Optimization of Deep Neural Networks for Efficient Resource Utilization

Master's thesis in Computer Science and Engineering

Namratha Sanjay

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

Optimization of Deep Neural Networks for Efficient Resource Utilization

Namratha Sanjay

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Optimization of Deep Neural Networks for Efficient Resource Utilization
Namratha Sanjay

© Namratha Sanjay, 2025.

Supervisor: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering
Advisor: Jonas Tallhage, Volvo Car Corporation
Examiner: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Optimization of Deep Neural Networks for Efficient Resource Utilization
Namratha Sanjay
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Deep neural networks (DNNs) are widely used in computer vision tasks such as image classification and semantic segmentation, but their high computational and memory demands limit deployment on resource-constrained edge devices. This thesis explores quantization as a model compression technique to improve inference efficiency while minimizing accuracy loss. Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) were applied to MobileNetV2 and ResNet50 for classification on the Mini-ImageNet dataset, and to FCN-ResNet18 for segmentation on the Cityscapes dataset. Additionally, mixed-precision QAT was investigated using first-order gradient-based sensitivity analysis to assign per-layer bit-widths. Maintaining activation precision at or above 6 bits during mixed-precision QAT enabled substantial compression—up to 7.8×—while keeping accuracy degradation under 1%. ResNet50 and MobileNetV2 attained compression ratios of 6.3× and 5.2×, respectively. FCN-ResNet18 preserved 57.3% mIoU with 7.8× compression and under 1% accuracy drop compared to the FP32 baseline. Conversely, reducing activation precision to 4 bits led to notable performance degradation, especially in lightweight models and segmentation tasks. Experiments were conducted on NVIDIA Tesla T4 GPU. The results demonstrate strong potential for deploying quantized DNNs on integer-based hardware such as mobile devices, embedded systems, and FPGAs.

Keywords: Neural Networks, Deep Learning, Network Compression, Quantization, Post-Training Quantization, Quantization Aware-Training ,Mixed-Precision Quantization, Network Acceleration, Resource-Constrained, Edge Device.

Acknowledgements

I would like to express my sincere gratitude to Annai Miranda and Siddhant Gupta for giving me the opportunity to carry out my master thesis at Volvo Cars on a challenging and engaging topic. I am especially thankful to my industrial supervisor Jonas Tallhage for his continuous support, insightful feedback, and encouragement throughout the project. I would also like to thank my academic supervisor, Pedro Petersen Moura Trancoso, for his guidance, thoughtful discussions, and valuable feedback that significantly shaped the direction and outcome of this thesis. This work is dedicated to my parents and my partner, whose unwavering support, belief in me, and constant encouragement made this journey possible. Lastly, I am deeply grateful to my family and friends for their patience, love, and understanding during this time.

Namratha Sanjay, Gothenburg, June 2025

Contents

List of Figures **xi**

List of Tables **xiii**

List of Listings **xiv**

List of Abbreviations **xv**

1 Introduction **1**
1.1 Motivation 2
1.2 Research Questions 2
1.3 Contributions 3
1.4 Ethical Considerations 3
1.5 Thesis Outline 3

2 Theory **4**
2.1 Artificial Neural Networks 4
2.1.1 Convolutional Neural Networks 5
2.1.1.1 Convolutional Layer 6
2.1.1.2 Fully Connected Layer 7
2.2 Evaluation Tasks 8
2.2.1 Image Classification 8
2.2.2 Semantic Segmentation 9
2.3 Quantization Techniques 9
2.3.1 Quantization Design Scheme 9
2.3.2 Post Training Quantization 12
2.3.3 Quantization Aware Training 12
2.4 Related Work 12
2.5 TensorRT Quantization 13
2.5.1 Implicit Quantization (PTQ) 13
2.5.2 Explicit Quantization (QAT) 14

3 Methods **15**
3.1 AI Frameworks 15
3.2 Quantization Techniques 16
3.2.1 Post Training Quantization 16
3.2.2 Quantization Aware Training 17

3.2.3 Gradient-Based Quantization 19
3.2.3.1 Gradient-Based Sensitivity Analysis 19
3.2.3.2 Bit-Width Allocation Using Pareto Frontier 20
3.3 Quantization Implementation 21
3.3.1 Quantization APIs 21
3.3.1.1 Initialization and Setup 21
3.3.1.2 Quantized Modules 22
3.3.1.3 Calibration and Scale Computation 23
3.3.1.4 Model Conversion and Export 23
3.3.1.5 Model Inference 23

4 Results **25**
4.1 Experimental Setup 25
4.1.1 Platform 25
4.1.2 Dataset 25
4.1.3 Experimental Models 26
4.1.4 Evaluation Metrics 26
4.2 Quantization Results for Image Classification 27
4.2.1 Base Model Result 27
4.2.2 Evaluation of Quantization Methods 28
4.3 Quantization Results for Segmentation 33
4.3.1 Base Model Result 33
4.3.2 Evaluation of Quantization Methods 33
4.4 Evaluation of Quantization Trade-offs 38
4.4.1 Discussion 38
4.5 Limitations and Future Directions 41

5 Conclusion **43**

Bibliography **44**

List of Figures

2.1 A simplified structure of a single neuron in a neural network. 4
2.2 Feedforward Neural Network 5
2.3 Convolution operation representation, where kernel K is convolved across Input Image I, resulting in feature maps. 7
2.4 Quantization value mapping of real values to INT8, adapted from [1]. 10

3.1 Post Training Quantization Workflow 17
3.2 pre-trained model network before and after undergoing QAT. 18
3.3 TensorRT deployment workflow for QAT models 19

4.1 Training and validation accuracy and loss for ResNet50 over 15 epochs. 28
4.2 Training and validation accuracy and loss for MobileNetV2 over 15 epochs. 28
4.3 Layer-wise sensitivity analysis for ResNet50 and MobileNetV2, derived from gradient-based metrics. Scores guided bit-width allocation in mixed-precision configurations by identifying which layers could be aggressively quantized and which required preservation. 30
4.4 Layer wise bit-width allocation of weights (W-bit), for ResNet50 - 4MP vs 2MP configuration. 32
4.5 Layer wise bit-width allocation of weights (W-bit), for MobileNetV2 - 4MP vs 2MP configuration. 32
4.6 Training and validation metrics (accuracy, loss, and IoU) for FCN-ResNet18 over 37 epochs. 33
4.7 Layer-wise sensitivity analysis of FCN-ResNet18, derived from first-order gradient-based metrics. Peaks correspond to layers that are more sensitive to quantization noise and were preserved at higher precision during bit allocation. 34
4.8 Layer wise bit-width allocation of weights (W-bit), for FCN-ResNet18 - 4MP vs 2MP configuration. 35
4.9 FCN-ResNet18-FP32 36
4.10 Predicted segmentation outputs of FCN-ResNet18 on PTQ and QAT quantization methods. 36
4.11 Predicted segmentation outputs for FCN-ResNet18 under 4MP-A8, 4MP-A6, and 4MP-A4 configurations. 37
4.12 Predicted segmentation outputs for FCN-ResNet18 under 2MP-A8, 2MP-A6, and 2MP-A4 configurations. 37

4.13 Accuracy drop for ResNet50, MobileNetV2, and FCN-ResNet18 across FP16, PTQ, QAT, and mixed-precision (4MP, 2MP) quantization methods. 38
4.14 Compression ratio comparison for ResNet50, MobileNetV2, and FCN-ResNet18 across FP16, PTQ, QAT, and mixed-precision (4MP, 2MP) quantization methods. 39
4.15 Trade-off between accuracy drop and compression ratio across quantization configurations for ResNet50, MobileNetV2, and FCN-ResNet18. Configurations such as 2MP-A8 and 4MP-A8 demonstrate favorable compression with minimal accuracy loss. MobileNetV2 exhibits high sensitivity under low-bit configurations (e.g., 2MP-A4), while FCN-ResNet18 maintains robustness even under aggressive compression. . 39
4.16 Inference speed-up comparison for ResNet50, MobileNetV2, and FCN-ResNet18 across FP16, PTQ, and QAT. 40

List of Tables

4.1 Model information of MobileNetV2 and ResNet50, including convolutional and fully connected layers 26
4.2 Baseline performance comparison between ResNet50 and MobileNetV2 (batch size = 1). 29
4.3 Comparison of implicit quantization methods on ResNet50 and MobileNetV2: accuracy drop and compression ratio relative to full-precision baseline 29
4.4 Comparison of mixed-precision QAT methods on ResNet50 and MobileNetV2. Accuracy drop and compression ratio are reported relative to the full-precision baseline. Since the bit-width of weights is fixed across configurations, model size and CR remain constant within each group. 31
4.5 Comparison of implicit quantization methods on ResNet50 and MobileNetV2 (batch size = 1), showing latency, throughput, and speed-up relative to the full-precision baseline. 32
4.6 Baseline performance of FCN-ResNet18 on Cityscapes with full precision, on input size 1024×2048 and batch size = 1 33
4.7 Comparison of implicit quantization methods on FCN-ResNet18: accuracy drop and compression ratio relative to full-precision baseline. . 34
4.8 Comparison of mixed-precision QAT methods on FCN-ResNet18. Accuracy drop and compression ratio are reported relative to the full-precision baseline. Since the bit-width of weights is fixed across configurations, model size and CR remain constant within each group. . 35
4.9 Comparison of implicit quantization methods on FCN-ResNet18 (batch size = 1), showing latency, throughput, and speed-up relative to the full-precision baseline. 36

Listings

2.1 TensorRT Implicit Quantization APIs 13
2.2 trtexec Command for Implicit Quantization 13
2.3 Code snippet of quantized modules 14
3.1 Default QuantConv2d (8-bit Per-Channel Quantization) 22
3.2 Custom QuantConv2d with bit-width configuration support 23

Abbreviations

Abbreviation	Full Form
DNN	Deep Neural Network
CNN	Convolutional Neural Network
NN	Neural Network
ANN	Artificial Neural Network
GPU	Graphics Processing Unit
ReLU	Rectified Linear Unit
SiLU	Sigmoid Linear Unit (also known as Swish)
AI	Artificial Intelligence
API	Application Programming Interface
FFNN	Feedforward Neural Network
LCNs	Locally Connected Networks
FCN	Fully Convolutional Network
SGD	Stochastic Gradient Descent
PTQ	Post-Training Quantization
QAT	Quantization-Aware Training
DS	Downsampling
STE	Straight-Through Estimator
SDK	Software Development Kit
QDQ	Quantize-Dequantize Node
ONNX	Open Neural Network Exchange
CLI	Command-Line Interface
MP	Mixed Precision
ISLVRC	ImageNet Large Scale Visual Recognition Challenge
DLA	Deep Learning Accelerator
mIoU	Mean Intersection over Union

1

Introduction

One of the most used deep learning models for image classification and detection, the convolutional neural network (CNN) achieves high accuracy compared to other machine learning algorithms. Due to their domain excellence, autonomous vehicles now integrally use them for both image classification and object detection as a part of their vision system. Object detection plays a vital role in recognizing and pinpointing objects in a vehicle's environment. Semantic segmentation, building upon object detection, takes this further by labeling each pixel in an image, helping the vehicle discern the precise boundaries and categories of all surrounding objects. This detailed understanding is critical for safe driving, as it allows the vehicle to distinguish between various road elements like lanes, sidewalks, and other vehicles, leading to more accurate decision-making. While object detection focuses on identifying specific objects, image classification categorizes the entire scene into a single class. For autonomous vehicles, this means identifying broader scenarios, such as whether the vehicle is on a city street, highway, or rural road, which influences its driving behavior; for example, adjusting speed limits or driving style.

However, CNNs are computationally- and memory-intensive, which poses significant challenges for inference deployment on Edge AI devices that have limited memory and power [2]. Optimizing and reducing the size of these networks remains an open research problem. Some of the most researched optimization techniques involve network architecture search, knowledge distillation, and network compression approaches which include pruning and quantization [3]. The focus of this thesis work is investigating the network quantization approach. As digital computers began to process abstract mathematical computations, the challenge emerged for efficiently representing, manipulating, and communicating the numerical values involved in those computations [4]. Quantization, frequently explored in the context of constrained representation of numerical quantities, involves mapping values from continuous space to discrete space. This process aims to minimize the number of bits needed for discrete representation while maintaining computation accuracy with minimal degradation. In the context of neural networks, quantization refers to reducing the numerical precision used for computations and tensor storage whether by lowering the bit-width of floating-point representations or by converting to integer formats. This approach can significantly decrease the size of neural networks while enhancing inference latency and throughput by leveraging high-throughput integer instructions [1].

1.1 Motivation

Inference of convolutional neural networks is typically done in central cloud servers. However, due to factors like the unreliability of transmission channels when exchanging data with a central server, adds latency, and concerns over security and data privacy, many CNN-based applications are now shifting to edge devices located near the data source. However this comes at the cost of edge devices having limited resources in computing capability and memory. Reducing the precision of weights and activations from 32-bit floating point (FP32) to 8-bit integer (INT8) reduces memory usage by a factor of 4 and can accelerate matrix multiplications by up to 16×, depending on hardware optimizations [1]. The leading research focus in academia and industry is on INT8 quantization, which can significantly improve the inference speed and reduce memory requirements without sacrificing much accuracy. While most of the research has adopted INT8 quantization, there are ongoing explorations to lower bit-widths as low as ternary and binary. Binarization represents an extreme type of quantization. BinaryNet [5] and XNORNet [6] are notable examples of this approach. However, their critical limitation is the considerable reduction in accuracy they cause. In contrast, using mixed precision quantized models allows a trade-off between accuracy and memory footprint [7], indicating that different network parts can operate at varying precisions. The authors of [7] employ Hessian second-order information to automatically determine the appropriate quantization precision for each layer. Although these methods have demonstrated promising results in reducing memory usage, the search space is exponentially large to determine the optimal bit-width for each layer.

1.2 Research Questions

This thesis focuses on evaluating the performance of neural networks using compression techniques designed to reduce precision and accelerate inference. For this purpose, open-source models and datasets, closely resembling those used by Volvo Cars, are employed. The compression algorithm is intended for deployment on an edge GPU. This work can serve as a foundation for future efforts by software engineers in the Safe Vehicle Automation department at Volvo Cars, facilitating implementation on various hardware platforms in their autonomous vehicles.
This thesis seeks to achieve the following objectives and provides quantitative results. The specific research questions to be addressed are listed below:

RQ1. What is a desirable compression technique to achieve an efficient inference with reduced model size and fast inference?

RQ2. How does the compression algorithm perform according to the objective determined in RQ1?

RQ3. What is the acceptable limit for quantization of neural network for efficient inference without much loss in accuracy?

1.3 Contributions

This thesis investigates quantization techniques aimed at enabling efficient deployment of deep neural networks on resource-constrained hardware platforms. The work includes the application of PTQ and QAT to standard models such as MobileNetV2 and ResNet50 for image classification on the Mini-ImageNet dataset, and FCN-ResNet18 for semantic segmentation on the Cityscapes dataset. A mixed-precision quantization method was employed using first-order gradient-based sensitivity analysis to guide layer-wise bit-width assignment, evaluating configurations with 2, 4, 6, and 8 bits. The thesis presents a comprehensive analysis comparing uniform 8-bit quantization with mixed-precision approaches in terms of accuracy retention, model compression, and inference speed-up. The findings highlight key engineering trade-offs between compression and task accuracy and identify precise thresholds necessary for maintaining acceptable performance. This work contributes a practical, sensitivity-aware quantization framework that can inform deployment decisions for integer-based edge devices such as embedded systems and FPGAs.

1.4 Ethical Considerations

The techniques explored in this thesis aim to improve the efficiency of neural networks for deployment on edge devices. However, it is important to acknowledge the broader ethical implications of such optimizations. While quantization reduces resource consumption, enabling wider deployment, it may also lead to unintended consequences such as biased performance across different user groups if not evaluated carefully. Furthermore, efficient deployment on low-cost hardware may accelerate the use of AI in sensitive domains without adequate oversight. These considerations underscore the need for responsible model evaluation, transparency in deployment contexts, and fairness-aware design practices.

1.5 Thesis Outline

This thesis is organized into five main chapters. Chapter 1 introduces the motivation for this work, the research questions addressed, and the specific contributions made. Chapter 2 provides the theoretical background, including key concepts related to convolutional neural networks, image classification and semantic segmentation tasks, as well as an overview of quantization techniques such as Post-Training Quantization, Quantization-Aware Training, and mixed-precision strategies. Chapter 3 describes the methodologies and implementation details used in this study, including gradient-based sensitivity analysis and bit-width allocation strategies. Chapter 4 presents the experimental setup and evaluation of quantized models, comparing various quantization strategies in terms of compression ratio, accuracy, and performance. Finally, Chapter 5 summarizes the findings, discusses limitations, and suggests future directions for deploying quantized models on resource-constrained hardware platforms.

2

Theory

CNNs have a wide range of applications, with image classification and semantic segmentation being two of the most crucial in vision systems. This chapter reviews key concepts of CNNs, focusing on relevant layers. Additionally, it offers an overview of the quantization technique and the TensorRT toolkit employed for quantization, alongside related work in this field.

2.1 Artificial Neural Networks

The architecture of an artificial neural network consists of individual units called neurons, designed to mimic the behavior of biological neurons in the brain [8]. Each neuron takes in inputs, applies weights and biases, and processes the result through an activation function to generate an output, as shown in figure 2.1, the figures in this section are adapted from standard neuron diagrams commonly found in neural network literature.

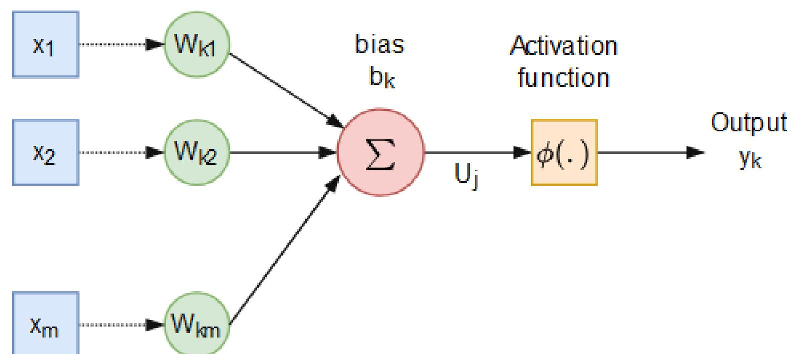


Figure 2.1: A simplified structure of a single neuron in a neural network.

A Neural network model receives *input* as a set of features provided for the learning process. For instance, in semantic segmentation, the input typically consists of an array of pixel values representing an image, where each pixel is assigned a label corresponding to a specific feature or region within the image. *Weights* scale inputs to highlight important features, while the transfer function aggregates inputs into a single value for the activation function. Activation functions introduce non-linearity, enabling the network to model complex relationships. Some of the most commonly

used activation functions include Relu [9], Sigmoid [10], Tanh [10]. Bias shifts the activation output, improving the network's ability to fit data. The output y_k from figure 2.1 can be represented as:

$$y_k = \phi\left(\sum_{i=1}^{m} w_{ki}x_i + b_k\right) \tag{2.1}$$

Where y_k is output, x_i is input, ϕ is the activation function, w_ki is the weight and b_k is bias. The simplest type of a neural network is the feedforward neural network as shown in figure 2.2 [11].

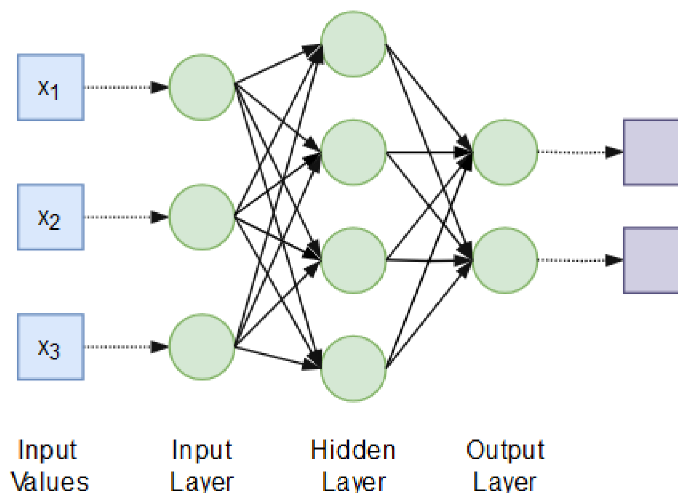


Figure 2.2: Feedforward Neural Network

In the above figure, information only traverses from a lower layer to a higher layer. There are no connections between neurons in the same layer, and there are no connections that transmit data from a higher layer to a lower layer.

2.1.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of neural network designed specifically for recognizing patterns in data that is best represented in a grid-like structure, such as images [12]. Although it is possible to flatten this data and use a standard Feedforward Neural Network (FFNN), doing so results in the loss of valuable spatial information. Instead, CNNs maintain the multidimensional nature of the data in both input and hidden layers, which would typically require a large number of weights. However, unlike FFNNs, where each unit in a layer is connected to every unit in the next layer, CNNs use sparse connections, meaning that each unit only interacts with a subset of the following units. This approach reflects the idea that a pixel in an image is more likely to be related to nearby pixels that form part of the same shape or object, rather than all pixels in the image. Additionally, CNNs employ weight sharing, where the same set of weights is applied across all unit pairs in a layer. Vision tasks are defined by the principles of locality and translation invariance. The exceptional performance of CNNs in these tasks is largely credited

to their biases of locality and weight sharing encoded into their short convolutions [13]. The reasoning behind this is, that these biases align with the characteristics of vision tasks, where local and spatially varying signals influence the output [14]. Unlike CNNs, Locally Connected Networks (LCN) incorporate only locality, while Fully Connected Networks (FCN) lack both locality and weight sharing, thus leading to higher sample complexity as compared to CNNs.

CNNs are also characterized by their ability to learn hierarchical features. Early layers capture simple features such as edges and corners [15], middle layers detect more complex patterns such as textures and object parts, and deeper layers combine these patterns into high-level representations, such as objects or classes. This hierarchical feature learning is a cornerstone of CNN's success in vision tasks [16]. Furthermore, CNNs include non-linear processing units, such as ReLU, Sigmoid or Swish activation functions, after convolutional layers. These activations introduce non-linearity, enabling CNNs to model complex, non-linear relationships between inputs and outputs [17]. Another essential component of CNNs is subsampling layers (e.g., MaxPooling, AveragePooling), which reduce the spatial dimensions of feature maps. These layers improve computational efficiency, add translational invariance, and help the model focus on the most important features, regardless of their location in the image [18]. CNNs are trained using backpropagation, where gradients are computed for all parameters (filters, biases) and updated via optimization techniques like SGD or Adam. In-order to improve generalization and prevent overfitting, CNNs often employ regularization techniques such as dropout and batch normalization, as well as data augmentation methods like cropping, flipping, and rotation. The scalability of CNNs allows them to handle large datasets and complex tasks effectively, making them a cornerstone of modern computer vision applications [16]. Transfer learning—using pretrained models on large datasets like ImageNet—enables CNNs to be fine-tuned for specific tasks, significantly reducing training time and computational resources [19].

However, CNNs are not without limitations. They require large labeled datasets and are computationally expensive, especially due to the high cost of operations in layers such as convolutional, fully connected, and pooling layers. To address their computational challenges, optimization techniques such as quantization can be employed. Quantization reduces the precision of weights and activations, for instance, from 32-bit floating-point to lower-precision formats like 8-bit integers, significantly improving inference speed and reducing memory usage without substantial loss in accuracy.

2.1.1.1 Convolutional Layer

Convolutional layers are the core of CNNs, as they hold the learned kernels (*weights*) responsible for extracting features to differentiate images in image classification and semantic segmentation tasks. A convolution operates as a sliding dot product, where a kernel moves across the input matrix, computing the dot product between the kernel and input at each position, creating *feature maps* that highlight patterns.

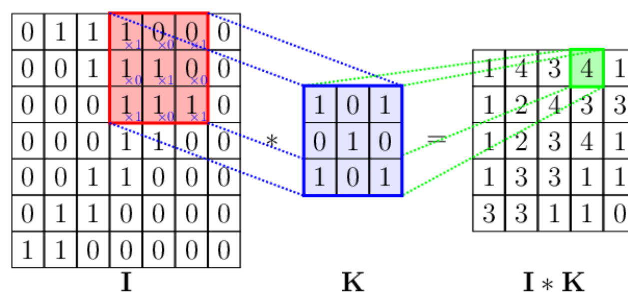


Figure 2.3: Convolution operation representation, where kernel K is convolved across Input Image I, resulting in feature maps.

Mathematically, for a given position *(i,j)*, the output feature map is:

$$(I * K)(i, j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \mathrm{I}(i + m, j + n) \cdot K(m, n) \tag{2.2}$$

The parameter sharing in convolutional layers significantly reduces the number of trainable parameters compared to fully connected layers, making CNNs more efficient. However, key hyperparameters such as kernel size, stride, and padding determine the spatial dimensions of the feature maps. The kernel size defines the receptive field of each neuron, the stride controls the step size of the kernel across the input image, and padding adds zero pixels to preserve spatial dimensions. The relationship between these parameters is given by:

$$\text{Output dimension} = \frac{I - K + 2 \times P}{S} + 1 \tag{2.3}$$

where I is the input image dimension, K is the kernel size, P is the padding size, and S is the stride.
Convolutional layers account for the majority of computational cost in CNNs due to their high FLOP count. For instance, AlexNet [20] has 60 million parameters and requires 729 million FLOPs per image. INT8 quantization significantly reduces memory and computation by replacing FP32 multiplications with integer operations. In models like MobileNetV2, the 1x1 pointwise convolutions contribute the most to computational cost and are primary targets for quantization. Similarly, depthwise convolutions, although computationally lighter, also benefit from quantization. In ResNet50, both the 3x3 convolutions for feature extraction and the 1x1 bottleneck convolutions for dimensionality reduction require quantization. In ResNet18-FCN, the 3x3 convolutions in residual blocks and the 1x1 convolutions in the segmentation head are quantization targets.

2.1.1.2 Fully Connected Layer

Fully connected (FC) layers play a crucial role in CNN architectures by transforming the high-dimensional feature maps into a final output representation suitable for classification. Each neuron in the FC layer is densely connected to all neurons in the previous layer, allowing the model to learn complex relationships between

extracted features and target labels. These layers are typically found at the end of CNN models and serve as the final decision-making stage.
FC layers perform matrix multiplications, where each input neuron contributes to every output neuron. Unlike convolutional layers, which utilize local spatial hierarchies, FC layers learn global feature relationships across the entire input. This makes them particularly effective in classification tasks, where the model needs to consolidate extracted features into a single prediction. However, due to their dense connections, FC layers introduce a high number of parameters, leading to large memory consumption and increased inference latency, especially in deep networks.
FC layers tend to be memory-bound rather than compute-bound, meaning quantization primarily benefits storage and memory bandwidth. Quantizing FC layers reduces the memory footprint by compressing parameters from FP32 to INT8, leading to a fourfold reduction in size.
For an input vector X and weight matrix W, an FC layer performs:

$$Y = WX + B \tag{2.4}$$

where B is the bias term. This dense operation scales poorly with increasing layer sizes, making it an ideal candidate for quantization. In models like MobileNetV2 and ResNet50, the final classification layer is fully connected and benefits from quantization. However, in ResNet18-FCN, which is a fully convolutional network, there are no FC layers, and instead, the 1x1 convolution layers serve a similar function. Quantizing FC layers significantly enhances inference efficiency, particularly in memory-constrained environments such as edge AI applications. Quantizing FC layers significantly enhances inference efficiency, particularly in memory-constrained environments such as edge AI applications.

2.2 Evaluation Tasks

Two types of evaluation tasks are conducted in this study to demonstrate the generality of the approach, ensuring its applicability across different model architectures and tasks. This section outlines the two evaluation tasks used: image classification and semantic segmentation.

2.2.1 Image Classification

Image classification is a fundamental computer vision task that assigns a single label to an image based on its visual content. In deep learning, it is typically formulated as a multiclass classification problem, where an input image is categorized into one of several predefined classes. Early classification models relied on manually crafted features and performed well on small datasets like CIFAR-10 and MNIST but struggled with real-world challenges such as lighting variations, background clutter, and intra-class diversity. The introduction of large-scale datasets like ImageNet [21] revolutionized the field by enabling deep learning models to generalize across diverse object categories. A pivotal moment came in 2012 when AlexNet [20] outperformed traditional feature-engineering approaches in the ILSVRC competition, demonstrating the power of deep convolutional networks. This success laid the foundation for

modern architectures such as ResNet50, MobileNetV2, and EfficientNet, which extract hierarchical patterns from images to achieve state-of-the-art performance. This thesis evaluates model performance on ImageNet-based classification tasks, where input images have a resolution of 224×224 pixels with three RGB channels. The analysis focuses on classification accuracy under different computational constraints, examining how model architecture and precision settings affect performance across various object categories.

2.2.2 Semantic Segmentation

Semantic segmentation extends image classification by assigning a category label to each pixel in an image, providing a dense prediction of object regions within a scene. This fine-grained understanding is essential for applications such as autonomous driving, medical imaging, and scene parsing. Unlike classification, which produces a single label per image, segmentation differentiates multiple objects and background regions simultaneously. This thesis evaluates segmentation models on the Cityscapes dataset [22], a widely used benchmark for urban scene understanding. The dataset includes high-resolution images with pixel-wise annotations for various object classes, making it suitable for assessing segmentation performance in real-world driving scenarios. The evaluation metric used is mean Intersection over Union (mIoU), which quantifies segmentation accuracy by measuring the overlap between predicted and ground-truth masks. The study analyzes segmentation performance under varying computational constraints, assessing how architectural choices and precision settings influence segmentation quality.

2.3 Quantization Techniques

In mathematics and DSP, quantization maps input values from a large, often continuous set to a smaller, finite set, similar to discretization in algorithms. In neural networks (NNs), it reduces the precision of weights and activations from a higher precision floating point to fixed-point integers or a lower floating point value. Quantization is classified into Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ) depending on when it is applied. QAT integrates quantization into the training process, introducing quantization noise to help models adapt while remaining in the FP32 range, often requiring fine-tuning, whereas PTQ is performed after training. NNs are generally robust to quantization, maintaining accuracy even at lower bit widths. Their large parameter space enables different parameter sets to achieve near-optimal performance, allowing fine-tuning to compensate for precision loss. This flexibility makes a careful quantization design crucial for optimizing efficiency based on the model architecture. This section discusses key quantization strategies that are employed in this thesis work:

2.3.1 Quantization Design Scheme

Quantization maps full-precision floating-point values to discrete low-precision integers. The two primary methods are uniform and non-uniform quantization.

Uniform quantization maintains a constant step size across the entire range, mapping real values $x \in [\beta, \alpha]$ to signed integer values with b bit-width in range $[-2^{b-1}, 2^{b-1} - 1]$, where values outside this range are clipped. It follows a linear transformation.

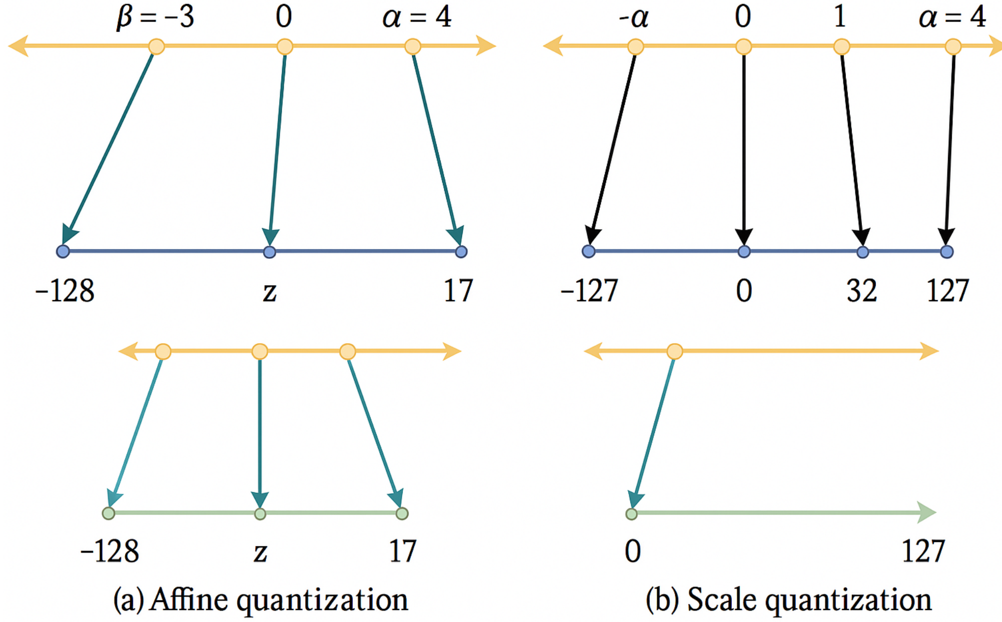


Figure 2.4: Quantization value mapping of real values to INT8, adapted from [1].

Affine quantization as illustrated in figure 2.4(a), follows the transformation function:

$$f(x) = S \cdot x + z \tag{2.5}$$

Where S is scale factor and z is zero-point and x, S, $z \in \mathbf{R}$:

$$S = \frac{2^{b-1}}{\alpha - \beta} \tag{2.6}$$

$$z = -\text{round}(\beta \cdot S) - 2^{b-1} \tag{2.7}$$

zero point is to determine the corresponding position of 0 in the domain of x within the range of values. Quantization operation is defined as:

$$\text{clip}(x, \beta, \alpha) = \begin{cases} \beta, & x < \beta \\ x, & \beta \leq x \leq \alpha \\ \alpha, & x > \alpha \end{cases} \tag{2.8}$$

$$x_q = quantize(x, b, S, z) = \text{clip}\left(\text{round}(S \cdot x + z), -2^{b-1}, 2^{b-1} - 1\right) \tag{2.9}$$

The corresponding dequantization operation, that converts from integer back to original input real value i.e FP32 is defined as:

$$x_{dq} = dequantize(x, b, S, z) = \frac{1}{S}(x_q - z) \tag{2.10}$$

Scale quantization uses only a scale factor, maintaining symmetric mapping around zero. The transformation function:

$$f(x) = S \cdot x \tag{2.11}$$

scale factor and the quantization operation here is represented as:

$$S = \frac{2^{b-1} - 1}{\alpha} \tag{2.12}$$

$$x_q = quantize(x, b, S) = \text{clip}\left(\text{round}(S \cdot x), -2^{b-1}, 2^{b-1} - 1\right) \tag{2.13}$$

Equation 2.14 defines the dequantization operation for scale quantization.

$$x_{dq} = dequantize(x, b, S) = \frac{x_q}{S} \tag{2.14}$$

Non-Uniform quantization uses a variable step size, offering higher precision for frequently occurring values through a non-linear transformation $f(x)$:

$$x_q = quantize(x, b, S, z) = \text{clip}\left(\text{round}(f(x)), q_{\min}, q_{\max}\right) \tag{2.15}$$

where $f(x)$ determines the mapping, and $q_{\min}, q_{\max}$ set the valid range. Techniques like *logarithmic* and *power-law quantization* help handle skewed distributions but increase computational cost. Therefore, we focus on uniform quantization-specifically scale quantization-due to its hardware efficiency—integer arithmetic (addition, multiplication) remains fast and computationally cheap, making it ideal for practical deployment.

Quantization Granularity defines how parameters are shared, from per-tensor (coarsest) to per-element (finest), with intermediate levels like per-channel scaling. In convolutional layers, per-channel quantization assigns unique scales to 3D kernels, optimizing precision while ensuring efficient computations. Per-tensor quantization is used for activations to simplify operations. In this thesis, we employ per-channel weight quantization alongside per-tensor activation quantization this was influenced by the suggestions given by authors in [23].

Calibration is the process of selecting α and β to define the range for model weights and activations, crucial for quantization. Common methods include:

- Max: Captures the full range using the maximum absolute value.
- Entropy: Minimizes information loss by optimizing KL divergence - a measure of how much the quantized distribution diverges from the original floating-point distribution. This is the default method used by TensorRT [24]
- Percentile: Clips extreme values to improve precision for most data points. For instance, 99% calibration removes the top 1% of values with the highest magnitudes.

Max calibration preserves the entire range but reduces precision, while entropy and percentile methods trade off outliers for better resolution of inlier values.

2.3.2 Post Training Quantization

This approach quantizes a pre-trained model without retraining/fine-tuning, using a small calibration dataset to establish the clipping range for quantizing weights and activations during deployment. While PTQ efficiently reduces model size, it often leads to greater accuracy loss compared to QAT.

2.3.3 Quantization Aware Training

QAT integrates quantization operations into a neural network before training, allowing the model to adapt to quantized weights and activations. The quantized model is then fine-tuned for a few iterations to recover any accuracy loss. During the forward pass, fake quantization is applied by first mapping activations and weights to discrete integer levels and then dequantizing them back to floating point, simulating quantization effects while keeping computations in FP32. Since quantization is not naturally compatible with gradient-based learning, QAT employs the Straight-Through Estimator (STE) to approximate gradients [25]. STE assigns a gradient of 1 within the valid range and 0 otherwise, enabling the model to learn as if quantization were a smooth operation. By incorporating low-precision arithmetic during training, QAT ensures that the final quantized model maintains high accuracy when deployed in an integer-based format for inference.

2.4 Related Work

Early research focused on PTQ, where per-channel weight quantization and per-layer activation quantization at 8-bit precision helped preserve model accuracy [23]. However, PTQ often suffered from accuracy degradation, particularly in activation-sensitive models. To address this, QAT was introduced, incorporating fake quantization during training to simulate low-precision arithmetic, thereby enabling better accuracy retention [26]. Recent advancements have expanded into non-uniform and adaptive bit-width quantization, aiming to optimize both efficiency and precision. A comprehensive survey highlighted techniques such as logarithmic and power-law quantization to better preserve frequently occurring values [27]. Mixed-precision quantization improves accuracy–efficiency trade-offs by assigning different bit-widths to layers based on their sensitivity—retaining higher precision in critical layers while using lower precision elsewhere. Zero shot Quantization (ZeroQ) [28] introduced a zero-shot quantization method that eliminates the need for a calibration dataset by generating synthetic data that mimics the statistical properties of real activations. This enables quantization of pre-trained models without requiring access to sensitive or proprietary data. Hessian Aware Weight Quantization (HAWQ) [7] leveraged second-order Hessian information, particularly the largest eigenvalues, to identify layers most sensitive to quantization. While effective, this approach is computationally expensive, and the bit-width search space grows exponentially with model depth, limiting its scalability. In contrast, Bit-Gradient Sensitivity Driven Mixed-Precision Quantization (BMPQ) [29] proposed a during-training approach that dynamically assigns bit-widths using first-order gradient-based bit-sensitivity analysis. It evaluates layer importance via normalized bit gradients and periodically

solves an Integer Linear Program (ILP) to satisfy hardware constraints. Similarly, Chauhan et al [30] proposed a post-training method that estimates sensitivity using expected average gradient norms in the ε-neighborhood of convergence. Their ILP-based optimization assigns bit-widths to minimize estimated accuracy loss while respecting model size constraints. Both approaches demonstrate that first-order information can serve as an efficient and scalable alternative to second-order sensitivity metrics.

2.5 TensorRT Quantization

TensorRT SDK provides APIs for optimizing and deploying machine learning inference on NVIDIA GPUs, seamlessly integrating with training frameworks like TensorFlow, PyTorch, and MXNet. It enhances neural network performance by leveraging lower-precision data types such as FP16 and INT8, improving efficiency while maintaining accuracy [31]. TensorRT supports two quantization methods: implicit (automatic) and explicit (manual) quantization, which are mutually exclusive.

2.5.1 Implicit Quantization (PTQ)

Implicit quantization is primarily used for PTQ, where the TensorRT builder applies quantization during model conversion via dynamic calibration without modifying the model. In this work, we use *trtexec*, a TensorRT CLI tool that acts as a wrapper around the TensorRT C++ and Python APIs to facilitate model conversion, benchmarking, and profiling. Specifically, we use trtexec to convert a pre-trained PyTorch model to ONNX and then to TensorRT. This approach is limited to INT8 precision and does not explicitly enforce precision levels. Instead, TensorRT dynamically selects the optimal precision during inference, utilizing INT8 where beneficial and falling back to FP32 or FP16 when necessary. For FP16 and INT8, TensorRT offloads computations to Tensor Cores, ensuring high-throughput execution of matrix operations. INT8 activations are scaled using calibration-generated dynamic ranges for accurate representation.

Listing 2.1: TensorRT Implicit Quantization APIs

```
# Create TensorRT builder and config
builder = trt.Builder(trt.Logger(trt.Logger.WARNING))
config = builder.create_builder_config()

# TensorRT will infer dynamic ranges
config.set_flag(trt.BuilderFlag.INT8)

# Calibration set-up
calibrator = Int8EntropyCalibrator(calib_dataloader)
config.int8_calibrator = calibrator
```

Listing 2.2: trtexec Command for Implicit Quantization

```
./trtexec \
--onnx=model.onnx \
--fp16 \
--int8 \
--useSpinWait \
--calib=calibration_data.cache \
--saveEngine=model.trt
```

2.5.2 Explicit Quantization (QAT)

Explicit quantization is used for QAT, where users manually insert IQuantizeLayer (Q) and IDequantizeLayer (DQ) nodes during training, allowing the model to adapt to low-precision arithmetic. This approach provides finer control over precision and accuracy, ensuring better quantization robustness. NVIDIA's PyTorch-Quantization Toolkit [32] facilitates QAT by adding QDQ nodes to pre-trained models before exporting them to ONNX and TensorRT. ONNX is an open-source format that ensures interoperability across deep learning frameworks, enabling seamless integration with TensorRT for efficient low-precision inference. The *TensorQuantizer* module is central to the API, handling tensor quantization, simulated quantization, and statistic collection. It works with *QuantDescriptor*, which defines the quantization method for a tensor. Built on top of TensorQuantizer are *quantized modules*, designed as drop-in replacements for PyTorch's full-precision counterparts. APIs used for explicit quantization [33]:

Listing 2.3: Code snippet of quantized modules

```
# PyTorch's module
fc1 = nn.Linear(in_features, out_features, bias=True)
conv1 = nn.Conv2d(in_channels, out_channels, kernel_size)

# Quantized version
quant_fc1 = quant_nn.Linear(
    in_features, out_features, bias=True,
    quant_desc_input=tensor_quant.QUANT_DESC_8BIT_PER_TENSOR,
    quant_desc_weight=tensor_quant.QUANT_DESC_8BIT_LINEAR_WEIGHT_PER_ROW)
quant_conv1 = quant_nn.Conv2d(
    in_channels, out_channels, kernel_size,
    quant_desc_input=tensor_quant.QUANT_DESC_8BIT_PER_TENSOR,
    quant_desc_weight=tensor_quant.QUANT_DESC_8BIT_CONV2D_WEIGHT_PER_CHANNEL)
```

3

Methods

This chapter discusses gradient-based mixed-precision quantization, as well as the workflow and implementation details thereof.

3.1 AI Frameworks

Deploying deep learning models in real-world applications requires an optimized pipeline for training, quantization, and inference. This study leverages PyTorch, TensorRT, and the NVIDIA's PyTorch-Quantization toolkit to develop, quantize, and optimize deep neural networks. PyTorch is a widely used deep learning framework that supports flexible model development, automatic differentiation, and GPU acceleration. In this work, PyTorch serves as the primary framework for training the baseline full-precision model using FP32 arithmetic. PyTorch supports PTQ and QAT by simulating low-bit precision inference using fake quantization layers. However, PyTorch's quantization framework is primarily optimized for CPU execution, relying on FBGEMM and QNNPACK backends for INT8 inference on x86 and ARM architectures [34]. It does not natively support GPU-based QAT or TensorRT integration. For this reason, this work employs NVIDIA TensorRT, a high-performance deep learning inference engine designed for hardware-accelerated INT8 and FP16 execution on NVIDIA GPUs. TensorRT utilizes CUDA-optimized kernels and NVIDIA Tensor Cores to accelerate matrix operations. Tensor Cores enable mixed-precision computation, allowing layers to execute at lower precision (FP16/INT8) while maintaining numerical stability. TensorRT supports both PTQ-based calibration and simulated QAT inference using the PyTorch-Quantization Toolkit [32], making it a highly-flexible inference solution. To facilitate deployment in TensorRT, pre-trained PyTorch models are first converted to the ONNX format, which serves as an interoperability standard between deep learning frameworks and inference engines. ONNX enables TensorRT-specific optimizations, including precision calibration, kernel tuning, and layer fusion, which leverage CUDA-accelerated operations. By mapping computationally intensive layers onto optimized CUDA kernels and Tensor Cores, TensorRT ensures efficient execution in mixed-precision inference. These optimizations significantly reduce inference latency, maximize GPU utilization, and enable real-time deployment of deep learning models in production environments.

3.2 Quantization Techniques

In our thesis work, we apply two popular methods for model quantization. One is Quantization Aware Training (QAT), which requires model training, and the other is Post Training Quantization (PTQ), which requires only a small amount of time for calibration.

3.2.1 Post Training Quantization

Unlike QAT, PTQ does not require fine-tuning to preserve accuracy. Instead, a small calibration dataset, typically a subset of the validation set, is used to compute scale values for each tensor in the network. This process begins by running inference on the calibration dataset using the pre-trained model in FP32, during which activation histograms are collected. These histograms capture the range and distribution of activations (layer outputs) and provide statistical insights necessary for quantization. Once the activation statistics are gathered, multiple quantized distributions are generated using different saturation thresholds. Activations are converted into INT8 format while experimenting with varying thresholds to determine an optimal balance between discretization (coarser representation) and truncation errors (clamping out-of-range values). The final scale value is selected based on this analysis, ensuring that quantization minimizes information loss while maintaining inference accuracy.

For calibration, this work employs *Entropy* calibration, the default and recommended strategy in TensorRT. Since calibration can be computationally expensive, TensorRT provides the capability to cache and reuse calibration tables, which significantly accelerates redeployment when running the same model on identical hardware. Beyond activations, TensorRT also quantizes weights using symmetric quantization or uniform scale quantization as discussed in section 2.3.1. The scale factor for weight tensors is determined based on their maximum absolute value, and for layers such as Convolution and FC layers, per-channel quantization is applied to preserve precision. Once calibration is complete, the pre-trained model is converted to ONNX format, from which TensorRT generates an INT8 inference engine. This engine is then used to perform quantized inference, allowing for the evaluation of model accuracy and computational performance in INT8 format. In addition to INT8 inference, this work also explores FP16 inference to assess performance trade-offs. Unlike INT8 quantization, FP16 inference does not require calibration since the model retains its original floating-point range. The entire PTQ workflow is illustrated in Figure 3.1, depicting the transition from FP32 to a fully quantized deployment-ready model in TensorRT.

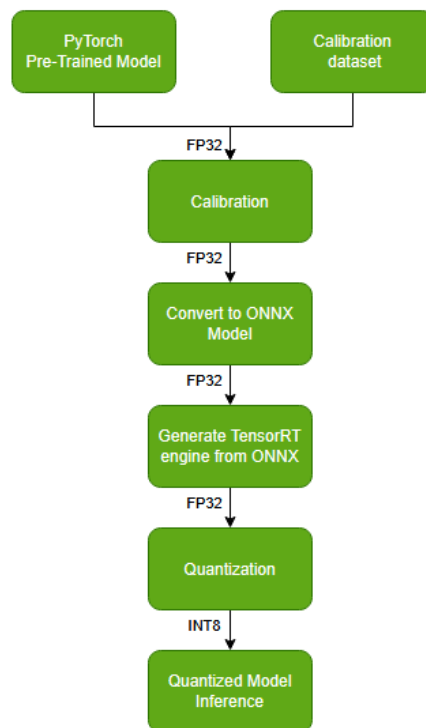


Figure 3.1: Post Training Quantization Workflow

3.2.2 Quantization Aware Training

QAT differs from PTQ by integrating quantization operations into the training phase itself. Instead of applying calibration after training, QAT simulates the effects of quantization within the forward pass, allowing the model to adapt to lower precision while mitigating accuracy degradation. This ensures that the network learns quantization-friendly representations, leading to improved performance during inference. The QAT process begins with a standard PyTorch pre-trained model, which is modified to introduce Quantize-Dequantize (QDQ) nodes, as illustrated in Figure 3.2. The left side of the figure represents the original PyTorch model, where all computations are performed in FP32. In contrast, the right side shows the QAT-modified version, where quantization nodes have been inserted. These QDQ nodes simulate INT8 quantization effects while keeping FP32 computation for backpropagation, enabling the model to adjust to quantization-induced errors.

This work utilizes the PyTorch-Quantization Toolkit [32], which provides APIs for training and evaluating quantized models. The process starts with *quant_modules.initialize()*, ensuring quantized modules replace their FP32 counterparts. We focus on quantizing convolution, FC, pooling, and residual layers, replacing them with their quantized equivalents—QuantConv2d [2.3], QuantLinear [2.3], QuantPooling, and QuantResidual. In the QAT-modified model, right side of Figure 3.2, the QuantConv2d module encapsulates standard Conv2d operations while adding quantization nodes around both inputs and weights. Weights and ac-

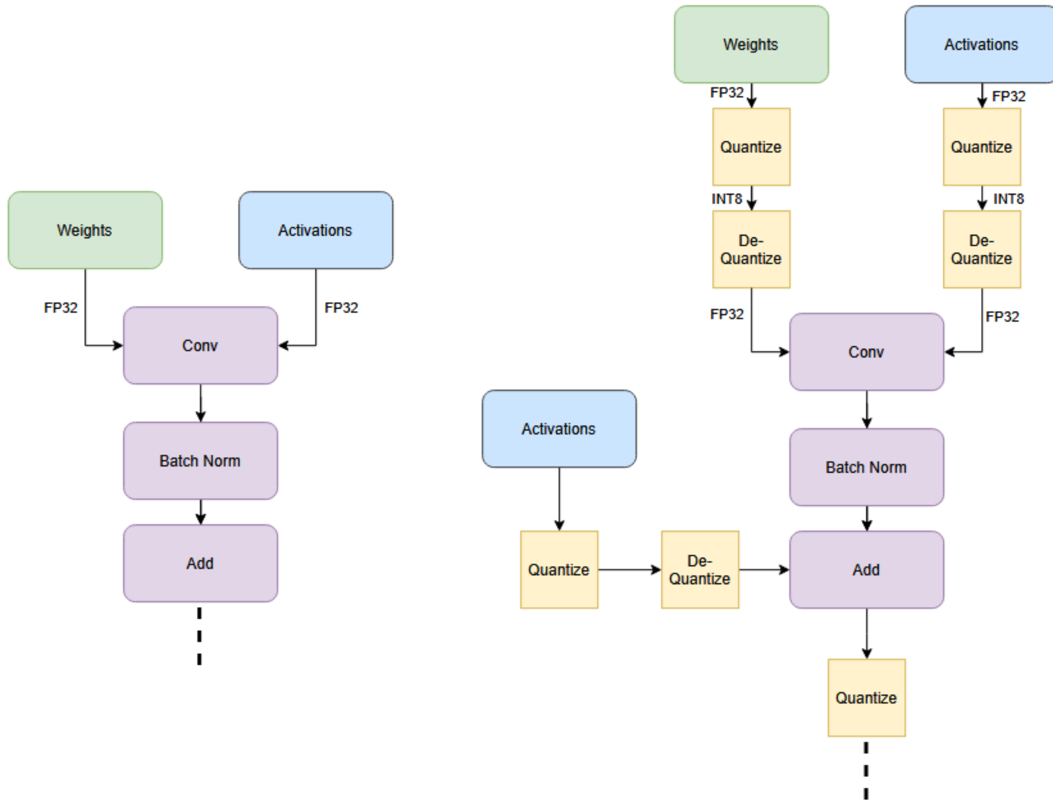


Figure 3.2: pre-trained model network before and after undergoing QAT.

tivations are quantized to INT8, used in computation, and then de-quantized back to FP32 before continuing through the network. Additionally, batch normalization layers are fused into convolution layers to stabilize activations and prevent discrepancies between training and inference.

Once fine-tuning is complete, the quantized model is exported to ONNX format, preserving the learned scale factors. The ONNX model is then processed using NVIDIA TensorRT, which generates an optimized INT8 inference engine. However, TensorRT does not support pre-quantized ONNX models with direct INT8 tensors or quantized operators. Instead, it expects models to use QuantizeLinear and DequantizeLinear operators for its own quantization optimizations. This limitation is particularly relevant for our gradient-based quantization method, where INT4-scale representation is used. Since TensorRT does not support INT4 computations, these models are evaluated in PyTorch instead. Finally, the optimized QAT-trained model is deployed for inference, leveraging hardware acceleration such as Tensor Cores. The QAT workflow in Figure 3.3 outlines the transition from pre-training, quantization-aware fine-tuning, ONNX conversion, TensorRT optimization, and final deployment.

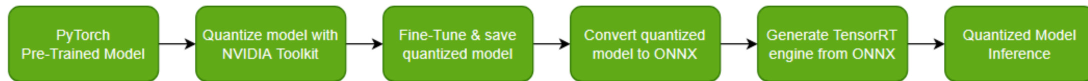


Figure 3.3: TensorRT deployment workflow for QAT models

3.2.3 Gradient-Based Quantization

Mixed-precision quantization involves assigning different bit-widths (2, 4, or 8 bits) to different layers of a neural network. This approach enables finer control over the trade-off between model compression and accuracy compared to uniform quantization, which applies the same precision (typically 8-bit) across all layers. The rationale is that not all layers contribute equally to the final output, layers that are more sensitive to quantization errors benefit from higher precision, while others can be quantized more aggressively. In this work, we implement a mixed-precision quantization framework guided by a gradient-based sensitivity analysis, inspired by the approach proposed in [30]. Similar to their method, we estimate layer-wise sensitivity using first-order gradient information, which is computationally efficient and scalable compared to second-order methods such as HAWQ [7], which rely on expensive Hessian eigenvalue computations. The procedure for estimating layer sensitivities is described in Algorithm 1. Using the resulting sensitivity scores, a constrained optimization strategy based on a Pareto frontier is applied to assign bit-widths across layers, subject to a model size constraint. This approach eliminates the need for manual tuning while ensuring minimal accuracy degradation. This optimization method is detailed in Algorithm 2. The proposed method is applied to ResNet50, MobileNetV2, and FCN-ResNet18 architectures. The resulting layer-wise bit-width configurations for 4MP and 2MP settings are presented and evaluated in Section 4.

3.2.3.1 Gradient-Based Sensitivity Analysis

To enable sensitivity-aware quantization, this thesis employs a perturbation-based sensitivity analysis method. Small Gaussian noise perturbations are introduced to the weights of each layer, and the impact on the loss function is measured via the L1 norm of the resulting first-order gradients.

Given a neural network with parameters $W = \{W_1, W_2, \ldots, W_L\}$, where W_l denotes the weights of the lth layer, a small perturbation δW_l is sampled as:

$$W_l' = W_l + \delta W_l, \quad \delta W_l \sim \mathcal{N}(0, \sigma^2), \quad \sigma = \frac{\epsilon}{\|W_l\|} \tag{3.1}$$

The scaling factor σ ensures uniform perturbation strength across layers of varying parameter scales. Sensitivity is quantified using the L1 norm of the gradient of the

loss with respect to the perturbed weights:

$$S_l = \left\|\nabla_{W'_l}\mathcal{L}\right\|_1 = \sum_{i=1}^{n}\left|\frac{\partial \mathcal{L}}{\partial W'^{i}_l}\right| \tag{3.2}$$

To reduce variance from individual perturbations, we average the sensitivity across N trials:

$$S_l = \frac{1}{N}\sum_{j=1}^{N}\left\|\nabla^{(j)}_{W'_l}\mathcal{L}\right\|_1 \tag{3.3}$$

The L1 norm is used for its simplicity and robustness to outliers; a higher L1 norm reflects greater sensitivity of the loss to weight perturbations, indicating that the corresponding layer should retain higher precision. Conversely, layers with lower L1 norms are less sensitive to perturbations and can be quantized more aggressively using lower precision.

Algorithm 1 Perturbation-Based Sensitivity Estimation

Input: Model $f(W)$, dataset D, loss function $\mathcal{L}$, perturbation scale ϵ, number of samples N
Output: Layer-wise sensitivities S_l, parameter sizes P_l
Set model to evaluation mode
for *each layer l in $f(W)$* **do**
  if *layer is Conv2D or Linear* **then**
    Save original weights W_l Compute parameter size $P_l \leftarrow \text{size}(W_l)$ Initialize $S_l \leftarrow 0$
    for *a batch (x, y) in D* **do**
      for $i \leftarrow 1$ **to** N **do**
        Apply perturbation: $W'_l = W_l + \delta W_l$, with $\delta W_l \sim \mathcal{N}(0, \epsilon/\|W_l\|)$ Compute loss $\mathcal{L} = \mathcal{L}(f(x; W'), y)$ Zero gradients and backpropagate $S_l \leftarrow S_l + \|\nabla_{W'_l}\mathcal{L}\|_1$ Restore W_l
      break
    Normalize: $S_l \leftarrow S_l/N$
return S_l, P_l

3.2.3.2 Bit-Width Allocation Using Pareto Frontier

Once sensitivity scores S_l are computed for each layer, a Pareto-frontier based approach is used to determine optimal bit-width allocation. This ensures that layers with higher sensitivity are assigned higher precision, while less sensitive layers are quantized more aggressively, minimizing accuracy loss under a given model size constraint [25].
The allocation is formulated as a constrained optimization problem:

$$\min_{\{b_\ell\}_\ell}\sum_{\ell} S_\ell \times \frac{1}{b_\ell}, \quad \text{subject to} \quad \sum_{\ell} P_\ell \times \frac{b_\ell}{8} \le M \tag{3.4}$$

Here, b_l is the bit-width assigned to layer l, S_l is its sensitivity score, P_l is the number of parameters, and M is the total model size constraint.
The Pareto frontier is used to select trade-offs where no other configuration achieves both lower accuracy loss and smaller model size. This automated strategy removes the need for manual bit-width tuning and supports precision-efficient quantization under deployment constraints.

Algorithm 2 Bit-Width Allocation with Pareto Optimization

Input: Sensitivities S_l, parameter sizes P_l, bit-width options B, target model size M
Output: Best and Pareto-optimal configurations
Initialize DP table with $dp[0][0] \leftarrow (0, \{\})$
for *each layer* l **do**
  for *each* $(s, config)$ *in previous table* **do**
    for *each* $b \in B$ **do**
      Compute size: $s' = s + P_l \cdot (b/8)$ **if** $s' \leq M$ **then**
        Compute loss: $L = S_l/b$ **if** s' *not seen or loss improves* **then**
          Store $(L, \text{updated config})$ in dp

Collect configurations from final DP table
Sort by total loss L
Normalize losses to percentiles
return configuration with lowest L and full Pareto frontier

3.3 Quantization Implementation

This section describes the quantization APIs and modifications required for implementing QAT with TensorRT and PyTorch-Quantization Toolkit. We discuss standard quantization modules, introduce custom modifications for mixed-precision quantization, and outline calibration and model export steps for deployment.

3.3.1 Quantization APIs

TensorRT explicit quantization outlines best practices for optimal placement of QDQ nodes, as their positioning plays a crucial role in model performance. It enables QAT-based deployment with 8-bit integer quantization, leveraging Tensor Cores for efficient execution. However, it does not support models trained with bit-widths lower than 8. Although the PyTorch-Quantization Toolkit provides APIs for custom quantization at lower bit-widths, it lacks direct compatibility for hardware deployment.

3.3.1.1 Initialization and Setup

The quantization setup begins by calling `quant_modules.initialize()`, which replaces standard FP32 modules in the model with their quantized counterparts. This

transformation can be reverted, if necessary, using `quant_modules.deactivate()`, thereby restoring the model to its original FP32 configuration.

3.3.1.2 Quantized Modules

We quantize layers such as convolution, FC, pooling, and residual layers. As an example, we consider the Conv2d layer. Listing 3.1 describes the default QuantConv2d[32] module provided by the toolkit, used for uniform-precision 8-bit per-channel quantization. However, customization is necessary for layers that are not natively supported, such as skip connections in ResNet-50 and inverted residual layers in MobileNetV2.

Listing 3.1: Default QuantConv2d (8-bit Per-Channel Quantization)

```
class QuantConv2d(_QuantConvNd):
    default_quant_desc_weight =
        tensor_quant.QUANT_DESC_8BIT_CONV2D_WEIGHT_PER_CHANNEL

    def __init__(self, in_channels, out_channels, kernel_size, stride=1,
        padding=0, ...):
        quant_desc_input, quant_desc_weight =
            _utils.pop_quant_desc_in_kwargs(self.__class__, **kwargs)
        super(QuantConv2d, self).__init__(in_channels, out_channels, kernel_size,
            stride, padding, ...,
                                  quant_desc_input=quant_desc_input,
                                      quant_desc_weight=quant_desc_weight)

    def forward(self, input):
        # Implicit quantization via _quant()
        quant_input, quant_weight = self._quant(input)
        return F.conv2d(quant_input, quant_weight, self.bias, self.stride,
            self.padding, self.dilation, self.groups)
```

Customization for Mixed-Precision Quantization: To support mixed-precision quantization, modifications are required to enable configurable bit-widths beyond the default uniform 8-bit quantization. Listing 3.2 presents a customized `QuantConv2d` module, referred to as `NewQuantConv2d`, designed to enable dynamic bit-width assignment. In this customized module, quantization is explicitly controlled using separate `TensorQuantizer` instances for inputs and weights. These are configured through `QuantDescriptor` definitions, replacing the default per-channel 8-bit quantization used in the original `QuantConv2d`. Unlike the default implementation, which automatically retrieves quantization descriptors, this customized version supports runtime updates via the `update_num_bits()` function, allowing flexible bit-width control per layer. This configurability is essential in mixed-precision settings, where different layers may benefit from varying levels of quantization precision. To integrate the bit-width configuration into the training workflow, each layer's precision is updated at runtime based on the layer-wise allocation produced by the Pareto-optimized sensitivity analysis described in Section 3.2.3. For each layer, the `update_num_bits()` function is called prior to training or fine-tuning, setting the input and weight precision according to its assigned bit-width. This enables a dynamic and flexible mixed-precision training environment using the toolkit, where layers operate with independent precision levels (e.g., 2-bit weights, 8-bit activations).

Listing 3.2: Custom QuantConv2d with bit-width configuration support

```
class NewQuantConv2d(nn.Conv2d):
    def __init__(self, in_channels, out_channels, kernel_size, stride=1,
        padding=0, ...):
        super(QuantConv2d, self).__init__(...)

        # Explicitly define quantizers
        self.input_quantizer = TensorQuantizer(QuantDescriptor(num_bits=8,
            calib_method='entropy', learn_amax=True))
        self.weight_quantizer = TensorQuantizer(QuantDescriptor(num_bits=8,
            calib_method='entropy', learn_amax=True))

    def update_num_bits(self, num_bits_input, num_bits_weight):
        # Allow dynamic bit-width updates
        self.input_quantizer.quant_desc = QuantDescriptor(num_bits=num_bits_input,
            ...)
        self.weight_quantizer.quant_desc =
            QuantDescriptor(num_bits=num_bits_weight, ...)
```

3.3.1.3 Calibration and Scale Computation

The calibration process involves two key steps: First, `compute_amax()` is used to load calibration results for all `TensorQuantizer` modules by determining the absolute maximum values of tensors. If a `MaxCalibrator` is used, it directly retrieves precomputed values; otherwise, additional calibration parameters may be applied to compute appropriate scale factors. Next, `collect_stats()` temporarily disables quantization and enables calibration mode to process a specified number of batches. During this phase, sample data is passed through the network to collect activation distributions necessary for accurate scale computation. Once the statistics are gathered, the function disables calibration mode and re-enables quantization, preparing the model for deployment.

3.3.1.4 Model Conversion and Export

After training with quantization-aware modifications, the model is converted to ONNX format, enabling deployment compatibility. `TensorQuantizer` is configured to use fake quantization by setting: `quant_nn.TensorQuantizer.use_fb_fake_quant = True`. The ONNX model is then optimized and converted into a TensorRT engine using the `trtexec` CLI tool, ensuring efficient INT8 execution on NVIDIA Tesla T4 GPU.

3.3.1.5 Model Inference

In the evaluation process, deserialization is performed by loading a precompiled TensorRT engine from disk using:
`trt.Runtime(TRT_LOGGER).deserialize_cuda_engine(f.read())`, allowing execution without re-compilation. Once deserialized, an execution context is created to manage dynamic batch sizes by setting the appropriate input shape. During inference, GPU memory is allocated for input and output tensors, and data is transferred using `cuda.memcpy_htod_async()` and `cuda.memcpy_dtoh_async()` for efficient communication. The inference is executed using `context.execute_v2()`, and latency, throughput, and accuracy are measured to assess model performance across

different batch sizes. Synchronization with `cuda.Stream()` ensures efficient GPU utilization, minimizing execution overhead.

4
Results

This chapter presents the experimental evaluation of quantized models for image classification and semantic segmentation. We compare FP32 baseline models with PTQ, QAT and mixed precision quantization to assess their impact on performance. The following subsections describe the experimental setup, results for image classification, and semantic segmentation, followed by a discussion of key findings.

4.1 Experimental Setup

4.1.1 Platform

The experimentation and inference were conducted on Volvo Cars' Data Science Platform, which provides a Jupyter Notebook server running a PyTorch-based Kubernetes pod hosted on an AWS EC2 instance with an NVIDIA Tesla T4 GPU. To leverage GPU parallelism, CUDA, a parallel computing platform designed for general-purpose GPU computing, was utilized. As discussed in Section 3.1, PyTorch was used for baseline model training, while NVIDIA's PyTorch-Quantization toolkit [32] enabled quantization-aware training. Furthermore, TensorRT8.4 was employed as the model inference engine.

4.1.2 Dataset

The ImageNet-1K dataset [21] is widely used for training deep learning models, particularly for **Image Classification** tasks. It contains 1,000 classes with 1.3 million images, making it a large-scale dataset. Due to its large size, a smaller subset was selected by choosing 100 classes from the original dataset, resulting in 50,000 training images and 10,000 validation images. This subset, known as Mini-ImageNet, is commonly used in few-shot learning and experiments requiring faster training.

The Cityscapes dataset [22], specifically developed for **Semantic Segmentation** tasks in urban environments, is selected for this study. It comprises 5,000 finely annotated and 20,000 coarsely annotated images collected from 50 cities. The dataset defines 19 evaluation classes, including roads, vehicles, pedestrians, and buildings. A standard subset consisting of 2,975 training images and 500 validation images is used to facilitate efficient model training and evaluation.

4.1.3 Experimental Models

The models used in image classification are ResNet50 [35] and MobileNetV2 [36]. ResNet50 is a deep convolutional neural network with a large number of parameters and layers, making it suitable for analyzing the behavior of quantization-aware training and mixed precision on high-capacity models. This model also includes residual skip connections, which aid in training deeper architectures.
In contrast, MobileNetV2 is a lightweight model optimized for efficiency and mobile deployment. These two models were selected to provide a comparative study on how quantization impacts both heavy and lightweight architectures. A summary of model characteristics such as number of convolutional layers, fully connected layers, total parameters, and FLOPs is presented in Table 4.1, further highlighting their structural differences.

Table 4.1: Model information of MobileNetV2 and ResNet50, including convolutional and fully connected layers

Model	**MobileNetV2**	**ResNet50**
Conv Layers	53	53
FC Layers	1	1
Total Parameters	3.4M	25.6M
FLOPs	0.3 GFLOPs	4.1 GFLOPs

For the semantic segmentation task, an FCN-ResNet18 model was implemented. While TensorRT's quickstart resources provide FCN-ResNet101 for segmentation, this model is computationally expensive and not well-suited for quantization-aware training under resource constraints. To address this, FCN-ResNet18 was constructed using a lightweight ResNet18 backbone, combined with the segmentation head architecture from FCN-ResNet101. This setup allowed for efficient experimentation while maintaining a fair approximation of the original segmentation pipeline.

4.1.4 Evaluation Metrics

To evaluate model performance and efficiency, particularly in the context of quantization, the following key metrics were used:

- **Top-1 Acc**: On the Mini-ImageNet dataset with 100 classes, Top-1 Accuracy measures the proportion of samples where the predicted class with the highest probability matches the ground truth label. It is defined as:

$$\textit{Top-1 Acc(\%)} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

- **mIoU**: Measures the average overlap between predicted and ground truth masks across all valid classes. It is calculated as:

$$mIoU(\%) = \frac{1}{C}\sum_{i=1}^{C}\frac{TP_i}{TP_i + FP_i + FN_i}$$

  where C is the number of valid classes, and TP_i, FP_i, and FN_i are the true positives, false positives, and false negatives for class i. In Cityscapes dataset,

C=19 and pixels labeled 255 (void) are ignored during training and evaluation.

- **Latency**: The time taken to process a single input, measured from the start to the end of inference. The average latency over N runs is:

$$\text{Latency (ms)} = \frac{1}{N}\sum_{i=1}^{N}(t_{\text{end},i} - t_{\text{start},i}) \times 1000$$

  where $t_{start,i}$and $t_{end,i}$ are the start and end times of run i.

- **Throughput**: The number of input images processed per second, calculated as:

$$\text{Throughput (img/s)} = \frac{\text{Total images}}{\text{Total inference time (s)}}$$

  When batch size = 1 and N=1, throughput is approximated as:

$$\text{Throughput (img/s)} \approx \frac{1000}{\text{Latency (ms)}}$$

- **Size (MB)**: Represents the storage size of the trained model in a disk.

- **Acc drop**: Quantization can reduce model accuracy. The drop in Top-1 Acc (classification) or mIoU (segmentation) is computed as the difference between the original model and quantized model accuracies
- **Compression Ratio**(CR): Quantifies how much the model has been compressed by comparing the original model size (S_{original}) to the quantized model size (S_{quant}):

$$\textit{Compression Ratio} = \frac{S_{\text{original}}}{S_{\text{quant}}}$$

- **Speed-Up**: Measures the improvement in inference speed due to quantization, defined as the ratio of latency of the original model to that of the quantized model:

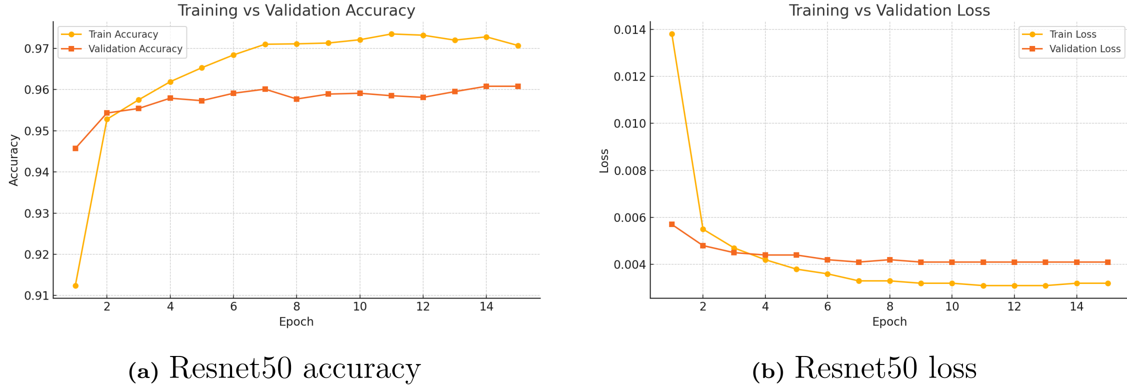
$$\textit{Speed-Up Factor} = \frac{\text{Latency}_{\text{original}}}{\text{Latency}_{\text{quant}}}$$

4.2 Quantization Results for Image Classification

4.2.1 Base Model Result

ResNet50 and MobileNetV2 were evaluated in full-precision on the Mini-ImageNet dataset using pretrained torchvision models adapted through transfer learning. The classification heads were modified to output 100 classes, and both models were fine-tuned for 15 epochs with cross-entropy loss function, SGD with learning rate 0.01, a cosine annealing scheduler. All experiments were conducted with a batch size of 1 to reflect real-time inference conditions, where inputs are processed individually rather than in batches. This setting allows for a fair comparison of model latency and

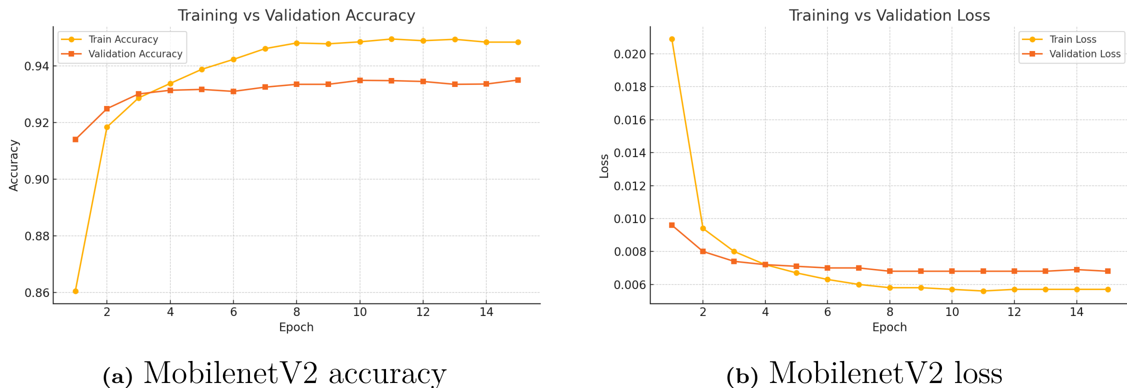
throughput in scenarios where low-latency responses are critical. Baseline inference metrics in table 4.2 shows that although ResNet50 attained higher accuracy, MobileNetV2 provided substantially lower latency and higher throughput, establishing efficient baselines for subsequent quantization experiments.



(a) Resnet50 accuracy

(b) Resnet50 loss

Figure 4.1: Training and validation accuracy and loss for ResNet50 over 15 epochs.



(a) MobilenetV2 accuracy

(b) MobilenetV2 loss

Figure 4.2: Training and validation accuracy and loss for MobileNetV2 over 15 epochs.

4.2.2 Evaluation of Quantization Methods

Upon establishing a baseline, experiments were conducted to explore various quantization methods, as shown in Table 4.3, involving different precision settings for weights and activations across layers. To facilitate a comparative analysis between uniform-precision and mixed-precision approaches, inference evaluations were performed using two distinct strategies. The first was an implicit quantization strategy, utilizing TensorRT's standard PTQ and QAT methods, where 8-bit integer precision for weights and activation was uniformly applied across all layers. The second was an explicit quantization strategy that leveraged gradient-based mixed-precision quantization with layer-wise varying precision. Table 4.2 presents the impact of various quantization methods on ResNet50 and MobileNetV2. Across both models, FP16 maintains high accuracy with minimal degradation and offers moderate

Table 4.2: Baseline performance comparison between ResNet50 and MobileNetV2 (batch size = 1).

Model	**Top-1 Acc(%)**	**Latency(ms)**	**Throughput(img/s)**	**Size(MB)**
ResNet50	96.10	12.57	79.57	95.19
MobileNetV2	93.50	6.21	163.48	10.17

compression. PTQ significantly reduces model size but leads to a considerable drop in accuracy, most notably 11.3% for MobileNetV2. In contrast, QAT substantially recovers the lost accuracy through fine-tuning over 10 epochs. To optimize performance, multiple experiments were conducted to identify the best combination of batch size, optimizer, and learning rate scheduler. A learning rate of 0.01 combined with a reduce-learning-rate-on-plateau scheduler proved effective in mitigating accuracy degradation from quantization.

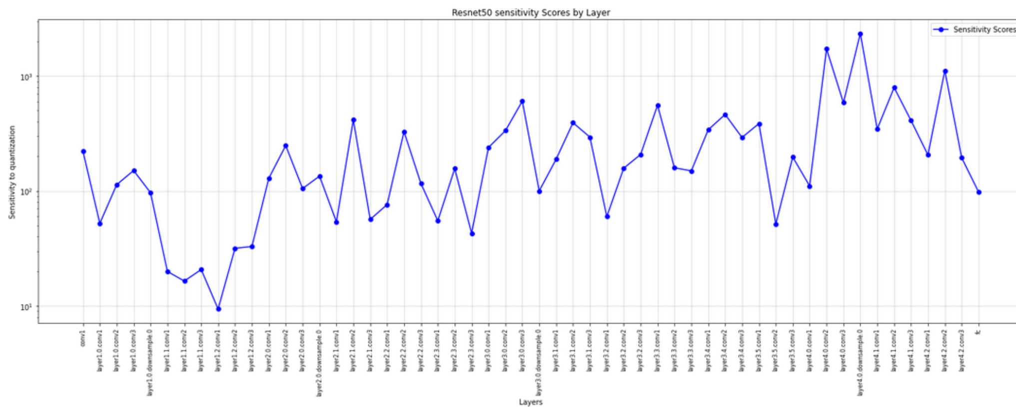
It is important to note that while convolutional, fully connected layers, and residual connections are quantized, TensorRT improves inference performance by folding batch normalization into the subsequent layer. This fusion not only reduces the number of separate operations performed at runtime, but also minimizes memory bandwidth usage and streamlines computations. Additionally, it combines element-wise additions with weighted layers, ensuring the output precision aligns with the first input's precision [31].

Table 4.3: Comparison of implicit quantization methods on ResNet50 and MobileNetV2: accuracy drop and compression ratio relative to full-precision baseline

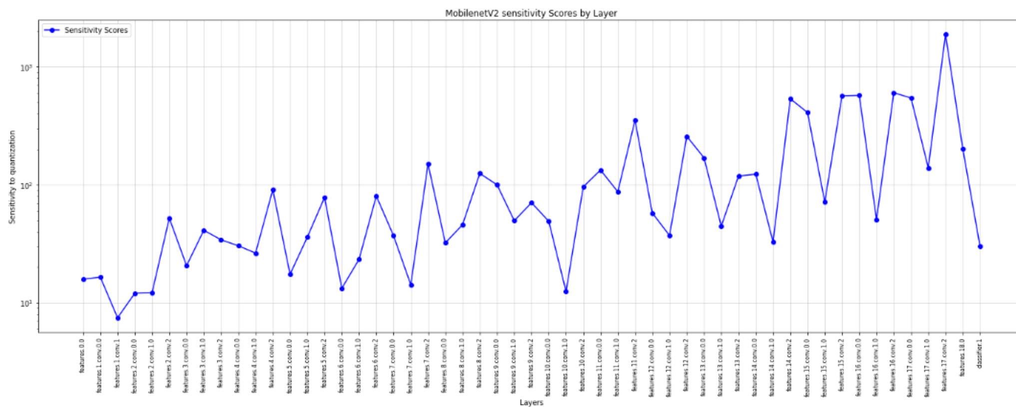
Model	**Method**	**Top-1 Acc(%)**	**Size(MB)**	**Acc drop(%)**	**CR**
ResNet50	FP16	96.02	47.89	0.08	1.99
	PTQ	87.86	25.41	8.24	3.74
	QAT	95.98	24.87	0.12	3.82
MobileNetV2	FP16	93.48	5.52	0.02	1.84
	PTQ	82.23	3.48	11.27	2.92
	QAT	91.41	3.19	2.09	3.18

To design effective mixed-precision configurations, gradient-based sensitivity analysis was conducted on ResNet50 and MobileNetV2. As shown in Figure 4.3, sensitivity scores vary across layers, with high-sensitivity layers requiring higher bit-widths (e.g., 8-bit or 4-bit) to preserve accuracy, while low-sensitivity layers can tolerate more aggressive quantization (e.g., 4-bit or 2-bit). For ResNet50, the initial convolutional layer and the final fully connected layer exhibited high quantization sensitivity, likely due to their critical role in encoding low-level features and pro-

ducing final class logits, respectively where information loss significantly impacts accuracy. Conversely, many of the intermediate residual blocks, especially those in the middle of the network, demonstrated greater robustness to reduced precision, allowing for compression to lower bit-widths such as 4-bit or 2-bit. In MobileNetV2, the expansion and projection layers located in the early and late stages showed higher sensitivity, which can be attributed to their higher information density and transformation capacity. On the other hand, the depthwise convolution layers in the bottleneck blocks are inherently lightweight and contribute less to representational capacity, making them more amenable to aggressive quantization (2-bit in the 2MP setting). These insights grounded in per-layer sensitivity analysis guided the bit-width allocation strategy in the 4MP and 2MP configurations, as shown in Table 4.4 and illustrated in Figures 4.4 and 4.5.



(a) ResNet50



(b) MobileNetV2

Figure 4.3: Layer-wise sensitivity analysis for ResNet50 and MobileNetV2, derived from gradient-based metrics. Scores guided bit-width allocation in mixed-precision configurations by identifying which layers could be aggressively quantized and which required preservation.

Table 4.4 presents the performance of both models under mixed-precision settings. Two weight configurations were explored—4MP (8- and 4-bit) and 2MP (8-, 4-, and 2-bit)—each paired with activation precisions of 8, 6, or 4 bits. All models

were trained for 10 epochs using quantization-aware training, following the same procedure as uniform 8-bit QAT. Experiments with 2-bit activations showed severe degradation and were therefore excluded. Layer-wise bit-widths were allocated using sensitivity scores and model size constraints, as defined in Algorithm 2. Entropy calibration was initially used to determine quantization scales, but for MobileNetV2, this method underperformed due to its skewed activation distributions. Instead, 99.9th percentile calibration was adopted. ResNet50, with more stable activations, continued to benefit from entropy calibration consistent with findings in [24].
For uniform INT8 models (Table 4.3), model size reflects on-disk footprint. For mixed-precision configurations (Table 4.4), size was estimated theoretically by summing each layer's parameter count multiplied by its assigned bit-width. This excludes activations and biases, and assumes weights are restored to FP32 at inference. Full W-bit and A-bit configurations are detailed in Table 4.4 and visualized in Figures 4.4 and 4.5.

Table 4.4: Comparison of mixed-precision QAT methods on ResNet50 and MobileNetV2. Accuracy drop and compression ratio are reported relative to the full-precision baseline. Since the bit-width of weights is fixed across configurations, model size and CR remain constant within each group.

Model	W-bit	A-bit	Top-1 Acc(%)	Size(MB)	Acc drop(%)	CR
ResNet50	4MP	8	95.42	18.76	0.68	5.07
		6	95.33	18.76	0.77	5.07
		4	90.10	18.76	6.0	5.07
	2MP	8	92.86	15.01	3.24	6.34
		6	92.52	15.01	3.58	6.34
		4	84.33	15.01	11.77	6.34
MobileNetV2	4MP	8	91.42	1.97	2.08	5.16
		6	86.73	1.97	6.77	5.16
		4	22.18	1.97	71.32	5.16
	2MP	8	87.67	1.81	5.83	5.61
		6	80.19	1.81	13.31	5.61
		4	14.81	1.81	78.69	5.61

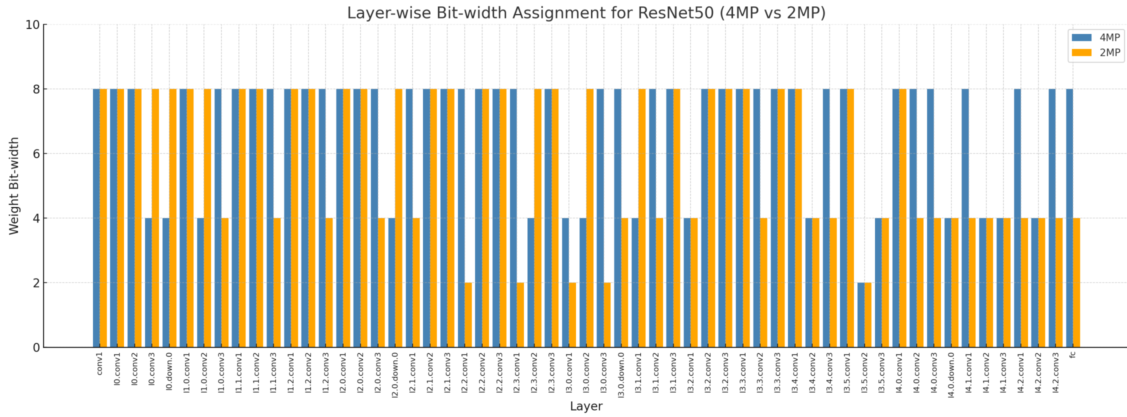


Figure 4.4: Layer wise bit-width allocation of weights (W-bit), for ResNet50 - 4MP vs 2MP configuration.

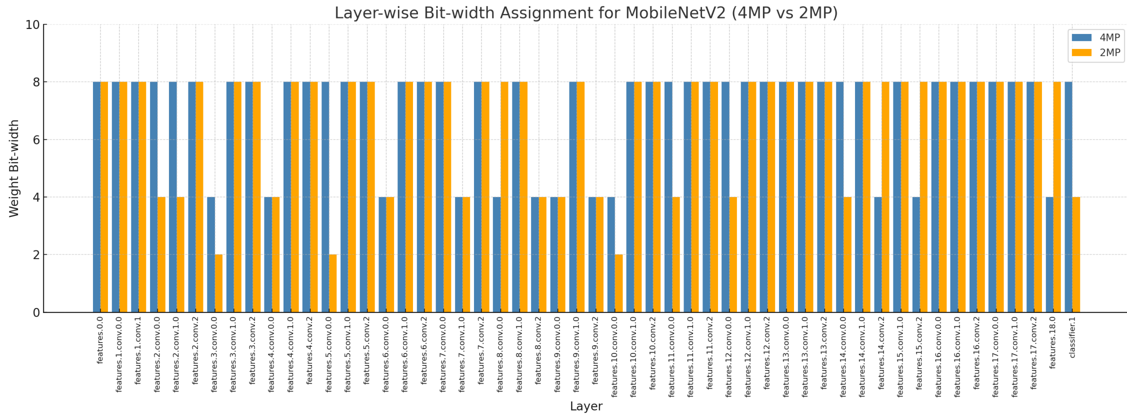


Figure 4.5: Layer wise bit-width allocation of weights (W-bit), for MobileNetV2 - 4MP vs 2MP configuration.

Table 4.5: Comparison of implicit quantization methods on ResNet50 and MobileNetV2 (batch size = 1), showing latency, throughput, and speed-up relative to the full-precision baseline.

Model	Method	Latency(ms)	Throughput(img/s)	Speed-up
ResNet50	FP16	6.56	152.37	1.91
	PTQ	4.46	224.04	2.81
	QAT	4.64	215.38	2.70
MobileNetV2	FP16	2.51	398.76	2.47
	PTQ	1.31	762.17	4.74
	QAT	2.03	492.96	3.05

4.3 Quantization Results for Segmentation

4.3.1 Base Model Result

The FCN-ResNet18 architecture was adopted as a baseline for semantic segmentation on the Cityscapes dataset. It was constructed by replacing the default ResNet101 backbone in the predefined FCN-ResNet101 model [35] with a lightweight ResNet-18 backbone. Feature maps were extracted from the layer4 block, and a custom classifier was designed to accommodate ResNet-18's 512 output channels, predicting 19 Cityscapes classes. To address the dataset's class imbalance, the model was trained using weighted cross-entropy loss, with higher weights assigned to underrepresented classes. The model was trained on images resized to 512×1024, and optimized with Adam with learning rate at 0.0003 and batch size of 8. Although training was scheduled for 50 epochs, early stopping was triggered at epoch 37 as validation accuracy and IoU stabilized. Figure 4.6 illustrate the training dynamics, showing validation pixel accuracy steadily improving to 92.3% and validation mIoU reaching 58.3%. Table 4.6 summarizes the baseline full-precision inference performance of FCN-ResNet18 on an input image resolution of 1024×2048, achieving a latency of 18.99ms and throughput of 52.64 with a batch size of 1. These baseline results serve as a reference point for evaluating the impact of subsequent quantization methods.

(a) Accuracy (b) Loss (c) IoU

Figure 4.6: Training and validation metrics (accuracy, loss, and IoU) for FCN-ResNet18 over 37 epochs.

Table 4.6: Baseline performance of FCN-ResNet18 on Cityscapes with full precision, on input size 1024×2048 and batch size = 1

Model	mIoU(%)	Latency(ms)	Throughput(img/s)	Size(MB)
FCN-ResNet18	58.26	18.99	52.64	82.66

4.3.2 Evaluation of Quantization Methods

Similar to the classification models, FCN-ResNet18 was evaluated using TensorRT with implicit quantization. As shown in Table 4.7, QAT was fine tuned for 5 epochs with a learning rate of 0.0001, it maintained segmentation quality with minimal degradation while providing substantial model compression. In contrast, PTQ led

to a noticeable decline in both pixel accuracy and mIoU, despite achieving a similar level of compression. Figures 4.9 and 4.10 visually confirm these differences: FP32 and QAT outputs maintain fine object boundaries, while PTQ introduces visible distortions, particularly around small or thin structures that are underrepresented in the Cityscapes dataset. These distortions result from quantization noise and the absence of retraining. Overall, QAT proves more effective, offering high compression with minimal loss in segmentation quality.

Table 4.7: Comparison of implicit quantization methods on FCN-ResNet18: accuracy drop and compression ratio relative to full-precision baseline.

Model	Method	mIoU(%)	Size(MB)	Acc drop(%)	CR
FCN-ResNet18	FP16	58.26	25.06	0.00	3.29
	PTQ	43.98	12.82	14.28	6.44
	QAT	57.99	12.66	0.27	6.52

To guide mixed-precision quantization for semantic segmentation, gradient-based sensitivity analysis was performed on FCN-ResNet18. As shown in Figure 4.7, the early backbone layers exhibit the highest sensitivity and were assigned higher precision (e.g., 8-bit or 4-bit) to preserve spatial features critical for dense pixel-wise prediction. In contrast, both intermediate residual blocks and the final classifier layer demonstrated lower sensitivity and were quantized more aggressively. This behavior reflects the reliance of semantic segmentation on accurate low-level feature encoding, while later layers, operating on semantically rich representations, are more tolerant to reduced precision. These insights guided the bit-width allocation in the 4MP and 2MP configurations evaluated in Table 4.8 and visualized in Figure 4.8.

Figure 4.7: Layer-wise sensitivity analysis of FCN-ResNet18, derived from first-order gradient-based metrics. Peaks correspond to layers that are more sensitive to quantization noise and were preserved at higher precision during bit allocation.

Qualitative segmentation outputs for the mixed-precision configurations are shown in Figures 4.11 and 4.12, illustrating that lower activation precision leads to noticeable visual degradation particularly in fine object boundaries and class separations.

These trends are quantitatively supported by the results in Table 4.8, which show a minimal performance drop when reducing activations from 8 to 6 bits, but a substantial degradation at 4-bit precision across both 4MP and 2MP configurations. Activation precision plays a critical role in preserving performance, particularly in dense prediction tasks where spatial detail is essential. These observations are consistent with prior work in object detection [25], further reinforcing the importance of maintaining adequate activation precision. Table 4.9 compares the inference performance of FCN-ResNet18 under different quantization strategies. PTQ achieves the highest speed-up, with the lowest latency and highest throughput. QAT also improves performance, achieving 9.40 ms latency and 106.27 FPS, corresponding to a 2.02× speed-up. While PTQ offers greater acceleration, it compromises segmentation quality, whereas QAT provides a better balance between accuracy and efficiency.

Table 4.8: Comparison of mixed-precision QAT methods on FCN-ResNet18. Accuracy drop and compression ratio are reported relative to the full-precision baseline. Since the bit-width of weights is fixed across configurations, model size and CR remain constant within each group.

Model	W-bit	A-bit	mIoU(%)	Size(MB)	Acc drop(%)	CR
FCN-ResNet18	4MP	8	57.52	10.53	0.74	7.84
		6	57.29	10.53	0.97	7.84
		4	47.74	10.53	10.52	7.84
	2MP	8	56.20	9.56	2.06	8.64
		6	54.05	9.56	4.21	8.64
		4	46.30	9.56	11.96	8.64

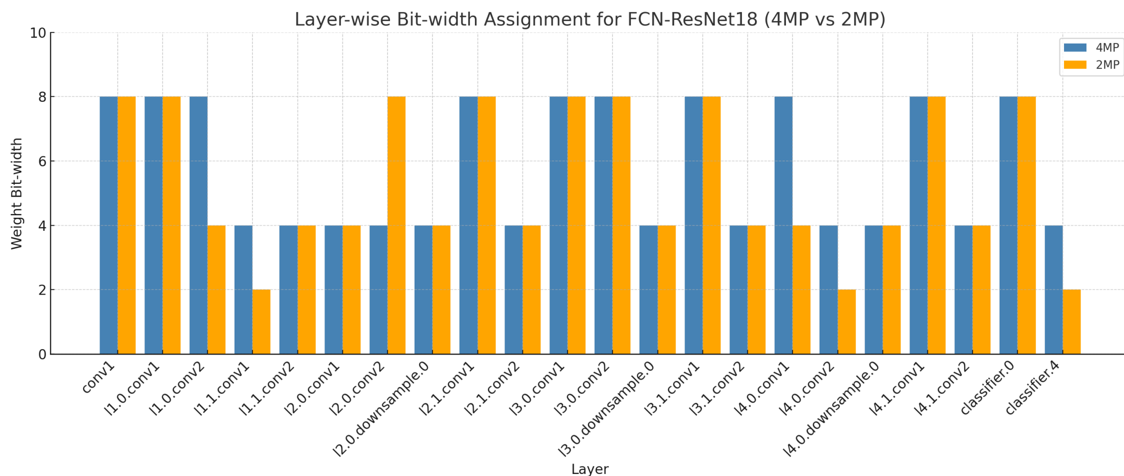


Figure 4.8: Layer wise bit-width allocation of weights (W-bit), for FCN-ResNet18 - 4MP vs 2MP configuration.

Table 4.9: Comparison of implicit quantization methods on FCN-ResNet18 (batch size = 1), showing latency, throughput, and speed-up relative to the full-precision baseline.

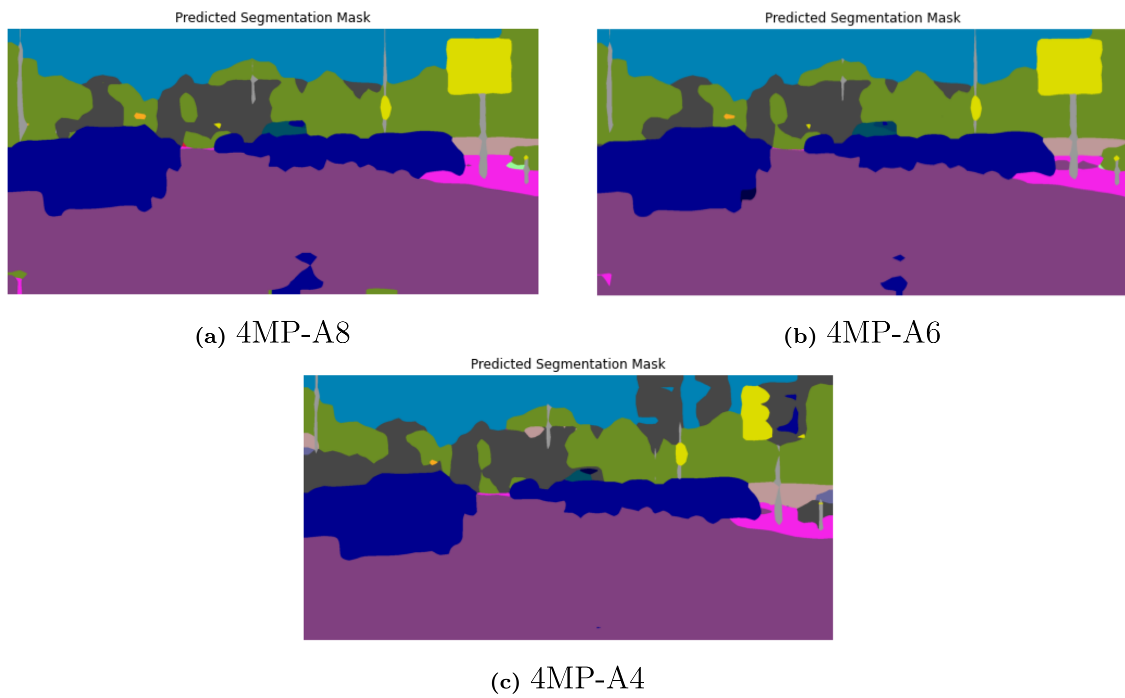
Model	Method	Latency(ms)	Throughput(img/s)	Speed-up
FCN-ResNet18	FP16	11.17	89.47	1.70
	PTQ	8.08	123.73	2.35
	QAT	9.40	106.27	2.02

Figure 4.9: FCN-ResNet18-FP32

(a) PTQ

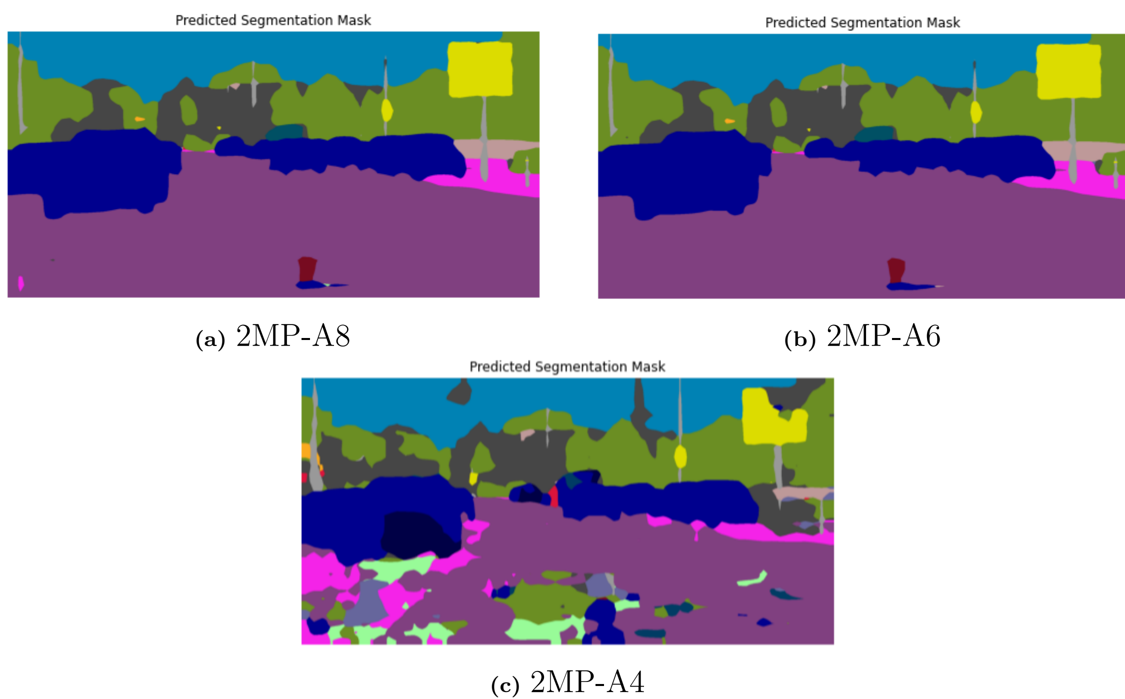
(b) QAT

Figure 4.10: Predicted segmentation outputs of FCN-ResNet18 on PTQ and QAT quantization methods.



(a) 4MP-A8 (b) 4MP-A6 (c) 4MP-A4

Figure 4.11: Predicted segmentation outputs for FCN-ResNet18 under 4MP-A8, 4MP-A6, and 4MP-A4 configurations.



(a) 2MP-A8 (b) 2MP-A6 (c) 2MP-A4

Figure 4.12: Predicted segmentation outputs for FCN-ResNet18 under 2MP-A8, 2MP-A6, and 2MP-A4 configurations.

4.4 Evaluation of Quantization Trade-offs

MobileNetV2 shows high sensitivity to quantization under 4MP-A4 and 2MP-A4 settings (Figure 4.13). Although ReLU6 helps by bounding activation values, the clipping at 6 can restrict expressiveness and amplify quantization errors at lower bit-widths. In contrast, FCN-ResNet18 performs more robustly in mixed-precision settings, maintaining accuracy more effectively. However, reducing activation precision below 6 bits leads to noticeable degradation in segmentation performance, likely due to the loss of fine-grained spatial information critical to semantic segmentation tasks. Among all configurations, 2MP-A8 consistently offers a favorable trade-off between accuracy retention and compression across all models.

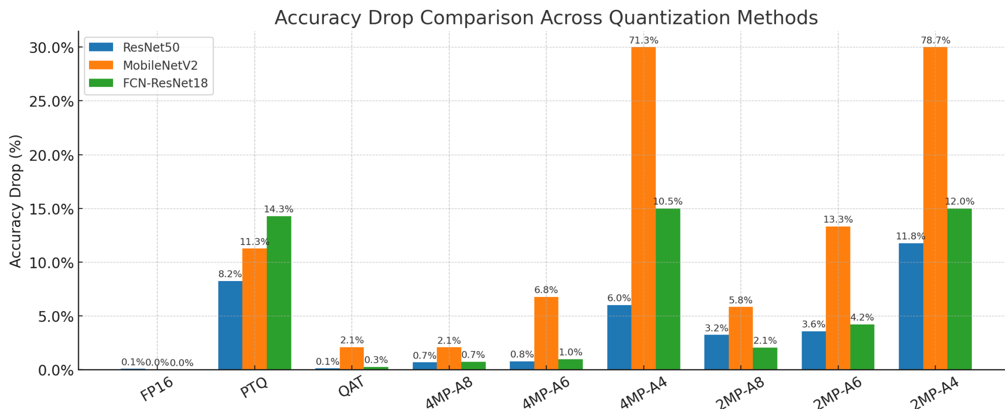


Figure 4.13: Accuracy drop for ResNet50, MobileNetV2, and FCN-ResNet18 across FP16, PTQ, QAT, and mixed-precision (4MP, 2MP) quantization methods.

FCN-ResNet18 achieves the highest compression ratio—up to 8.64×—particularly with 2MP configurations (Figure 4.14). This is attributed to its smaller baseline model size, concentration of weights in fewer layers, and a lightweight decoder that tolerates aggressive quantization. ResNet50 and MobileNetV2 show steady but comparatively lower gains. Across all models, mixed-precision quantization (MP-QAT) significantly outperforms uniform precision approaches like PTQ and QAT.
MobileNetV2 consistently achieves the highest inference speed-up across all quantization methods as shown in figure 4.16. This is due to its lightweight architecture, characterized by depthwise separable convolutions and a significantly lower parameter count compared to ResNet50 and FCN-ResNet18. While all models benefit from quantization in terms of throughput, the structural efficiency of MobileNetV2 allows it to capitalize more effectively on low-precision operations. PTQ remains the most effective in accelerating inference, followed by QAT and FP16.

4.4.1 Discussion

This section discusses the key findings of the study in light of the research questions posed in Section 1.2. The experiments focused on image classification using MobileNetV2 and ResNet50, and semantic segmentation using FCN-ResNet18, applying quantization as a model compression strategy to improve inference efficiency. The observations drawn from the results help clarify the trade-offs involved and the

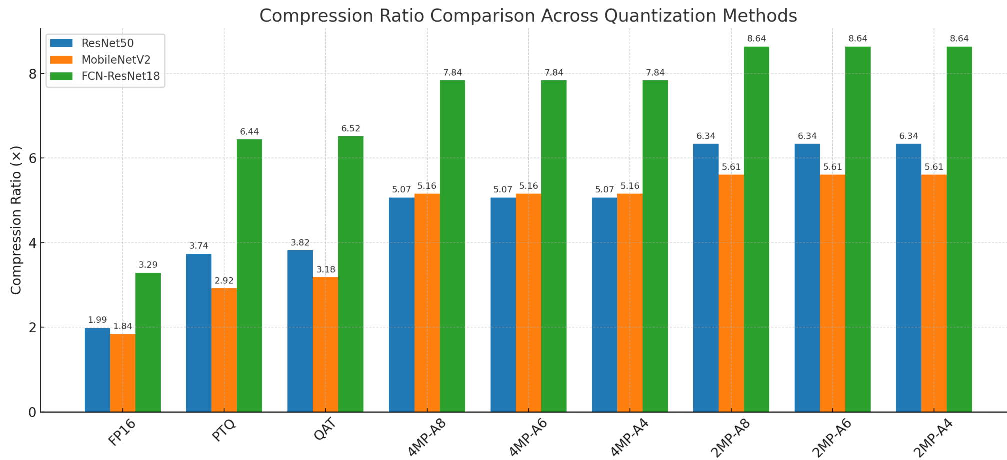


Figure 4.14: Compression ratio comparison for ResNet50, MobileNetV2, and FCN-ResNet18 across FP16, PTQ, QAT, and mixed-precision (4MP, 2MP) quantization methods.

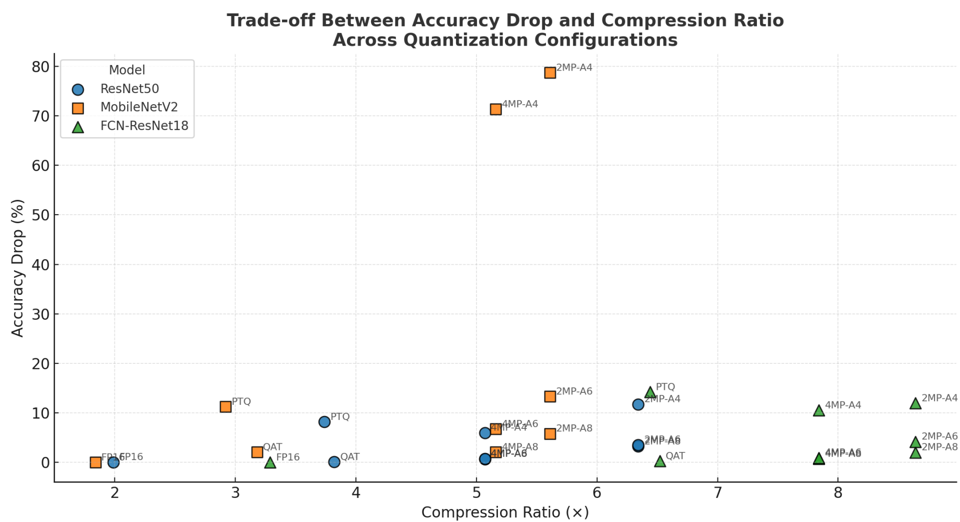


Figure 4.15: Trade-off between accuracy drop and compression ratio across quantization configurations for ResNet50, MobileNetV2, and FCN-ResNet18. Configurations such as 2MP-A8 and 4MP-A8 demonstrate favorable compression with minimal accuracy loss. MobileNetV2 exhibits high sensitivity under low-bit configurations (e.g., 2MP-A4), while FCN-ResNet18 maintains robustness even under aggressive compression.

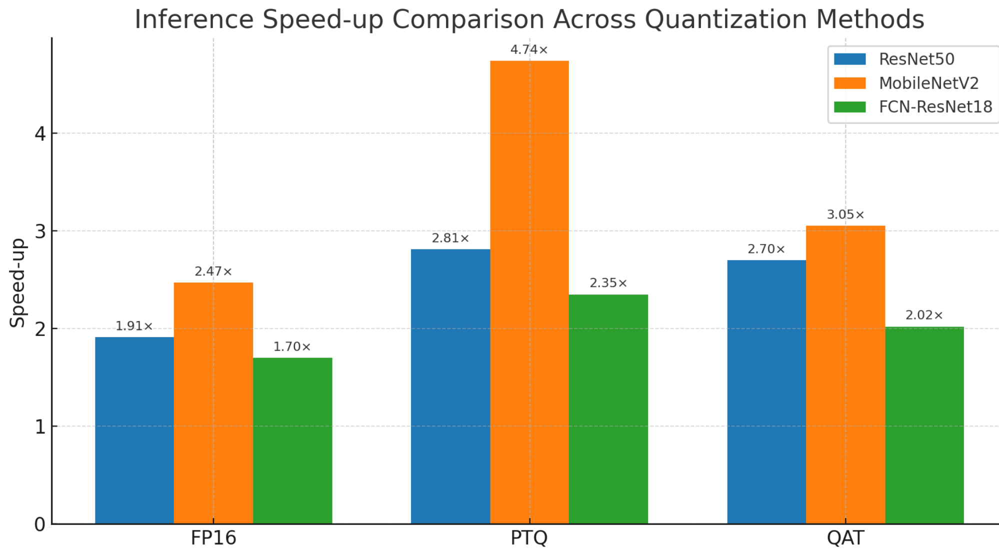


Figure 4.16: Inference speed-up comparison for ResNet50, MobileNetV2, and FCN-ResNet18 across FP16, PTQ, and QAT.

practical viability of different quantization methods.

RQ1: What is a desirable compression technique to achieve an efficient inference with reduced model size and fast inference?

Among the evaluated techniques, QAT emerges as the most desirable strategy across all models. In classification tasks, both ResNet50 and MobileNetV2 preserved accuracy while achieving notable reductions in model size and inference latency through QAT. This was particularly critical for MobileNetV2, where PTQ led to substantial performance degradation under low-bit configurations. In contrast, QAT maintained robustness even with constrained precision. For semantic segmentation, FCN-ResNet18 also retained high mIoU scores with QAT, confirming its effectiveness across vision tasks. However, mixed-precision configurations like 2MP-A8 offer a highly practical alternative. As shown in Figure 4.15, which visualizes the trade-off between compression ratio and accuracy drop, 2MP-A8 achieves compression ratios of up to 8.64× with only a modest reduction in accuracy. This makes it particularly attractive for deployment scenarios where memory footprint or model size is a primary constraint. While QAT ensures minimal accuracy loss, the choice between QAT and 2MP-A8 ultimately depends on an application's tolerance for degradation versus its need for compactness. In summary, QAT provides the most reliable performance when accuracy is critical, whereas 2MP-A8 offers an efficient trade-off for resource-constrained environments.

RQ2: How does the compression algorithm perform according to the objective determined in RQ1?

The performance of each quantization method varied depending on the model architecture and the specific task. PTQ proved useful in scenarios where retraining is

not feasible such as with proprietary models or when computational resources are limited. This was evident in MobileNetV2 and ResNet50, where PTQ enabled faster inference and reduced model size. However, this came at the cost of reduced accuracy, particularly in MobileNetV2, which is more sensitive to aggressive quantization. In contrast, QAT demonstrated greater adaptability across both classification and segmentation tasks. For instance, in FCN-ResNet18, QAT preserved segmentation quality while still achieving notable improvements in efficiency. Additionally, mixed-precision quantization guided by gradient based sensitivity analysis proved effective in optimizing both performance and resource usage. Assigning higher precision to more sensitive layers and lower precision to robust ones allowed significant improvements in MobileNetV2 and FCN-ResNet18 without uniform performance degradation. This layer-wise, sensitivity aware strategy highlights the effectiveness of mixed-precision approaches when tailored to the specific requirements of the deployment environment.

RQ3: What is the acceptable limit for quantization of neural networks for efficient inference without much loss in accuracy?

The results show that the impact of quantization on accuracy depends on how weights and activations are jointly compressed. While 6-bit activations often maintained performance when paired with moderately quantized weights, aggressive configurations like 2MP-A4 led to significant degradation especially in MobileNetV2 and FCN-ResNet18. As illustrated in Figure 4.15, accuracy drops sharply when both weights and activations are pushed to lower bit-widths. In contrast, configurations like 2MP-A8 achieve high compression with minimal accuracy loss. Thus, instead of a fixed threshold, acceptable quantization limits depend on the balance between weight and activation precision, and should be guided by layer-wise sensitivity.

4.5 Limitations and Future Directions

While the findings demonstrate the effectiveness of quantization strategies in improving inference latency and reducing model size in deep neural networks, several limitations should be considered. All experiments were conducted on a cloud-based NVIDIA Tesla T4 GPU. Although this platform supports hardware-accelerated inference, it does not fully represent the constraints and variability encountered in real-world edge deployment scenarios, such as on mobile or embedded devices. The mixed-precision quantization approach in this study employed layer-wise bit-width assignment based on first order gradient based sensitivity analysis. While this method is computationally efficient, it may not fully capture interdependencies between layers that influence model robustness under quantization. Although improvements in inference speed and model size were observed, power consumption and energy efficiency were not directly measured. Future work could address these gaps by incorporating energy profiling and evaluating performance across a broader range of hardware platforms. Additionally, more advanced sensitivity metrics such as second-order derivative based methods or data-driven profiling may further improve precision assignment. It is also important to recognize that quantization, if

applied without sufficient understanding of the system requirements, can lead to reduced model reliability, unintended biases, or incorrect outputs. Therefore, engineers must consider the safety and accuracy requirements of the target application when selecting a quantization method and deciding the extent of bit precision. Striking the right balance between computational efficiency and result fidelity remains a key responsibility in the deployment of quantized neural networks. Overall, this work provides a strong foundation for deploying quantized models on edge hardware that supports integer arithmetic, enabling efficient inference in resource-constrained environments.

5

Conclusion

In this work, low-bit quantization techniques were explored to improve the efficiency of deep neural networks for image classification and semantic segmentation tasks. MobileNetV2 and ResNet50 were used for classification on Mini-Imagenet dataset, while FCN-ResNet18 was used for segmentation on the Cityscapes dataset. The baseline models were first trained using standard 32-bit floating point precision, and then evaluated under various quantization strategies. PTQ and QAT were applied as standard 8-bit quantization methods. The models were further evaluated using low-bit mixed precision quantization, where layer-wise first-order sensitivity analysis was used to guide precision assignment. Several bit-width combinations (2, 4, 6, and 8) were tested and compared against the uniform 8-bit PTQ and QAT configurations. The results showed that mixed precision quantization achieved comparable performance to 8-bit QAT, while offering greater compression. However, aggressive quantization of activations to 4 bits led to significant accuracy degradation, particularly in MobileNetV2 and FCN-ResNet18. Sensitivity-based bit assignment proved more effective than uniform fixed-precision allocation, but the overall accuracy differences were marginal. Although the experiments were conducted on a cloud-based Tesla T4 GPU, the findings provide a strong foundation for deploying quantized models on edge hardware that supports integer-based computation, such as mobile devices, embedded systems, and FPGAs.

Bibliography

[1] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," 2020, arXiv:2004.09602. [Online]. Available: https://arxiv.org/abs/2004.09602

[2] M. P. Véstias, "A survey of convolutional neural networks on edge with reconfigurable computing," *Algorithms*, vol. 12, no. 8, 2019. [Online]. Available: https://www.mdpi.com/1999-4893/12/8/154

[3] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and quantization for deep neural network acceleration: A survey," 2021. [Online]. Available: https://arxiv.org/abs/2101.09671

[4] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," 2021. [Online]. Available: https://arxiv.org/abs/2103.13630

[5] I. Hubara, D. Soudry, and R. E. Yaniv, "Binarized neural networks," 2016. [Online]. Available: https://arxiv.org/abs/1602.02505

[6] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," 2016. [Online]. Available: https://arxiv.org/abs/1603.05279

[7] Z. Dong, Z. Yao, A. Gholami, M. Mahoney, and K. Keutzer, "Hawq: Hessian aware quantization of neural networks with mixed-precision," 2019. [Online]. Available: https://arxiv.org/abs/1905.03696

[8] M. C. Nwadiugwu, "Neural networks, artificial intelligence and the computational brain," 2020. [Online]. Available: https://arxiv.org/abs/2101.08635

[9] A. F. Agarap, "Deep learning using rectified linear units (relu)," 2019. [Online]. Available: https://arxiv.org/abs/1803.08375

[10] S. R. Dubey, S. K. Singh, and B. B. Chaudhuri, "Activation functions in deep learning: A comprehensive survey and benchmark," 2022. [Online]. Available: https://arxiv.org/abs/2109.14545

[11] J. Patterson and A. Gibson, *Deep Learning: A Practitioner's Approach*, 1st ed. O'Reilly Media, Inc., 2017.

[12] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning ap-

plied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[13] A. Lahoti, S. Karp, E. Winston, A. Singh, and Y. Li, "Role of locality and weight sharing in image-based tasks: A sample complexity separation between cnns, lcns, and fcns," 2024. [Online]. Available: https://arxiv.org/abs/2403.15707

[14] R. Gens and P. M. Domingos, "Deep symmetry networks," in *Neural Information Processing Systems*, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:267009

[15] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," 2013. [Online]. Available: https://arxiv.org/abs/1311.2901

[16] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[17] V. Nair and G. Hinton, "Rectified linear units improve restricted boltzmann machines vinod nair," vol. 27, 06 2010, pp. 807–814.

[18] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," 01 2010, pp. 92–101.

[19] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" 2014. [Online]. Available: https://arxiv.org/abs/1411.1792

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, pp. 84 – 90, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:195908774

[21] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[22] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," 2016. [Online]. Available: https://arxiv.org/abs/1604.01685

[23] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018. [Online]. Available: https://arxiv.org/abs/1806.08342

[24] S. Migacz, "Nvidia 8-bit inference with tensorrt," in *GPU Technology Conference*, 2017.

[25] Z. Dong, Z. Yao, Y. Cai, D. Arfeen, A. Gholami, M. W. Mahoney, and K. Keutzer, "Hawq-v2: Hessian aware trace-weighted quantization of neural networks," 2019. [Online]. Available: https://arxiv.org/abs/1911.03852

[26] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, 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 (CVPR)*, 2018, pp. 2704–2713. [Online]. Available: https://doi.org/10.1109/CVPR.2018.00286

[27] L. Wei, M. Zhong, C. Yang, and Q. Yao, "Advances in the neural network quantization: A comprehensive review," *Applied Sciences*, vol. 14, p. 7445, 08 2024.

[28] Y. Cai, Z. Yao, Z. Dong, A. Gholami, M. W. Mahoney, and K. Keutzer, "Zeroq: A novel zero shot quantization framework," 2020. [Online]. Available: https://arxiv.org/abs/2001.00281

[29] S. Kundu, S. Wang, Q. Sun, P. A. Beerel, and M. Pedram, "Bmpq: Bit-gradient sensitivity driven mixed-precision quantization of dnns from scratch," 2021. [Online]. Available: https://arxiv.org/abs/2112.13843

[30] A. Chauhan, U. Tiwari, and V. N. R, " Post Training Mixed Precision Quantization of Neural Networks using First-Order Information ," in *2023 IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*. IEEE Computer Society, 2023, pp. 1335–1344. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICCVW60793.2023.00144

[31] NVIDIA, "NVIDIA TensorRT Documentation," 2022. [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-843/developer-guide/index.html

[32] NVIDIA, "NVIDIA PyTorch-Quantization Toolkit," 2022. [Online]. Available: https://github.com/NVIDIA/TensorRT/tree/main/tools/pytorch-quantization

[33] NVIDIA, "Nvidia pytorch-quantization toolkit documentation - userguide," 2022. [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/pytorch-quantization-toolkit/docs/index.html#document-userguide

[34] PyTorch, "PyTorch Quantization," 2022. [Online]. Available: https://pytorch.org/docs/stable/quantization.html

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[36] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2019. [Online]. Available: https://arxiv.org/abs/1801.04381