Article

Deep Learning Performance Characterization on GPUs for Various Quantization Frameworks

Muhammad Ali Shafique [1], Arslan Munir [2,*] and Joonho Kong [3]

[1] Department of Electrical and Computer Engineering, Kansas State University, Manhattan, KS 66506, USA; alishafique@ksu.edu
[2] Department of Computer Science, Kansas State University, Manhattan, KS 66506, USA
[3] School of Electronic and Electrical Engineering, Kyungpook National University, Daegu 41566, Republic of Korea; joonho.kong@knu.ac.kr
* Correspondence: amunir@ksu.edu

Abstract: Deep learning is employed in many applications, such as computer vision, natural language processing, robotics, and recommender systems. Large and complex neural networks lead to high accuracy; however, they adversely affect many aspects of deep learning performance, such as training time, latency, throughput, energy consumption, and memory usage in the training and inference stages. To solve these challenges, various optimization techniques and frameworks have been developed for the efficient performance of deep learning models in the training and inference stages. Although optimization techniques such as quantization have been studied thoroughly in the past, less work has been done to study the performance of frameworks that provide quantization techniques. In this paper, we have used different performance metrics to study the performance of various quantization frameworks, including TensorFlow automatic mixed precision and TensorRT. These performance metrics include training time and memory utilization in the training stage along with latency and throughput for graphics processing units (GPUs) in the inference stage. We have applied the automatic mixed precision (AMP) technique during the training stage using the TensorFlow framework, while for inference we have utilized the TensorRT framework for the post-training quantization technique using the TensorFlow TensorRT (TF-TRT) application programming interface (API). We performed model profiling for different deep learning models, datasets, image sizes, and batch sizes for both the training and inference stages, the results of which can help developers and researchers to devise and deploy efficient deep learning models for GPUs.

Keywords: optimization; deep learning; quantization; performance; TensorRT; automatic mixed precision

Citation: Shafique, M.A.; Munir, A.; Kong, J. Deep Learning Performance Characterization on GPUs for Various Quantization Frameworks. *AI* **2023**, *4*, 926–948. https://doi.org/10.3390/ai4040047

Academic Editors: Gianni D'Angelo and Walter Richardson

Received: 26 August 2023
Revised: 27 September 2023
Accepted: 11 October 2023
Published: 18 October 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Deep learning-based artificial intelligence has gained tremendous attention in recent years. Deep learning models are being used in a wide range of applications [1–3], such as computer vision [4,5], machine translation [6], natural language processing [7], and recommender systems [8]. In addition, deep learning techniques have achieved great success in real-time applications such as self-driving cars [9,10], unmanned aerial vehicles (UAVs) [11], and autonomous robots [12–14]. In all of these applications, the primary goal is to achieve high accuracy, which can be generally achieved using large and complex models. While such large and complex deep neural networks (DNNs) ensure high accuracy, they entail problems and challenges as well, such as long training times, high inference latency, low throughput, high energy consumption, and large memory usage. These challenges can be addressed using various optimization frameworks for obtaining the desired performance levels of deep learning models during the training and inference stages. In this paper, we study the performance of quantization frameworks such as AMP and TensorRT, which are low-precision formats, and characterize the behavior of classification models used to

address these challenges. Several of the challenges and problems of large and complex deep learning models are discussed in detail below.

1.1. First Problem: Training Time

For large DNNs, the first challenge is their high training time. The training time of recent DNN models can be extended by up to several weeks, which greatly slows the DNN model development and deployment process. For example, [15] showed that ResNet-101, which is less than 1% more accurate than ResNet-50, takes one week to train on four Maxwell M40 GPUs. Similarly, while ResNet-152 is 0.05% more accurate than ResNet-101, it takes 3.5 additional days of training time compared to ResNet-101. When a deep learning model takes a long time to train, there can be implications such as continuous usage of resources, power consumption, a slow development process, and challenges in scalability and maintenance. In addition, it greatly impacts the progress of new DNN designs and slows down the fine-tuning, evaluation, testing, and deployment processes of DNNs.

1.2. Second Problem: Inference Latency/Throughput for Real-Time Applications

Many real-time applications, such as advanced driver assistance systems (ADAS) and autonomous driving, utilize DNNs to perform tasks such as detection of obstacles [16–18] and pedestrians [19]. Along with the accuracy of deep learning models, latency and throughput become prominent factors in such safety-critical applications. A recent study [20] using the ImageNet dataset showed that the number of model layers in ResNet-152 (2015) has grown by 19$\times$ compared to Alexnet (2012). This has increased the giga-floating point operations (GFLOPs) of the model significantly while reducing the error rate by 12.5%. These large models lead to more memory references, resulting in high latency and low throughput. The increase in latency and decrease in throughput limits the attainable performance of real-time deep learning applications such as self-driving cars and autonomous robots.

1.3. Third Problem: Large Memory Usage

Large memory usage is another common challenge when working with large and complex deep learning models. Modern DNNs can contain millions or even billions of parameters. As models become more complex, their memory requirements increase significantly. For instance, GPT-3 [21] consists of hundreds of billions of parameters, leading to massive memory requirements. This issue affects both the training and inference stages of deep learning. The effects of these large models on memory requirements during training process are even worse than in the inference stage, as the activation values need to be stored during the training stage to allow for backpropagation. As the model becomes larger and more complex, the number of layers and activation values increases significantly, resulting in large memory requirements. During the deployment stage, large models affect inference performance in resource-constrained environments. This is because such models require significant memory capacity in order to efficiently process inputs and weights.

Various optimization techniques have been used to address these challenges, including weight pruning [22–25], weight clustering [26], and gradient accumulation and quantization [27–30]. Deep compression methods are developed using combinations of optimization techniques such as pruning, clustering, and quantization in order to reduce the maximum degree of redundancy in the model [31]. In addition to these deep learning optimization techniques, optimization frameworks can be used to enhance the performance of deep learning models [32]. For instance, TensorRT is a popular optimization framework that is used to optimize DNNs for more efficient inference.

Deep learning optimization techniques such as quantization and pruning have been studied and used thoroughly in the past; however, few works have studied the performance of the frameworks that provide these quantization techniques. The objective of the present study is to characterize the behavior of different models when using TensorFlow AMP for the training stage and Nvidia TensorRT quantization techniques for the inference stage. Memory utilization and training time are profiled during the training stage, while latency

and throughput are measured for the inference stage. Furthermore, model accuracy is quantified to assess the effect of TensorFlow AMP on TensorRT post-training quantization. We then analyze the quantization techniques with regard to the various deep learning performance metrics. Our main contributions are as follows:

- We study the performance of various quantization frameworks, such as the AMP and TensorRT low-precision formats, to address challenges of deep learning models such as training time, inference latency, and memory usage. We demonstrate the performance of these quantization frameworks through different deep learning classification models.
- We benchmark the training time and memory utilization with and without the AMP technique for various models during the training stage.
- We benchmark the latency and throughput with different precision modes of TensorRT post-training quantization during the inference stage.
- We profile deep learning performance across various deep learning models using various metrics, including accuracy, memory usage, training time, latency, and throughput, by varying the image sizes, batch sizes, and datasets.
- We quantify and analyze the accuracy of different deep learning models for different quantization precision modes, such as FP32, FP16, and INT8, with and without the AMP technique.

This remainder of this article is organized in the following manner. Section 2 provides a comprehensive review of previous works and studies related to quantization techniques, highlighting the key advancements and findings in the field. In Section 3, the concepts of quantization are explained in the context of deep learning, covering the fundamental principles and techniques. Section 4 describes the framework that utilizes quantization techniques for efficient inference. The methodology employed in this paper is presented in Section 5, which includes a detailed description of the datasets, models, hardware, and metrics used for experimentation. Section 6 presents the results obtained from our experiments and analyzes the impact of quantization techniques on various models across different batch sizes. Lastly, Section 7 concludes the paper by summarizing our key findings and provides suggestions for future exploration in the field of quantization for deep learning.

2. Related Work

Continuous pursuit of high accuracy has led researchers to develop very large and complex neural network architectures such as deep convolutional neural networks (CNNs). However, using these large CNNs is not suitable for mobile or embedded platforms such as smartphones, augmented reality (AR)/virtual reality (VR) devices, and drones. These mobile and/or embedded platforms require smaller models that are a better fit for their limited memory and computational resources. As a result, there is a growing field of research dedicated to reducing model size and inference time while maintaining high accuracy. One way to do this involves reducing the precision of the weights and activations of the models by converting them from higher-bit to lower-bit representations. This approach has led to new lightweight network architectures such as Binary Neural Networks (BNN [33]) and Ternary Weight Networks (TWN [34]).

Quantization techniques have demonstrated remarkable performance improvements in training and inference of DNN models [35–39]. Similarly, the advancements in half-precision and mixed-precision training [40,41] have played an important role in efficient DNN execution. By enabling low-precision computation with efficient dataflow in hardware accelerators, quantization leads to much better latency and throughput in deep learning inferences.

Quantization reduces the computational and memory requirements of DNNs, making them suitable for deployment on resource-constrained edge devices. This enables edge AI [42], reduces power consumption and latency, improves real-time processing, and addresses memory constraints in edge devices. In one study by Ravi et al. [43], a lightweight

vision transformer model was deployed on a Xilinx PYNQ Z1 field-programmable gate array (FPGA) board by applying quantization. The work in [44] utilized quantization with federated learning to improve the efficiency of data exchange between cloud nodes and edge servers. The TensorRT optimization framework [45] has been used to easily accelerate and deploy various deep learning applications on edge devices. This framework is comprised of many optimization techniques, including the quantization technique. It was used by Wang et al. [46] to accelerate the YOLOv4 architecture on a Jetson Nano for the application of detecting of dirty eggs. In another study by Chunxiang et al. [47], the YOLOX model was optimized with TensorRT to allow its deployment on low-cost embedded devices. The performance of the CenterNet model was accelerated with TensorRT during a video analysis by Tao et al. [48]. In addition, TensorRT plays an important role in autonomous vehicle applications. Trajectory prediction is a critical task in self-driving due to limited computation resources and strict inference timing constraints. Optimization of these prediction models with TensorRT has resulted in low latency and high throughput [49].

Modern DNN accelerators, such as tensor processing units (TPUs) [50] deployed in Google Coral [51], are highly optimized for artificial intelligence (AI) workloads, as they are application-specific integrated circuits (ASICs) used to accelerate deep learning workloads. High-bandwidth memory (HBM), which provides much higher bandwidth compared to dual in-line memory modules (DIMMs), is often deployed along with TPUs. However, HBM typically has limited capacity compared to a DIMM. Quantization plays an important role by reducing the memory requirement of the models, enabling more models to fit into memory with a limited size such as HBM. Quantization can alleviate memory bandwidth bottlenecks for large and unquantized models by reducing access to the large-capacity DIMM [52].

3. Quantization in Deep Learning

Neural networks require a great deal of memory and computing power. Whether running in the cloud or on smaller devices such as smartphones or edge devices, optimizing the memory and computing power of DNNs is a very important way to reduce the required computing resources and costs. One way to do this is quantization [53], which involves using lower-precision data types (as shown in Figure 1) to represent the network's weights and activations. By using fewer bits or simple data types, it is possible to reduce the required amount of memory and computation, making the network run more efficiently and faster.

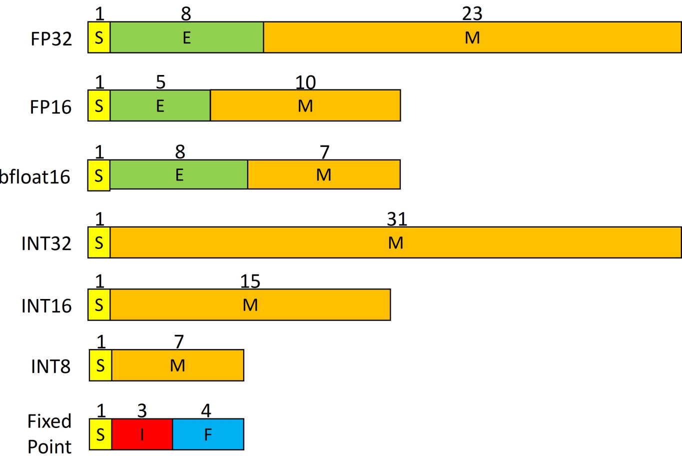


Figure 1. Numerical representation of different data types.

Quantization [54] is an optimization technique that represents weights and activations ranging from higher-precision to lower-precision data types. Using a data type with a lower number of bits yields benefits such as reduced memory usage, lower energy consumption, and faster execution of operations [55]. Additionally, quantization enables models to be

deployed on embedded devices, which often support only integer data types [56]. The most commonly used lower-precision data types in quantization are:

- FP16: half-precision IEEE floating point format.
- bfloat16: brain floating point format.
- INT8: eight-bit integer format.

These common quantization approaches are discussed in more detail in the following.

3.1. Quantization to FP16/bfloat16

Quantization from higher precision to lower precision, such as FP16/bfloat16, is a straightforward process because these data types share the same numerical representation scheme. However, compatibility of the hardware and sensitivity of small values of gradients for lower precision format are the factors that need to be considered when quantizing higher precision to lower precision.

3.2. Quantization to INT8

Quantizing from the higher precision value to the INT8 value is a more complex task compared to quantization to FP16 or bfloat16. Unlike FP32, which can represent a large range of real number values, INT8 can only represent 256 discrete values. The goal is to determine the optimal approach for mapping the range $[f_{min}, f_{max}]$ of higher precision values into the limited space of INT8.

3.2.1. Asymmetric Signed Quantization

Let us consider a floating-point value x in $[f_{min}, f_{max}]$. We can map this value to a signed integer value x_q with the range $[q_{min}, q_{max}]$ using the following Equation (1):

$$x_q = clip(round(x/S) + Z, -2^{n-1}, 2^{n-1} - 1) \tag{1}$$

where:

- n is the number of bits in lower precision format after quantization. In the case of signed INT8, the range will be $[-128, 127]$.
- The range $[f_{min}, f_{max}]$ is determined during the calibration stage.
- S is the scale factor and is a positive FP32 value:

$$S = \frac{f_{max} - f_{min}}{2^n - 1}. \tag{2}$$

- Z is the zero-point (which may be called the quantization bias or offset), which is the INT8 value corresponding to a value of 0 in the FP32 space:

$$Z = -round(\frac{f_{min}}{S}) - 2^{n-1}. \tag{3}$$

The example shown in Figure 2 below maps the FP32 value to the signed INT8 precision using asymmetric signed quantization.

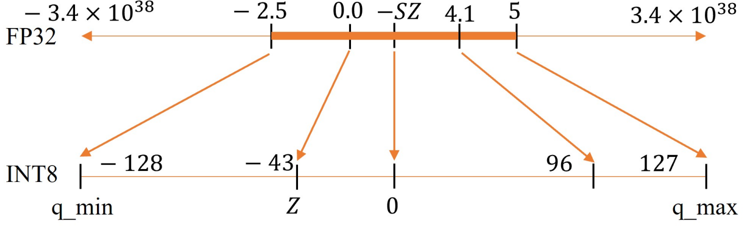


Figure 2. Asymmetric signed quantization example from FP32 to signed INT8.

$$S = \frac{5 - (-2.5)}{255} = 0.0294 \quad (4)$$

$$Z = -round(\frac{-2.5}{0.0294}) - 128 = -43 \quad (5)$$

$$x_q = clip(round(4.1/0.0294) - 43, -128, 127) = 96 \quad (6)$$

3.2.2. Symmetric Signed Quantization

A commonly used approach used in quantization is symmetric signed quantization [57]. In this scheme, $|f_{min}| = |f_{max}|$ and the zero-point = 0. The quantized value for symmetric signed quantization can be calculated using Equation (7):

$$x_q = clip(round(x/S), q_{min}, q_{max}). \quad (7)$$

Based on the precision range for signed integers, there are two types of symmetric signed quantization, which are discussed below.

In *symmetric signed quantization with full range*, the range of the integer is considered as $[-2^{n-1}, 2^{n-1} - 1]$. For INT8, this is $[-128, 127]$, and S is calculated using Equation (8):

$$S = \frac{2 \times max(|f_{min}|, |f_{max}|)}{2^n - 1}. \quad (8)$$

However, in *symmetric signed quantization with restricted range*, the precision range of a signed integer is $[-2^{n-1} + 1, 2^{n-1} - 1]$. For INT8, this is $[-127, 127]$ and S is computed using Equation (9):

$$S = \frac{max(|f_{min}|, |f_{max}|)}{2^{n-1} - 1}. \quad (9)$$

The simple example below demonstrates the mapping of the value 3.86 in FP32 to the INT8 space using symmetric signed quantization with a restricted range, as depicted in Figure 3.

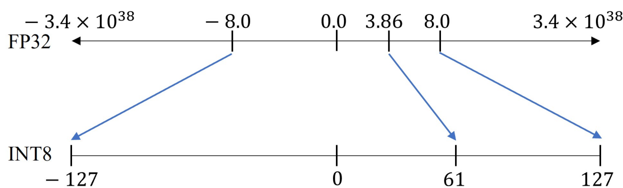


Figure 3. Symmetric signed quantization scheme example with restricted range of INT8.

$$S = \frac{max(|f_{min}|, |f_{max}|)}{2^{n-1} - 1} = \frac{8}{127} = 0.0629 \quad (10)$$

$$x_q = round\left(\frac{3.86}{0.0629} + 0\right) = 61 \quad (11)$$

3.3. Calibration

During quantization, calibration is the step that determines the range of model tensors, which include weights and activations [58]. While it is relatively easy to compute the range for weights, as the actual range is known at the time of quantization, it is less clear for activations. The activation of a neural network layer varies based on the input data fed to the model. Therefore, a representative set of input data samples is required to estimate the range of activations. This set of input data samples is called the calibration dataset. Quantization of activations is a data-dependent process that requires input data samples. For this purpose, different approaches can be applied to address this issue.

These quantization approaches are summarized in Table 1 based on accuracy effects and calibration data requirements.

Table 1. Quantization techniques with accuracy effects and data requirements.

Quantization Mode	Data Requirement	Accuracy
Post-training dynamic [59]	Not required	Small decrease
Post-training static [59]	Unlabelled data	Small decrease
Quantization-aware training [60]	Training data	Negligible decrease
Mixed precision [41,61]	Training data	Negligible decrease

3.4. Post-Training Dynamic Quantization

Generally, weights are quantized to lower precision, as they are known before the inference stage and their range can be computed for the quantization. Activation values are unknown prior to the inference stage; therefore, it is hard to find the scale factor and zero-point for the quantization of activation tensors. In this type of quantization, the range of activation tensors is calculated dynamically during the inference stage using input samples; this is called post-training dynamic quantization [62]. This technique yields excellent results with minimal effort, however, it can be slower than static quantization due to the calculation overhead introduced by computing the range of activation during the inference stage.

3.5. Post-Training Static Quantization

During the quantization process, the range for each activation is typically computed prior to the inference stage [59]. This requires a calibration dataset that can be passed through the model and profiling of activation values. To achieve this, the following steps are required:

- Perform a certain number of forward passes on a calibration dataset, which is usually around 150–200 samples, and record activation values.
- Compute the ranges for each activation using calibration techniques such as min-max, moving average of min-max, or histogram.
- Calculate the scale factor and zero-point using the range of activation tensors used to perform quantization.

3.6. Quantization Error

Model accuracy may be reduced due to post-training quantization in deep learning. When the weights and activation values of the model are quantized to a low-precision format, errors may be introduced due to rounding and clipping operations in quantization.

- **Rounding errors** are a type of numerical error that occur when a real number with high precision is approximated by a low-precision value. When rounding numbers, especially during calculations involving floating-point arithmetic, the result may not be exact and may differ slightly from the true value. Rounding errors can accumulate in quantization and affect the accuracy of the model.
- **Clipping errors** occur when a high-precision value is mapped or quantized to a discrete set of values or a limited range. This mapping introduces error because the quantized value may not precisely represent the original value.

The total quantization error is the sum of the rounding and clipping errors over a given dataset, as depicted in Equation (12):

$$error_{quant} = \sum_{dataset} error_{rounding} + error_{clipping}. \tag{12}$$

Quantization may lead to information loss, as values are mapped to a smaller set of discrete levels. This information loss can impact the model's ability to differentiate between similar inputs, resulting in a reduction in accuracy, as shown in Figure 4.

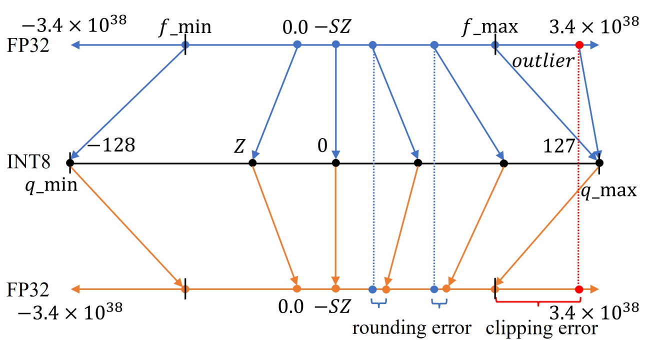


Figure 4. Quantization error in FP32 and INT8 precision formats.

3.7. *Accuracy Recovery Techniques in Quantization*

Many deep learning models experience accuracy loss due to quantization errors. To address this problem, several techniques can be used to retain accuracy, including:

- Quantization-Aware Training (QAT)
- Mixed Precision Training

3.7.1. Quantization-Aware Training

The objective of this training is to increase the performance (accuracy) of the model by simulating inference stage quantization [60]. To achieve this, the DNN weights and activations are approximated in a low-precision format during training without actually reducing the precision. The neural network's forward and backward passes implement low-precision weights, and the loss function adjusts the quantization errors that may occur due to the low-precision values. This technique allows the model to perform more accurately during the inference stage, as it familiarizes the model with the quantization effect during training.

3.7.2. Mixed Precision Training

This method [61] similarly helps the model to familiarize itself with the quantization effect during training while reducing the precision of model tensors such as weights, activations, and gradients. Therefore, certain operations are performed in a lower-precision format while necessary information is stored in single-precision for critical components of the network. Mixed precision training speeds up computational processes and significantly reduces training time. The use of Tensor cores in the Nvidia Hopper, Ampere, Turing, and Volta architectures significantly improves the overall training speed, particularly for complex models. For mixed precision training, two steps are essential:

- The model tensors are converted into lower precision where applicable.
- Loss scaling is incorporated to preserve small gradient values.

The training dataset works as calibration dataset to compute the range of weights and activation tensors for scale factor and zero-point of quantization.

Automatic Mixed Precision is an extension of mixed precision training that automatically reduces the precision of appropriate model tensors during training and scales up the loss to preserve the small gradients in low-precision formats.

Quantization-aware training and mixed precision training differ in terms of the following aspects. The primary goal of mixed precision training is to decrease training time and memory usage by reducing the precision of network data wherever appropriate. Quantization-aware training does not prioritize this aspect; rather, it makes the network aware of the quantization effect by emulating quantized data in the network. Mixed

precision training leverages a real low bit-width format, which accelerates both forward and backward passes in neural network training due to hardware support, such as in Tensor cores. Quantization-aware training, on the other hand, does not require an actual low bit-width format or corresponding hardware support. A summary of quantization techniques is depicted in Figure 5.

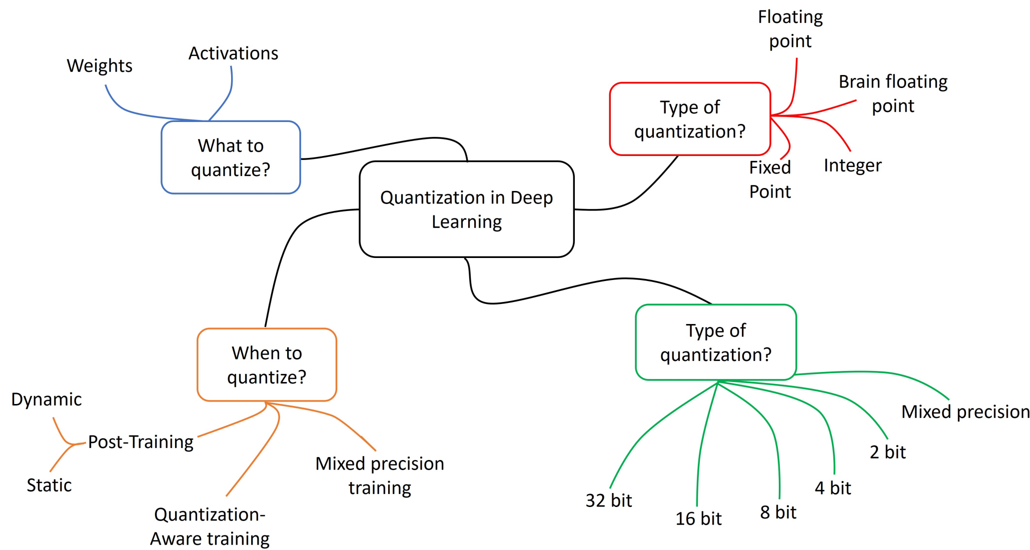


Figure 5. Categorization of quantization.

4. TensorRT (TRT) and TensorFlow-TRT Frameworks

Nvidia's TensorRT (TRT) [63] is a high-performance deep learning inference framework. It works as a deep-learning compiler that is specifically designed to optimize TensorFlow/PyTorch models for efficient inference on NVIDIA devices. TensorFlow-TensorRT (TF-TRT) is an application programming interface (API) of Nvidia's TRT in TensorFlow.

Nvidia TensorRT is a powerful inference optimizer that enables performing inference with lower-precision on GPUs. By integrating TensorRT with TensorFlow, users can easily apply TensorRT optimization techniques to their TensorFlow models. The optimization process targets the supported TensorFlow model layers while leaving unsupported operations for native execution in TensorFlow.

TF-TRT utilizes many of TensorRT's capabilities to accelerate inference performance. Among of these capabilities are:

- Different precision modes
- Post-training quantization
- INT8 quantization

This section provides an overview of the above capabilities and illustrates how to use them.

4.1. *Quantization with Different Precision Modes*

TensorRT can convert activations and weights to lower precisions, resulting in faster inference during runtime. The precision mode, determined by the "precision_mode" argument, can be set to FP32, FP16, or INT8. Utilizing lower precision can provide higher performance with supported hardware such as Tensor cores.

The FP16 mode, with supported hardware such as Tensor cores or half-precision hardware, boosts inference performance with little accuracy loss. On the other hand, the INT8 precision mode utilizes Tensor cores or integer hardware instructions, offering the best performance in terms of latency and throughput. However, INT8 quantization may introduce quantization errors due to rounding and clipping operations, leading to accuracy degradation.

Different precision modes such as FP32, FP16, and INT8 can be set independently. TensorRT has the flexibility to choose a higher-precision kernel for the part of the model if it leads to a lower overall runtime or if a low-precision implementation is unavailable. This mixed selection of precision modes offers better performance in terms of latency and throughput in the inference stage.

4.2. Post-Training Quantization in TF-TRT

TF-TRT predominantly uses post-training quantization (PTQ). PTQ is applied to pretrained models to reduce their size and improve throughput with a small reduction in accuracy.

During the calibration of post-training static quantization, TensorRT utilizes "calibration" data to estimate the scale factor and zero-point for each tensor based on its dynamic distribution and range. A representative input data loader should be passed during the quantization process to ensure meaningful scale factors for activations. Using a large and diverse dataset for calibration, such as the test dataset or its subset, can provide a better range and distribution for activations.

4.3. INT8 Quantization

TensorRT supports 8-bit integer precision mode. It converts high-precision values into INT8 precision values using symmetric signed quantization.

The scaling factor S is provided as follows:

$$S = \frac{max(|f_{min}|, |f_{max}|)}{127} \tag{13}$$

where f_{min} and f_{max} provide a range of floating point values for the given tensor.

For a given scale S, quantization/dequantization operations can be represented as follows:

$$x_q = quantize(x, S) = clip(round(\frac{x}{S}), -128, 127) \tag{14}$$

where:

- x_q is a quantized value in the range $[-128, 127]$.
- x is a floating-point value of the tensor.

Using the same formula, de-quantization can be performed through a multiplication operation using Equation (15). De-quantization is an important step, as certain model operations are not supported by TensorRT; in such cases, de-quantization is required in order to keep the model subnetworks compatible with one another.

$$x = dequantize(x_q, S) = x_q \times S \tag{15}$$

TF-TRT Workflow

After installing the TensorRT API for the Tensorflow framework and obtaining a trained TensorFlow model, the model is exported in the saved format. TF-TRT then applies different optimization techniques to the supported layers. The result is a TensorFlow graph in which the supported layers replaced by TensorRT-optimized engines. The complete workflow of TF-TRT is shown in Figure 6.

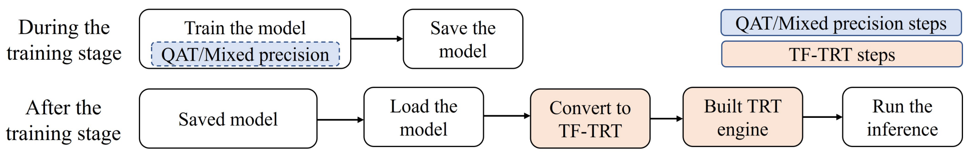


Figure 6. TF-TRT workflow during and after the training stage.

TF-TRT operations involve three steps:

- **Model Partitioning:** TensorRT scans the TensorFlow model to split the subnetworks that can be optimized based on supported operations.
- **Layer Conversion:** supported TensorFlow layers within each subnetwork are converted into TensorRT layers.
- **Engine Optimization:** the subnetworks are transformed into TensorRT engines, as shown in Figure 7.

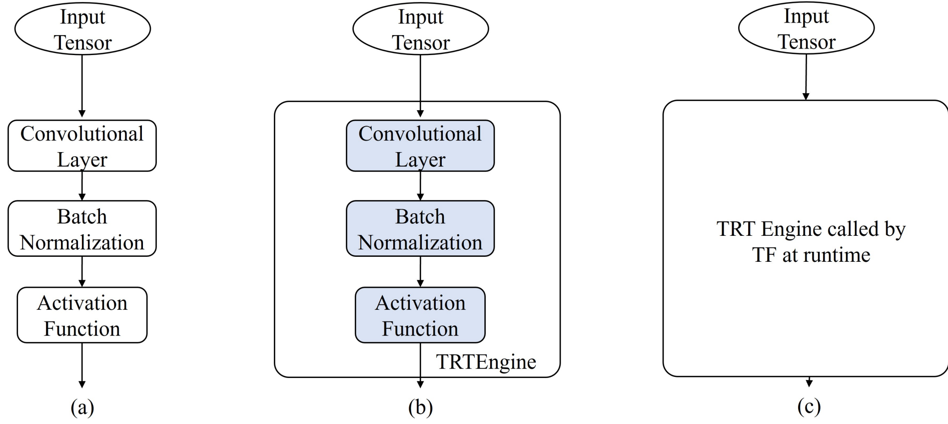


Figure 7. (**a**) TensorFlow model for conversion, (**b**) partitioning of supported TensorFlow layers for TRT Engine, and (**c**) conversion to TRT Engine.

TF-TRT automatically scans and partitions the TensorFlow model network into compatible subnetworks for optimization and execution by TensorRT. During the conversion process, TensorRT performs critical transformations and optimizations, including constant folding, pruning unnecessary nodes, and layer fusion. The aim of TF-TRT is to convert as many operations as possible into a single TensorRT engine that will lead to maximum performance in terms of latency and throughput.

5. Methodology

This section outlines the methodology adopted in this study. It consists of a detailed explanation of the various aspects involved in the experimentation, including the datasets used for the study, the models employed, the hardware used, and the metrics considered for performance evaluation.

5.1. Datasets

This study utilized two datasets, CIFAR-10 and Cats_vs_Dogs, to evaluate the effect of different quantization approaches on various deep learning models. The CIFAR-10 dataset, shown in Figure 8, consists of 32×32 color images with ten classes. The total number of images in the dataset is 60,000, and each class contains an equal number of images, for a total of 6000 images per class. The CIFAR-10 dataset is divided into training images, validation, and test images. There are 50,000 training images and 5000 each of validation and test images. In this paper, the size of the CIFAR-10 images used for model training and optimization was $48 \times 48 \times 3$. The main reason for resizing the CIFAR-10 images was due to MobileNet_v1. Because of the lightweight and efficient architecture of MobileNet_v1, it trained quickly both with AMP and without AMP on the CIFAR dataset. Therefore, in order to observe good training time behavior with AMP and without AMP, a size of $48 \times 48 \times 3$ was used.

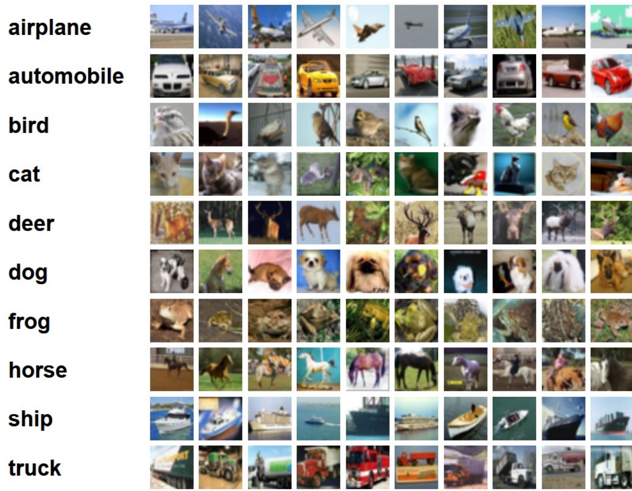


Figure 8. CIFAR-10 dataset with ten examples in each class [64].

The second dataset used for the model training and optimization was cats_vs_dogs, shown in Figure 9. It consists of color images with two classes. There are 3000 total images and each class contains an equal number of images, for 1500 per class. The dataset is divided into training images and validation images, including 2000 Training images and 500 each of validation and test images. In this paper, the size of the cats_vs_dogs images used for model training and optimization was $128 \times 128 \times 3$. These datasets were processed and trained in the TensorFlow framework using the 22.03 Nvidia container [65].

Figure 9. Cats_vs_dogs dataset with five examples in each class [66].

5.2. Models

The following models were used to observe the effect of optimizations during the training and inference stages.

5.2.1. VGG16

This is a CNN model comprised of sixteen layers, which consist of thirteen convolution layers with a kernel size of 3×3 and three fully connected layers after the convolutional layers [5]. All layers use the ReLU activation function for except the last layer, which is equipped with a softmax activation function.

5.2.2. MobileNet_v1

This is another CNN model, commonly used in mobile and embedded vision applications [67]. This network is comprised of 28 layers. It employs depthwise separable convolutions, which enable lightweight DNNs and helps to reduce latency and computational requirements on mobile and embedded devices. It has gained popularity in various applications, such as object detection, classification, and localization in resource-constrained mobile and embedded systems.

5.2.3. ResNet-50

ResNet (short for Residual Network) is a deep CNN model that utilizes the concepts of residual learning and skip connections to avoid the exploding/vanishing gradients

problem [68]. This concept enables the model to become a large and deep network. This model consists of convolutional layers and identity blocks followed by a final softmax layer.

5.3. Hardware

The hardware used in this paper was an Nvidia Quadro RTX 4000 [69], which consists of Turing architecture with a compute capability of 7.5. The compute capability of a GPU determines the set of features and general specifications of the GPU [70]. It has 2304 CUDA cores for deep learning computation and 288 tensor cores that support quantized data and mixed-precision processing. The GPU memory uses GDDR6 technology with a capacity of 8 GB.

5.4. Metrics

The following are the metrics used to quantify model performance across different quantization techniques.

- **Accuracy**: a metric that summarizes the performance of a classification model by calculating the fraction of correct predictions over the total number of predictions, as provided by Equation (16):

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}. \tag{16}$$

- **Average Training Time**: the training time of the model for a specific number of epochs. The speedup of AMP in terms of average training time against the case without AMP can be computed by dividing the total training time without AMP by the total training time with AMP, as shown in Equation (17):

$$\text{Speedup}_{\text{Training time}} = \frac{\text{Training time}_{\text{without AMP}}}{\text{Training time}_{\text{with AMP}}}. \tag{17}$$

- **Memory Usage in Training**: the size of the memory used during the training process. The improvement in memory usage can be quantified by calculating the memory reduction factor, as provided in Equation (18):

$$\text{Reduction Factor}_{\text{Memory}} = \frac{\text{Memory}_{\text{without AMP}}}{\text{Memory}_{\text{with AMP}}}. \tag{18}$$

- **Latency in Inference**: the time taken by the model to predict one input unit. The reduction in latency can be computed by the speedup, as provided by Equation (19):

$$\text{Speedup}_{\text{Latency}} = \frac{\text{Latency}_{\text{without quantization}}}{\text{Latency}_{\text{with quantization}}}. \tag{19}$$

- **Throughput in Inference**: the measure of the number of images predicted by the model in one second. The increase in throughput can be computed by the improvement factor (IF), as in Equation (20):

$$\text{Improvement Factor}_{\text{Throughput}} = \frac{\text{Throughput}_{\text{with quantization}}}{\text{Throughput}_{\text{without quantization}}}. \tag{20}$$

Because the training time speedup, memory reduction factor, latency speedup, and throughput improvement factor are the ratios of two quantities, these terms have no units.

6. Results and Discussion

This section presents the experimental results in terms of the training time, memory usage, accuracy, latency, and throughput, displaying the outcomes of different batch sizes, image sizes, and datasets. The batch size is the number of images used in one iteration

of the training or inference model. This section then discusses the impact of quantization techniques on various models for different batch sizes.

6.1. Training Stage

In the training stage, we trained three models: VGG16, ResNet-50, and MobileNet_v1. These models were trained with and without the AMP optimization technique. The performance of the base models and optimized models were recorded for different batch sizes, datasets, and image sizes to observe model behavior. Two metrics were measured during the training stage, namely, the average training time and the memory utilization. The images in the CIFAR-10 dataset were resized to ($48 \times 48 \times 3$), while those in the Cats_vs_Dogs dataset were resized to ($128 \times 128 \times 3$).

6.1.1. Training Time

The AMP optimization technique reduces the training time significantly for large batch sizes. The training time speedup shows an increasing trend with an increase of batch sizes (Figure 10); however, a variation in speedup is observed for different batch sizes and for different models as shown in Tables 2 and 3. For MobileNet_v1 and ResNet-50, the training time speedup is recorded as even less than 1 for the batch size of 32. This is mainly due to two reasons:

- AMP has a component processing time (quantization/dequantization and loss scaling) in addition to the training time.
- VGG16 has 138 million parameters, which is far more than MobileNet_v1 or ResNet-50, which have 4.2 million and 13 million parameters, respectively.

Table 2. Training time and speedup calculation with AMP for CIFAR-10 (image size: $48 \times 48 \times 3$).

Batch Size	Training Time (ms)								
	Without AMP			With AMP			Speedup		
	VGG16	ResNet-50	MBNet	VGG16	ResNet-50	MBNet	VGG16	ResNet-50	MBNet
32	480.14	659.36	253.27	265.44	711.65	367.09	1.81	0.93	0.69
64	373.22	403.44	138.8	187.99	360.74	192.23	1.99	1.12	0.72
128	310.14	300.74	94.49	143.61	193	110.48	2.16	1.56	0.86
256	291.32	269.95	78.14	121.49	115.46	57.29	2.40	2.34	1.36
512	256.16	257.65	74.6	112.28	89.54	37.87	2.28	2.88	1.97
1024	304.14	231.16	70.93	109.43	75.22	34.61	2.78	3.07	2.05

MBNet = MobileNet_v1.

Table 3. Training time and speedup calculation with AMP for cats_and_dogs (image size: $128 \times 128 \times 3$).

Batch Size	Training Time (ms)								
	Without AMP			With AMP			Speedup		
	VGG16	ResNet-50	MBNet	VGG16	ResNet-50	MBNet	VGG16	ResNet-50	MBNet
8	114.81	120.95	59.83	63.51	130.59	70.84	1.81	0.93	0.84
16	78.94	86.82	41.96	50.19	72.84	39.25	1.57	1.19	1.07
32	70.79	69.94	35.29	45.4	44.89	22.28	1.56	1.56	1.58
64	72.61	63.52	33.85	41.92	37.53	16.26	1.73	1.69	2.08
128	79.8	61.87	33.45	42.36	33.99	21.15	1.88	1.82	1.58
256	OOM	OOM	36.19	49.12	31.5	15.09	N/A	N/A	2.40

OOM = Out of Memory, MBNet = MobileNet_v1, N/A = Not applicable.

For models such as MobileNet_v1 and ResNet-50, which have low batch sizes and small parameter counts, the processing time for AMP components becomes significant compared to the low-precision format time reduction effect, causing a reduction in the speedup of training time. In the case of large batch sizes or models such as VGG16 with a large number of parameters, the AMP component processing time becomes insignificant, providing a great speedup in training time. While the AMP optimization technique provides a great speedup, a variation in this speedup is observed for different batch sizes. This can be avoided by profiling the behavior of the model for different batch sizes and selecting the optimal batch size that provides a desirable speedup in training time.

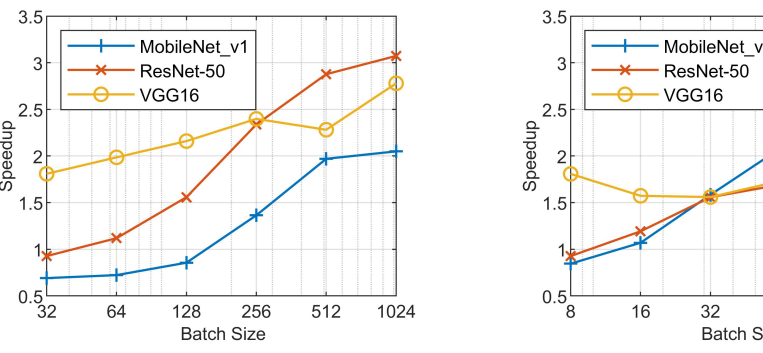


Figure 10. Training time speedup comparison with AMP for CIFAR-10 (**left**) and CATS_VS_DOGS (**right**).

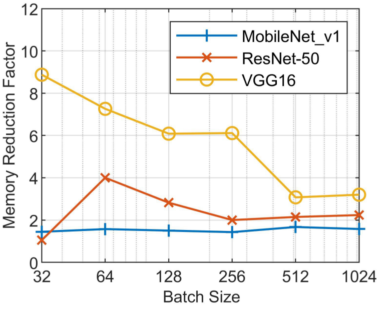
6.1.2. Memory Usage

We observed that memory usage was optimized for both datasets by the quantization techniques. For MobileNet_v1 and ResNet-50, memory utilization was optimized by roughly the same degree, particularly with large batch sizes, as shown in Figure 11. For VGG16, memory was optimized drastically for small batch sizes, up to 12× in the Cats_vs_Dogs dataset and 8.88× in the CIFAR-10 dataset, as shown in Table 4. The memory reduction factor decreased with the increase in batch size for VGG16 in both datasets, with the AMP technique providing roughly 3× memory optimization for a batch size of 128 on the Cats_vs_Dogs dataset. The reason for this effect is that at low batch sizes the AMP memory reduction effect is more significant for large parameter models such as VGG16, while for either a large batch size or small parameter number, as in such models as MobileNet_v1 and ResNet-50, the memory reduction factor shows negligible differences.

Table 4. Memory reduction calculation with AMP for CIFAR-10 (image size: 48 × 48 × 3).

Batch Size	Memory Usage (GB)								
	Without AMP			With AMP			Memory Reduction Factor		
	VGG16	ResNet-50	MBNet	VGG16	ResNet-50	MBNet	VGG16	ResNet-50	MBNet
32	1.42	0.22	0.13	0.16	0.21	0.09	8.88	1.05	1.44
64	1.67	1.00	0.22	0.23	0.25	0.14	7.26	4.00	1.57
128	2.13	1.1	0.36	0.35	0.39	0.24	6.09	2.82	1.50
256	3.24	1.32	0.7	0.53	0.66	0.49	6.11	2.00	1.43
512	3.78	2.66	1.47	1.23	1.24	0.88	3.07	2.15	1.67
1024	5.79	5.23	2.79	1.81	2.34	1.77	3.20	2.24	1.58

MBNet = MobileNet_v1.



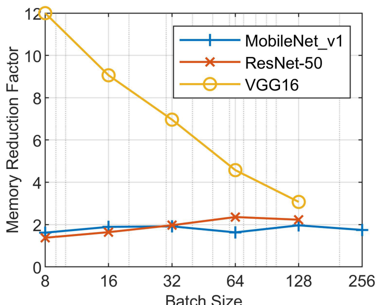


Figure 11. Memory performance comparison with AMP for CIFAR-10 (**left**) and CATS_VS_DOGS (**right**).

For the batch size of 256, ResNet-50 and VGG16 could not be trained on our 8 GB RTX4000 GPU due to out-of-memory issues, while the AMP technique allowed training to take place on this GPU, as shown in Table 5. This represents a significant advantage of AMP optimization technique.

Table 5. Memory reduction calculation with AMP for cats_and_dogs (image size: 128 × 128 × 3).

Batch Size	Memory Usage (GB)								
	Without AMP			With AMP			Memory Reduction Factor		
	VGG16	ResNet-50	MBNet	VGG16	ResNet-50	MBNet	VGG16	ResNet-50	MBNet
8	2.64	0.33	0.21	0.22	0.24	0.13	12.00	1.38	1.62
16	2.99	0.59	0.34	0.33	0.36	0.18	9.06	1.64	1.89
32	3.69	1.18	0.67	0.53	0.6	0.35	6.96	1.97	1.91
64	4.39	2.59	1.14	0.96	1.1	0.7	4.57	2.35	1.63
128	4.85	4.5	2.57	1.58	2.02	1.31	3.07	2.23	1.96
256	OOM	OOM	4.34	2.91	4.11	2.49	N/A	N/A	1.74

OOM = Out of Memory, MBNet = MobileNet_v1, N/A = Not applicable.

It was observed that the memory improvement factor during the training stage was reduced on both datasets, while training time speedup varied across different models at different batch sizes with the increasing trend. This is mainly because AMP includes additional components during the training stage, as it includes storage of weights in FP32, conversion of FP32 to FP16, and scaling of gradients. These additional components have less of an effect with large batch sizes; however, when the batch size is small, the processing time for these additional AMP components becomes significant and restricts the attainable speedup of training time. Therefore, a large batch size may be preferred for the training stage if a large training time speedup is required. Large batch sizes reduce the memory reduction factor and provide a great speedup in training time. The optimal batch size that provides both a desirable training time speedup and memory reduction factor can be selected by profiling the model performance for various batch sizes.

6.2. Inference

In this paper, the TF-TRT framework is used for inference optimizations. The key capabilities of the TF-TRT framework include different precision modes, such as FP32, FP16, and INT8 quantization. TF-TRT provides significant performance improvements during the inference stage by reducing the latency and increasing the throughput. In this paper, the performance of TF-TRT precision modes such as FP32, FP16, and INT8 has been profiled for different batch sizes, datasets, and image sizes. Speedup and improvement factor graphs are shown and results are discussed in this subsection.

AMP helps to adjust model weights during the training stage due to quantization errors, with the model saved in FP32 format when it has been trained. After training,

post-training quantization converts the model to low-precision formats, which significantly increases the latency speedup and throughput improvement factor when using supported hardware such as Tensor cores.

It has been observed that for small image sizes, in the CIFAR-10 dataset, as the batch size increases, latency speedup almost stays the same as shown in Tables 6 and 7. This effect is observed for all models. On the other hand, when image size is increased in Cats_vs_Dogs, then latency speed up shows a decreasing trend with the increase of batch size as shown in Figures 12 and 13. Similar behavior is observed for throughput performance. With a small dataset, memory access time for the input batch is insignificant as compared to the low-precision format time reduction effect for model weights and activation values. However, with large image sizes, memory access time becomes significant and it grows as the batch size increases as shown in Tables 8 and 9. This causes a reduction in latency speedup and throughput improvement as shown in Figures 14 and 15.

Table 6. CIFAR-10 latency speedup results with TF-TRT precision modes for different models.

Batch Size	Latency (ms)											
	VGG16				ResNet-50				MobileNet_v1			
	Base	**FP32**	**FP16**	**INT8**	**Base**	**FP32**	**FP16**	**INT8**	**Base**	**FP32**	**FP16**	**INT8**
8	39.3	4.7	1.5	1.2	47	4.2	1.9	1.9	39.8	1.9	1.4	1.3
16	39.4	5.6	1.5	1.4	47.9	4.2	2.1	1.9	39.8	1.9	1.4	1.4
32	38.1	7.9	1.7	1.5	48.1	4.6	2.1	2	37.3	1.9	1.5	1.4
64	55.2	14.2	3	1.8	60.2	6	2.2	2.1	45.7	2.2	1.6	1.5

Table 7. Cats_vs_Dogs latency speedup results with TF-TRT precision modes for different models.

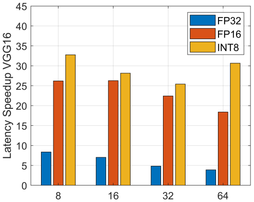
Batch Size	Latency (ms)											
	VGG16				ResNet-50				MobileNet_v1			
	Base	**FP32**	**FP16**	**INT8**	**Base**	**FP32**	**FP16**	**INT8**	**Base**	**FP32**	**FP16**	**INT8**
8	47.8	10.6	2.8	2	49.1	4.6	2.6	2.1	40.8	2.2	1.9	1.8
16	51.9	19.1	5.1	3.1	51.4	7.9	3.3	2.8	41.2	2.8	2.3	2.3
32	54.2	37.7	10.1	5.6	52.8	15	5.2	3.5	43.2	4.8	3.1	3.2
64	83.3	76	19.6	11.2	77.3	28.9	9.1	6.3	58.6	9.2	4.7	3.9

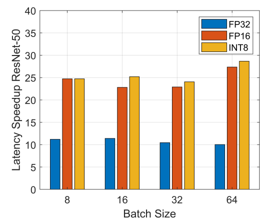
Table 8. CIFAR-10 throughput improvement factor results with TF-TRT precision modes for different models.

Batch Size	Throughput (Images per Second)											
	VGG16				ResNet-50				MobileNet_v1			
	Base	**FP32**	**FP16**	**INT8**	**Base**	**FP32**	**FP16**	**INT8**	**Base**	**FP32**	**FP16**	**INT8**
8	204	1689	5470	5455	170	1916	4105	4123	201	4286	5545	5947
16	406	2858	10,554	10,780	334	3771	7747	8301	402	8430	11,271	11,817
32	840	4026	18,326	21,065	666	6887	15,254	16,245	857	16,890	20,984	23,040
64	1160	4491	21,475	35,960	1063	10,618	30,599	30,975	1402	28,950	40,877	42,037

Table 9. Cats_vs_Dogs throughput improvement factor results with TF-TRT precision modes for different models.

Batch Size	Throughput (Images per Second)											
	VGG16				ResNet-50				MobileNet_v1			
	Base	FP32	FP16	INT8	Base	FP32	FP16	INT8	Base	FP32	FP16	INT8
8	167	757	2858	4081	163	1747	3086	3774	196	3563	4105	4470
16	308	839	3149	5224	311	2031	4859	5783	388	5788	7041	7108
32	591	848	3169	5711	606	2127	6154	9077	730	6716	10,280	10,024
64	768	842	3271	5718	828	2218	7033	10,172	1093	6967	13,670	16,368





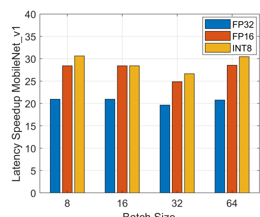
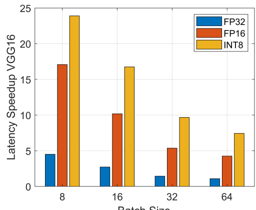
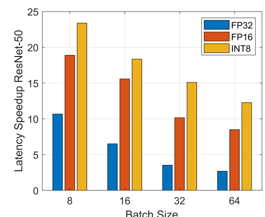


Figure 12. CIFAR-10 latency speedup comparison with TF-TRT for VGG16 (**left**), ResNet-50 (**middle**), and Mobilenet_v1 (**right**) models.





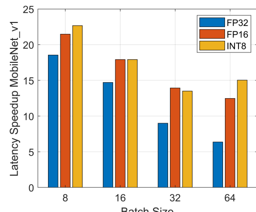
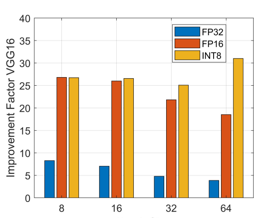
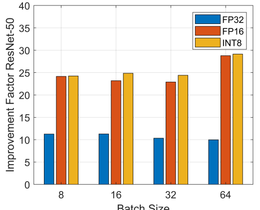


Figure 13. Cats_vs_Dogs latency speedup comparison with TF-TRT for VGG16 (**left**), ResNet-50 (**middle**), and Mobilenet_v1 (**right**) models.





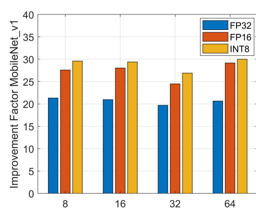
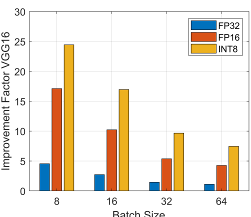
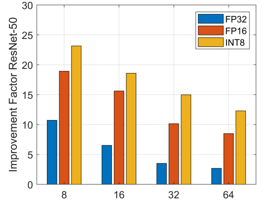


Figure 14. CIFAR-10 throughput improvement factor comparison with TF-TRT for VGG16 (**left**), ResNet-50 (**middle**), and Mobilenet_v1 (**right**) models.





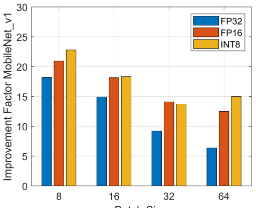


Figure 15. Cats_vs_Dogs throughput improvement factor comparison with TF-TRT for VGG16 (**left**), ResNet-50 (**middle**), and MobileNet_v1 (**right**) models.

Low-precision formats offer a large latency speedup and throughput improvement factor for VGG16; however, this is not the case for MobileNet_v1. This effect is due to the difference in model parameters. VGG16, with a large number of parameters, shows

great latency speedup and throughput improvement in low-precision formats. In the case of MobileNet_v1, because it inherently has fewer parameters and efficient architecture, the post-training quantization effect of TF-TRT does not show significant speedup between low-precision formats.

The results reveal that MobileNet_v1 exhibits less speedup and improvement in performance metrics as compared to ResNet-50 and VGG16 for AMP and various TensorRT low-precision formats. This is mainly due to the depthwise separable convolutional layers in MobileNet_v1, which reduce the number of parameters and computations used in convolutional operations. Hence, MobileNet_v1 shows worse quantization performance in terms of training time and memory.

6.3. Model Accuracy

Model accuracy was compared for different precision modes of post-training quantization with and without AMP training. We observed that accuracy drops slightly in FP16 and INT8 quantization without AMP training, whereas in certain cases accuracy for FP16 is slightly increased. This is mainly due to the fact that quantization provides regularization in these cases. Therefore, lowering the precision from FP32 to FP16 results in a slight increase in accuracy on the CIFAR-10 test dataset for VGG16 without the AMP technique, as shown in Table 10.

With AMP training, model accuracy becomes more consistent with regard to the quantization techniques for TensorFlow FP32, TFTRT FP32, and TFTRT FP16, as shown in Table 11. Therefore, AMP not only provides memory reduction and training time speedup, it can help to adjust model weights during the training stage in order to compensate for quantization error due to low-precision values of weights and activations.

Table 10. CIFAR-10 accuracy for different models.

Model Accuracy				
Model	**TensorFlow FP32**	**TF-TRT FP32**	**TF-TRT FP16**	**TF-TRT INT8**
MobileNet_v1 without AMP	85.383	85.383	85.332	83.921
MobileNet_v1 with AMP	84.576	84.576	84.576	84.274
VGG16 without AMP	89.120	89.120	89.171	88.894
VGG16 with AMP	88.709	88.709	88.709	88.960
ResNet-50 without AMP	86.895	86.895	86.845	85.721
ResNet-50 with AMP	87.01	87.01	87.01	88.15

Table 11. Cats_vs_Dogs accuracy for different models.

Model Accuracy				
Model	**TensorFlow FP32**	**TF-TRT FP32**	**TF-TRT FP16**	**TF-TRT INT8**
MobileNet_v1 without AMP	96.484	96.484	96.289	95.898
MobileNet_v1 with AMP	96.289	96.289	97.465	96.484
VGG16 without AMP	92.578	93.554	92.773	92.359
VGG16 with AMP	93.516	93.516	93.516	93.724
ResNet-50 without AMP	94.335	94.335	94.110	93.825
ResNet-50 with AMP	94.117	94.117	94.214	94.017

6.4. Quantization in Other Deep Learning Frameworks

In addition to AMP and TRT/TF-TRT, there are several other quantization libraries and frameworks available for deep learning models. These libraries offer tools and techniques to quantize model weights and activations in order to make them more efficient

for deployment on resource-constrained devices. TensorFlow Lite is one of those quantization libraries; it includes tools for quantization and post-training quantization (PTQ). It is commonly used for deploying models on mobile and edge devices. PyTorch provides support for quantization-aware training and post-training quantization, enabling efficient deployment of models. The Open Neural Network Exchange (ONNX) runtime, an inference engine for ONNX models, offers support for quantized models, allowing for faster and more memory-efficient inference. The CMSIS (Cortex Microcontroller Software Interface Standard) library includes CMSIS-NN, which provides quantization functions for optimizing models on ARM Cortex-M processors and microcontrollers. BNN-PYNQ is a library for quantized neural networks targeting FPGA platforms. Intel's OpenVINO toolkit includes tools for quantizing and optimizing deep learning models for Intel hardware, including central processing units (CPUs) and accelerators.

These libraries provide varying levels of support for different hardware platforms and model architectures. The choice of a quantization library may depend on the target hardware, the specific deep learning framework being used, and the level of customization and optimization required for an application. We clarify that our work can be extended to these frameworks and other deep learning models to determine key factors affecting the performance of quantization for these frameworks on different deep learning models.

7. Conclusions

In this paper, we have studied the performance of different quantization techniques during the training and inference stages. The behavior of VGG16, ResNet-50, and MobileNet_v1 were observed with different batch sizes, datasets, and image sizes. For the AMP and TF-TRT low-precision formats, we observed the trends in different performance metrics such as accuracy, memory usage, training time, latency, and throughput. In order to select the appropriate batch size for training and inference, model profiling can help to see the trend of different metrics in the training and inference stages and choose the optimal batch size that provides the most desirable performance.

In this work, we have directed our attention to classification models in order to determine the prime factors in classification model architectures that affect quantization performance. We found that model parameters are a major factor that affects quantization performance. VGG16 has a large number of parameters, while MobileNet_v1 and ResNet-50 reduce their parameters using depthwise separable convolutional layers and 1×1 filters, resulting in lesser speedup and improvement factors after quantization as compared to VGG16.

This work can be extended to other deep learning models, such as natural language processing, graph neural networks, pose estimation, and segmentation models, in order to obtain insights into the effect of quantization on the performance of these models. The characterization of the optimization techniques adopted in deep learning frameworks can help researchers to adopt best practices for using these optimizations to obtain efficient results in the training and inference stages. This characterization can enable researchers to develop more efficient deep learning optimization techniques by understanding their performance on different models and datasets.

Author Contributions: Conceptualization, M.A.S. and A.M.; methodology, M.A.S. and A.M.; software, M.A.S.; validation, M.A.S. and A.M.; formal analysis, M.A.S.; investigation, A.M.; resources, M.A.S. and A.M.; data curation, M.A.S.; writing—original draft preparation, M.A.S. and A.M.; writing—review and editing, M.A.S., A.M. and J.K.; supervision, A.M.; project administration, A.M.; funding acquisition, M.A.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The source code for this work is available at: https://github.com/alishafique3/Deep_Learning_Performance_Characterization_for_Quantization_Frameworks (accessed on 10 October 2023).

Acknowledgments: The authors of this study acknowledge the University of Engineering and Technology (UET) Lahore for support and affiliation with author Muhammad Ali Shafique.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. LeCun, Y.; Bengio, Y.; Hinton, G. Deep Learning. *Nature* **2015**, *521*, 436–444. [CrossRef] [PubMed]
2. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjeland, A.K.; Ostrovski, G.; et al. Human-level Control through Deep Reinforcement Learning. *Nature* **2015**, *518*, 529–533. [CrossRef] [PubMed]
3. Silver, D.; Huang, A.; Maddison, C.J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* **2016**, *529*, 484–489. [CrossRef] [PubMed]
4. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet Classification with Deep Convolutional Neural Networks. *Commun. ACM* **2017**, *60*, 84–90. [CrossRef]
5. Simonyan, K.; Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv* **2015**, arXiv:1409.1556.
6. Costa-jussà, M.R.; Fonollosa, J.A. Latest trends in hybrid machine translation and its applications. *Comput. Speech Lang.* **2015**, *32*, 3–10. [CrossRef]
7. Otter, D.W.; Medina, J.R.; Kalita, J.K. A survey of the usages of deep learning for natural language processing. *IEEE Trans. Neural Networks Learn. Syst.* **2020**, *32*, 604–624. [CrossRef]
8. Zhang, S.; Yao, L.; Sun, A.; Tay, Y. Deep Learning Based Recommender System: A survey and New Perspectives. *ACM Comput. Surv. (CSUR)* **2019**, *52*, 1–38. [CrossRef]
9. Ohn-Bar, E.; Trivedi, M.M. Looking at Humans in the Age of Self-driving and Highly Automated Vehicles. *IEEE Trans. Intell. Veh.* **2016**, *1*, 90–104. [CrossRef]
10. Bojarski, M.; Del Testa, D.; Dworakowski, D.; Firner, B.; Flepp, B.; Goyal, P.; Jackel, L.D.; Monfort, M.; Muller, U.; Zhang, J.; et al. End to end Learning for Self-driving Cars. *arXiv* **2016**, arXiv:1604.07316.
11. Lu, H.; Li, Y.; Mu, S.; Wang, D.; Kim, H.; Serikawa, S. Motor Anomaly Detection for Unmanned Aerial Vehicles Using Reinforcement Learning. *IEEE Internet Things J.* **2017**, *5*, 2315–2322. [CrossRef]
12. Hadidi, R.; Cao, J.; Woodward, M.; Ryoo, M.S.; Kim, H. Distributed Perception by Collaborative Robots. *IEEE Robot. Autom. Lett.* **2018**, *3*, 3709–3716. [CrossRef]
13. Pfeiffer, M.; Schaeuble, M.; Nieto, J.; Siegwart, R.; Cadena, C. From Perception to Decision: A Data-driven Approach to End-to-end Motion Planning for Autonomous Ground Robots. In Proceedings of the 2017 IEEE International Conference on Robotics and Automation (ICRA), Singapore, 29 May–3 June 2017; pp. 1527–1533.
14. Merck, M.L.; Wang, B.; Liu, L.; Jia, C.; Siqueira, A.; Huang, Q.; Saraha, A.; Lim, D.; Cao, J.; Hadidi, R.; et al. Characterizing the Execution of Deep Neural Networks on Collaborative Robots and Edge Devices. In Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning), Chicago, IL, USA, 28 July–1 August 2019; pp. 1–6.
15. Han, S. Efficient Methods and Hardware for Deep Learning. Available online: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture15.pdf (accessed on 28 October 2022).
16. Yasrab, R.; Gu, N.; Zhang, X. An Encoder-Decoder Based Convolution Neural Network (CNN) for Future Advanced Driver Assistance System (ADAS). *Appl. Sci.* **2017**, *7*, 312. [CrossRef]
17. Aladem, M.; Rawashdeh, S.A. A Single-Stream Segmentation and Depth Prediction CNN for Autonomous Driving. *IEEE Intell. Syst.* **2020**, *36*, 79–85. [CrossRef]
18. Yang, M.; Wang, S.; Bakita, J.; Vu, T.; Smith, F.D.; Anderson, J.H.; Frahm, J.M. Re-thinking CNN Frameworks for Time-Sensitive Autonomous-Driving Applications: Addressing an Industrial Challenge. In Proceedings of the 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Montreal, QC, Canada, 16–18 April 2019; pp. 305–317.
19. Hasan, I.; Liao, S.; Li, J.; Akram, S.U.; Shao, L. Pedestrian Detection: Domain Generalization, CNNs, Transformers and Beyond. *arXiv* **2022**, arXiv:c2201.03176.
20. Muhammad Rastegari, M.C. Efficient Methods and Hardware for Deep Learning. Available online: https://nips.cc/Conferences/2016/Schedule?showEvent=6234 (accessed on 7 January 2023).
21. Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language Models are Few-Shot Learners. In *Proceedings of the Advances in Neural Information Processing Systems*; Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H., Eds.; Curran Associates, Inc.: Red Hook, NY, USA, 2020; Volume 33, pp. 1877–1901.
22. Han, S.; Mao, H.; Dally, W.J. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. *arXiv* **2016**, arXiv1510.00149.
23. Yu, J.; Lukefahr, A.; Palframan, D.; Dasika, G.; Das, R.; Mahlke, S. Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. *ACM SIGARCH Comput. Archit. News* **2017**, *45*, 548–560. [CrossRef]
24. Lin, J.; Rao, Y.; Lu, J.; Zhou, J. Runtime Neural Pruning. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 1–11.

25. Wen, W.; Wu, C.; Wang, Y.; Chen, Y.; Li, H. Learning Structured Sparsity in Deep Neural Networks. *Adv. Neural Inf. Process. Syst.* **2016**, *29*, 1–9.
26. Son, S.; Nah, S.; Lee, K.M. Clustering Convolutional Kernels to Compress Deep Neural Networks. In Proceedings of the European Conference on Computer Vision (ECCV), Munich, Germany, 8–14 September 2018; pp. 216–232.
27. Courbariaux, M.; Bengio, Y.; David, J.P. Training Deep Neural Networks with Low Precision Multiplications. *arXiv* **2014**, arXiv:1412.7024.
28. Gong, Y.; Liu, L.; Yang, M.; Bourdev, L. Compressing Deep Convolutional Networks using Vector Quantization. *arXiv* **2014**, arXiv:1412.6115.
29. Vanhoucke, V.; Senior, A.; Mao, M.Z. Improving the Speed of Neural Networks on CPUs. In Proceedings of the Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011, Granada, Spain, 12–17 December 2011.
30. Köster, U.; Webb, T.; Wang, X.; Nassar, M.; Bansal, A.K.; Constable, W.; Elibol, O.; Gray, S.; Hall, S.; Hornof, L.; et al. Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks. In *Proceedings of the Advances in Neural Information Processing Systems*; Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R., Eds.; Curran Associates, Inc.: Red Hook, NY, USA, 2017; Volume 30.
31. Ye, S.; Zhang, T.; Zhang, K.; Li, J.; Xie, J.; Liang, Y.; Liu, S.; Lin, X.; Wang, Y. A Unified Framework of DNN Weight Pruning and Weight Clustering/Quantization using ADMM. *arXiv* **2018**, arXiv:1811.01907.
32. Hadidi, R.; Cao, J.; Xie, Y.; Asgari, B.; Krishna, T.; Kim, H. Characterizing the Deployment of Deep Neural Networks on Commercial Edge Devices. In Proceedings of the 2019 IEEE International Symposium on Workload Characterization (IISWC), Orlando, FL, USA, 3–5 November 2019; pp. 35–48.
33. Hubara, I.; Courbariaux, M.; Soudry, D.; El-Yaniv, R.; Bengio, Y. Binarized neural networks. *Adv. Neural Inf. Process. Syst.* **2016**, *29*, 1–9.
34. Liu, B.; Li, F.; Wang, X.; Zhang, B.; Yan, J. Ternary weight networks. In Proceedings of the ICASSP 2023–2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Rhodes Island, Greece, 4–10 June 2023; pp. 1–5.
35. Banner, R.; Hubara, I.; Hoffer, E.; Soudry, D. Scalable methods for 8-bit training of neural networks. *Adv. Neural Inf. Process. Syst.* **2018**, *31*, 1–9.
36. Chmiel, B.; Ben-Uri, L.; Shkolnik, M.; Hoffer, E.; Banner, R.; Soudry, D. Neural gradients are near-lognormal: Improved quantized and sparse training. *arXiv* **2020**, arXiv:2006.08173.
37. Faghri, F.; Tabrizian, I.; Markov, I.; Alistarh, D.; Roy, D.M.; Ramezani-Kebrya, A. Adaptive gradient quantization for data-parallel sgd. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 3174–3185.
38. Kim, J.; Yoo, K.; Kwak, N. Position-based scaled gradient for model quantization and pruning. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 20415–20426.
39. Wang, N.; Choi, J.; Brand, D.; Chen, C.Y.; Gopalakrishnan, K. Training deep neural networks with 8-bit floating point numbers. *Adv. Neural Inf. Process. Syst.* **2018**, *31*, 1–10.
40. Ginsburg, B.; Nikolaev, S.; Kiswani, A.; Wu, H.; Gholaminejad, A.; Kierat, S.; Houston, M.; Fit-Florea, A. Tensor Processing Using Low Precision Format. U.S. Patent App. 15/624,577, 28 December 2017.
41. Micikevicius, P.; Narang, S.; Alben, J.; Diamos, G.; Elsen, E.; Garcia, D.; Ginsburg, B.; Houston, M.; Kuchaiev, O.; Venkatesh, G.; et al. Mixed Precision Training. *arXiv* **2018**, arXiv:1710.03740.
42. Munir, A.; Blasch, E.; Kwon, J.; Kong, J.; Aved, A. Artificial Intelligence and Data Fusion at the Edge. *IEEE Aerosp. Electron. Syst. Mag.* **2021**, *36*, 62–78. [CrossRef]
43. Ravi, A.; Chaturvedi, V.; Shafique, M. ViT4Mal: Lightweight Vision Transformer for Malware Detection on Edge Devices. *ACM Trans. Embed. Comput. Syst.* **2023**, *22*, 1–26. [CrossRef]
44. Tonellotto, N.; Gotta, A.; Nardini, F.M.; Gadler, D.; Silvestri, F. Neural network quantization in federated learning at the edge. *Inf. Sci.* **2021**, *575*, 417–436. [CrossRef]
45. Nvidia. Nvidia TensorRT. Available online: https://developer.nvidia.com/tensorrt (accessed on 11 March 2023).
46. Wang, X.; Yue, X.; Li, H.; Meng, L. A High-efficiency Dirty-egg Detection System based on YOLOv4 and TensorRT. In Proceedings of the 2021 International Conference on Advanced Mechatronic Systems (ICAMechS), Tokyo, Japan, 9–12 December 2021; pp. 75–80.
47. Chunxiang, Z.; Jiacheng, Q.; Wang, B. YOLOX on Embedded Device with CCTV & TensorRT for Intelligent Multicategories Garbage Identification and Classification. *IEEE Sens. J.* **2022**, *22*, 16522–16532.
48. Tao, L.; Hong, T.; Guo, Y.; Chen, H.; Zhang, J. Drone Identification Based on CenterNet-TensorRT. In Proceedings of the 2020 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB), Paris, France, 27–29 October 2020; pp. 1–5.
49. Wang, Z.; Guo, J.; Hu, Z.; Zhang, H.; Zhang, J.; Pu, J. Lane Transformer: A High-Efficiency Trajectory Prediction Model. *IEEE Open J. Intell. Transp. Syst.* **2023**, *4*, 2–13. [CrossRef]
50. Akkad, G.; Mansour, A.; Inaty, E. Embedded Deep Learning Accelerators: A Survey on Recent Advances. *IEEE Tran. Artif. Intell.* **2023**, *1*, 1–19. [CrossRef]
51. Google. Coral. Available online: https://coral.ai/ (accessed on 23 September 2023).
52. Lotti, A.; Modenini, D.; Tortora, P.; Saponara, M.; Perino, M.A. Deep Learning for Real Time Satellite Pose Estimation on Low Power Edge TPU. *arXiv* **2022**, arXiv:2204.03296.

53. Jacob, B.; Kligys, S.; Chen, B.; Zhu, M.; Tang, M.; Howard, A.; Adam, H.; Kalenichenko, D. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *arXiv* **2017**, arXiv1712.05877.
54. Li, Z.; Li, H.; Meng, L. Model Compression for Deep Neural Networks: A Survey. *Computers* **2023**, *12*, 60. [CrossRef]
55. Li, L.; Li, Q.; Zhang, B.; Chu, X. Norm Tweaking: High-performance Low-bit Quantization of Large Language Models. *arXiv* **2023**, arXiv:2309.02784.
56. Kaur, I.; Jadhav, A.J. Survey on Computer Vision Techniques for Internet-of-Things Devices. In Proceedings of the 2023 IEEE International Conference on Industry 4.0, Artificial Intelligence, and Communications Technology (IAICT), Bali, Indonesia, 13–15 July 2023; pp. 244–250. [CrossRef]
57. Gholami, A.; Kim, S.; Dong, Z.; Yao, Z.; Mahoney, M.W.; Keutzer, K. A Survey of Quantization Methods for Efficient Neural Network Inference. *arXiv* **2021**, arXiv:2103.13630.
58. Nagel, M.; Fournarakis, M.; Amjad, R.A.; Bondarenko, Y.; Van Baalen, M.; Blankevoort, T. A white paper on neural network quantization. *arXiv* **2021**, arXiv:2106.08295.
59. TensorFlow. Model Optimization. Available online: https://www.tensorflow.org/model_optimization/guide/quantization/post_training (accessed on 23 September 2023).
60. Kirtas, M.; Oikonomou, A.; Passalis, N.; Mourgias-Alexandris, G.; Moralis-Pegios, M.; Pleros, N.; Tefas, A. Quantization-aware training for low precision photonic neural networks. *Neural Netw.* **2022**, *155*, 561–573. [CrossRef] [PubMed]
61. Nvidia. Train with Mixed Precision. Available online: https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html (accessed on 23 September 2023).
62. Yao, Z.; Wu, X.; Li, C.; Youn, S.; He, Y. ZeroQuant-V2: Exploring Post-training Quantization in LLMs from Comprehensive Study to Low Rank Compensation. *arXiv* **2023**, arXiv:2303.08302.
63. Nvidia. Accelerating Inference in TensorFlow with TensorRT User Guide. Available online: https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html (accessed on 21 January 2023).
64. Krizhevsky, A. *Learning Multiple Layers of Features from Tiny Images*; University of Toronto: Toronto, ON, Canada, 2009.
65. Nvidia. Contents of the TensorFlow Container. Available online: https://docs.nvidia.com/deeplearning/frameworks/tensorflow-release-notes/rel_22-03.html (accessed on 17 February 2023).
66. Elson, J.; Douceur, J.J.; Howell, J.; Saul, J. Asirra: A CAPTCHA that Exploits Interest-Aligned Manual Image Categorization. In Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS), Alexandria, VI, USA, 31 October–2 November 2007.
67. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv* **2017**, arXiv:1704.04861.
68. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778.
69. Nvidia. Real Time Means Real Change Nvidia Quadro RTX 4000. Available online: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/quadro-rtx-4000-data-sheet-us-nvidia-830682-r6-web.pdf (accessed on 19 November 2022).
70. Nvidia. Compute Capability 7.x. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability-7-x (accessed on 10 March 2023).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.