. I would also like to thank, dr. ir. Sibylle Hess for her help during the defense of this thesis.

Lastly, I would like to thank my family and friends for their endless support and motivation during the period of this thesis.

Contents

List of Figures

List of Tables

Acronyms

DFQ	Data Free Quantization
MPQ	Mixed Precision Quantization
NMS	Non maximum suppression
NN	Neural Network
QAT	Quantization Aware Training
SOTA	State-of-the-art
tinyML	Tiny Machine Learning
PTQ	Post Training Quantization
AP	Average Precision
mAP	Mean Average Precision
TP	True Positive
FP	False Positive
FN	False Negative
IoU	Intersection Over Union
RMSE	Root Mean Square Error
UPQ	Uniform Precision Quantization

Notations

y	Output from a layer
x	Input to a layer
w	Weights of a layer
b	Bias of a layer
a	Activation of a layer
r	Real value of a model parameter
S	Scaling factor
Z	Adjustment value for quantization
$\hat{r}$	Quantized value of r
$\tilde{r}$	Dequantized value of $\hat{r}$
α	Minimum value of set of model parameter quantized
β	Maximum value of set of model parameter quantized
n	Number of levels in the quantized range
n_{obj}	Maximum number of detections in object detection model
B	Precision
$\mathcal{D}^u$	Dequantization function for *quantized_bits*
$\mathcal{Q}^u$	Quantization function for *quantized_bits*
$\mathcal{D}^{nu}$	Dequantization function for *quantized_po2*
$\mathcal{Q}^{nu}$	Quantization function for *quantized_po2*
$\mathcal{F}$	Activation Function of a layer
$\#$	Number of
$\sim$	Approximately
W_B	B bit precision for weights of hidden layers
A_B	B bit precision for activation of hidden layers
I_B	B bit precision for weights and activation of input layers
O_B	B bit precision for weights and activation of output layers
F_N	Set of first N Layers
S	Set of suspected layers
H	Set of hidden layers
W'_B	B bit precision for weights for layers $H - S$
A'_B	B bit precision for activation for layers $H - S$

Chapter 1

Introduction

Machine learning is a potent and evolving field in computer science that is revolutionizing all fields of technology. Machine learning solutions are becoming an indispensable part of digital technology. A *neural network* (NN) is one of the popular machine learning techniques that is heavily used in use cases such as classification, object detection, face recognition and pattern recognition. In fact, this technique has potential applications in many fields of technology and is currently widely used in industrial products such as healthcare, security, automated vehicles, smartphones and home automation.

Even though NN has potential applications in numerous fields, its deployment in resource-constrained devices is challenged due to the high power and computing demands of state-of-the-art (SOTA) machine learning models. To circumvent this issue, NN models can be run on cloud servers. However, using cloud servers for some of the use cases imposes challenges like high latency on time-sensitive data, dependency on a stable internet connection, and privacy and security issues on confidential data.

To mitigate such challenges, one solution is to run NN models on edge computing resources, which brings computation and data storage closer to the data source [7], reducing the need for data transfer to servers for computation and eliminating any latency issues associated with it. Such a method also avoids data privacy and security concerns as confidential data stays locally. However, these devices typically have limited computational and memory resources.

To address these issues in edge devices, a new subfield called tinyML has emerged, that aims at making the deployment of NNs on smaller and resource-constrained embedded devices possible. More precisely, the goal in tinyML is to develop machine learning solutions that achieve a significant reduction in inference time, power consumption, and model size, without compromising the performance needed to support the use cases. Techniques applied to reduce model size are the primary focus of this thesis work.

Model size reduction can be achieved by using several techniques such as *quantization* [8], *pruning* [9], and *knowledge distillation* [10]. However, all such methods may come at the cost of accuracy. Pruning involves deleting non-significant parameters from an existing NN whereas knowledge distillation involves moving knowledge from a large NN to a smaller one. Quantization is the process of reducing the precision of model parameters of a NN to save data storage. Even though these methods are often used in combination in NNs, the focus of this work is the quantization technique in isolation.

1.1 Context

NXP Semiconductors is a semiconductor design and manufacturing company that delivers technology solutions for multiple market segments, such as automotive, health care and smart homes. Machine learning deployment on production devices is increasingly becoming a key component of these solutions developed at NXP for their customers. This thesis is sponsored by NXP under the supervision of the machine learning group, to investigate the applicability of SOTA quantization methods in a variety of real-life use case models that need to be deployed on edge device products. Research solutions proposed by academia are typically tested on research NN models, whereas production devices need the deployment of custom models trained by customers. To this extent, this work aims to support the productization of quantization methods in custom NNs that will enable efficient NN-based products and solutions.

1.2 Research Objectives

The following are the key research objectives and contributions discussed in this thesis work:

- Evaluate a set of state-of-the-art NN quantization techniques by experimenting them on a set of different state-of-the-art (SOTA) NN model architectures.
- Investigate the applicability of the considered quantization techniques for production use cases (image classification and object detection) in order to achieve size reduction while retaining the performance of the models.

1.3 Related Work

Many SOTA NNs were primarily developed and evaluated based on model fidelity without considering the computational complexity and resource demands [11]. This has led to over-parameterized NN models. i.e. more parameters than necessary. Due to resource limitations, deploying such models in edge devices may not be practical. Therefore, significant research has been conducted to reduce the model's size. For this purpose, several NNs like EfficientNet [12], SqueezeNet [13], DenseNet [14], etc were developed with lesser depth in the NN architecture. On the other hand, techniques like knowledge distillation [10], pruning [9] and quantization can also be used to reduce the models' size. These techniques can be applied to any model architecture.

As mentioned above, the focus of this work is on quantization. Quantization is a mathematical technique that is not new and finds application in many fields [15]. However, the implementation of quantization in NNs is a relatively new topic that is researched extensively as a solution for the size reduction of NNs. Several methods are available in the literature to quantize NN. In this work, we investigate the best quantization method for typical use cases of NXP customers.

The related work is continued in chapter 2 along with a detailed explanation of the concepts used in this work.

1.4 Thesis Outline

This document consists of 5 chapters. In Chapter 2, the background knowledge required for this work, such as the basics of NNs, the theory behind quantization, different machine learning frameworks for quantization, etc are explained briefly . In Chapter 3, the research and experiment plan for this work is outlined along with the chosen datasets and NN architectures. In chapter 4, the results of each experiment

are discussed along with a detailed comparison of the performance achieved. Finally, in Chapter 5, key findings from this work are summarized and the scope of future work is suggested.

Chapter 2

Background

In this chapter, the key concepts of machine learning used in this work are briefly explained. The tasks, image classification and object detection are discussed as use-cases of NN. The theory of quantization and its implementation in NN are outlined. Finally, the different quantization frameworks used in this work are also explained.

2.1 Neural Network

NNs are computing systems inspired by the biological neural networks used to perform intelligent decision-making tasks. A NN is made up of several layers of interconnected nodes, where each node is an artificial neuron that computationally mimics the behaviour of a biological neuron. As shown in Figure 2.1.1, a NN consists of three types of layers - an input layer, an output layer, and one or more hidden layers.

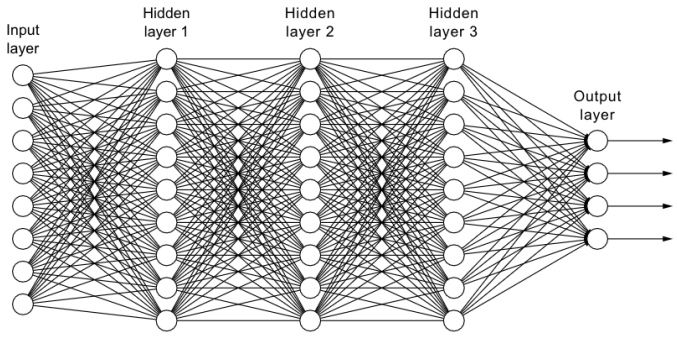


Figure 2.1.1: *Fully connected NN as printed in [1]*

Each of the layers play a key role in the functionality of the network as below:

- **Input Layer:** The input layer takes input data. For instance, in a computer vision task, each input node gets assigned one of the three channels of a pixel value, where these three channels correspond to the colors red, blue and green.
- **Output Layer:** The output layer provides the prediction computed by the network. For instance, in a classification task, there is one output node per output class.
- **Hidden Layer:** There can be a number of intermediate layers between the input and output layers. Nodes in these layers perform a computation based on the output of the preceding layer nodes they are connected to.

The layers in a NN can be of different types such as *fully connected layers*, *convolution layers*, *concatenation layers* and *pooling layers* [16].

The computation performed by a node of a fully connected layer or convolutional layer with n inputs and m outputs can be expressed as,

$$y_j = \mathcal{F}(\sum_i w_{ij}x_i + b_j). \tag{2.1.1}$$

where,

- x_i is the i^{th} input such that $i \in \{1, 2, ..., n\}$.
- y_j is the j^{th} output such that $j \in \{1, 2, ..., m\}$.
- w_{ij} is the weight associated with the input x_i and the output y_j.
- b_j is the bias associated with j^{th} output.
- $\mathcal{F}$ is an activation function.

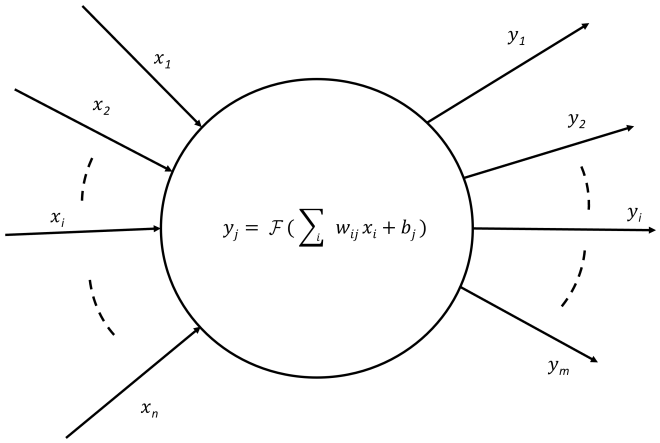


Figure 2.1.2: *Single node and parameters*

The weights(w) and bias(b) are the model parameters of a NN. Training a NN for a specific task requires finding the appropriate values for these model parameters. Some popular activation functions are ReLU, Sigmoid and Tanh [17].

2.1.1 Training

In this thesis, we focus on NNs that are trained using *supervised learning*. Supervised learning is performed using a set of labeled data. In such a set, each data sample is labeled with the correct prediction, called the true label. The labeled dataset is divided into two subsets - a *training dataset* and a *test dataset*. The training process as detailed below operates on the training dataset to derive the model parameters which is the best fit for the task at hand. The trained model is finally evaluated using the test dataset which determines the performance of the trained model.

There are two stages in the process of training NNs - *forward propagation* and *backward propagation*. Training the NN model starts with the assignment of random model parameter values (weights and biases) and the outputs y_i are calculated for every node, starting from the input layer up to the output layer. This stage is termed as *forward-propagation*. The output computed in the forward propagation pass is compared with the true label. The difference, or error, is then quantified by a *loss function*. During the next stage,

the gradient of the loss function from the previous stage is calculated with respect to the weights and biases. Based on the gradients, the weights and biases are adjusted via a backward pass, starting from the output layer to the input layer. This second stage is called *backward-propagation*.

During training, several such forward and backward propagation stages are repeated on the training dataset. After each iteration, the model parameters are updated. Training the model with the entire dataset once is called an *epoch* and training may need several such epochs to arrive at a model that minimizes the loss function.

To use a trained NN model, the architecture and model parameters need to be stored. By default, the parameters are stored as *float32* to ensure high precision on modern computers. A NN generally consists of millions of parameters and saving all the values in a high precision standard results in significantly high memory usage as well as heavy computational cost, which ultimately leads to heavy energy consumption. For instance, storing an image classification model of *ResNet* in float32 requires $\approx$ 7 GB [18] of storage. Deploying such NNs in edge devices, which typically have limited computing resources, is extremely challenging. In order to support NN-based use cases on resource constrained devices, there is a strong need for a more efficient approach to limit computational resource usage. One such popular technique to reduce the memory footprint of the NN model parameters is *quantization*, which will be detailed in section 2.4.

This thesis evaluates SOTA quantization methods for two popular use cases: Image classification and object detection. These two use cases are discussed in the next two sections.

2.2 Image Classification

In image classification, the task is to assign a class label to an input image, where the class label is selected from a predefined set of classes. An example is a cat-and-dog classifier, where it is the task to predict for an input image whether it depicts a cat or a dog. More practical use-case examples for image classification are: medical imaging, face recognition, and traffic control systems. Some of the most popular image classification model architectures are *ResNet* [19], *Inception* [20] and *MobileNet* [21]. To build an image classification NN model, the NN is trained with a set of labeled training images. These should all be of the same dimension.

The input and output of an image classification task can be defined as:

- **Input :** An image containing a single type of object to classify, with dimensions matching the input layer of the trained NN.
- **Output**: The predicted probability for each class in the predefined set of classes is obtained for the output layer nodes. The number of output nodes depends on the number of classes.

The class label of the highest probability prediction will be selected as the predicted class of the image classification model for a specific input. The performance of a classification model can be measured using the *accuracy* metric which is defined as follows:

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}. \tag{2.2.1}$$

Although we use the accuracy [22] to measure the quality of a model, we note that alternative measures exist, such as recall and precision. We refer to sections 3.1 and 3.2.1 for the datasets and architectures we consider for image classification in this work.

2.3 Object Detection

In object detection, the task is to both locate and classify objects in an image. Within an image, there can be multiple objects and these objects may or may not be from the same class. To locate an object, a rectangular bounding box is computed that as good as possible fits around the object.

Object detection model architectures are more complex compared to image classification models, as the task includes both locating (object localization) and classifying (object classification) the object. Some of the popular SOTA NN models for object detection are *RetinaNet*, *CenterNet* and *EfficientDet*. Object detection NN models are used in applications such as autonomous vehicles, object tracking, facial recognition and surveillance.

The input and the output for an object detection task can be defined as:

- **Input :** An image containing single or multiple objects to locate and classify, with dimensions matching the input layer of the trained NN.
- **Output :** A typical object detection model has two output layers, both of which are concatenation layers [12]. The nodes of one of the output layers provide the predictions related to the object localization task. The nodes of the other layer provide the predictions for the object classification task. In general, all the object detection architectures will specify the maximum number of objects that can be detected in an image before training which is denoted as (n_{obj}). Results from both layers are obtained as below:
    1. Localization layer output: An array containing the bounding boxes of each detected object.
    2. Classification Layer output: An array containing the predicted probabilities in the same order as the detected objects.

The class label of the highest probability prediction will be selected as the class for each object detected.

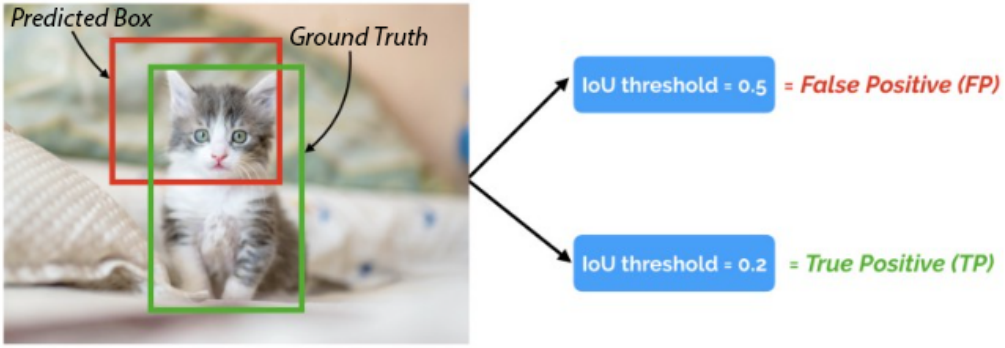


Figure 2.3.1: *Image localization with ground truth and prediction* [2]

When training an object detector, a prediction should be compared with the ground truth. For this, we need a criterion on whether a predicted bounding box is sufficiently the same as a ground-truth bounding box. This is commonly done via a metric called Intersection over Union (IoU). This metric is defined as:

$$Intersection\ over\ Union = \frac{Area\ of\ Overlap}{Area\ of\ Union}. \tag{2.3.1}$$

where *Area of Overlap* refers to the intersecting area between the *Predicted* and *Ground Truth* bounding boxes. *Area of Union* refers to the union area between the *Predicted* and *Ground Truth* bounding boxes (Figure 2.3.1). If IoU is greater than a predefined threshold value, the prediction is considered positive. Otherwise, the prediction is considered negative. A typical value for the threshold is 0.5.

The performance of an object detection model is commonly evaluated by *mean average precision* (mAP). There are two different methods in use for calculating the mAP. One is called the COCO version and the other the VOC version, referring to the data sets for which they were introduced [23]. In this work, the COCO version is used from the *object_detection.metric.coco_evaluation* library to calculate the average precision (AP) for a given class. The mAP is derived from this by taking the average AP value over all classes.

We refer to sections 3.2.2 and 3.1 for the architectures and data sets we consider for object detection in this thesis.

Non Maximum Suppression (NMS)

One of the challenges with a NN trained for object detection is that the same object may be detected multiple times. This can be clearly seen in Figure 2.3.2(a) which has multiple bounding boxes around the same object. This may result in the added effort of having to choose the most suitable detection box. To solve this, a post-processing algorithm called Non Maximum Suppression (NMS) [24] is often used. NMS deletes the duplicate localizations of the same object as shown in Figure 2.3.2(b) and retains the most suitable one.

(a) (b)

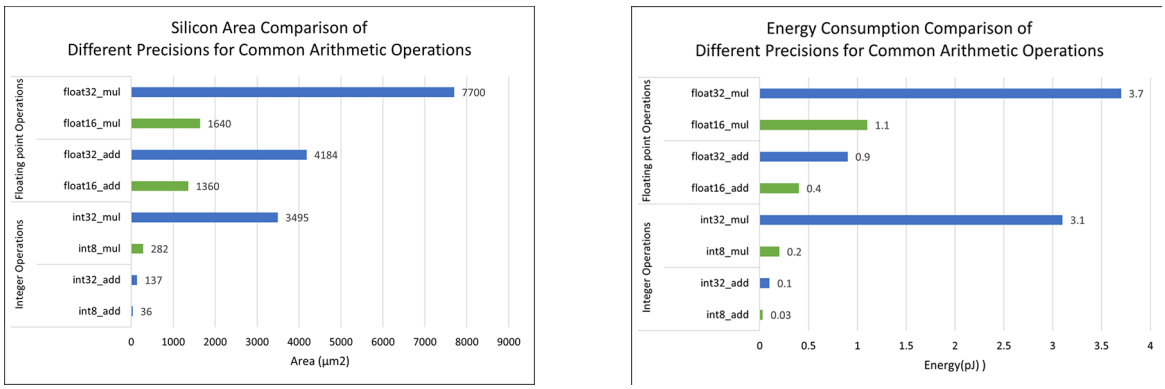
Figure 2.3.2: *(a) Before NMS (b) After NMS*

2.4 Quantization

Quantization is a mathematical operation that involves mapping the values of a large set into a smaller one. This technique is extensively used in the field of digital signal processing. With the rise of machine learning, quantization has found its way to help reduce the size of NN models. This essentially reduces the computation resource requirements, which is particularly important for edge devices. In a NN, quantization can be applied to the model parameters by reducing the precision of the values. The objective here is to apply quantization techniques without reducing the model's accuracy significantly. For instance, one can aim for quantizing the model parameters i.e., weights and biases, which are usually represented in 32 bits, to lower-precision data types, such as 16, 8 or 4 bits.

Figure 2.4.1 shows the work of Dally [3], where the energy consumption and silicon chip area requirement (for addition and multiplication operations) at different precisions are shown. It can be observed, for key computational operations needed for NN execution, that lower-precision data consumes significantly less energy and requires a smaller chip area. Quantization of a NN is therefore an efficient technique to reduce the power consumption and chip area requirements for NNs on edge devices. Additionally, reducing

the precision leads to less memory accesses and lower precision computations, which translate to reduced inference time for the quantized NN models.



(a) (b)

Figure 2.4.1: *(a) Silicon area and (b) Energy consumption for common arithmetic operations* [3]

Let R be a set of values that is quantized by the same quantization parameters and let $r \in R$. During quantization, the floating point value r is mapped into a discrete lower precision value $\hat{r}$ as seen in Figure 2.4.2.

If the discrete values in the quantized space (vertical axis in figure 2.4.2 (a)) are evenly spaced, the quantization using this type of mapping is called uniform quantization. If the discrete values in the quantized space (vertical axis in figure 2.4.2 (b)) are unevenly spaced, the quantization using this type of mapping is called non-uniform quantization.

(a) (b)

Figure 2.4.2: *(a) uniform (b) non uniform quantization*

For each of these quantization method, we explain one of the popular method with their mathematical definition below.

Uniform quantization: Quantized_bits

In [25], the authors present a uniform quantization approach, to which we refer as *quantized_bits*, that quantizes a floating point number $r \in R$ to B bit precision using the mapping ($\mathcal{Q}^u$) defined as follows:

$$\hat{r} = \mathcal{Q}^u(r, S, Z, B) = \left\lfloor \frac{clip(r, \alpha, \beta)}{S} - Z \right\rceil \quad \forall \quad r \in R \tag{2.4.1}$$

$$clip(r, \alpha, \beta) = \begin{Bmatrix} \alpha & \text{if } r < \alpha \\ r & \text{if } \alpha \leq r \leq \beta \\ \beta & \text{if } r > \beta \end{Bmatrix} \quad \forall \quad r \in R \tag{2.4.2}$$

$$S = \frac{\beta - \alpha}{n - 1} \tag{2.4.3}$$

$$Z = \frac{\alpha}{S} \tag{2.4.4}$$

where,

- R is a set of values to be quantized.
- $\hat{r}$ is the quantized version of r.
- B is a precision to which the values are quantized.
- S is a scaling factor that is used to scale down the values of R.
- Z is an adjustment value that ensures that if $r = 0$ then $\hat{r} = 0$.
- $\lfloor \bullet \rceil$ is a *rounding* operator. Generally rounds the values to the closest integer. We refer to section 2.4.1 for a detailed explanation of rounding operations.
- *clip* is the clipping function used to clip the values of R to a new range $[\alpha,\beta]$. We refer to section 2.4.2 for a detailed explanation of clipping.
- $[\alpha, \beta]$ is the clipping range.
- n is the number of levels in the quantized range and is determined as $n = 2^B$.

The parameters S, Z, B, α and β of the equation (2.4.1) are called the quantization parameters.

After quantization, a dequantization function is used to retrieve (approximate) values of R from a quantized set. Let $\hat{R}$ denote the set containing the quantized versions of the elements in R such that $\hat{r} \in \hat{R}$. The dequantization function for quantized_bits denoted by $\mathcal{D}^u$ is defined as follows:

$$\tilde{r} = \mathcal{D}^u(\hat{r}, Z, S) = (\hat{r} + Z) \cdot S \quad \forall \quad \hat{r} \in \hat{R} \tag{2.4.5}$$

where, $\tilde{r}$ is the dequantized version of $\hat{r}$.

In this case, $\tilde{R}$ is used to denote the dequantized set of $\hat{R}$ such that $\tilde{r} \in \tilde{R}$.

Non-uniform quantization: Quantized_po2

For a set of real numbers R that can be quantized to B bit precision, *quantized_po2* [25] ($\mathcal{Q}^{nu}$) can be defined as:

$$\hat{r} = \mathcal{Q}^{nu}(r, B) = clip(\ \lfloor log_2(|r|) \rceil\ ,\ 0\ ,\ 2^{B-1}\ ) \quad \forall \quad r \in R \tag{2.4.6}$$

The parameter B of the equation (2.4.6) is the quantization parameter. The input value can be retrieved (approximately) back from the quantized values $\hat{r}$ using a *dequantization* function $\mathcal{D}^n$ as follows:

$$\tilde{r} = \mathcal{D}^{nu}(\hat{r}) = \begin{cases} 0 & \text{if } \hat{r} = 0 \\ 2^{\hat{r}} & \text{otherwise} \end{cases} \tag{2.4.7}$$

After dequantization of $\hat{r}$ using either uniform and non-uniform quantization methods, we can be in one of the following cases:

- $\tilde{r} \neq r$. In this case, the difference between the dequantized and original values is known as *quantization error*. Quantization error arises mainly due to rounding and clipping errors which are introduced by the rounding operator($\lfloor \cdot \rceil$) and the clipping function (*clip*).
- $\tilde{r} = r$. This is an ideal case of quantization with no quantization errors.

Quantization in Neural Networks

As discussed, Quantization is applied to a NN to generate a smaller model by reducing the data precision of model parameters. However, models that are quantized directly by reducing bit precision may lead to a drop in accuracy. Therefore it is imperative to identify quantization techniques that do not cause a loss in performance. As there are many techniques available in the literature, *quantization simulation* is used as a technique by machine learning practitioners to evaluate the model performance against the selected quantization method. In addition, quantization simulation can also be performed on a better computationally capable system, similar to where the models are trained. Compared to an edge device where the deployment is done, a host system can make use of resources like GPU that help with faster and easier evaluation. The best-performing quantized model is finally selected for deployment.

We denote the function that simulates a quantized inference run by *quantizer function*. This function takes inputs into an implementation by taking a quantization step followed by a dequantization step as to evaluate the error introduced by the the quantization method. For instance, if we apply the quantized_bits method, the quantizer function $\mathcal{G}^u$ is defined by:

$$\tilde{r} = \mathcal{G}^u(r, S, Z, B) = D^u(\mathcal{Q}^u(r, S, Z, B), Z, S) \qquad \forall \quad r \in R \tag{2.4.8}$$

where, the parameters are the same as defined for equation (2.4.1) and (2.4.5). Each set of R values has its own quantization parameters. Precision of r and $\tilde{r}$ are usually $float32$.

In order to quantize a NN effectively, it may be needed to apply different quantization parameters to different parts of a NN. Typically, different layers require different quantization parameters [8]. In this thesis, we focus on the quantization simulation of a NN performed per-layer basis. Hence, separate quantizers may be added to the weights, the biases, and the activations (or the inputs and the outputs) of each layer. Furthermore, different types of layers may require different quantization methods. For example, the convolutional layer has multiple filters and therefore can be quantized channel-wise or layer-wise. This specific example is discussed in detail in section 2.4.3, *quantization granularity*. We also refer to section 2.4.4 to explain how some commonly used layers are quantized.

Furthermore, different types of layers may require different quantization methods. For example, the convolutional layer has multiple filters and therefore can be quantized channel-wise or layer-wise. This specific example is discussed in detail in section 2.4.3, *quantization granularity*. We also refer to section 2.4.4 to explain how some commonly used layers are quantized.

Adding quantizers to NN can affect the accuracy and inference time of the NN models. The accuracy is affected due to clipping and rounding errors introduced by the quantization and the dequantization steps in quantizers. On the other hand, the inference time of the model is impacted negatively due to extra calculations involved for quantization and dequantization steps. In addition, the quantized NN created after quantization simulation has a smaller model size compared to the non-quantized model. The effect on size is due to the quantization of the NN's weights and biases. However, quantizing the output values of activations does not affect the model's size because these values are only generated during the inference of the model.

Some of the key methods that have an impact on the performance of quantization in NN are discussed in the following subsections.

2.4.1 Rounding operation

Rounding operation generally rounds the values to the nearest integer which we refer to as *Rounding-to-nearest* technique. Rounding can cause errors which is referred to as *rounding error*. It was shown by Gupta et al. [26] that Rounding-to-nearest values do not always give the best performance. In another work by Nagel et al. [27], the authors show a new approach to round only the NN weights. They propose using the minimization of the Hessian matrix, which is part of the Taylor series approximation of the loss difference obtained by rounding weights. This is expressed as follows,

$$\mathbb{E}[\mathcal{L}(\mathrm{x}, \mathrm{y}, \mathrm{w} + \Delta\mathrm{w}) - \mathcal{L}(\mathrm{x}, \mathrm{y}, \mathrm{w})] \approx \Delta\mathrm{w}^T \cdot \mathrm{g}^{(\mathrm{w})} + \frac{1}{2}\Delta\mathrm{w}^T \cdot \mathbf{H}^{(\mathrm{w})} \cdot \Delta\mathrm{w}, \tag{2.4.9}$$

where,

- $\mathbb{E}[\cdot]$ is the expectation operator
- $\mathscr{L}(\cdot)$ is the network loss function
- w, x and y are weight, input and output variables of the layer to be quantized.
- $\mathrm{g}^{(\mathrm{w})}$ is the gradient associated with the loss function with respect to weights.
- $\mathbf{H}^{(\mathrm{w})}$ is the Hessian matrix associated with the loss function with respect to weights.

For a trained NN, the gradient term $\mathrm{g}^{(\mathrm{w})}$ will be approximately zero. Therefore, the minimization process involves minimizing the Hessian matrix $\mathbf{H}^{(\mathrm{w})}$ by suitably rounding the weights. $\mathbf{H}^{(\mathrm{w})}$ can be approximated as follows,

$$\mathbf{H} \approx \mathbf{E} + \mathbf{K} + \mathbf{C} \tag{2.4.10}$$

where, **E**, **K** and **C** are three submatrices of the Hessian matrix [28]. The relation between **E**,**K**,**C** and **H** is depicted in Figure 2.4.3 and three submatrices are minimized separately. Later in this work, this method of rounding is referred to as *Squant_hessian*.

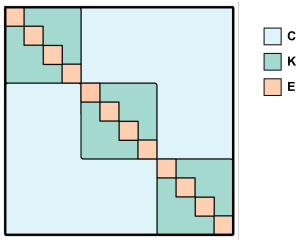


Figure 2.4.3: *Submatrices **E**,**K** and **C** in the Hessian matrix*

2.4.2 Clipping

Clip function is introduced in quantization in order to reduce the rounding error.

The clipping function will diminish the range of R ($[min(R), max(R)]$) into a new clipping range ($[\alpha, \beta]$). With this approach, the distance between α and β may take smaller values than the distance between min(R) and max(R), which reduces the rounding error. However, reducing the clipping range as per

equation (2.4.2) will assume more values in R as outliers (i.e $r > \beta$ and $r < \alpha$ where $r \in R$). This process can introduce an error in these values, which is termed as *clipping error*.

Let the distribution of values in R be as shown in Figure 2.4.4.

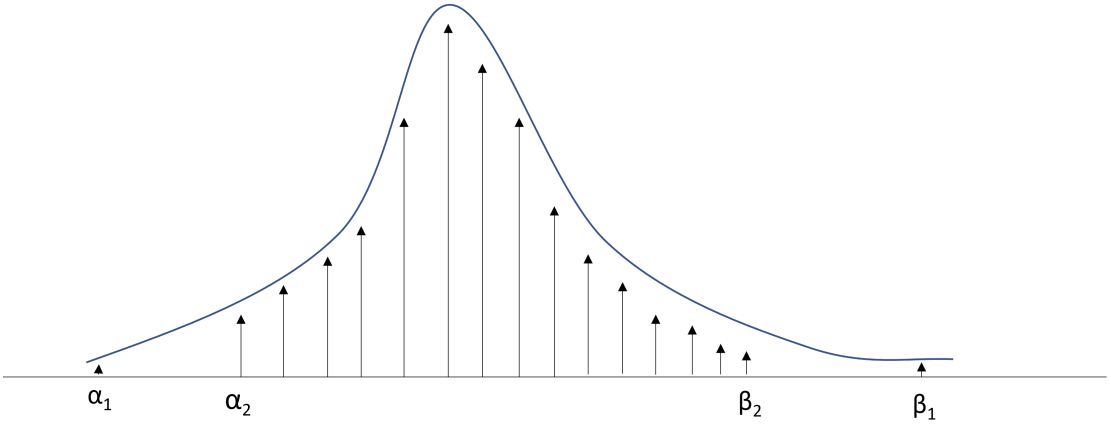


Figure 2.4.4: *Example Distribution of R*

As can be verified in the figure, α_1 and β_1 are respectively the minimum and the maximum values in R and it could be assumed that α_1 and β_1 are outliers. If $\alpha = \alpha_1$ and $\beta = \beta_1$, the clipping range is relatively large and can result in significant rounding errors. On the other hand, if $\alpha = \alpha_2$ and $\beta = \beta_2$, the range becomes narrower which will help reduce the rounding error. Even though there will be considerable clipping errors in the clipped values α_1 and β_1, the impact can be low as they are outliers, meaning that they correspond to rare scenarios.

Given below are some examples of clipping performed on different sets of R using the equation (2.4.2). R' represents the dataset clipped between the range $[\alpha, \beta]$).

Example - 1 : For $R = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, $\alpha = 3$ and $\beta = 7$. $R' = \{3, 3, 3, 4, 5, 6, 7, 7, 7, 7\}$

Example - 2 : For $R = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, $\alpha = 2$ and $\beta = 9$. $R' = \{2, 2, 3, 4, 5, 6, 7, 8, 9, 9\}$

Example - 3 : For $R = \{10, 112, 113, 114, 115, 116, 117, 118, 119, 120\}$, $\alpha = 100$ and $\beta = 120$. $R' = \{100, 112, 113, 114, 115, 116, 117, 118, 119, 120\}$

Example - 4 : For $R = \{10, 112, 113, 114, 115, 116, 117, 118, 119, 120\}$, $\alpha = min(R) = 10$ and $\beta = max(R) = 120$. $R' = \{10, 112, 113, 114, 115, 116, 117, 118, 119, 120\}$

In Examples 1-3, $R \neq R'$, as the clipping operation assigns the values outside the clipping range to α or β. The clipping error in these cases is the difference between R and R'. Whereas in Example-4, $R = R'$ and α and β are respectively min(R) and max(R).

The range of the clipped dataset in Example-4 is *110*, which is much larger than *20* in Example-3. During quantization, the rounding error will be higher for Example-4 compared to Example-3 due to its larger range. Therefore, we can conclude that reducing the clipping error can increase the rounding error and vice versa. These examples highlight the importance of carefully setting the clipping range for effective quantization.

Below, we will discuss methods to compute values α' and β' that represent lower and upper-bounds on the values to be quantized, respectively. Next, we discuss how α and β can be obtained from α' and β'. A commonly used clipping technique is *min-max*, where the clipping range α' and β' depends on $min(R)$ and $max(R)$ respectively. However, min-max clipping may result in large errors due to outliers. To solve

this, an MSE-based range setting can be used. In this method, we minimize the MSE of the quantizer outputs with respect to the quantizer input to find the optimal values of α' and β' as follows:

$$\arg\min_{\alpha',\beta'} \left\| R - \tilde{R}\left(\alpha', \beta'\right) \right\|^2 \tag{2.4.11}$$

Where, R is the set of values to be quantized and $\tilde{R}$ is output from the quantizer when provided the values of R as input. The equation (2.4.11) can be minimized using several techniques to find the appropriate values of α' and β'. We refer to the work of Nagel et. al. [29] for a detailed explanation of one such method.

The clipping range selected by the user can either be symmetric or asymmetric, which is defined below.

Symmetric and Asymmetric Quantization:

In the case of symmetric quantization, $\alpha = -\beta$ and $\beta = max(|\alpha'|, |\beta'|)$. In this case, $Z = 0$ and the range of the quantization space is $[-(2^{B-1} - 1), 2^{B-1} - 1]$. An example of such a mapping is shown in Figure 2.4.5(a). During symmetric clipping, as the clipping range depends on $|\alpha'|$ or $|\beta'|$, the difference between these two values determines the rounding error. As this difference increases, the rounding error increases. Hence, symmetric quantization is often used only when the values of R are balanced, i.e, when $|\alpha'| - |\beta'|$ is small. Symmetric quantization is widely used to quantize the weights and biases of a NN, which is proven to significantly reduce the computation cost compared to asymmetric quantization with limited loss in accuracy [30].

In the case of asymmetric quantization, $\alpha = \alpha'$ and $\beta = \beta'$. This method uses the full range of quantization space $[-2^{B-1}, 2^{B-1} - 1]$. An example of such a mapping is shown in Figure 2.4.5(b). During asymmetric clipping, the clipping range is guaranteed to be small compared to symmetric quantization and hence it is possible to achieve lower rounding error. Asymmetric quantization is widely used in practice to quantize the outputs of activation functions [31].

(a) (b)

Figure 2.4.5: *Quantization based on range of clipping function (a) Symmetric (b) Asymmetric*

2.4.3 Quantization granularity

The weights and biases of a convolutional layer can be quantized layer-wise or channel-wise. For example, Let R be the set of weights of a convolutional layer to be quantized. Let R_i be defined as the weights of the i^{th} filter of R. Let us assume that the convolutional layer has only three filters, i.e, $R = R_1 \cup R_2 \cup R_3$.

In layer-wise quantization, all the weights of the entire layer are clipped to the same range, as seen in figure 2.4.6. In this example, the clipping range of R is determined by the filter R_2, which has a larger clipping range than R_1 and R_3. This increases the rounding errors for values in R_1 and R_3 during quantization. Due to this, layer-wise quantization may lead to sub-optimal accuracy.

In channel-wise quantization, each filter's weights have separate clipping ranges and are quantized separately, as seen in figure 2.4.6. This figure shows that the clipping range of R_1, R_2 and R_3 are different

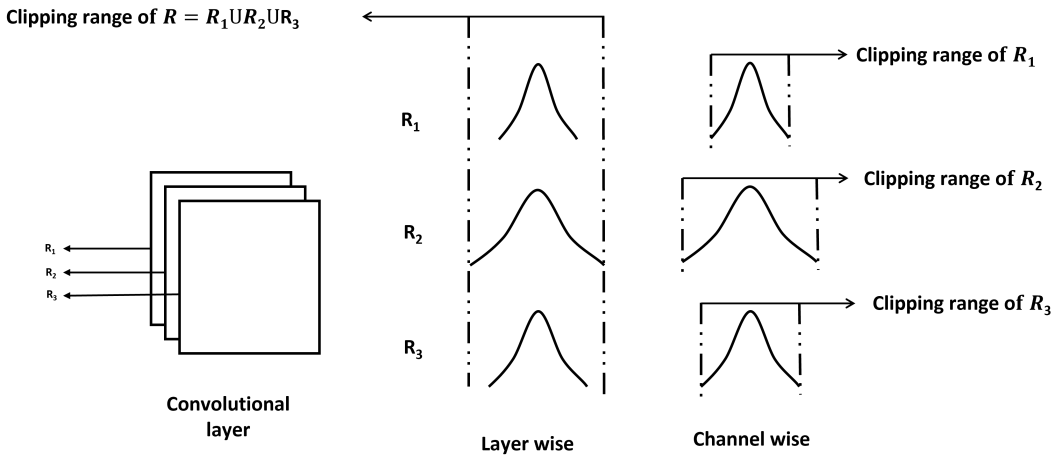


Figure 2.4.6: *Quantization granularity*

and fit each filter's distributions unlike layer-wise quantization. Due to this, channel-wise quantization can improve the quantized model accuracy [31]. Therefore, channel-wise quantization is typically used as the standard method for quantizing the weights of convolutional layers [8]. However, a drawback of channel-wise quantization is that storing multiple quantization parameters for each filter is required.

Even though the above examples showcase only the case of weights of the convolutional layers, the same behaviour can be expected for the biases of convolutional layers.

2.4.4 Quantization of different NN layers

Quantization of NN layers can be performed in different ways as introduced in the above sections 2.4.2-2.4.3. Table 2.4.1 shows a summary of some of these methods used for quantizing some commonly used layers in NN.

Table 2.4.1: *Methods for quantization of different layers*

Type of layers	Channel-wise	Layer-wise	Symmetric	Asymmetric
ADD		✓		✓
AVERAGE_POOL_2D		✓		✓
CONCATENATION		✓		✓
CONV_2D	✓		✓	
DEPTHWISE_CONV_2D	✓		✓	
FULLY_CONNECTED		✓	✓	
LOGISTIC		✓		✓
MAX_POOL_2D		✓		✓
MUL		✓		✓
SOFTMAX		✓		✓
RELU		✓		✓
SUM		✓		✓
SUB		✓		✓
MAXIMUM		✓		✓

Quantization in a NN can be implemented using several techniques and some of the relevant methods are discussed below.

2.5 Types of NN Quantization

Quantization of a NN may result in a performance loss of the model (accuracy or mAP). This performance loss can be reduced by adjusting the model parameters by retraining the model or tuning the quantization parameters without the retraining model. The former method is called *quantization aware training* (QAT) [11,25] and the latter is called *post training quantization* (PTQ) [11,28]. During quantization, the different layers in a NN may be quantized to the same or different precision, which are called *uniform precision quantization* (UPQ) [15] and *mixed precision quantization* (MPQ) [42] respectively.

We discuss both QAT and PTQ based on uniform quantization in the subsequent text below. Note that these methods can be easily adapted to non-uniform quantization.

2.5.1 Quantization aware training

QAT involves retraining the NN with quantizers to update the NN model parameters such that the drop in accuracy is reduced while quantizing the model. Figure 2.5.1 shows the typical pipeline for QAT. Each block in the pipeline is explained below.

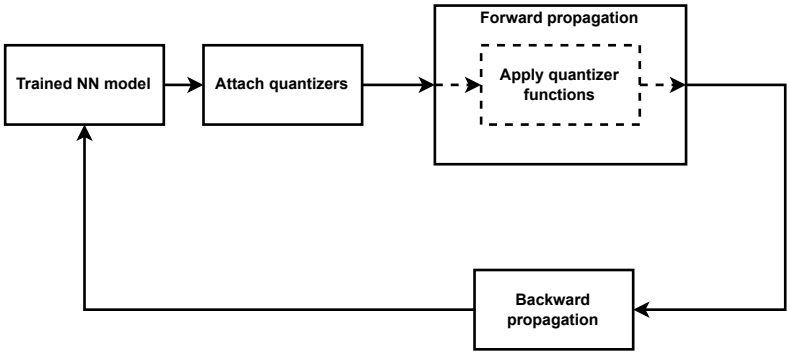


Figure 2.5.1: *Quantization aware training pipeline*

Trained NN model: This represents the trained model that needs to be quantized where the models have the same precision as when it was trained. A typical precision value for such a trained model is *float32*.

Attach quantizers: Separate quantizers are attached to weights, biases, and activations of each layer of a NN, to apply quantization simulation.

Forward propagation: The forward propagation is carried out on the trained neural network by applying the attached quantizer functions. During this forward pass the quantization parameters are calculated for every quantizer.

Backward propagation: During the backward pass, the weights and biases of the models are updated similar to the normal training of a NN.

QAT retraining involves several epochs with forward and backward passes using the method mentioned above. The performance of these models depends on how many epochs the model is retrained. However, a drawback of QAT method is that it requires labeled data for effective quantization.

While saving the quantized model, the weights and the biases are quantized and saved along with the quantization parameters of the quantizers used in the last epoch. On the other hand, for activation quantizers, only the quantization parameters are saved. However, the quantization parameters of the activation quantizers vary with every batch of input as the clipping range of the activations is dependent on the input. In order to find a suitable clipping range for the activation quantizers, we collect the clipping ranges of all the batches and based on a criterion, the clipping range of the activation quantizers is calculated. We refer to section 2.7 for an explanation of this criterion.

2.5.2 Post training quantization

The quantization method applied to an already trained NN model without retraining is called Post Training Quantization (PTQ). Unlike QAT, PTQ does not require labeled data for quantization. However, this may lead to poor model performance. Therefore, when labeled data is unavailable, PTQ may use representative data to quantize the NN model. Representative data is a sample subset of the training or testing dataset that is not labeled. Figure 2.5.2 shows the typical pipeline for PTQ with the representative dataset. The *Trained NN model* and *Attach quantizers* block in the pipeline are the same as that in the QAT method mentioned in section 2.5.1. Each remaining block in the pipeline is explained separately below.

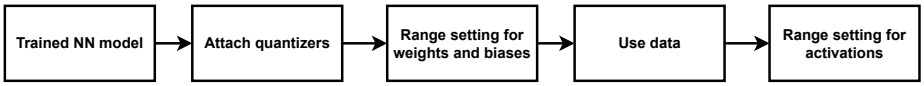


Figure 2.5.2: *Post training quantization*

Range setting for weights and biases: The range of weights and biases (α and β) are calculated, using which the remaining quantization parameters of weights and biases can be calculated and stored. These stored parameters are then used by the respective quantizers of weights and biases.

Use data: The model is then inferred using the representative dataset to find the quantization parameters of the activation quantizers.

Range estimation of activation quantizers: Similar to the case of QAT, the clipping range of every batch of input is collected and the final range is estimated using a criterion. We refer to 2.7 for a detailed explanation of this criterion.

PTQ can also be done without the use of any data and the technique is called *data free quantization* (DFQ) which is discussed in the following section.

Data free quantization

The method works similarly to PTQ except that in DFQ, the range estimation is done only with the model parameters. The lack of data, especially during range estimation, could lead to severe degradation in the model's performance (accuracy or mAP). Nevertheless, the method is attractive as it could circumvent the data security and privacy issues associated with PTQ. In order to overcome the degradation of accuracy, active research is currently being conducted [28, 32, 33]. Techniques such as cross-layer range equalization and bias corrections are popular techniques used in DFQ.

Cross-layer equalization is used to equalize the range of all layers of the model as closely as possible. This could cause big differences in the output range of the activation values and therefore bias absorption is used to avoid this. For a detailed explanation of these methods, refer to the work of Nagel et. al. [32].

2.5.3 Uniform precision quantization

In uniform precision quantization (UPQ), all the layers of the NN model are quantized to the same precision. UPQ can be beneficial in edge devices where a set of executing commands is usually of the same bit precision. Even though this approach is quicker to implement, UPQ may lead to a significant degradation in accuracy in the cases of low precision quantizations [34].

2.5.4 Mixed precision quantization

Contrary to UPQ, mixed precision quantization is more fine-grained, where each layer of the model may use different bit precision. This may lead to better performance for the quantized model. However, for this technique, the search space for choosing the optimum combination of layer precision is exponential to the number of layers. In order to address this huge search space, several methods have been proposed in the literature. For instance, Wang et al. [35] have proposed a method using reinforcement learning, Wu et al [34] used mixed precision in *Nueral Architecture Search* and Jiaxing et al. [36] devised a bayesian based method. All these methods are however computationally expensive.

To reduce the computational cost of selecting the precision of each layer, the *TensorFlowLite.experimental* library provides layer statistics data [37]. This statistical data can be used to identify the layers (*suspected layers*) whose quantization can significantly degrade the model's performance. Hence quantizing the values of all the layers except for the suspected layers will lead to a quantized model with less degradation in accuracy.

The ratio between scale factor and *root mean squared error* (RMSE) for every layer is calculated to identify the suspected layers. The RMSE of a layer is the root mean square error of the corresponding quantized layer outputs with respect to the non-quantized layer outputs. The scale factor is the quantization parameter used to quantize the model. If the ratio of a particular layer exceeds a user-defined threshold value, the layer will be considered a suspected layer.

2.6 Machine Learning Frameworks for Quantization

ML frameworks are sets of tools that are used for developing ML solutions like creating NN models for tasks such as image classification and object detection. Even though there are several ML frameworks in the industry, in this work, we use *TensorFlow* and *Keras* as the customers of NXP use *TensorFlow* and *Keras* frameworks for NN-based ML solutions.

TensorFlow Lite is a tool developed within the *TensorFlow* framework for easy and efficient deployment of *TensorFlow* models into edge devices. The models developed with *TensorFlow Lite* can be deployed on several platforms such as Android, iOS, Linux, and micro-controllers (using C library). Due to this versatility, the *TensorFlow Lite* framework is widely used in the industry. This tool also provides the option for quantizing the NN models. However, it only provides support for quantization up to 8-bit precision. In order to quantize NN models to sub-8-bit precision, we choose the $QKeras$ framework. *QKeras* framework is an extension to the *Keras* framework which supports the quantization of layers to sub-8-bit precision. $QKeras$ framework is used in this work because it is extensively researched in NXP Semiconductors.

However, both the frameworks (*TensorFlow Lite* and *QKeras*) mentioned above require labeled data or unlabeled data for the quantization of NN. In practical scenarios, this data might not be available due to privacy and security reasons which leads one to use DFQ techniques. The *SQuant* framework implements the SOTA DFQ method and claims to show the best performance among other DFQ methods in literature [28]. Even though it is implemented in *PyTorch*, the concepts used in this method can be implemented in *TensorFlow Lite* or *QKeras*.

The quantized models of the *TensorFlow Lite* framework can be quantized and saved, while for *SQuant* and *QKeras*, this is not yet possible to the best of our knowledge as they are relatively new frameworks. However, *QKeras* provides tooling to estimate the model's size and is soon expected to have support for saving and deployment.

2.7 Quantization of NN Models in Frameworks

Frameworks that offer quantization functionality are typically configurable in how they apply quantization. Options that are offered may include the different types of quantization methods (Uniform or Non-uniform), selection of clipping range (Min-max or MSE-based range setting / Symmetric or Asymmetric) and rounding operation methods (Rounding-to-nearest or Squant-hessian).

As mentioned in the previous section, the frameworks used in this work are *TensorFlow Lite*, *QKeras* and *SQuant*. Each of these frameworks may offer different quantization configurations and types of quantization (PTQ, QAT, or DFQ). This will be discussed in the remainder of this section. Types of quantization and quantizer configuration of all these frameworks are explained in the following subsections.

2.7.1 TensorFlow Lite

TensorFlow Lite supports both PTQ and QAT and can quantize to 16 or 8-bit precision. This framework provides support to save the quantized model which in turn helps with the measurement of the model size. PTQ and QAT methods in this framework work based on the pipelines explained respectively in section 2.5.2 and 2.5.1.

The quantizers of both methods are configured as:

- Rounding Operation: *Rounding-to-nearest*
- Quantization method: *Quantized_bits.*
- Clipping: Min-max
    1. Weights and Biases: Symmetric
    2. Activations: Asymmetric

The difference in both methods is based on the criteria of how the clipping range of activation quantizers is estimated, which is explained below.

PTQ

As mentioned in section 2.5.2, the α and the β values are collected for every batch of inputs from the representative data. The minimum and and maximum of these collected values are considered to be clipping range for activation quantizers.

QAT

As mentioned in section 2.5.1, the α and the β values are collected for every batch of inputs of a training epoch. These collected values are then aggregated using exponential averages to calculate the clipping range of activation quantizers. We refer to the work of Jacob et. al. [11] for a detailed explanation of how the exponential average is calculated and the clipping range is defined.

2.7.2 QKeras

QKeras only provides QAT and it is implemented similarly as in *TensorFlow Lite*. Additionally, *QKeras* supports quantization to sub-8 precisions and provides different choices of mapping functions. In this work, we use the *quantized_bits* and *quantized_po2* mapping functions of this framework.

2.7.3 SQuant

SQuant uses the DFQ method for quantization. Except for a few differences, *SQuant* follows the pipeline explained in section 2.5.2 for PTQ.

The PTQ in *SQuant* differs mainly in the fact that the trained models used for quantization are first applied with cross-layer equalization and bias absorption techniques. The resultant model is then passed into the PTQ pipeline. Since there is no data in DFQ, the blocks including 'use data' and beyond, (in the pipeline shown in Figure 2.5.2) are not applicable for DFQ.

This framework adds quantizers only to the weights and the activations of each layer. It must be noted that, at the time of writing this report, the method does not enable the quantization of biases.

The quantizers of this framework are configured as follows:

- Rounding Operation:
  1. Weights: *Squant-hessian*
  2. Activations: *Rounding-to-nearest*

     This is because using *Squant-hessian* for the activation quantizers will result in a large penalty on inference time as activation outputs are only available during model inference.
- Quantization method: *Quantized_bits.*
- Clipping:
  1. Weights: Symmetric and MSE-based range setting.
  2. Activations: Symmetric and BN method.

     BN method is based on batch normalization where the batch normalization parameters are used to find the clipping range. We refer to the work of Nagel et. al. [32] for a detailed explanation of this method.

Chapter 3

Research Plan

In this section, the datasets and the NN model architectures used for the experiments in this work are explained briefly. Finally, the experiment setup, the experiment plan, and the machine setup used in this work are provided.

3.1 Datasets

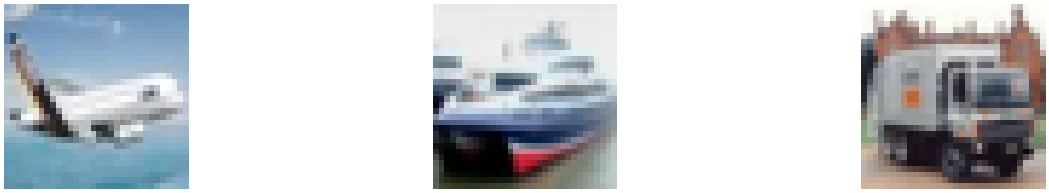
In this work, the datasets used are *Cifar10* [4], *Oxford iiit pet* [5] and *Pascal-VOC* [6]. A brief explanation of these datasets is given in the following subsections, with a high-level overview provided in Table 3.1.1.

Table 3.1.1: *Basic information of datasets used in this work*

Dataset	**# classes**	**# training**	**# testing**
Oxford iiit pet (Oxford)	37	3673	3662
Cifar10	10	50000	10000
Pascal-VOC	20	15672	16720

3.1.1 Cifar10

The *Cifar10* dataset contains small ($32 \times 32 \times 3$) color images of 10 classes and is commonly used to explain machine learning concepts to beginners or for conducting research experiments. Figure 3.1.3 shows a few example images from this dataset with their respective class names. In this work, this dataset is used only for image classification tasks.



(a) *Airplane* (b) *Ship* (c) *Truck*

Figure 3.1.1: *Cifar10 dataset example images* [4]

3.1.2 Oxford iiit pet dataset

Compared to *Cifar10*, this dataset consists of images that are large and diverse with large varieties of classes (37). However, each class contains only very few images. Such a dataset is very typical for a real-life machine learning task. Some of the images in this dataset are shown in Figure 3.1.2. Hereafter, in this work, this dataset will be referred to as *Oxford* and this dataset is used only for image classification tasks.

(a) *Chihuahua*

(b) *Bengal*

(c) *Abyssinian*

Figure 3.1.2: *Oxford dataset example images* [5]

3.1.3 Pascal-VOC

The *Pascal-VOC* dataset is a very widely used dataset for object detection tasks and has 20 classes. It has relatively large images with a significantly larger collection compared to most of the datasets used for object detection training. Each image in the dataset has a *xml* file associated with it containing annotations, class labels, bounding boxes, etc. However, in this work the *xml* annotations files couldn't be used directly in the already existing training *api's*, as some of the annotations were missing namely, *truncated*, *difficulty* and *pose*. Therefore, these annotation features were given appropriate values before they were used in this work.

This dataset consists of two subsets namely, *VOC2007* and *VOC2012*. In this work *VOC2012* is used for training and *VOC2007* is used for testing.

(a)

(b)

(c)

Figure 3.1.3: *Pascal-VOC dataset example images* [6]

3.2 Model Architectures

In this work, SOTA model architectures are used to perform the tasks of image classification and object detection. Table 3.2.1 gives a summary of the architectures considered in this work which are discussed in detail in the following subsections.

* in Table 3.2.1 denotes these values were not able to be retrieved due to the file format the models were saved in.

Table 3.2.1: *Basic information of NN architectures used in this work*

Task	Architecture	# layers	# parameter (Millions)	Training image size
Image classification	*MobileNet*	90	20.6	128×128 × 3
	ResNet50	179	57.8	128×128 × 3
	InceptionV3	315	30.8	128×128 × 3
Object detection	EfficientDet-Lite0	237	3.2	224×224 × 3
	SSD_MobileNet	*	*	320×320 × 3
	CenterNet	*	*	512×512 × 3
	RetinaNet	*	*	512×512 × 3

3.2.1 Image classification

The models used for image classification are from the libraries *Keras.application* [38] for *TensorFlow* models and *Torchvision.models* for *PyTorch* [39] models. All the models are downloaded with the pre-trained weights of the *ImageNet* dataset and retrained with *Cifar10* and *Oxford* dataset. The models are however retrained after a slight modification of adding two extra dense layers before the output layer. These layers were added to solve some errors we encountered during the training of multiple model architectures. Since the goal of our work is to study quantization for arbitrary neural networks, we believe this addition to SOTA architectures does not restrict the applicability of our results.

Every model is trained for 50 epochs with varying batch sizes depending on the availability of memory resources. The performance of the image classification tasks is measured using the metric *accuracy*.

MobileNet

MobileNet architecture was particularly developed to be used in edge devices [21] and hence this model architecture is considered. For image classification tasks, version 1 of *MobileNet* is used.

ResNet50

ResNet is a popular SOTA architecture used for several real-world applications and this model mainly prioritizes the accuracy of the model [19]. It is therefore worth investigating the performance of quantization on this model. However, ResNet is a family of architectures. They vary in the number of layers from which they consist. To limit the resources we need for training and quantization, we performed our experiments on the version with 50 layers. This architecture is called *ResNet50*.

InceptionV3

Inception is a SOTA architecture that was developed to reduce inference time without compromising model performance. In this work we focus on *InceptionV3*.

3.2.2 Object detection

The models used for object detection are from the *TensorFlow 2 Detection Model Zoo* and trained using the *TensorFlow 2 Detection API* [40].

All the models used in this work are downloaded with the pre-trained weights of the *COCO* dataset and retrained with *Pascal-VOC*. Each model is trained for 100,000 steps with a batch size of 8. The performance of the object detection tasks is measured using mAP.

EfficientDet

This architecture was chosen due to its availability in multiple versions of the same at different depths. This gives the option to compare the quantization of *EfficientDet* models at different depths. The backbone of this architecture is *EfficientNet* [12]. This work uses the lite version of *EfficientDet* known as *EfficientDet-Lite0*.

SSD_MobileNet

This model architecture uses the approach of *Single Shot MutliBox Detector* [41] with the backbone architecture *MobileNet* (Version 2).

CenterNet

This model architecture is based on the *Feature Pyramid Network* [42] with the backbone *MobileNet* [43].

RetinaNet50

The model architecture is based on both *Single Shot MutliBox Detector* [41] and *Feature Pyramid Network* [42] with the backbone *ResNet50*.

3.3 Experiment Setup

This work uses the quantization frameworks *TensorFlow Lite*, *QKeras* and *SQuant*. This section lists the details about the models that are used by each framework for quantization experiments.

3.3.1 TensorFlow Lite

In this work, the *TensorFlow Lite* framework is used to evaluate the quantization of both image classification and object detection tasks, and the models used for conducting experiments in this framework are trained in the *TensorFlow* framework.

For the image classification task, *InceptionV3*, *ResNet50* and *MobileNet* architectures are trained with both the *Oxford* and the *Cifar10* datasets.

For the object detection task, *EfficientDet-Lite0*, *SSD_MobileNet*, *CenterNet* and *RetinaNet* architectures are trained with the *Pascal VOC* dataset.

3.3.2 QKeras

In this work, *QKeras* is used only for image classification. The models that can be quantized using this method should be developed using *QKeras* layers. Since this is a relatively new framework, a model zoo is not available. The reconstruction of model architecture with *QKeras* layers is very time-consuming. Hence, the *TensorFlow* models created for the *TensorFlow Lite* experiments containing the *Keras* layers are converted into *QKeras* layers for experiments.

3.3.3 SQuant

In this work *SQuant* is used only for image classification task and the input models for this framework have to be trained in the *PyTorch* framework.

For the image classification task, *InceptionV3*, *ResNet50* and *MobileNet* architectures are trained with both the *Oxford* and the *Cifar10* datasets.

3.4 Experiment Plan

This research considers the practical implementation of quantization for neural network deployments and is therefore mostly experiment-driven. Research objectives are outlined below for both image classification and object detection tasks separately.

- **Image classification**

  Objective - 1 : Investigate 16-bit precision quantization for both QAT and PTQ using *TensorFlow Lite*.

  Objective - 2 : Investigate 8-bit precision quantization for both QAT and PTQ using *TensorFlow Lite*.

  Objective - 3 : Investigate different bit precision (8,16 and 32) quantizations for QAT using *QKeras* for the mapping functions, *quantized_bits* and *quantized_po2*.

  Objective - 4 : Investigate 8 and sub-8 bit precisions (6 and 4) quantizations for DFQ using *SQuant*

- **Object detection**

  Objective - 5 : Investigate 16 and 8-bit precision quantizations for both PTQ using *TensorFlow Lite*.

Several experiments are conducted for every objective and their respective goals, results and observations are discussed in chapter 4.

3.5 Machine Setup

The hardware and software specifications for the experiments conducted in this work are as follow:

Hardware:

- CPU : Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz (8 Cores)
- GPU : NVIDIA GeForce RTX 3080 (10 GB)
- RAM : 62GB

Software:

- CUDA tool kit versions*

  PyTorch: 11.1

  TensorFlow: 11.2

*Two *CUDA Tool Kit* versions are used due to the incompatibility of the toolkit versions with *TensorFlow object detection API 2* and *SQuant*.

Chapter 4

Results

In this chapter, the details of the experiments conducted to investigate the performance of different quantization techniques in different frameworks are explained. To this end, image classification and object detection tasks are considered. For these tasks, NN models with different architectures are trained with different datasets. Quantization simulation is performed to evaluate the performance of each model which is then compared to the original non-quantized models.

There are five objectives addressed in this chapter: four for image classification and one for object detection. Each objective is explained in the following subsections with respective experiments. The results of every experiment are then discussed.

Annotations Used:

In the tabulated results, notations are used to represent the different layers in a NN. As shown in Figure 4.0.1, the NN can be considered to be made up of the input (I), the output (O) and the hidden layers (H). The set of the first N layers is denoted by FN. The suspected layers are labeled as S. The quantization precision (B) of the weights (W) and the activation (A) of the hidden layers(H) are denoted as W_B and A_B respectively. On the other hand, when $S \neq \emptyset$, the quantization precision of the weights and the activations are respectively denoted as W'_B and A'_B, where $W'_B = W_B - S$ and $A'_B = A_B - S$. The quantization precision of the input (I) and the output (O) layers are respectively denoted by I_B and O_B.

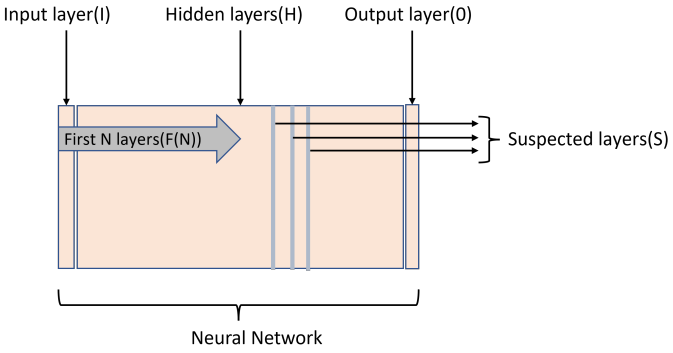


Figure 4.0.1: *Notations used for explaining the experiment results.*

In the next two sections, we discuss our results for the image classification and the object detection use cases, respectively.

4.1 Image Classification

Objectives 1-4 in section 3.4 relate to the evaluation of quantization in the image classification tasks. In the subsections below, we discuss the experiments related to these objectives.

4.1.1 Objective - 1

Investigate the quantization of NN models to 16-bit precision using the *TensorFlow Lite* for both PTQ and QAT.

Experiment 1

Goal: Evaluate the accuracy of the *TensorFlow Lite* models for QAT with 16-bit precision (i.e. $W_{16}A_{16}I_{32}O_{32}$) and compare them with the accuracy of the non-quantized model (i.e. $W_{32}A_{32}I_{32}O_{32}$).

Result: After quantization, it is observed that even though the quantization precision for the NN layers is set as $W_{16}A_{16}I_{32}O_{32}$, the obtained quantized model uses the precision $W_8A_8I_{32}O_{32}$. To the best of our knowledge, the option of 16-bit QAT is not available in *TensorFlow Lite*.

The next experiment is to evaluate 16-bit quantization using the PTQ method.

Experiment 2

Goal: Evaluate the accuracy of the *TensorFlow Lite* models for PTQ with 16-bit precision (i.e. $W_{16}A_{16}I_{32}0_{32}$) and compare them with the accuracy of the non-quantized model (i.e. $W_{32}A_{32}I_{32}0_{32}$).

Result: Table 4.1.1 shows the results of experiment 2 where the accuracy and the size of the quantized and the non-quantized models are tabulated.

Table 4.1.1: *Accuracy and model size of PTQ at 16-bit precision.*

Dataset	**Models**	$\mathbf{W_{32}A_{32}I_{32}0_{32}}$		$\mathbf{W_{16}A_{16}I_{32}0_{32}}$	
		Accuracy (%)	**Size(MB)**	**Accuracy (%)**	**Size (MB)**
Oxford	ResNet50	41.94	443	41.94	111
	MobileNet	73.51	159	73.45	40
	InceptionV3	60.18	238	60.21	59
Cifar10	ResNet50	89.96	443	89.95	111
	MobileNet	93.63	159	93.66	40
	InceptionV3	92.81	238	92.82	59

Using the values of Table 4.1.1, the percentage change in accuracy and model compression are calculated and plotted in Figure 4.1.1.

From Figure 4.1.1, two main observations can be made: (1) the change in accuracy is $< 1\%$ for all the NN models evaluated. (2) the model compression achieved is the same for all models, viz. around 74%. It is interesting to note here that even though the model is quantized from 32-bit to 16-bit, the model compression is not 50 %. This may be because the non-quantized NN models are saved in *H5* format and may contain more information than the model architecture and the model parameters. However, in the case of quantized models, the models are saved in *tflite* format, which is ready for deployment.

Objective - 1 : Result Summary

From experiments 1 and 2, it is found that QAT at 16-bit precision is not supported by *TensorFlow Lite* while PTQ at 16-bit precision performs effectively (i.e. $< 1\%$ change in accuracy).

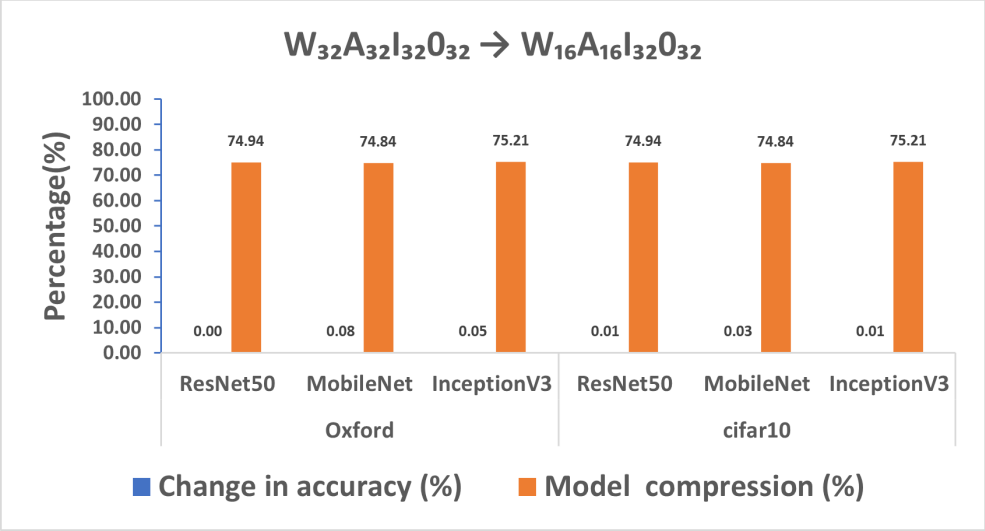


Figure 4.1.1: *Percentage change in accuracy and model compression for PTQ at 16-bit precision.*

4.1.2 Objective - 2

Investigate the quantization of NN models to 8-bit precision using the *TensorFlow Lite* for both PTQ and QAT.

Experiment 3

Goal: Evaluate the accuracy of the *TensorFlow Lite* models for QAT with 8-bit precision (i.e. $W_8A_8I_8O_8$) and compare them with the accuracy of the non-quantized model (i.e. $W_{32}A_{32}I_{32}O_{32}$).

Result: Table 4.1.2 shows the result of experiment 3 where the accuracy and the size of the quantized and the non-quantized models are tabulated.

Table 4.1.2: *Accuracy and model size of QAT at 8-bit precision.(i.e.$W_8A_8I_8O_8$).*

Dataset	**Models**	**$W_{32}A_{32}I_{32}O_{32}$**		**$W_8A_8I_8O_8$**	
		Accuracy (%)	**Size (MB)**	**Accuracy (%)**	**Size (MB)**
Oxford	ResNet50	48.41	443	2.40	56
	MobileNet	73.92	159	2.81	20
	InceptionV3	41.69	238	2.62	30
cifar10	ResNet50	90.71	443	10.12	56
	MobileNet	93.73	159	10.00	20
	InceptionV3	89.01	238	10.00	30

Using the values of Table 4.1.2, the percentage change in accuracy and model compression are calculated and plotted in Figure 4.1.2.

From Figure 4.1.2, two main observations can be made: (1) the change in accuracy is very high ($> 85\%$) and may not be acceptable for quantization tasks. (2) the model compression achieved is the same for all models, viz. around 87%.

To investigate this high change in accuracy for 8-bit quantization, further experiments are undertaken as follows.

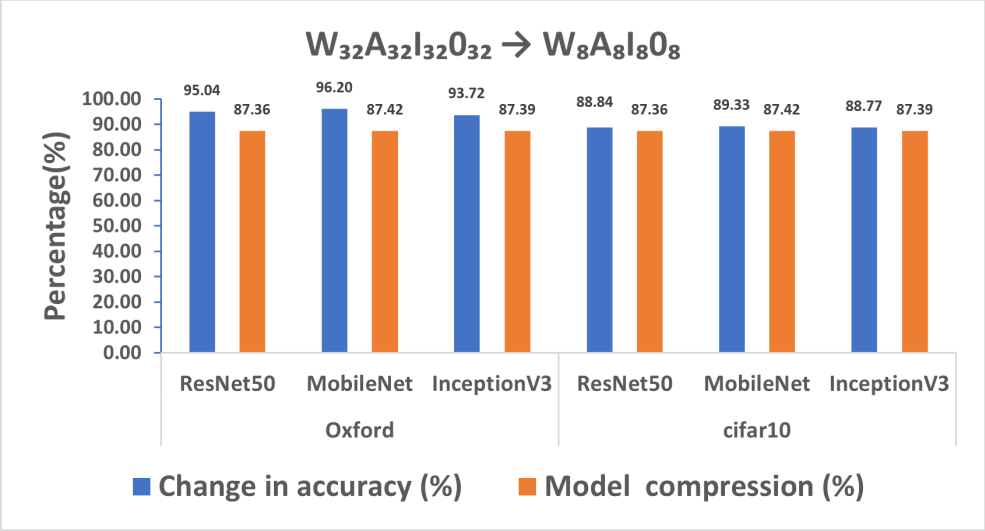


Figure 4.1.2: *Percentage change in accuracy and model compression for QAT at 8-bit precision.*

Experiment 4

Goal: Evaluate the accuracy of the *TensorFlow Lite* models for QAT with 8 or 32-bit precision for the input and the output layers while quantizing the weights and activations of the other layers to 8-bits (i.e. $W_8A_8I_{32}O_{32}$, $W_8A_8I_{32}O_8$ and $W_8A_8I_8O_{32}$).

Result: Table 4.1.3 shows the result of experiment 4, where the accuracy of the quantized models is tabulated. In this experiment, the models have the same size as in experiment 3. Therefore the model sizes are not provided in the table.

Table 4.1.3: *Accuracy of QAT at 8-bit precision with different precision for inputs and output layers.*

Dataset	**Models**	**Accuracy (%)**		
		$\mathbf{W_8A_8I_{32}O_{32}}$	$\mathbf{W_8A_8I_{32}O_8}$	$\mathbf{W_8A_8I_8O_{32}}$
Oxford	ResNet50	48.49	48.22	2.40
	MobileNet	74.03	73.92	2.81
	InceptionV3	41.80	41.83	2.62
cifar10	ResNet50	90.92	90.69	10.12
	MobileNet	93.78	93.88	10.00
	InceptionV3	89.00	89.06	10.00

Using the values of Tables 4.1.2 and 4.1.3, Figure 4.1.3 is plotted which shows the percentage change in accuracy at different precisions.

From Figure 4.1.3, it is observed that the change in accuracy is high ($> 85\%$) in cases where the input layer has 8-bit precision while the change in accuracy is $< 1\%$ where the input layer has 32-bit precision.

From experiments 3 and 4, it can be concluded that for the models used in this work, QAT is effective for 8-bit precision when the input layer is kept at 32-bit precision.

The next experiment is to evaluate 8-bit quantization using the PTQ method.

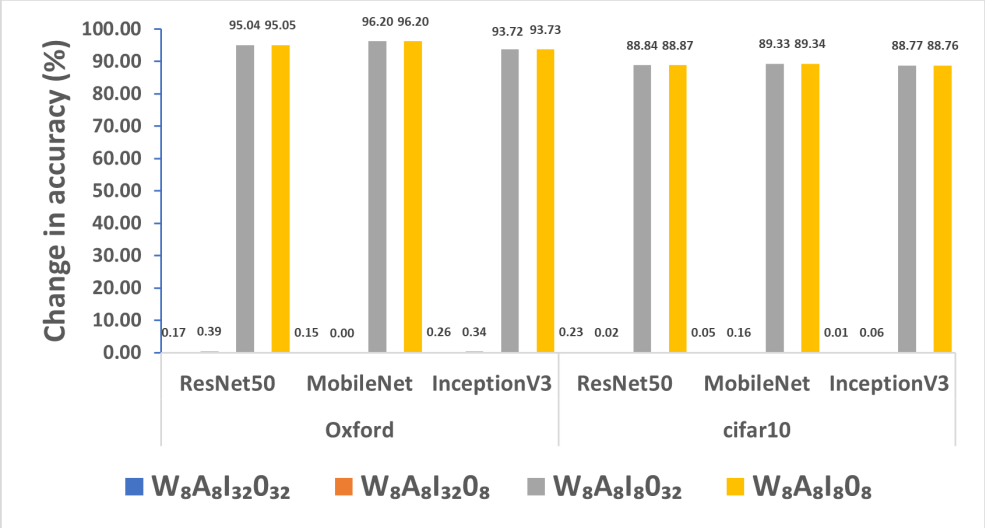


Figure 4.1.3: *Percentage change in accuracy QAT at 8-bit precision.*

Experiment 5

Goal: Evaluate the accuracy of the *TensorFlow Lite* models for PTQ with 8 or 32-bit precision for input and output layers while quantizing the weights and the activations of the other layers to 8-bits (i.e. $W_8A_8I_8O_8$, $W_8A_8I_{32}O_{32}$, $W_8A_8I_{32}O_8$ and $W_8A_8I_8O_{32}$). The accuracy of these quantized models is then compared with the accuracy of the non-quantized model ($W_{32}A_{32}I_{32}O_{32}$).

Results: Table 4.1.4 shows the results of experiment 5 where the accuracy of the quantized and the non-quantized models are tabulated. In this experiment, models have the same size as in experiment 4. Therefore the model sizes are not provided in the table.

Table 4.1.4: *Accuracy of PTQ at 8-bit precision with different precision for inputs and output layers.*

Dataset	**Models**	**Accuracy (%)**				
		$\mathbf{W_{32}A_{32}I_{32}0_{32}}$	$\mathbf{W_8A_8I_{32}0_{32}}$	$\mathbf{W_8A_8I_{32}0_8}$	$\mathbf{W_8A_8I_80_{32}}$	$\mathbf{W_8A_8I_80_8}$
Oxford	ResNet50	41.94	41.94	35.66	2.73	2.73
	MobileNet	73.51	73.51	72.99	2.73	2.73
	InceptionV3	60.18	60.21	59.55	2.73	2.73
cifar10	ResNet50	89.96	89.96	26.00	10.00	10.00
	MobileNet	93.63	93.65	93.10	9.58	9.58
	InceptionV3	92.81	92.83	29.4	10.00	10.00

Using the values of Table 4.1.4, Figure 4.1.4 is plotted which shows the percentage change in accuracy at different precisions.

Figures 4.1.3 and 4.1.4 show similar observations for the change in accuracy in the cases $W_8A_8I_{32}0_{32}$, $W_8A_8I_80_8$ and $W_8A_8I_80_{32}$. However, the case of $W_8A_8I_{32}0_8$ shows different values for different model architectures. These observations show that quantizing NN using PTQ can be done effectively, provided that we do not quantize the input and the output layers. The following experiment investigates why the accuracy of models has an enormous impact on quantizing the input layer.

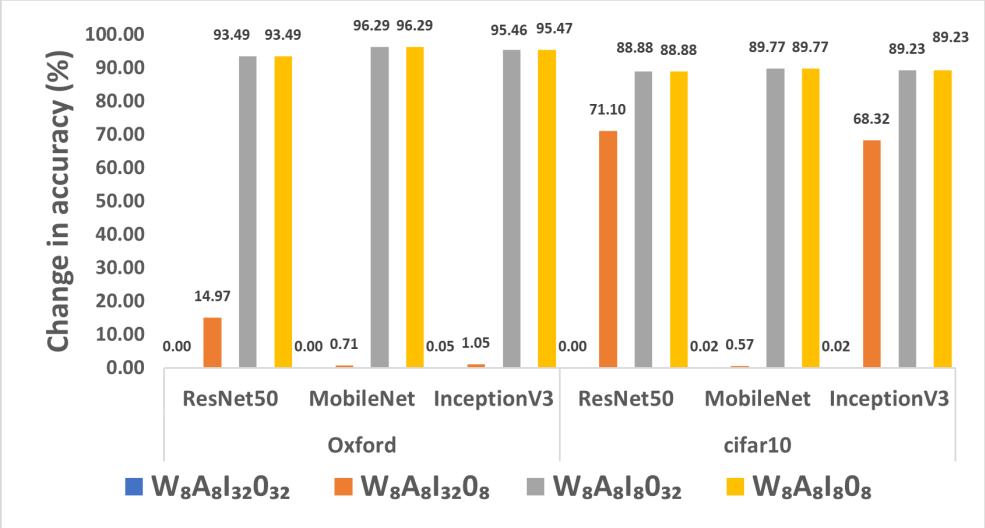


Figure 4.1.4: *Percentage change in accuracy PTQ at 8-bit precision.*

Experiment 6

Goal: Investigate the drop in accuracy due to the quantization of the NN models consisting of an 8-bit precision input layer.

Experiment setup: A NN model with a single dense layer is created with the biases as zeros and the weights as an identity matrix. This NN model is expected to return the same output value as the input value. This model is then quantized to 8-bits.

Result: Table 4.1.5 shows the range used for the input and the output of the model before and after quantization.

Table 4.1.5: *Input and output to single layer model.*

	Output	
Input	**Original Model** (before)	**Quantized Model** (after)
[103,150]	[103,150]	[0,255]

As expected, the model gives the exact values of the input as the output before quantization. However, the output values of the NN are very different after quantization to 8 bits. The result of the output values of the quantized models is particularly surprising as the images in the considered datasets consisted of 8-bit values with a range 0-255. This discrepancy in the output values of the quantized model is due to the min-max clipping in *TensorFlow Lite*, where the original minimum and maximum of the input are shifted (i.e. to 0 and 255 respectively), causing deviations in the predicted outputs.

Objective - 2 : Result Summary

From experiments 3 - 6, two observations can be highlighted: (1) the quantization to 8-bit precision is effective using both PTQ and QAT, wherein for QAT, we quantize all the layers except the input layers, and for PTQ, we quantize all the layers expect the input and the output layers. (2) the size of the quantized models for both PTQ and QAT is approximately the same.

4.1.3 Objective - 3

While in objectives 1 and 2 we quantize using the *TensorFlow Lite*, in this objective, we investigate the quantization of NN models to different bit-precisions using the *QKeras* for QAT.

Experiment 7

Goal: Evaluate the accuracy of the *QKeras* models with 8, 16 and 32 bit precisions (i.e. $W_8A_8I_80_8$, $W_{16}A_{16}I_{16}0_{16}$ and $W_{32}A_{32}I_{32}0_{32}$) using the mapping functions *quantized_bits* and *quantized_po2*, and compare them with the accuracy of the non-quantized model (i.e. $W_{32}A_{32}I_{32}0_{32}$).

Result: Table 4.1.6 shows the accuracy of the models obtained from this experiment. This table only contains the *ResNet50* model architecture because the architectures *MobileNet* and *InceptionV3* give errors during the conversion of the models from *Keras* layers to *QKeras* layers. As mentioned before in section 3.3.2, for the *QKeras* framework, the NN models have to be built using layers from the *QKeras* framework.

Table 4.1.6: *Accuracy of QKeras models for oxford dataset.*

	Accuracy(%)				
	Non-quantized	**Quantized**			
		Quantized_bits			**Quantized_po2**
Model	$W_{32}A_{32}I_{32}0_{32}$	$W_8A_8I_80_8$	$W_{16}A_{16}I_{16}0_{16}$	$W_{32}A_{32}I_{32}0_{32}$	$W_8A_8I_80_8$
ResNet50	48.19	2.81	2.67	2.62	2.64

Using the values of Table 4.1.6, Figure 4.1.5 is plotted which shows the percentage change in accuracy at different precisions.

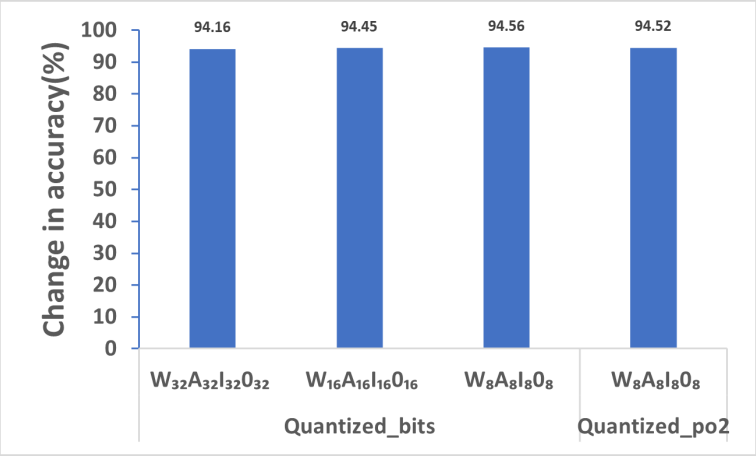


Figure 4.1.5: *Percentage change in the accuracy of ResNet50 at different precisions quantized using quantized_bits and quantized_po2 mapping functions.*

From Figure 4.1.5 it can be observed that the change in accuracy is very high ($> 85\%$). This high change in accuracy might be due to the conversion of *Keras* layers to *QKeras* layers. This needs to be investigated in the future because the work by Coelho et al. [25] shows better results for the quantization of NN models trained using the *QKeras* layers.

Objective - 3: Result Summary

It can be observed that the accuracy drop is high ($> 85\%$) when using *QKeras* for quantization of the NN models converted from *Keras* layers into *QKeras* layers.

4.1.4 Objective - 4

While in the previous experiments, we studied *TensorFlow Lite* and *QKeras*, in this objective we investigate the quantization of NN models to different bit precisions using the *SQuant* for DFQ.

Experiment 8

Goal: Evaluate the accuracy of the *SQuant* models with 8, 6 and 4-bit precisions, where the input layer is at 8-bit precision. (i.e. $W_8A_8I_8O_8$, $W_6A_6I_8O_6$ and $W_4A_4I_8O_4$). These accuracies are then compared with the accuracy of the non-quantized model (i.e. $W_{32}A_{32}I_{32}O_{32}$).

Result: Table 4.1.7 shows the result of experiment 8 where the accuracy of the quantized and the non-quantized models are tabulated.

Table 4.1.7: *Accuracy of DFQ at 8-bit precision expect for input layer at 8 bit.*

Dataset	**Models**	**Accuracy (%)**			
		$\mathbf{W_{32}A_{32}I_{32}O_{32}}$	$\mathbf{W_8A_8I_8O_8}$	$\mathbf{W_6A_6I_8O_6}$	$\mathbf{W_4A_4I_8O_4}$
Oxford	ResNet50	97.23	67.60	65.49	50.04
	MobileNet	96.01	95.85	95.10	87.59
	InceptionV3	95.57	95.61	89.37	70.04
Cifar10	ResNet50	90.90	52.81	50.81	14.74
	MobileNet	81.59	80.83	79.73	55.81
	InceptionV3	90.85	90.77	88.42	60.13

Using the values of Table 4.1.7, Figure 4.1.6 is plotted which shows the percentage change in accuracy at different precisions.

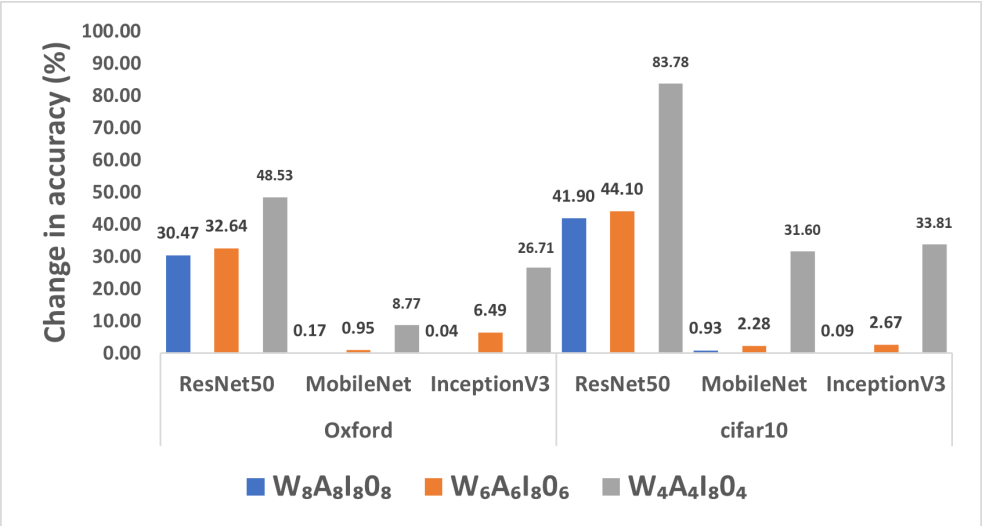


Figure 4.1.6: *Percentage change in accuracy PTQ at 8-bit precision.*

From Figure 4.1.6, it is observed that for 8-bit quantization, the percentage change in accuracy is $< 1\%$ except for *ResNet50*. For 6 and 4 bit quantization, the change in accuracy is suboptimal($< 50\%$) expect for *ResNet50* at 4 bit precision where the change in accuracy is severe.

To investigate this suboptimal change in accuracy during quantization at different precisions used in this experiments, further experiments are undertaken as follows.

Experiment 9

Goal: Evaluate the accuracy of the *SQuant* models with 8, 6 and 4-bit precisions where the input layer is at 32-bit precision. (i.e. $W_8A_8I_{32}O_8$, $W_6A_6I_{32}O_6$ and $W_4A_4I_{32}O_4$). These accuracies are then compared with the accuracy of the non-quantized model (i.e. $W_{32}A_{32}I_{32}O_{32}$).

Result: Table 4.1.8 shows the result of experiment 9 where the accuracy of the quantized and the non-quantized models are tabulated.

Table 4.1.8: *Accuracy of DFQ at 8-bit precision expect for input layer precision at 32 bit.*

Dataset	**Models**	**Accuracy (%)**			
		$\mathbf{W_{32}A_{32}I_{32}O_{32}}$	$\mathbf{W_8A_8I_{32}O_8}$	$\mathbf{W_6A_6I_{32}O_6}$	$\mathbf{W_4A_4I_{32}O_4}$
Oxford	ResNet50	97.23	97.17	96.67	96.10
	MobileNet	96.01	95.91	95.52	92.15
	InceptionV3	95.57	95.49	91.61	82.78
Cifar10	ResNet50	90.90	90.87	90.77	87.02
	MobileNet	81.59	81.21	79.76	61.85
	InceptionV3	90.85	90.60	89.45	80.06

Using the values of Table 4.1.8, Figure 4.1.7 is plotted which shows the change in accuracy at different precisions.

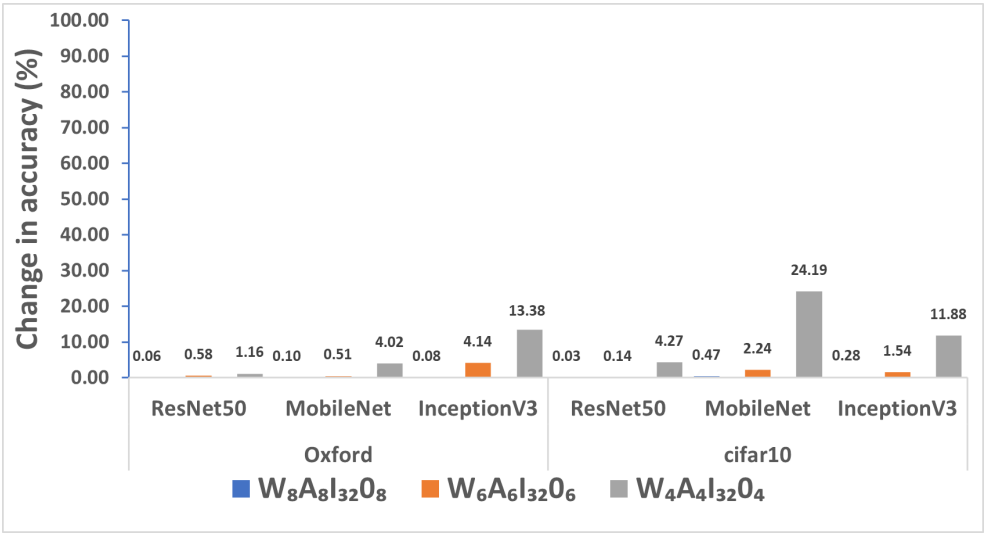


Figure 4.1.7: *Percentage change in accuracy PTQ at 8-bit precision.*

From Figure 4.1.7, it is observed that the percentage change in accuracy for all the quantized models with precision $W_8A_8I_{32}O_8$ is $< 1\%$. On the other hand, the models with precisions $W_6A_6I_{32}O_6$ and $W_4A_4I_{32}O_4$ show change in accuracy of up to almost $< 5\%$ and $< 25\%$, respectively.

Objective - 4 : Result Summary

From experiments 8 and 9, it can be observed that even though the *SQuant* uses DFQ, the method is effective, provided the input layer is left unquantized. Furthermore, the change in accuracy of the quantized

model with the precision $W_8A_8I_8O_8$, is $< 50\%$ for the *SQuant* whereas it is $> 87\%$ for the *TensorFlow Lite*.

Image Classification- Result summary

From the results obtained from all the experiments of image classification, the following conclusions can be made for the models used in this work.

For 16-bit quantization, PTQ using *TensorFlow Lite* was the best-performing choice among the ones investigated. However, the input and the output layers of these 16-bit precision models are kept at 32-bit precision.

Furthermore, for 8-bit quantization, both QAT and PTQ of the *TensorFlow Lite* showed a similar change in accuracy($< 1\%$). Additionally, the change in accuracy for the 8-bit quantization using DFQ of the *SQuant* was observed to be similar to PTQ and QAT of the *TensorFlow Lite*. It is to be noted that for these models, the input layer was kept at 32-bit precision during the 8-bit quantization, except for PTQ where the output layer is also kept at 32 bit precision.

Finally, for 6 and 4-bit quantizations using DFQ in the the *SQuant*, the change in accuracy was found to be $< 5\%$ and $< 25\%$ respectively. For these quantizations, similar to 8-bit quantization, the input layer is kept at 32-bit precision.

4.2 Object Detection

Objective 5 in section 3.4 relates to the object detection task. In the subsections below, we discuss the experiments related to this objective. In this objective, the model is evaluated using the metric mAP, unlike image classification where the metric accuracy is used to evaluate the models.

4.2.1 Objective - 5

Investigate the quantization of NN to 16 and 8 bit precisions using the *TensorFlow Lite* models for PTQ.

Experiment 10

Goal: Evaluate the mAP of the *TensorFlow Lite* models with 16 and 8-bit precisions and compare them with the non-quantized model of the *EfficientDet-Lite0* architecture.

Result: The quantization of *EfficientDet-Lite0* failed due to the errors from *tflite.converter()* library. Further investigation confirmed that *tflite.converter()* does not support *EfficientDet-Lite0*. However, using *tflite_model_maker()* library, *EfficientDet* models, quantized with QAT, can be obtained directly. But this method doesn't give a non-quantized model for comparison.

Further research on the quantization of object detection models showed that it might be possible to quantize the model architectures from *TensorFlow GitHub* [40] repository. The model architectures *SSD_MobiletNet*, *CenterNet*, and *RetinaNet* are chosen from this repository. In the following experiment, an attempt is made to quantize these models using the *TensorFlow Lite*.

Experiment 11

Goal: Investigate whether the *TensorFlow Lite* support 16 and 8-bit quantizations of *CenterNet*, *RetinaNet* and *SSD-MobiletNet*model architectures.

Result: Only *SSD_MobiletNet* was successfully quantized to 16 and 8-bit precisions. After several failed attempts to fix the errors for other model architectures, it was decided to proceed with experiments only on *SSD_MobiletNet*, due to time constraints.

SSD_MobiletNet models are trained using *TensorFlow 2 Detection API* which does not support QAT.

The following experiment evaluates the mAP of the *SSD_MobiletNet* at different precisions of quantization.

Experiment 12

Goal: Evaluate the mAP of the *TensorFlow Lite* models with 16 and 8 bit precisions and compare them with the non-quantized model for the *SSD_MobiletNet* architecture.

Result: Table 4.2.1 shows the results of experiment 12 where the mAP and the size of the quantized and the non-quantized models are tabulated.

Table 4.2.1: *mAP and size of PTQ at 8 and 16 bit precisions for Pascal-Voc.*

Precision configuration	mAP	Size(MB)
$W_{32}A_{32}I_{32}O_{32}$	53.71	25
$W_{16}A_{16}I_{32}O_{32}$	52.88	5.7
$W_8A_8I_{8/32}O_{8/32}$	0.09	3.7

Using the values of Table 4.2.1, Figure 4.2.1 is plotted, which shows the percentage change in accuracy and model compression at different precisions.

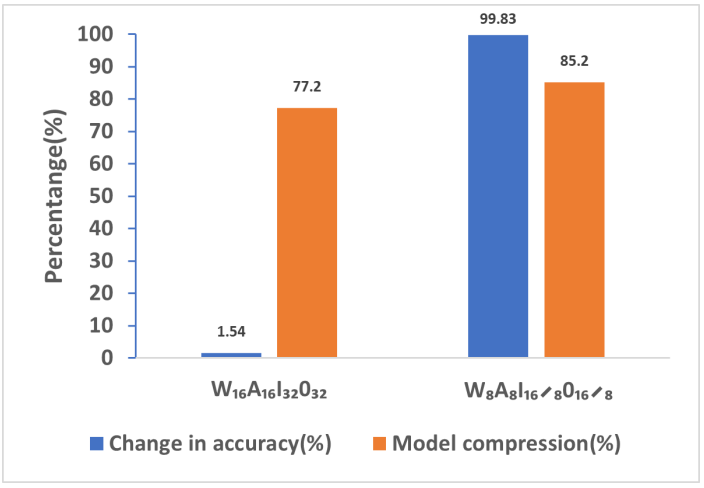


Figure 4.2.1: *Percentage change in accuracy and model compression for SSD_MobileNet at different precisions for Pascal-Voc.*

From Figure 4.2.1, two main observations can be made:(1) the change in accuracy is $< 2\%$ for 16-bit quantization and the change in accuracy is very high ($> 95\%$) for 8-bit quantization. (2) the model compression for 16 and 8-bit quantization are approximately 77% and 85%.

Further experiments are undertaken to investigate the high change in accuracy for 8-bit quantization.

Experiment 13

Goal: Evaluate the mAP of the *TensorFlow Lite* models quantized to 8-bit precision while considering the suspected layers (explained in subsection 2.5.4) and compare them with the non-quantized model of

the *SSD_MobiletNet* architecture.

Result:

The threshold of the suspected layers (S) (explained in subsection 2.5.4) used in this experiment is 0.7. Let the number of layers in this set be denoted as $|S|$.

Let FN be the set consisting of the first N layers. We then define a set $\mathcal{NQ}$ such that $\mathcal{NQ} = S \cup FN$, This will be the set of layers that we do not quantize. FN is incorporated in $\mathcal{NQ}$ as it is recommended to add some first layers to the set of suspected layers.

The threshold value of 0.7 results in $|S| = 1$, i.e, a single suspected layer. However, to study the effect of the quantization of layers in S, we quantize this layer (in which case $|S| = 0$) or leave them unquantized ($|S| = 1$). In this experiment, the Table 4.2.2 shows the results of this experiment with different values of FN and S. It is to be noted that W'_B and A'_B are the sets of W_B and A_B without the weights and the activations of the layers in NQ.

Table 4.2.2: *mAP of PTQ at 8-bit precision with suspected layers as NQ.*

Serial Number	**Precision configuration**	**NQ**		**mAP**	**change in mAP**
		|FN|	**|S|**		
1	$W_8A_8I_{32}O_{32}$	0	0	5.10	90.44
2	$W'_8A'_8I_{32}O_{32}$	1	0	51.65	3.15
3	$W'_8A'_8I_{32}O_{32}$	5	0	52.63	1.31
4	$W'_8A'_8I_{32}O_{32}$	0	1	7.00	86.87
5	$W'_8A'_8I_{32}O_{32}$	1	1	52.67	1.24
6	$W'_8A'_8I_{32}O_{32}$	5	1	52.62	1.33

From Table 4.2.2 two cases are observed, (1) from row 2 and 5, we find little difference in the mAP when S changes from 0 to 1. (2) from row 2 and 3, we observe only a little increment in the mAP when we increase FN from $F1$ to $F5$.

Object Detection (Objective - 5) - Result summary

We were only able to quantize *SSD_MobileNet* using PTQ in *TensorFlow Lite*. From Experiment 12, we observe that 16-bit precision quantization with an input and output layer precision of 32-bit produces a change in mAP $< 2\%$ while the 8-bit quantization has a high($> 95\%$) change in accuracy. Finally, from experiment 13, it follows that 8-bit quantization produces a change in mAP $< 2\%$ using the suspected layers method. However, from this experiment, we observe that the impact of leaving the suspected layers non-quantized on the mAP of the quantized models is insignificant. Furthermore, it is found to be sufficient to keep only the first layer non-quantized.

Chapter 5

Conclusion

In this work, we researched SOTA quantization methods and experimented with those implemented by machine learning frameworks such as *TensorFlow Lite*, *SQuant* and *QKeras*. We evaluated these methods on SOTA NN models used for image classification and object detection tasks. The effect of quantization is measured in terms of percentage change in accuracy (or mAP) and model size reduction (or model compression), comparing the performance of the quantized and the non-quantized models. To this end, we conducted extensive set of experiments using different combinations of quantization techniques, datasets and model architectures. Based on the experimental results, we provide some recommendations for the quantization of NN models. It must be emphasized that the recommendations are based on the models that are investigated during this work and results may vary for other models or datasets.

Feature support for quantization methods in the machine learning frameworks used in this work vary widely for different usecase scenarios. Based on our experimental observations, we recommend using *TensorFlow Lite* or *SQuant* if the representative data is available or unavailable, respectively. *SQuant* does not support the generation of quantized models. Therefore it is advised to implement the concepts used by *SQuant* in *TensorFlow Lite* which can generate deployment-ready quantized models. If the available representative data is labelled, both PTQ and the QAT methods can be used. However if the data is unlabelled, only PTQ can be used as QAT requires labelled data. Finally, if no representative data is available, only DFQ method can be used, as both PTQ and QAT methods require representative data.

For the image classification task, all the models could be quantized to 16 and 8-bit precisions, achieving a model size reduction of $\sim 75\%$ and $\sim 85\%$ respectively, with both precisions performing at $< 1\%$ change in accuracy. The following are the recommendations for 16 and 8-bit quantization to obtain the best performance:

1. For 16-bit quantization using PTQ in *TensorFlow Lite*, it is best to quantize all the layers except the input and the output layers of the NN.
2. For 8-bit quantization using QAT in *TensorFlow Lite* or DFQ in *SQuant*, it is best to quantize all the layers except the input layer of the NN.
3. For 8-bit quantization using PTQ in *TensorFlow Lite*, it is best to quantize all the layers except the input and the output layers of the NN.

For sub-8-bit quantization, we limited our experiments to *SQuant* and *QKeras* as *TensorFlow Lite* does not support this bit precision. Our key observations for sub-8-bit quantization are as follows.

1. Models quantized using *QKeras* performed very poorly with $> 95\%$ change in accuracy. This high

change in accuracy might be due to the conversion of *Keras* layers to *QKeras* layers of the NN.

2. Quantization using *SQuant* to 6 and 4-bit precisions performed better with a change in accuracy of $< 5\%$ and $< 25\%$ respectively, with the input layer left unquantized.

For object detection, experiments are conducted only with PTQ in *TensorFlow Lite* where *SSD_MobileNet* was the only model architecture (among the other four architectures considered) that was successfully quantized without errors. *SSD_MobileNet* quantization to 16 and 8-bit precision achieves a model size reduction $\sim 75\%$ and $\sim 85\%$ respectively, with both the precisions performing at $< 2\%$ change in mAP.

The following are the recommendations for 16 and 8-bit quantization for object detection based on the *SSD_MobileNet*:

1. For 16-bit quantization, it is best to quantize all the layers except the input and the output layers.
2. For 8-bit quantization, it is best to quantize all the layers except for certain layers in the NN. The layers left unquantized are found using the suspected layer concept as mentioned in subsection 2.5.4.

From all the research conducted using the different SOTA frameworks in this work, it is observed that the quantization of image classification tasks has better support than object detection tasks. We suspect this is because of the difference in the model architectures and underlying complexity. Further development may be required to support the quantization of object detection models developed using *TensorFlow 2 Detection API* [40].

5.1 Future Work

1. For the quantization of object detection models, we were only able to quantize *SSD_MobileNet* successfully. Further investigation is required to either solve the quantization errors for the tested architectures or find model architectures that have quantization support for *TensorFlow Lite*.
2. During 8 bit quantization using *TensorFlow Lite*, keeping the suspected layers non-quantized did not have any appreciable impact. In this work, the suspected layers were determined based on a threshold value obtained by RMSE and Scale factor. Exploring other methods to calculate the threshold values of each layer will be interesting for future research.
3. Further investigation of quantization in *QKeras* framework by using SOTA model architectures built with *QKeras* layers. This is based on the premise of the work done by Coelho et al. [25] which shows promising results for the quantization of NN models trained using the *QKeras* layers.
4. Investigate the quantization of object detection models using *SQuant* to study how effective DFQ can perform for the models of this task.

Bibliography

[1] O. Dürr, B. Sick, and E. Murina, "Probabilistic deep learning with python," *Keras and TensorFlow Probability. isbn: 9781617296079*, 2020.

[2] S. Yohanandan. (2020) map (mean average precision) might confuse you! [Online]. Available: https://towardsdatascience.com/map-mean-average-precision-might-confuse-you-5956f1bfa9e2

[3] W. Dally, "High-performance hardware for machine learning," *Nips Tutorial*, vol. 2, p. 3, 2015.

[4] A. Krizhevsky, "The cifar-10 dataset." [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

[5] O. M. Parki, "The oxford-iiit pet dataset." [Online]. Available: https://www.robots.ox.ac.uk/~vgg/data/pets/

[6] M. Everingham, "Visual object classes challengenbsp;2012 (voc2012)." [Online]. Available: http://host.robots.ox.ac.uk/pascal/VOC/voc2012/index.html

[7] E. Hamilton, "What is edge computing: The network edge explained," *cloudwards. net. Retrieved*, pp. 05–14, 2019.

[8] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *arXiv preprint arXiv:2103.13630*, 2021.

[9] S. Xu, A. Huang, L. Chen, and B. Zhang, "Convolutional neural network pruning: A survey," in *2020 39th Chinese Control Conference (CCC)*. IEEE, 2020, pp. 7458–7463.

[10] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *International Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, 2021.

[11] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2704–2713.

[12] B. Koonce, "Efficientnet," in *Convolutional neural networks with swift for tensorflow*. Springer, 2021, pp. 109–123.

[13] ——, "Squeezenet," in *Convolutional Neural Networks with Swift for Tensorflow*. Springer, 2021, pp. 73–85.

[14] T. Zhou, X. Ye, H. Lu, X. Zheng, S. Qiu, and Y. Liu, "Dense convolutional network and its application in medical image analysis," *BioMed Research International*, vol. 2022, 2022.

[15] R. M. Gray and D. L. Neuhoff, "Quantization," *IEEE transactions on information theory*, vol. 44, no. 6, pp. 2325–2383, 1998.

[16] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 international conference on engineering and technology (ICET)*. Ieee, 2017, pp. 1–6.

[17] A. Apicella, F. Donnarumma, F. Isgr, and R. Prevete, "A survey on modern trainable activation functions," *Neural Networks*, vol. 138, pp. 14–32, 2021.

[18] A. Albanie, "Sise of resent model, albanie/convnet-burden: Memory consumption and flop count estimates for convnets." [Online]. Available: https://github.com/albanie/convnet-burden

[19] I. Z. Mukti and D. Biswas, "Transfer learning based plant diseases detection using resnet50," in *2019 4th International Conference on Electrical Information and Communication Technology (EICT)*, 2019, pp. 1–6.

[20] X. Xia, C. Xu, and B. Nan, "Inception-v3 for flower classification," in *2017 2nd International Conference on Image, Vision and Computing (ICIVC)*, 2017, pp. 783–787.

[21] W. Wang, Y. Li, T. Zou, X. Wang, J. You, and Y. Luo, "A novel image classification approach via dense-mobilenet models," *Mobile Information Systems*, vol. 2020, 2020.

[22] R. Yacouby and D. Axman, "Probabilistic extension of precision, recall, and f1 score for more thorough evaluation of classification models," in *Proceedings of the first workshop on evaluation and comparison of NLP systems*, 2020, pp. 79–91.

[23] R. Padilla, S. L. Netto, and E. A. Da Silva, "A survey on performance metrics for object-detection algorithms," in *2020 international conference on systems, signals and image processing (IWSSIP)*. IEEE, 2020, pp. 237–242.

[24] J. Hosang, R. Benenson, and B. Schiele, "Learning non-maximum suppression," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4507–4515.

[25] C. N. Coelho, T. Aarrestad, V. Loncar, H. Zhuang, A. Kuusela, J. Ngadiuba, S. Summers, A. A. Pol, S. Li, and M. Pierini, "arxiv: Ultra low-latency, low-area inference accelerators using heterogeneous deep quantization with qkeras and hls4ml," Tech. Rep., 2020.

[26] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International conference on machine learning*. PMLR, 2015, pp. 1737–1746.

[27] M. Nagel, R. A. Amjad, M. Van Baalen, C. Louizos, and T. Blankevoort, "Up or down? adaptive rounding for post-training quantization," in *International Conference on Machine Learning*. PMLR, 2020, pp. 7197–7206.

[28] C. Guo, Y. Qiu, J. Leng, X. Gao, C. Zhang, Y. Liu, F. Yang, Y. Zhu, and M. Guo, "Squant: On-the-fly data-free quantization via diagonal hessian approximation," *arXiv preprint arXiv:2202.07471*, 2022.

[29] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization," *arXiv preprint arXiv:2106.08295*, 2021.

[30] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," *arXiv preprint arXiv:2004.09602*, 2020.

[31] S. Garg, A. Jain, J. Lou, and M. Nahmias, "Confounding tradeoffs for neural network quantization," *ArXiv*, vol. abs/2102.06366, 2021.

[32] Y. Cai, Z. Yao, Z. Dong, A. Gholami, M. W. Mahoney, and K. Keutzer, "Zeroq: A novel zero shot quantization framework," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 13 169–13 178.

[33] V. Chikin and M. Antiukh, "Data-free network compression via parametric non-uniform mixed precision quantization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 450–459.

[34] B. Wu, Y. Wang, P. Zhang, Y. Tian, P. Vajda, and K. Keutzer, "Mixed precision quantization of convnets via differentiable neural architecture search," *arXiv preprint arXiv:1812.00090*, 2018.

[35] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 8612–8620.

[36] Jiaxing, H. Bai, J. Wu, and J. Cheng, "Bayesian automatic model compression," *IEEE Journal of Selected Topics in Signal Processing*, vol. 14, no. 4, pp. 727–736, 2020.

[37] "tf.lite.experimental.QuantizationDebugger TensorFlow v2.11.0." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/lite/experimental/QuantizationDebugger

[38] K. Team, "Keras documentation: Keras applications." [Online]. Available: https://keras.io/api/applications/

[39] PyTorch, "Models and pre-trained weights." [Online]. Available: https://pytorch.org/vision/stable/models.html

[40] G. Tensorflow, "Models/tf2_detection_zoo.md at master · tensorflow/models," May 2021. [Online]. Available: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md

[41] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*. Springer, 2016, pp. 21–37.

[42] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2117–2125.

[43] Z. Jiao, C. Ma, C. Lin, X. Nie, and A. Qing, "Real-time detection of pantograph using improved centernet," in *2021 IEEE 16th Conference on Industrial Electronics and Applications (ICIEA)*. IEEE, 2021, pp. 85–89.