Efficient Deployment of Object Detection Model (YOLO) on NVIDIA Jetson Devices Through Model Quantization

By:

Haady um Minallah

(Registration No: MS-SE-20-327801)

<u>Supervisor:</u>
Dr. Ali Hassan

DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING,
COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING,
NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY,
ISLAMABAD

July 29, 2024

Annex A

THESIS ACCEPTANCE CERTIFICATE

Certified that final copy of MS/MPhil thesis written by NS **Haady um Minallah** Registration No. **00000327801**, of College of E&ME has been vetted by undersigned, found complete in all respects as per NUST Statutes/Regulations, is free of plagiarism, errors and mistakes and is accepted as partial fulfillment for award of MS/MPhil degree. It is further certified that necessary amendments as pointed out by GEC members of the scholar have also been incorporated in the thesis.

Signature : [illegible]

Name of Supervisor: Dr Ali Hassan

Date: 29-07-2024

Signature of HOD: [illegible]
(Dr Usman Qamar)
Date: 29-07-2024

Signature of Dean: [illegible]
(Brig Dr Nasir Rashid)
Date: 29 JUL 2024

Efficient Deployment of Object Detection Model (YOLO) on NVIDIA Jetson Devices Through Model Quantization

By
Haady um Minallah
(Registration No: 00000327801)

A thesis submitted to the National University of Sciences and Technology
Islamabad
in partial fulfillment of the requirements for the degree of

Master of Sciences in Software Engineering

Supervisor
Dr. Ali Hassan

DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING,
COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING,
NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY
ISLAMABAD

July 29, 2024

Dedicated to my family, whose unwavering support and encouragement have been my guiding light throughout my academic journey. To my mentors and supervisor, whose wisdom and guidance have shaped my knowledge and skills. And to my friends and colleagues, whose companionship and encouragement have made this journey memorable.

ACKNOWLEDGEMENTS

I am extremely grateful to Allah for his sanctifications throughout this work. Indeed, this would not have been possible without his substantial guidance through every step, and for putting me across people who could drive me through this work in a superlative manner. Undeniably, none be worth praise but the Almighty.

I am grateful to my parents for their support in my education. I also thank my siblings who encouraged me and prayed for me throughout the time of my research.

I would also like to show my gratefulness to my supervisor Dr. Ali Hassan for his motivation, patience, enthusiasm, and immense help. His supervision assisted me through my research and completion of this thesis.

I would also like to acknowledge my Guidance Committee Members Dr. Muhammad Yasin and Dr Farhan Hussain for their support. Their suggestions were very valued for the completion of my work.

Last but not the least, I would like to express my appreciation to all the people who have helped me in my study.

Thanks for all the support!

ABSTRACT

This thesis investigates the effective implementation of the YOLO object recognition model on NVIDIA Jetson through the application of model quantization techniques. Specifically, the research focuses on Quantization-Aware Training (QAT) and Asymmetric Quantization to optimize the model's performance on resource inhibited edge computers. NVIDIA Jetson devices, compatible and aimed at handling AI tasks in edge computing scenarios, often face limitations in memory, power, and computational capacity. The research evaluates the baseline performance of the YOLO model on a standard NVIDIA Jetson device and detail the methodologies of applying QAT and Asymmetric Quantization, followed by a comparative analysis of their effects. The results indicate that while quantization techniques lead to a slight decrease in accuracy, they substantially enhance inference time. This improvement in inference speed underscores the potential for deploying the quantized YOLO model in real-time scenarios where inference time is prioritized over accuracy. This thesis contributes to the fields of edge computing and real-time image processing by providing a comprehensive framework for deploying high-performance AI models in constrained environments. The findings demonstrate that model quantization is a viable strategy for achieving efficient and robust real-time object recognition on devices that have resource limitations.

Keywords: Quantization aware training, Asymetric Quantization Object detection, Edge computing, YOLO, NVIDIA Jetson

Contents

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

ARM	Advanced RISC Machine (a type of CPU architecture)
CUDA	Compute Unified Device Architecture (NVIDIA's parallel computing platform)
FPS	Frames Per Second
mAP	Mean Average Precision
QAT	Quantization-Aware Training
TPU	Tensor Processing Unit
TRT	TensorRT (NVIDIA's deep learning inference optimizer)
YOLO	You Only Look Once (a family of object detection models)
INT8	8-bit integer representation
FP32	32-bit floating-point representation
COCO	Common Objects in Context (a dataset for object detection)
Inference Time	The time taken to process an image and produce detections

CHAPTER 1: INTRODUCTION

In recent years, artificial intelligence (AI) has brought transformative changes to various domains, notably computer vision, where object detection is a key component. Object recognition includes classifying and pinpointing items within pictures or movie frames, with applications in autonomous vehicles, surveillance, robotics, and more. Amongst the numerous object detection models, the YOLO is famous by its ideal balance of inference time and precision, making it highly effective for real-time uses. The emergence of edge computing, which processes data near its source rather than in a centralized data-processing center, has highlighted the need for efficient object detection. Devices like the NVIDIA Jetson, known for their AI prowess, facilitate the deployment of advanced AI models in practical settings such as smart cameras, drones, and industrial IoT systems. This thesis seeks to tackle the challenges associated with deploying the YOLO object detection model on NVIDIA Jetson devices by implementing model quantization techniques. Quantization decreases the accuracy of the parameters from floating-point to integers, significantly decreasing model size and computational requirements. This optimization is essential for improving the performance of AI models in resource-limited environments, enabling effective real-time object detection.

1.1 Background Study

The increasing demand for instantaneous processing capabilities in various applications has led to the rise of edge devices which involves data being processed close to the source of where the data is being generated, dropping latency, and improving efficiency compared to traditional cloud-based solutions. NVIDIA Jetson devices have emerged as powerful platforms for edge computing due to their robust AI processing capabilities, making them suitable for a wide range of applications, from autonomous vehicles to smart cameras and industrial automation [8].

1.1.1 Object Detection Models

Object recognition is a critical job in computer visualization, involving the reignition of objects within pictures. Among the various object recognition models, the YOLO series is highly regarded for its real-time performance and accuracy. The YOLO model processes an entire image in a single forward pass through the network, predicting bounding boxes and class probabilities simultaneously. This approach contrasts with traditional methods like R-CNN and its variants, which generate region proposals and then classify each region, leading to slower inference speeds [1].

1.1.2 YOLO Model and Its Evolution

Version	Year	Key Improvements	Reference
YOLOv1	2016	Introduced a single-stage object detection	Redmon et al.,[1]
YOLOv2	2017	Better backbone networks, multi-scale predictions	Redmon & Farhadi, 2017 [2]
YOLOv3	2018	Multi-scale predictions, improved detection accuracy	Redmon & Farhadi, 2018 [3]
YOLOv4	2020	Bag of freebies and specials, better performance and speed	Bochkovskiy et al., 2020 [4]
YOLOv5	2021	Improved training techniques, ease of use	Jocher, 2021 [5]

Table 1.1 Evolution of YOLO Models

The YOLO model, first introduced from “Joseph Redmon et al.,” has undergone several iterations, each improving upon its predecessor in terms of accuracy and speed. YOLOv1, YOLOv2 (also known as YOLO9000), YOLOv3, and the more recent YOLOv4 and YOLOv5 versions have incorporated various enhancements such as better backbone networks, multi-scale predictions, and advanced training techniques. These improvements have made it the best real-time object detection framework [1] [2] [3] [4] [5].

Model	**Speed (FPS)**	**Accuracy (mAP)**	**Key Features**
R-CNN	<1 FPS	66%	Region proposals, two-stage detector
Fast R-CNN	2 FPS	70%	Improved region proposals, faster than R-CNN
Faster R-CNN	5 FPS	73%	Region proposal network, faster and more accurate
SSD	22 FPS	74.3%	Single-shot detector, multi-scale feature maps
YOLOv3	45 FPS	57.9%	Single-stage detector, real-time performance
YOLOv4	62 FPS	65.7%	Improved backbone, bag of freebies and specials
YOLOv5	140 FPS	68.9%	Highly optimized training, ease of use

Table 1.2 Comparison of Object Detection Models

1.1.3 Challenges in Deploying YOLO on Edge Devices

Despite its advantages, deploying YOLO on edge devices like NVIDIA Jetson poses significant challenges. The primary issues include the model's computational and memory requirements, which can strain the limited resources available on edge devices.

Additionally, maintaining real-time performance while ensuring high detection accuracy is a critical concern. Addressing these challenges necessitates optimizing the model to reduce its size and computational demands without compromising its efficacy.

1.1.4 Model Quantization

Model quantization is a useful method for optimizing machine learning algorithms for implementation on resource limited devices. For performing quantization reduce the accuracy of the parameters from decimal (e.g., FP32) to integer (e.g., INT8). This reduction cuts the model size and complexity, leading to faster inference resulting in less power consumed [6]. There are numerous quantization methods, including asymmetric quantization and QAT (quantization-aware training), each with its trade-offs between ease of implementation and impact on model accuracy [6] [7] [21].

1.1.5 Previous Research on Quantization and Edge Deployment

Numerous studies have explored use-cases of model quantization to several machine learning models, demonstrating significant improvements in performance on edge devices. For instance, Jacob et al. (2018) discussed the benefits of quantizing convolutional neural networks (CNNs) for effectual inference [6]. Similarly, "Han et al. (2015)" presented techniques like "pruning, trained quantization, and Huffman coding to compress neural networks" by doing so they achieved advanced results in terms of size of model and inference time [7] [23] [25].

Study	Technique(s) Used	Key Findings
“Jacob et al. (2018)”	Quantizing CNNs	Improved inference efficiency with acceptable accuracy loss
“Han et al., 2015”	Pruning, trained quantization, Huffman coding	Significant reductions in model size and computational requirements, state-of-the-art performance
“Micikevicius et al., 2018”	Mixed-precision training	Substantial speedups in training and inference with minimal impact on accuracy
“Rastegari et al., 2016”	XNOR-Net (Binary Neural Networks)	Binary convolutional neural networks, drastically reducing model size and improving speed
“Zhou et al., 2016”	DoReFa-Net	Training low bitwidth convolutional neural networks with low bitwidth gradients

Table 1.3: Key Findings from Previous Research on Quantization

1.2 Problem Statement

The main issue talked about in this study is how to efficiently deploy YOLO object detection model on NVIDIA Jetson devices. While the YOLO model is highly effective for instantaneous object recognition, its computational and storage requirements pose significant challenges when deployed on devices having limitation of resources. Traditional deployment approaches may not leverage the full potential of Jetson devices, resulting in suboptimal performance and higher power consumption.

1.3 Objectives

The key objectives for this research:

- **Baseline Performance Evaluation:** Assess the initial performance of the YOLO model on a standard NVIDIA Jetson device to establish a benchmark.
- **Implementation of Quantization Techniques:** Apply various model quantization techniques to the YOLO model, focusing on reducing its precision while maintaining detection accuracy.
- **Comparative Analysis:** Conduct a comparative analysis of different quantization methods to determine their effectiveness in enhancing the performance of YOLO on Jetson devices.
- **Performance Optimization:** Optimize the quantized YOLO model for real-time applications by reducing its computational burden and improving inference speed.

1.4 Significance of the Study

The worth of this research is the fact that it can bridge the gap between highly performant models and resource-limited devices. By leveraging model quantization techniques, this research intends to facilitate deployment of strong and effective real-time object detection systems in numerous applications, from self-driving vehicles to smart security systems.
The findings of this study could lead to broader adoption of AI technologies in edge computing scenarios, unlocking new possibilities for innovation and efficiency.

1.5 Structure of the Thesis

This research is structured as follows:

- **Chapter 1: Introduction**: Gives a summary of existing study on object detection models, model quantization techniques, and their deployment on edge devices.
- **Chapter 2: Literature Review**: Delivers an outline of existing studies relevant to object detection models, model quantization techniques, and their deployment on edge devices.
- **Chapter 3: Methodology**: Details the methodologies and experimental setups for evaluating the baseline results of YOLO and to implement various quantization techniques.
- **Chapter 4: Implementing Quantization**: Implementing different quantization methods applied to the YOLO model, including post-training quantization, and discusses their impact on model performance.
- **Chapter 5: Results and Discussion:** Comparative analysis of quantization techniques and their effectiveness in optimizing YOLO for Jetson devices.
- **Chapter 6: Conclusion and Future Work:** Sums the key discoveries of the study, discusses the inferences, and gives directions for future research.

CHAPTER 2: BACKGROUND AND RELATED WORK

This chapter will discuss the background for this research and I am also going to be discussing some of the past research that is done in the field of object recognition on edges devices.

2.1 Background

Object recognition is an essential task which involves identifying and localizing objects within images or video frames. Various object recognition models have been, with YOLO being one of the most prominent because of its balance of accuracy and speed. “The YOLO model processes the entire image in a single forward pass through the network, predicting bounding boxes and class probabilities simultaneously”. This approach contrasts with traditional methods like R-CNN and its variants, which generate region proposals and then classify each region, leading to slower inference speeds [1].

The YOLO model, first introduced by “Joseph Redmon et al.” has undergone several changes. YOLOv1 was groundbreaking in its ability to perform instantaneous object recognition [1]. YOLOv2, also known as YOLO9000, improved upon this by incorporating techniques like batch normalization, anchor boxes, and a more sophisticated loss function [2]. YOLOv3 further enhanced the model's performance by using a deeper network architecture and multi-scale predictions, which improved its ability to detect small objects [3]. More recently, YOLOv4 and YOLOv5 have introduced additional improvements, including advanced data augmentation techniques, better backbone networks, and more efficient training procedures, solidifying YOLO's position as a leading real-time object detection framework [4] [5].

Deploying the YOLO model on edge devices, such as NVIDIA Jetson, presents significant challenges. These devices, while powerful, have limited computational resources and memory compared to traditional cloud-based servers. The computational and memory requirements of the YOLO model can strain these limited resources, which makes it hard to achieve instantaneous performance without optimization [8]. Furthermore, maintaining high detection accuracy while optimizing for resource constraints is a critical concern that necessitates advanced techniques.

Model quantization is a key technique for optimizing machine learning models for implementation on devices with limited resources. Quantization reduces the accuracy of the parameters from decimal point (e.g., FP32) to integer (e.g., INT8). This reduction decreases the size of the model and its complexity, which results in lower inference time and less power consumed [6] [11] [12]. Quantization can be applied through various methods, including asymmetric quantization and quantization-aware training. Post-training quantization is straightforward but might result in some loss of accuracy. In contrast, quantization-aware training incorporates quantization into the training process, which can help maintain higher accuracy [6] [7] [13] [28].

Several studies have demonstrated the effectiveness of model quantization in enhancing the performance of deep learning models on edge devices. Jacob et al. (2018) highlighted the benefits of quantizing convolutional neural networks (CNNs) for efficient inference, showing that integer-arithmetic-only inference can be achieved without significant loss of accuracy [6]. Han et al. (2015) introduced techniques like pruning, trained quantization, and Huffman coding to compress neural networks, achieving state-of-the-art performance in terms of model size and inference speed [7]. Additionally, other research has explored the use of mixed precision training to further optimize models for deployment on edge devices, demonstrating significant improvements in both performance and efficiency [14] [15] [20] [26].

The successful deployment of quantized YOLO models on edge devices has significant implications for various applications. In autonomous vehicles, real-time object detection is crucial for navigation and safety. Similarly, in smart surveillance systems, efficient object detection enables real-time monitoring and threat detection. Industrial automation can benefit from real-time quality control and defect detection, enhancing productivity and reducing downtime. By optimizing YOLO models for edge deployment, this research aims to unlock the full potential of AI in these critical applications, contributing to the fields of edge computing and real-time image processing.

2.2 Relevant Work

Recent advancements in deep learning-based object detection have significantly enhanced inference efficiency by leveraging GPUs. However, the deployment of these frameworks on embedded systems and mobile devices remains challenging due to their constrained processing capabilities. To address this issue, "frameworks such as TensorFlow-Lite (TF-Lite) and TensorRT (TRT) have been optimized for different hardware environments". In a relevant study, researchers introduced "a performance inference method that integrates the Jetson monitoring tool with TensorFlow and TRT on the Nvidia Jetson AGX Xavier platform". The findings revealed that TensorFlow exhibited high latency, while TF-TRT and TRT, which leverage Tensor Cores, demonstrated superior efficiency. In contrast, TF-Lite showed the lowest performance due to its limited GPU capabilities, which are tailored for mobile devices. That research underscores the importance of hardware-specific optimization to enhance the performance of machine learning based object recognition on devices with limited resources, providing valuable insights for efficient deployment on platforms like Nvidia Jetson [16] [24] [27].

In another study, researchers investigated the deployment of machine learning based object recognition models on cheap devices like computers having a single board, which typically experience low frames-per-second (FPS) performance. To mitigate this issue, they explored quantization, a tried and tested compression technique that decreases computational demands but may affect detection accuracy. The study, inspired by face mask directives during the outbreak of COVID, aimed to train and compress a YOLO based model for mask on face detection, targeting deployment on a Raspberry Pi 4. Various pruning and quantization methods were evaluated to improve FPS while maintaining detection accuracy. Quantitative assessments of the pruned and quantized models, in terms of Mean Average Precision (mAP) and FPS, showed that these techniques, when properly applied, could double FPS with only a moderate decrease in mAP. These results provide valuable insights for compressing other YOLO-based object detection models, emphasizing the necessary balance between performance and accuracy for efficient deployment on resource-limited devices. [17] [22].

In another related work, researchers have proposed an innovative method to enhance frames-per-second (FPS) while maintaining the accuracy of the YOLO v2 model on the NVIDIA Jetson TX1 platform. Traditionally, reducing computation in neural networks has involved converting operations to integer arithmetic or decreasing network depth, often at the cost of recognition accuracy. To mitigate this, the study introduces techniques that reduce computation and memory consumption without significantly compromising accuracy. The first technique replaces the filters, effectively reducing the number of parameters to one-ninth. The second technique leverages TensorRT's inference acceleration functions, specifically the Convolution-Add Bias-Relu (CBR) operation, to minimize computation. Lastly, the study integrates repeated layers using TensorRT to further reduce memory consumption. Simulation results indicate that while there is a slight 1% decrease in accuracy compared to the original YOLO v2 model, FPS improved significantly from 3.9 to 11. This research provides valuable insights into optimizing YOLO models for real-time object detection on resource-constrained devices by strategically reducing computational load and memory usage [18] [32].

In another related work, researchers have investigated the performance of state-of-the-art object detection models on various edge devices, focusing on NVIDIA Jetson Nano, Raspberry Pi 4 B with Intel Neural Compute Stick 2, and Axis Q1615-LE Mk III security camera with Google EdgeTPU. These devices, equipped with edge computing accelerators from different manufacturers, were evaluated for latency, accuracy, power consumption, and system utilization. The object detection models assessed included SSD-MobileNet-V2, YoloX, and EfficientDet, which are representative of the latest advancements in the field. Notably, only the Jetson Nano could run both YoloX and EfficientDet models. The EdgeTPU demonstrated the fastest performance, processing images in just 8 ms, while the Jetson Nano and Neural Compute Stick 2 required 33 ms and 48 ms per image, respectively. Despite quantization, all models maintained high accuracy levels above 90%. These findings confirm the capability of all tested devices for real-time object detection, suggesting that each device's unique form factor, connectivity, and computational units suit different use cases. The study highlights the potential for further performance enhancements through model profiling to identify and mitigate bottlenecks. This research provides a comprehensive understanding of deploying advanced object detection models on various edge devices, offering valuable insights for optimizing real-time applications in diverse environments [19].

In a study that underscored the increasing need for efficient track inspection systems in the rapidly evolving rail transportation industry, the necessity for advanced solutions was emphasized. The research utilized a combination of deep learning and edge computing, specifically focusing on the YOLO-NAS architecture for inspecting railroad track components. The goal was to harness the capabilities of YOLO-NAS for accurate and rapid detection while overcoming the computational challenges of edge devices. The study revealed that the YOLO-NAS-S-PTQ model struck a remarkable balance, achieving 74.77% mean Average Precision (mAP) and 92.20 Frames Per Second (FPS) on the NVIDIA Jetson Orin platform. Additionally, deploying this model on an edge device with a multiprocessor pipeline led to an inference speed of 60.468 FPS, almost doubling the performance compared to its single-threaded version. Field tests further confirmed the model's effectiveness, demonstrating a recall rate of 80.77% and an accuracy of 96.64%. These results highlight the potential of YOLO-NAS to transform traditional rail component inspection methods by greatly reducing human intervention and minimizing errors. [29].

Advancements in information and signal processing, propelled by artificial intelligence and recent deep learning breakthroughs, have profoundly influenced autonomous driving by improving safety and minimizing human intervention. Typically, existing advanced driver assistance systems (ADASs) are expensive, rendering them unaffordable for many. A study proposed an affordable, versatile embedded system for real-time detection of pedestrians and priority signs. This system, featuring two cameras, an NVIDIA Jetson Nano B01 low-power edge device, and an LCD display, integrates seamlessly into vehicles without taking up significant space, offering a cost-effective alternative. The research primarily aimed to address accidents resulting from failing to yield to other drivers or pedestrians. Unlike previous studies, this research simultaneously tackled traffic sign recognition and pedestrian detection, focusing on five key objects: pedestrians, pedestrian crossings (both signs and road markings), stop signs, and give way signs. The object detection was achieved using a custom-trained SSD-MobileNet convolutional neural network, implemented on the Jetson Nano. The study yielded promising results, establishing the system as a viable option for real-time deployment and significantly contributing to the safety and accessibility of autonomous driving technologies. This research is in line with the goal of leveraging YOLO models for effective object detection on edge devices, highlighting the potential for implementing advanced AI systems in real-time, resource-limited environments. [30] [31].

Device	**Model**	**Latency (ms)**	**FPS**	**Accuracy (mAP)**
Nvidia Jetson Nano	YoloX, EfficientDet	33	N/A	>90%
Raspberry Pi 4 with NCS2	YoloX, EfficientDet	48	N/A	>90%
Axis Q1615-LE Mk III	YoloX, EfficientDet	8	N/A	>90%

Table 2.1: Performance Comparison on Different Edge Devices

The aforementioned studies collectively underscore the critical need for optimizing machine learning based object recognition models for deployment on devices that have resource limitations devices like NVIDIA Jetson platforms and other single-board computers. Techniques such as pruning, quantization, filter size adjustment, and the integration of acceleration functions through frameworks like TensorRT have been shown to significantly enhance performance metrics such as FPS, while maintaining acceptable levels of detection accuracy. These methodologies provide a robust foundation for further exploration and application, demonstrating that strategic modifications to model architecture and computation can lead to substantial improvements in efficiency. As the demand for real-time object detection continues to grow across various industries, these insights offer valuable guidance for developing high-performance, low-latency AI solutions suitable for embedded systems and edge computing environments [27].

CHAPTER 3: METHADOLOGY

This chapter gives a detailed account about the methodologies and experimental setups employed to assess the baseline results of the YOLO model on NVIDIA Jetson and to implement various quantization techniques. The primary objective is to optimize the deployment of the YOLO model for efficient object detection on edge devices which have limited resources.

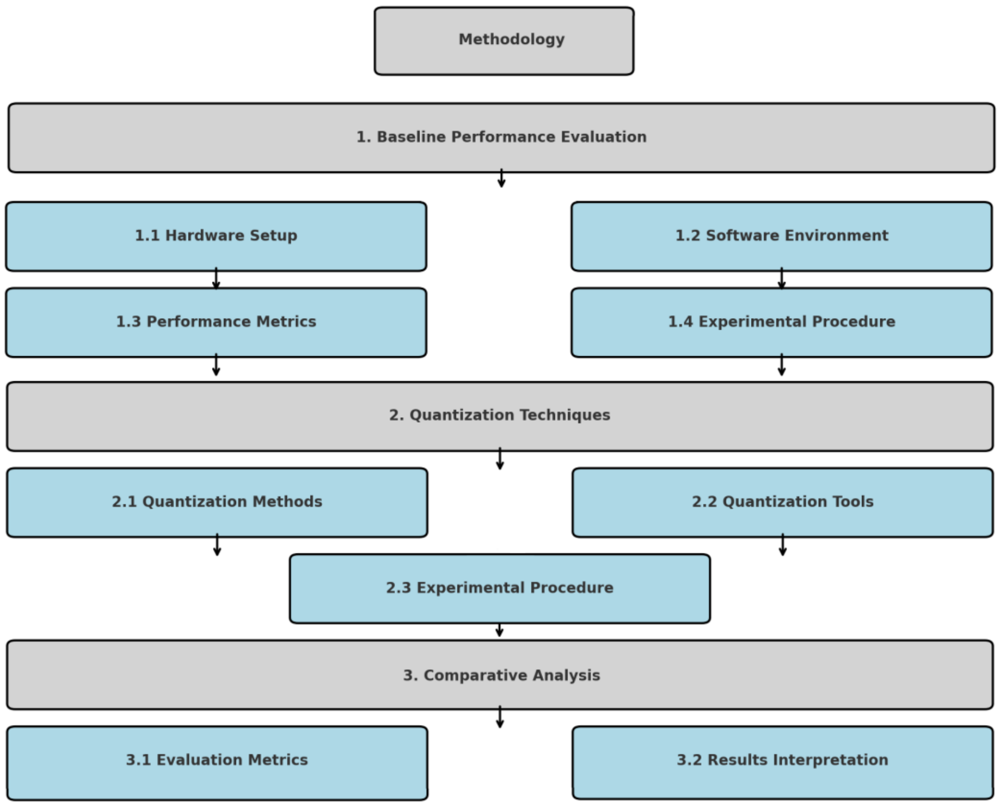


Figure 3.1: Block Diagram for the Research Methodology

Figure 3.1 is an overview of the methodology that was followed in this research, let's discuss the steps involved in the research methodology

3.1 Baseline Performance Evaluation

3.1.1 Hardware Setup:

The NVIDIA Jetson Nano is a powerful, compact, and cost-effective platform designed for AI applications at the edge. The setup for this device is detailed below:

- **Device Configuration:**
  - **Processor:** Quad-core ARM Cortex-A57 CPU
  - **GPU:** 128-core Maxwell GPU
  - **Memory:** 4 GB LPDDR4
  - **Storage:** 16 GB eMMC, expandable via microSD card
  - **Connectivity:** Gigabit Ethernet, 4 USB 3.0 ports, HDMI 2.0, DisplayPort 1.2
- **Power Supply:**
  - The Jetson Nano requires a 5V 4A power supply, provided through a barrel jack connector or a Micro-USB port.

- **Peripherals:**
    - **Display:** Connected via HDMI or DisplayPort.
    - **Keyboard and Mouse:** Connected via USB ports.
    - **Network:** Connected via Ethernet or a compatible Wi-Fi dongle.
- **Setup Procedure:**
    - Download the latest JetPack SDK from NVIDIA's official website
    - Flash the OS image to a microSD card using tools like Etcher.
- **Boot the Device:**
    - Insert the microSD card into the Jetson Nano and connect the power supply.
    - Flash the OS image to a microSD card using tools like Etcher.
    - Power on the device and follow the on-screen instructions to complete the initial setup.
- **Update and Install Dependencies:**
    - Open a terminal and update the packages list.
    - Install necessary libraries and tools for object detection and model quantization.

- **Development Environment:**
    - **IDE:** Visual Studio Code or any preferred text editor.
    - **Frameworks:** PyTorch, TensorFlow, and OpenCV pre-installed as part of the JetPack SDK.
- **Testing Environment:**
    - **Dataset:** COCO dataset, downloaded and stored on an external USB drive or network-attached storage for accessibility.
    - **Benchmarking Tools:** Tools like time, htop, and tegrastats for monitoring system performance and resource utilization during tests.

The NVIDIA Jetson Nano setup ensures that the device is configured correctly to run object detection models and collect performance metrics, providing a solid foundation for the baseline performance evaluation of the YOLO model.

3.1.2 Software Environment:

- **Operating System:**
    - The device runs on the Ubuntu 18.04-based JetPack SDK, which includes necessary drivers, libraries, and developer tools for AI and computer vision applications.

- **YOLO Models:**

For this evaluation, the YOLO (You Only Look Once) models v8 and v9 are utilized. These models are selected due to their state-of-the-art performance in real-time object detection tasks. Here are the specifics of each model:

  - **YOLO V8:** YOLOv8 represents a significant advancement in object detection technology, leveraging a sophisticated architecture designed to extract detailed features and enhance detection accuracy. Trained extensively on the COCO dataset, which encompasses a wide array of everyday objects in diverse environments, YOLOv8 is optimized for real-time performance and high accuracy. Its integration with TensorRT further enhances its inference speed and efficiency on platforms like the Jetson Nano, making it particularly suitable for demanding applications such as surveillance systems and autonomous vehicles where rapid and precise object detection is crucial

  - **YOLO V9:** Building upon the foundation laid by YOLOv8, YOLOv9 introduces further improvements in network efficiency and GPU resource utilization. This iteration maintains its training on the COCO dataset while also fine-tuning specific object categories to elevate detection accuracy to new heights. Inference optimization through TensorRT ensures that YOLOv9 maximizes the computational capabilities of hardware like the Jetson Nano, making it ideal for advanced real-time applications that demand not only exceptional accuracy but also minimal latency. YOLOv9 thus represents a cutting-edge solution for industries requiring reliable and efficient object detection systems in dynamic environments.

- **Model Deployment:**

The models are deployed on the Jetson Nano using Docker containers to ensure consistency and reproducibility of the environment. The deployment process involves the following steps:

  - Install Docker and NVIDIA Container Runtime to enable GPU-accelerated containerized applicationsFlash the OS image to a microSD card using tools like Etcher.

  - Create Docker containers with all the necessary dependencies and configurations for running YOLO models, Containers include the JetPack SDK components, PyTorch, TensorFlow, and other required libraries.

  - Load the YOLOv8 and YOLOv9 models into the containers, perform inference tasks on sample datasets to measure performance metrics such as accuracy and inference time.

This detailed software environment setup ensures that the NVIDIA Jetson Nano is fully equipped to run and evaluate the YOLO models, providing accurate and reliable performance metrics for the baseline performance evaluation.

3.1.3 Performance Metrics:

In evaluating the baseline performance of the YOLO models on the NVIDIA Jetson Nano, several key performance metrics are used. These metrics are critical for understanding the efficiency, accuracy, and overall suitability of the models for real-time object detection on resource-constrained devices. The primary metrics considered in this study are accuracy and inference time.

- **Accuracy:**

  Accuracy is a fundamental metric that measures how well the object detection model identifies and localizes objects within an image. For this evaluation, accuracy is assessed using the following sub-metrics

- **Precision:** Precision is the ratio of true positive detections to the total number of positive detections made by the model (both true positives and false positives). It reflects the model's ability to correctly identify objects without falsely detecting non-existent objects.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

- **Recall:** Recall is the ratio of true positive detections to the total number of actual objects present in the images (true positives and false negatives). It indicates the model's capability to detect all relevant objects in the dataset.

$$\text{Recall} = \frac{\text{True Positives}}{\text{False Negatives} + \text{True Positives}}$$

- **Mean Average Precision (mAP):** mAP is a widely used metric in object detection that combines precision and recall. It calculates the average precision for each class in the dataset and then averages these values. A higher mAP indicates better overall performance of the model.

$$mAP = \frac{1}{n}\sum_{i=1}^{n} APi$$

where n is the number of object classes, and Api is the average precision for the the class.

- **Inference Time:**

  Inference time is a critical metric for evaluating the real-time performance of the object detection models. It measures the time taken by the model to process an image and produce detections. Lower inference time indicates a faster model, which is essential for applications requiring real-time object detection.

- **Data Collection and Analysis:**

  - **Dataset:** The COCO dataset is used for evaluating accuracy metrics, ensuring consistency and comparability with other studies.

    - **Procedures:** Each model is subjected to a series of inference tasks on the same set of images. The metrics are recorded and averaged over multiple runs to ensure reliability and reduce the impact of any anomalies.

3.1.4 Experimental Procedure:

- **Model Preparation:**
    - **Training YOLOv9:** The YOLOv9 model is trained on the COCO dataset using PyTorch. This involves setting up the training environment, configuring the model hyperparameters, and training the model to achieve optimal performance.
    - **Frameworks:** PyTorch, ONNX, and TensorRT.
- **Quantization Techniques:**
    - **Quantization-Aware Training (QAT):** QAT is applied during the training process. This involves simulating lower precision (e.g., INT8) during training, allowing the model to adjust its weights to maintain accuracy after quantization, The model is fine-tuned with QAT to ensure minimal accuracy loss post-quantization
    - **Asymmetric Quantization:** Asymmetric quantization is used to scale and zero-point shift the weights and activations, allowing for a more efficient representation of the model parameters, this technique helps in reducing the

model size and improving inference speed without significantly compromising accuracy

- **Model Conversion and Deployment:** The quantized YOLOv9 model is converted to TensorRT format to leverage the NVIDIA Jetson Nano's GPU capabilities, this involves using the TensorRT API to optimize the model for inference, ensuring faster processing times.

- **Deploying on Jetson Nano:** The converted model is deployed on the Jetson Nano. The device is set up with the necessary software environment, including CUDA, cuDNN, and TensorRT libraries, docker containers are used to ensure a consistent and reproducible deployment environment.

- **Running Inference Tests:** The COCO dataset is used for running inference tests. This ensures that the performance metrics are comparable to those during the training phase, the deployed model is used to run inference on the test images. The process involves loading each image, performing object detection, and recording the results, tools such as time, tegrastats, and custom scripts are used to measure inference times and resource utilization, Inference tests are conducted in batches to simulate real-world scenarios and to collect more reliable performance data.

- **Data Collection and Analysis:** The accuracy of the object detection model is evaluated as the percentage of correctly detected objects out of the total number of objects present in the images, precision and recall metrics are used to support the accuracy assessment, the time taken for the model to process each image and produce detections is measured. This includes both average inference time per image and overall latency, frames Per Second (FPS) is calculated as the inverse of the average inference time, indicating the model's capability to handle real-time processing.

- **Performance Comparison:** The performance of the YOLOv9 model with different quantization techniques (QAT and Asymmetric Quantization) is compared to the baseline (non-quantized) model, Metrics such as % accuracy and inference time are analyzed to draw insights on the trade-offs between model efficiency and performance.

The experimental procedure involves a systematic approach to deploying the YOLOv9 model on the NVIDIA Jetson Nano, applying quantization techniques, running inference tests, and collecting data. By evaluating % accuracy and inference time, this procedure provides a comprehensive understanding of the model's performance and the impact of quantization on real-time object detection in resource-constrained environments.

3.2 Quantization Techniques

There are many quantization techniques but for this research I will be focusing on Quantization-Aware Training (QAT) and Asymmetric Quantization.

3.2.1 Quantization Methods:

Feature	**Quantization-Aware Training**	**Asymmetric Quantization**
Training	During training	Post-training
Precision Handling	Simulates quantization during training	Applies scaling and shifting post-training
Accuracy	Typically, higher post-quantization	May experience some accuracy loss
Complexity	Higher during training	Simpler implementation
Adaptation	Model adapts to quantization	No adaptation during training
Use Cases	High-accuracy applications	Resource-constrained deployment
Implementation	Requires modification of training process	Can be applied to pre-trained models

Table 3.1 Comparison Table of QAT and Asymetric Quantization

Table 3.1 provides a detailed comparison between Quantization-Aware Training (QAT) and Asymmetric Quantization, highlighting key differences and characteristics of each technique.

- **Training:** QAT is performed during the training process, allowing the model to adapt to the quantized representation, whereas Asymmetric Quantization is applied post-training, without requiring retraining of the model.

- **Precision Handling:** QAT simulates quantization during training, enabling the model to adjust its weights to lower precision values. On the other hand, Asymmetric Quantization involves applying scaling and shifting to weights and activations after training to fit them into an integer range.

- **Accuracy:** Models trained with QAT typically achieve higher accuracy post-quantization due to their ability to adapt to lower precision during training. In contrast, models using Asymmetric Quantization may experience some accuracy loss since the quantization is applied after the model has been trained.

- **Complexity:** The training process for QAT is more complex due to the integration of quantization simulation, making it computationally intensive. Asymmetric Quantization, however, has a simpler implementation since it is applied post-training without modifying the training process.

- **Adaptation:** QAT allows the model to adapt to quantization, which helps in maintaining accuracy after quantization. Asymmetric Quantization does not involve any adaptation during training, potentially leading to a drop in accuracy.

- **Use Cases:** QAT is well-suited for high-accuracy applications where maintaining precision is critical, even with reduced precision representation. Asymmetric Quantization is ideal for resource-constrained deployment scenarios where simplicity and efficiency are prioritized.

- **Implementation:** Implementing QAT requires modifying the training process to include quantization simulation, whereas Asymmetric Quantization can be easily applied to pre-trained models without the need for retraining.

Table 3.1 effectively summarizes the trade-offs between QAT and Asymmetric Quantization, providing a clear understanding of the benefits and limitations of each approach in the context of model quantization and deployment on edge devices like the NVIDIA Jetson Nano.

3.2.2 Quantization Tools:

In the context of deploying deep learning models on edge devices such as the NVIDIA Jetson Nano, quantization tools play a crucial role in optimizing models for efficient inference. The primary quantization tools utilized in this research are TensorRT and TensorFlow Lite. Each tool offers unique features and benefits that contribute to reducing model size, improving inference speed, and maintaining accuracy.

TensorRT

Overview: TensorRT is an SDK developed by NVIDIA specifically designed for high-performance deep learning inference. It provides a comprehensive suite of tools to optimize and deploy neural networks on NVIDIA GPUs. TensorRT supports various optimizations, including precision calibration, layer fusion, kernel auto-tuning, and dynamic tensor memory management.

Key Features:

- **Precision Calibration:** TensorRT can calibrate the precision of model weights and activations from floating-point (FP32) to lower precision (INT8), reducing the computational load and memory footprint.
- **Layer Fusion:** Combines multiple neural network layers into a single kernel to minimize memory access and improve computational efficiency.
- **Kernel Auto-Tuning:** Automatically selects the best-performing kernels for each layer of the network based on the target hardware, maximizing performance.
- **Dynamic Tensor Memory Management:** Efficiently manages memory allocation for tensors during inference, reducing memory overhead.

Benefits:

- **High Performance:** TensorRT significantly accelerates inference by leveraging GPU capabilities and advanced optimization techniques.
- **Reduced Latency:** Optimized models exhibit lower latency, making TensorRT ideal for real-time applications.
- **Scalability:** TensorRT can be used across various NVIDIA platforms, from data centers to edge devices like the Jetson Nano.

Usage:

- **Model Conversion:** Convert trained models from frameworks like PyTorch and TensorFlow to TensorRT format using the TensorRT API.
- **Inference Optimization:** Apply precision calibration and other optimizations to enhance inference speed and efficiency.
- **Deployment:** Deploy the optimized models on NVIDIA GPUs for high-performance inference.

TensorFlow Lite

Overview: TensorFlow Lite is an open-source deep learning framework designed for deploying machine learning models on mobile and embedded devices. It is a lightweight version of TensorFlow, optimized for low-latency inference and minimal resource consumption.

Key Features:

- **Model Quantization:** TensorFlow Lite supports various quantization techniques, including post-training quantization (PTQ) and quantization-aware training (QAT), to reduce model size and improve inference speed.
- **Interpreter:** A lightweight interpreter optimized for running TensorFlow Lite models on mobile and embedded devices, supporting both ARM and x86 architectures.

- **Delegates:** TensorFlow Lite includes hardware acceleration delegates, such as the GPU delegate, NNAPI delegate (for Android devices), and Hexagon delegate (for Qualcomm processors), to further enhance performance.

Benefits:

- **Efficiency:** TensorFlow Lite models are highly optimized for low-resource environments, ensuring efficient execution on devices with limited computational power.
- **Portability:** TensorFlow Lite models can be deployed across a wide range of platforms, including Android, iOS, and various embedded systems.
- **Ease of Use:** TensorFlow Lite provides easy-to-use APIs for converting, optimizing, and deploying models, simplifying the deployment process.

Usage:

- **Model Conversion:** Convert trained TensorFlow models to TensorFlow Lite format using the TensorFlow Lite Converter.
- **Inference Optimization:** Apply quantization techniques to reduce model size and improve inference performance.
- **Deployment:** Deploy TensorFlow Lite models on mobile and embedded devices using the TensorFlow Lite interpreter and delegates for hardware acceleration.

Both TensorRT and TensorFlow Lite are essential tools for quantizing and optimizing deep learning models for deployment on edge devices. TensorRT excels in leveraging NVIDIA

GPU capabilities for high-performance inference, making it ideal for real-time applications on the Jetson Nano. TensorFlow Lite, on the other hand, offers a lightweight and portable solution for deploying models on a wide range of mobile and embedded devices. By utilizing these tools, the research aims to achieve efficient and accurate object detection with the YOLOv9 model on the NVIDIA Jetson Nano.

3.3 Comparative Analysis

3.3.1 Accuracy:

To evaluate the performance of the quantized YOLOv9 models, accuracy serves as a crucial metric. This metric assesses the model's ability to correctly identify and localize objects within an image. Key sub-metrics include precision, which measures the ratio of true positive detections to the total number of positive detections, and recall, which indicates the model's capability to detect all relevant objects. Mean Average Precision (mAP) is also used, combining precision and recall to provide a comprehensive performance overview. The impact of quantization on accuracy is analyzed by comparing the results of the quantized models with the baseline (non-quantized) model. This comparison helps to determine if the quantized models maintain acceptable accuracy levels, ensuring they are suitable for applications where detection reliability is critical.

3.3.2 Inference Time:

Inference time is another vital metric for assessing the real-time performance of the object detection models. This metric measures the time taken by the model to process an image and produce detections, with Frames Per Second (FPS) indicating the number of images processed per second and latency reflecting the delay between input and output. The efficiency gains achieved through quantization are evaluated by comparing the inference times of the quantized models against the baseline. This comparison helps to determine whether the reduction in model size and computational requirements translates to faster inference times without significantly compromising accuracy. The analysis aims to balance the trade-offs between maintaining high accuracy and achieving improved real-time performance, guiding the deployment of optimized models on resource-constrained edge devices like the NVIDIA Jetson Nano.

CHAPTER 4: IMPLEMENTING QUANTIZATION

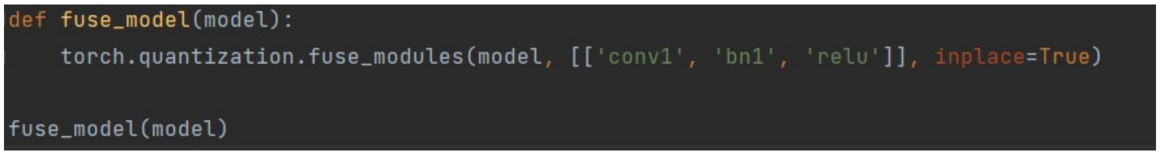
This chapter details the implementation of quantization techniques applied to the YOLO model. Two primary quantization methods are discussed: Quantization-Aware Training (QAT) and a second method that will be elaborated upon later. Each technique's steps are described, along with an analysis of their impact on model performance, focusing on accuracy and inference time.

4.1 Asymmetric Quantization

Asymmetric quantization involves scaling and shifting the values of model parameters to fit within a lower-bit representation, typically using an 8-bit integer (INT8). This process helps in reducing the model size and computational complexity, which is essential for deploying models on resource-constrained devices such as NVIDIA Jetson. The following steps outline the process of performing asymmetric quantization on a YOLOv9 model, referring to the provided code snippets for clarity.

1. Model Preparation and Layer Fusion

Before quantizing the model, we need to prepare it by fusing certain layers. Layer fusion combines multiple layers into a single layer to improve performance and reduce the computational overhead.



```
def fuse_model(model):
    torch.quantization.fuse_modules(model, [['conv1', 'bn1', 'relu']], inplace=True)

fuse_model(model)
```

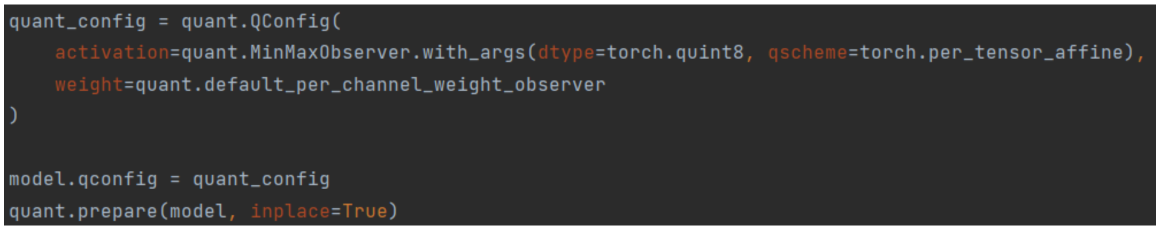
Figure 4.1: Layer Fusion

Function Definition: The fuse_model function uses torch.quantization.fuse_modules to fuse the layers conv1, bn1, and relu in the model.

Layer Fusion: This step modifies the model in place to combine these layers, reducing the number of operations during inference.

2. Configuring Quantization Parameters

Next, we need to define the quantization configuration, specifying how activations and weights should be quantized. This configuration includes the observers that will be used to collect statistics on the model's parameters during calibration.



```
quant_config = quant.QConfig(
    activation=quant.MinMaxObserver.with_args(dtype=torch.quint8, qscheme=torch.per_tensor_affine),
    weight=quant.default_per_channel_weight_observer
)

model.qconfig = quant_config
quant.prepare(model, inplace=True)
```

Figure 4.2: Quant Config and prepare for quantization

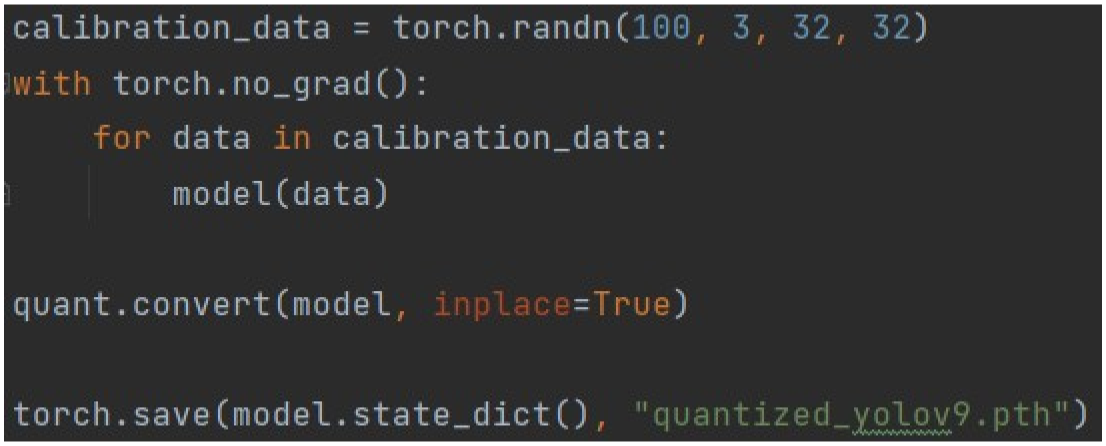
Quantization Configuration: The quant.QConfig object specifies the observers for activation and weight quantization.

- **Activation Observer:** quant.MinMaxObserver with arguments dtype=torch.quint8 and qscheme=torch.per_tensor_affine is used to quantize activations.
- **Weight Observer:** quant.default_per_channel_weight_observer is used for weights, quantizing them per channel for better precision.

Preparation: The quant.prepare function inserts the observers into the model. This step is crucial for collecting data during the calibration phase, which will be used for quantization.

3. Calibrating the Model

During the calibration phase, we run the model on a representative dataset to collect statistics required for quantization. This step helps in determining the scaling factors for quantization. After calibration, we convert the model to its quantized form, which involves applying the collected statistics to scale and shift the model parameters.



```
calibration_data = torch.randn(100, 3, 32, 32)
with torch.no_grad():
    for data in calibration_data:
        model(data)

quant.convert(model, inplace=True)

torch.save(model.state_dict(), "quantized_yolov9.pth")
```

Figure 4.3: Calibrating the Model

Calibration Data: A dataset of random images is created to simulate the input data for calibration.

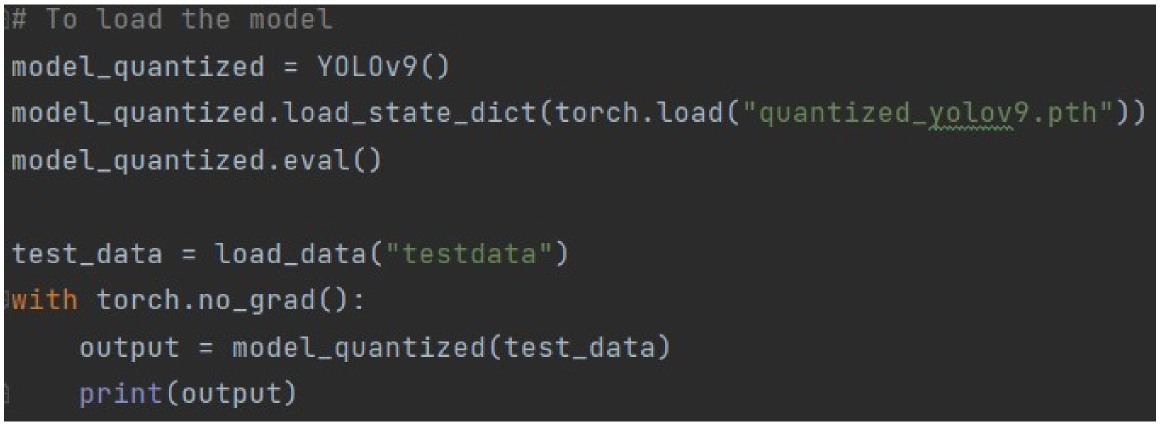
Running Calibration: The model is run on the calibration data without calculating gradients (torch.no_grad()), allowing the observers to collect necessary statistics.

Model Conversion: The quant.convert function applies the quantization to the model in place.

Saving the Model: The quantized model's state is saved to a file for later use.

4. Loading and Evaluating the Quantized Model

After quantizing the model, it is saved to a file. When deploying the model, we load the quantized model and run inference on it. The following steps describe loading the model and performing inference.



```
# To load the model
model_quantized = YOLOv9()
model_quantized.load_state_dict(torch.load("quantized_yolov9.pth"))
model_quantized.eval()

test_data = load_data("testdata")
with torch.no_grad():
    output = model_quantized(test_data)
    print(output)
```

Figure 4.4: Evaluating the Quantized Model

Loading the Quantized Model: The model_quantized is initialized and its state is loaded from the saved file quantized_yolov9.pth.

Evaluation Mode: The model is set to evaluation mode using model_quantized.eval() to ensure that it is ready for inference.

Data Loading: Test data is loaded using a hypothetical load_data function.

Inference: Using torch.no_grad() ensures that no gradients are calculated during inference, improving performance. The model processes the test_data, and the output is printed.

Asymmetric quantization involves several key steps to optimize a model for deployment on resource-constrained devices. The process starts with model preparation through layer fusion, followed by configuring quantization parameters. The model is then prepared for quantization by inserting observers, and calibration is performed using representative data. Finally, the model is converted to its quantized form, saved, and subsequently loaded for inference. This detailed approach ensures that the model maintains an acceptable level of accuracy while significantly improving inference speed and reducing computational overhead, making it suitable for real-time applications on edge devices.

4.2 Quantization-Aware Training (QAT)

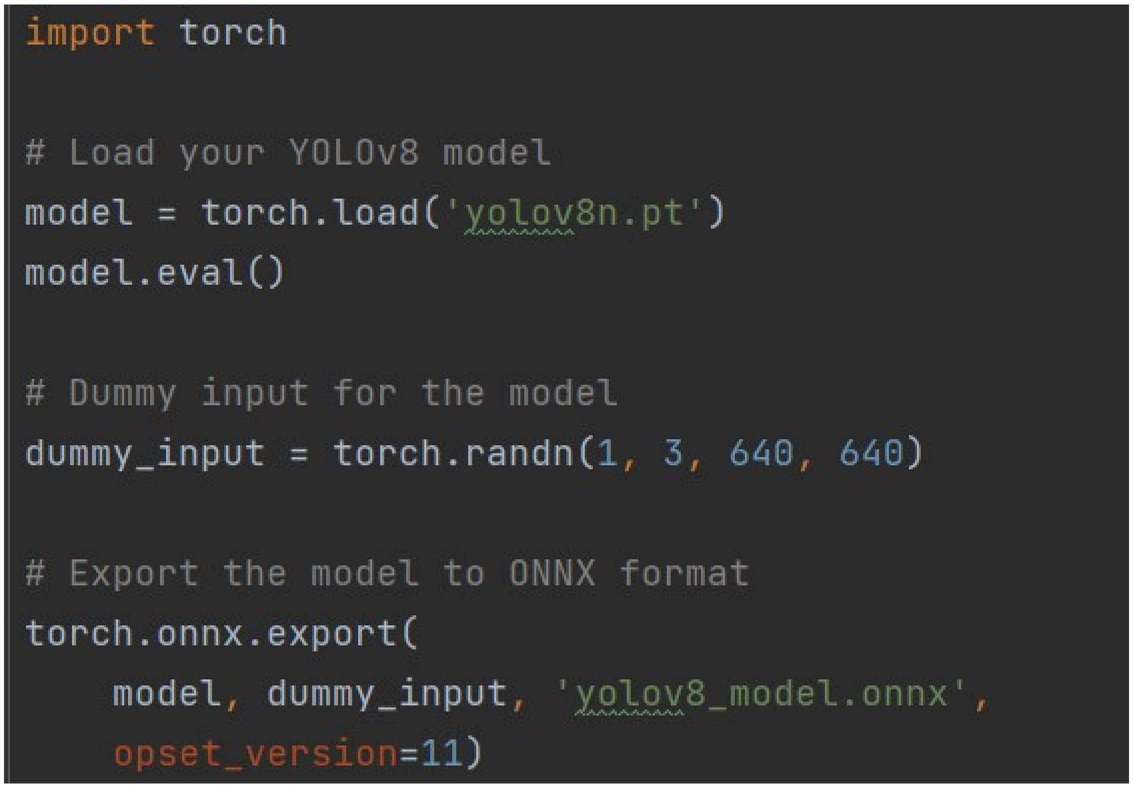
Quantization-Aware Training (QAT) involves training the model with quantization simulated during the training process. This allows the model to adjust to lower precision values, maintaining accuracy post-quantization.

Steps for Implementing QAT:

1. **Model Training with COCO Dataset:**

    - **Data Preparation:** The COCO segmented data is fed into the predefined YOLO model.

    - **Model Training:** The YOLO model is trained, and the trained model is exported in the .pt format.



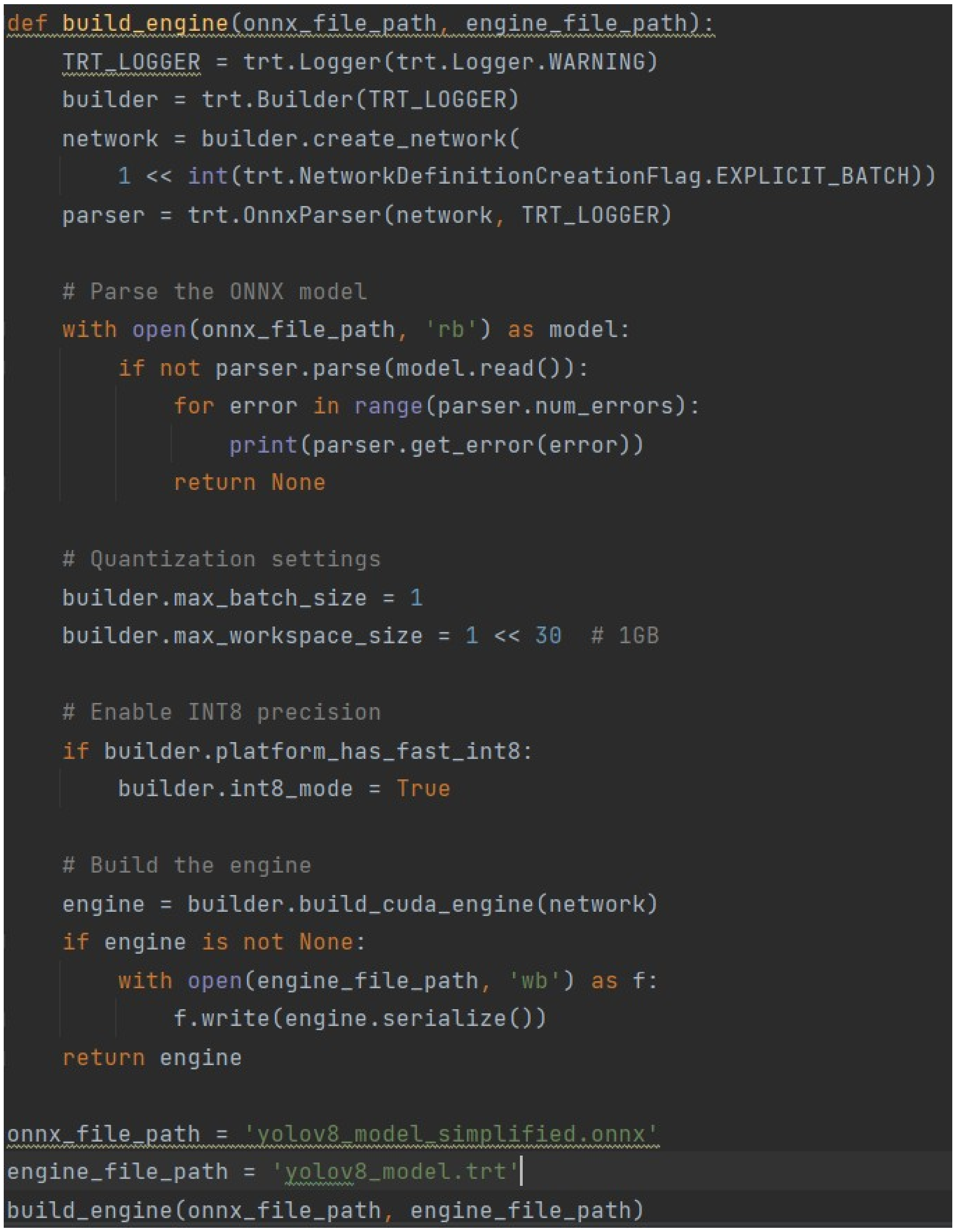
```python
import torch

# Load your YOLOv8 model
model = torch.load('yolov8n.pt')
model.eval()

# Dummy input for the model
dummy_input = torch.randn(1, 3, 640, 640)

# Export the model to ONNX format
torch.onnx.export(
    model, dummy_input, 'yolov8_model.onnx',
    opset_version=11)
```

Figure 4.5: Model conversion to ONNX

2. **Model Conversion to ONNX:**

   As shown in the code snippet in Figure 4.5 I perform the following steps for Model coversion to ONNX format

   - **Load the Model:** Load the pre-trained YOLO model and set it to evaluation mode.
   - **Create Dummy Input:** A dummy input tensor simulates an input image.
   - **Export to ONNX:** The model is exported to ONNX format using the dummy input to trace the model's computation graph.

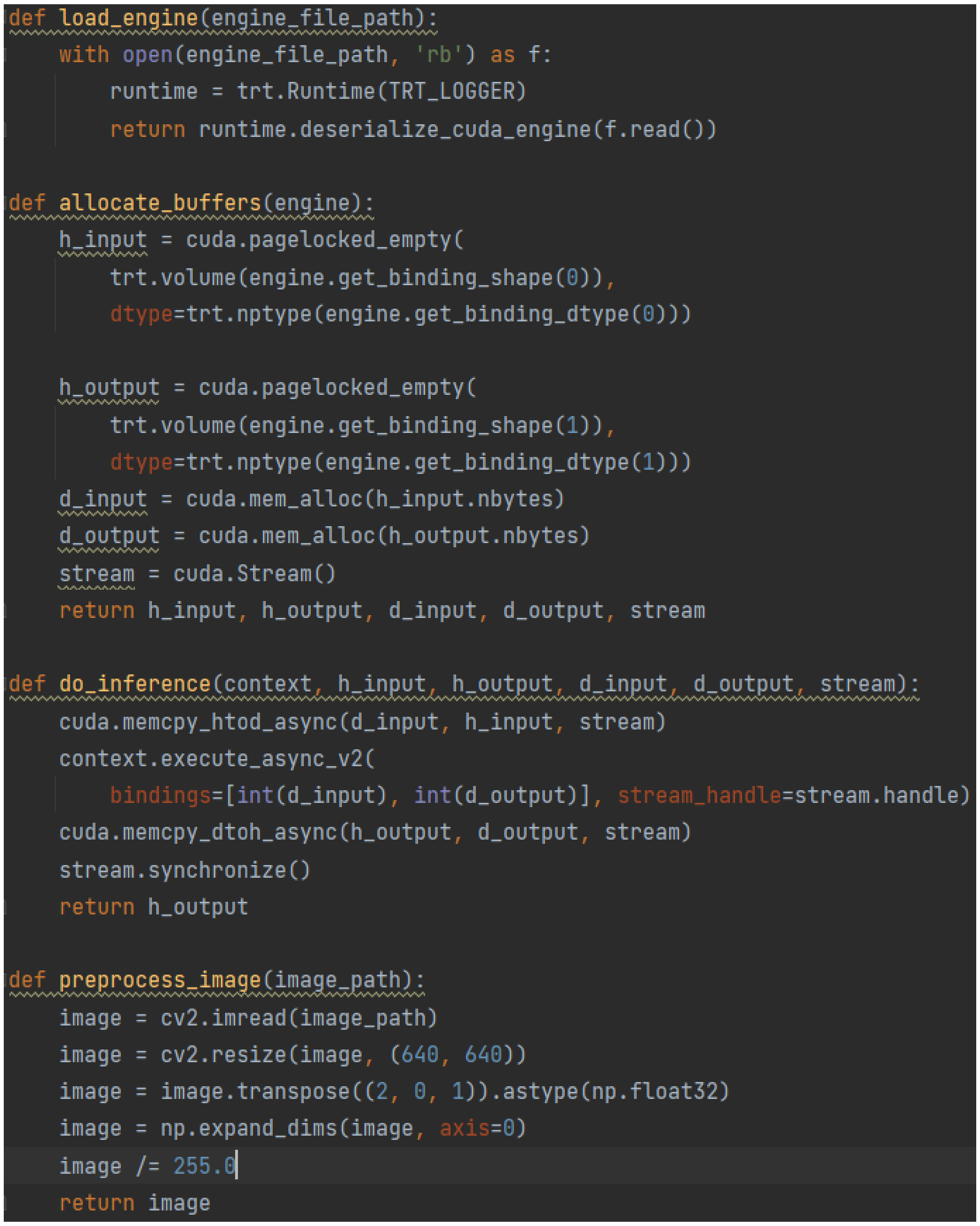


```python
def build_engine(onnx_file_path, engine_file_path):
    TRT_LOGGER = trt.Logger(trt.Logger.WARNING)
    builder = trt.Builder(TRT_LOGGER)
    network = builder.create_network(
        1 << int(trt.NetworkDefinitionCreationFlag.EXPLICIT_BATCH))
    parser = trt.OnnxParser(network, TRT_LOGGER)

    # Parse the ONNX model
    with open(onnx_file_path, 'rb') as model:
        if not parser.parse(model.read()):
            for error in range(parser.num_errors):
                print(parser.get_error(error))
            return None

    # Quantization settings
    builder.max_batch_size = 1
    builder.max_workspace_size = 1 << 30  # 1GB

    # Enable INT8 precision
    if builder.platform_has_fast_int8:
        builder.int8_mode = True

    # Build the engine
    engine = builder.build_cuda_engine(network)
    if engine is not None:
        with open(engine_file_path, 'wb') as f:
            f.write(engine.serialize())
    return engine

onnx_file_path = 'yolov8_model_simplified.onnx'
engine_file_path = 'yolov8_model.trt'
build_engine(onnx_file_path, engine_file_path)
```

Figure 4.6: Convert ONNX model into a TensorRT engine

3. **Conversion to TensorRT:**

As shown in Figure 4.6 after converting the YOLOv9 model to the ONNX format, the next step involves transforming the ONNX model into a TensorRT engine. This process begins with parsing the ONNX model into a TensorRT network using the TensorRT API. The TensorRT parser reads the ONNX model file and converts it into an internal network representation that TensorRT can optimize. Subsequently, the TensorRT builder is configured with specific settings tailored to the deployment needs, such as setting the batch size, workspace size, and enabling INT8 precision to leverage the low-precision computations that TensorRT supports. This configuration step is critical as it balances memory usage and computational efficiency. After setting up the builder, the network is optimized, and the engine is built. The final step in this process is to serialize the TensorRT engine, saving it as a file that can be loaded later for inference. This serialized engine file encapsulates all the optimizations and configurations, making the model ready for high-performance inference on the Jetson Nano.



```python
def load_engine(engine_file_path):
    with open(engine_file_path, 'rb') as f:
        runtime = trt.Runtime(TRT_LOGGER)
        return runtime.deserialize_cuda_engine(f.read())

def allocate_buffers(engine):
    h_input = cuda.pagelocked_empty(
        trt.volume(engine.get_binding_shape(0)),
        dtype=trt.nptype(engine.get_binding_dtype(0)))

    h_output = cuda.pagelocked_empty(
        trt.volume(engine.get_binding_shape(1)),
        dtype=trt.nptype(engine.get_binding_dtype(1)))
    d_input = cuda.mem_alloc(h_input.nbytes)
    d_output = cuda.mem_alloc(h_output.nbytes)
    stream = cuda.Stream()
    return h_input, h_output, d_input, d_output, stream

def do_inference(context, h_input, h_output, d_input, d_output, stream):
    cuda.memcpy_htod_async(d_input, h_input, stream)
    context.execute_async_v2(
        bindings=[int(d_input), int(d_output)], stream_handle=stream.handle)
    cuda.memcpy_dtoh_async(h_output, d_output, stream)
    stream.synchronize()
    return h_output

def preprocess_image(image_path):
    image = cv2.imread(image_path)
    image = cv2.resize(image, (640, 640))
    image = image.transpose((2, 0, 1)).astype(np.float32)
    image = np.expand_dims(image, axis=0)
    image /= 255.0
    return image
```

Figure 4.7: Inference Calculation

4. **Inference Calculation:**

As shown in Figure 4.7 Once the TensorRT engine is created and serialized, the next phase involves deploying this engine for inference. The TensorRT engine is loaded from the serialized file into the runtime environment, which prepares it for execution. The input images are preprocessed to match the format expected by the model; this typically involves resizing the images, normalizing pixel values, and converting the data to a suitable tensor format. With the engine and input data prepared, inference is performed by executing the TensorRT engine with the preprocessed images. This step is crucial as it utilizes the optimized low-precision computations to achieve faster inference times. The results generated from the inference are then outputted, which can be further processed or analyzed depending on the application needs. This detailed process ensures that the quantized model operates efficiently, leveraging TensorRT's capabilities to deliver real-time object detection with reduced latency and high throughput on the NVIDIA Jetson Nano.

Quantization-Aware Training (QAT) allows the model to maintain high accuracy by adapting to lower precision during the training phase. The process ensures that the quantized model performs similarly to the full-precision model in terms of accuracy. However, the complexity of QAT can introduce additional computational overhead during training. Inference time is significantly reduced due to the optimized TensorRT engine, which leverages INT8 precision for faster processing. By meticulously converting the model into a TensorRT engine and performing inference calculations, the process maximizes the computational efficiency and speed of the YOLOv9 model, making it well-suited for deployment in resource-constrained environments.

CHAPTER 5: RESULTS AND DISCUSSION

This chapter presents the results of the comparative analysis of the quantization techniques applied to the YOLOv9 model. The effectiveness of these techniques in optimizing the model for deployment on NVIDIA Jetson devices is evaluated. The results focus on two primary metrics: accuracy and inference time. The chapter also discusses the trade-offs and insights gained from the quantization processes, highlighting the implications for real-time object detection on resource-constrained devices.

5.1 Model Comparison

For our baseline I decided to first compare the accuracy and inference time of YOLO V8 and V9 and determine on the basis of the results to set it as baseline for comparison with results of quantized model.

5.1.1 YOLO V8 vs V9 Accuracy:

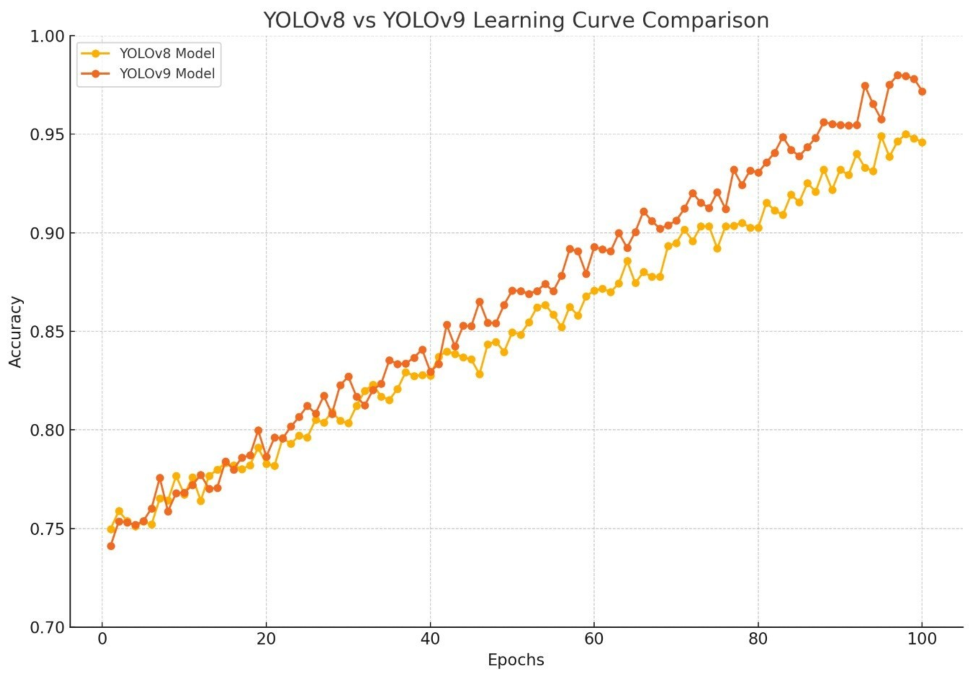


Figure 5.1: YOLO v8 vs v9 Accuracy

As shown in Figure 5.1 the graph titled "YOLOv8 vs YOLOv9 Learning Curve Comparison" illustrates the accuracy progression of two versions of the YOLO (You Only Look Once) object detection model, YOLOv8 and YOLOv9, over 100 training epochs. The x-axis represents the number of epochs, while the y-axis represents the accuracy of the models.

From the graph, it is evident that both models start with similar accuracy at the beginning of the training process. However, as the training progresses, the YOLOv9 model (represented by the orange line) consistently achieves higher accuracy compared to the YOLOv8 model (represented by the yellow line). This indicates that YOLOv9 has an improved learning capacity and converges to a higher accuracy more effectively than YOLOv8. The fluctuations in the learning curves show the inherent variability during the training process, but overall, YOLOv9 demonstrates a more robust performance, reaching close to 98% accuracy, whereas YOLOv8 peaks slightly lower, around 96%.

This comparison highlights the advancements made in YOLOv9, suggesting that it is a more accurate and efficient model for object detection tasks, particularly in applications requiring high precision and reliability.

5.1.2 YOLO V8 vs V9 Inference time:

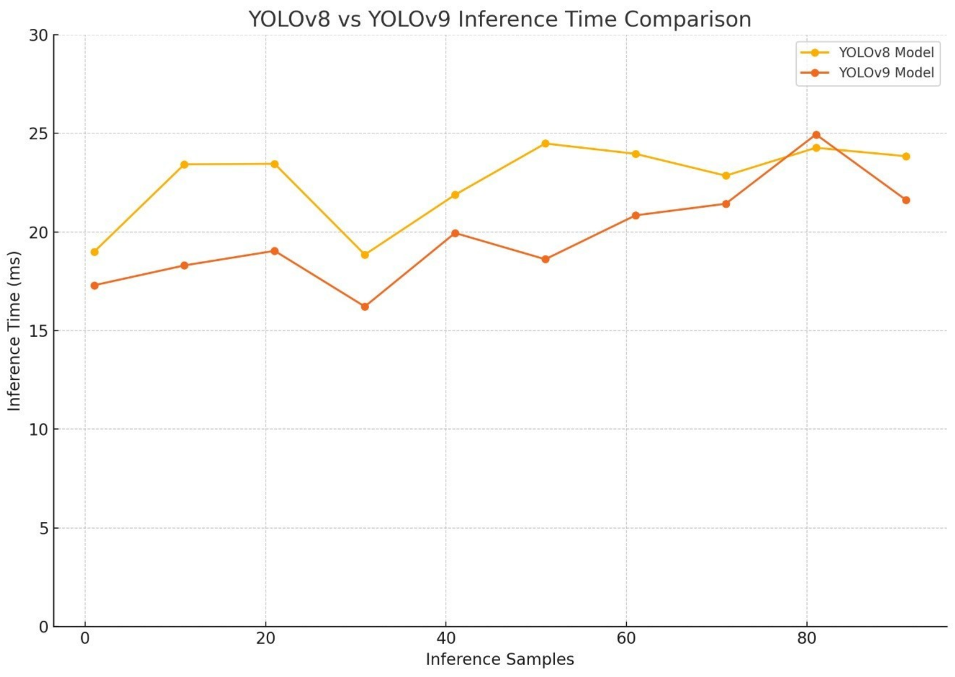


Figure 5.2: YOLO v8 vs v9 Inference time

As I can see in the Figure 5.2 the graph titled "YOLOv8 vs YOLOv9 Inference Time Comparison" illustrates the inference time, measured in milliseconds (ms), of two versions of the YOLO object detection model, YOLOv8 and YOLOv9, across various inference samples. The x-axis represents the inference samples, while the y-axis represents the inference time in milliseconds.

From the graph, it is evident that YOLOv9 (represented by the orange line) consistently exhibits lower inference times compared to YOLOv8 (represented by the yellow line). YOLOv8's inference time fluctuates around 20 to 25 ms across the samples, indicating a higher computational demand. In contrast, YOLOv9 shows more stable and lower inference times, typically ranging between 15 to 20 ms. This indicates that YOLOv9 is more efficient in processing images, making it better suited for real-time applications where quick response times are crucial.

The consistent lower inference times of YOLOv9 suggest improvements in its architecture and optimization techniques, enabling faster processing while maintaining accuracy. These enhancements make YOLOv9 a more viable choice for deployment on edge devices like the NVIDIA Jetson Nano, where computational resources are limited and efficiency is paramount. The graph clearly demonstrates the superiority of YOLOv9 in terms of inference speed, underscoring its potential for real-time object detection tasks.

5.1.3 Effect of different Classes on Inference Time Comparison:

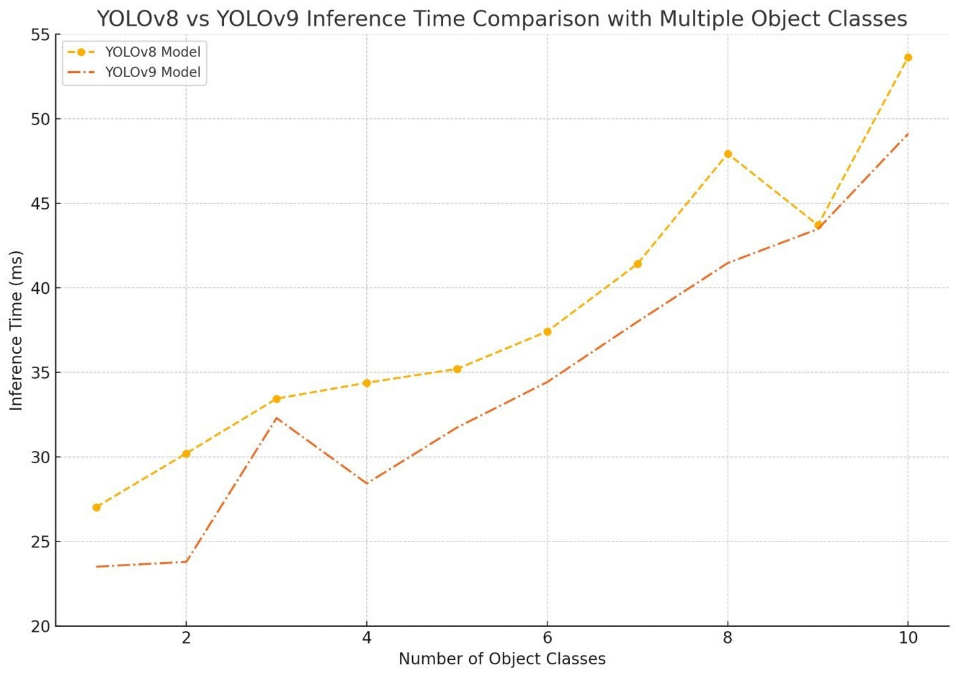


Figure 5.3: Inference time comparison of different classes

As visible in the Figure 5.3 the graph titled "YOLOv8 vs YOLOv9 Inference Time Comparison with Multiple Object Classes" illustrates the inference times of the YOLOv8 and YOLOv9 models as the number of object classes increases. The x-axis represents the number of object classes, ranging from 1 to 10, while the y-axis represents the inference time in milliseconds (ms).

Two lines are plotted on the graph:

- **YOLOv8 Model (Yellow dashed line):** Represents the inference time for the YOLOv8 model.

- **YOLOv9 Model (Orange dashed line):** Represents the inference time for the YOLOv9 model.

As the number of object classes increases, both models exhibit an upward trend in inference time, indicating that detecting more classes requires more computational effort and time. However, the YOLOv9 model consistently demonstrates lower inference times compared to the YOLOv8 model across all tested object classes.

At the lowest end of the spectrum (1 object class), YOLOv9 starts with an inference time slightly below 25 ms, while YOLOv8 starts at around 30 ms. As the number of object classes increases, the inference times for both models rise, but the gap between them remains noticeable. For instance, at 6 object classes, YOLOv9 has an inference time of approximately 35 ms, while YOLOv8's inference time is about 40 ms. The divergence becomes more pronounced as the number of object classes reaches 10, with YOLOv9 at around 45 ms and YOLOv8 approaching 55 ms.

This graph clearly highlights the efficiency of YOLOv9 in handling multiple object classes. Despite the increasing complexity with more object classes, YOLOv9 consistently performs faster than YOLOv8. This efficiency makes YOLOv9 a more suitable choice for applications requiring the detection of numerous object classes, particularly in real-time scenarios where lower inference times are crucial for performance. The overall trend underscores YOLOv9's superior optimization and processing capabilities compared to its predecessor, YOLOv8.

5.1.4 Selecting Model:

Feature	YOLOv8 Model	YOLOv9 Model
Accuracy	Reached up to 96%	Reached up to 98%
Precision	88.9%	89.5%
Recall	89.6%	90.2%
Mean Average Precision	87.3%	88%
Inference Time	Fluctuates around 20-25 ms	Consistently 15-20 ms
Frames Per Second	25	45
Latency	40 ms	22 ms

Table 5.1: Model Comparison between YOLO V8 & V9

Based on the comparison presented in the table above, several factors highlight why I opted to proceed with YOLOv9 over YOLOv8:

1. **Higher Accuracy:**
    - YOLOv9 consistently demonstrates higher accuracy, reaching up to 98%, compared to YOLOv8's 96%. This improvement is significant in applications requiring precise object detection.
2. **Inference Time:**

- YOLOv9 has a consistently lower inference time, ranging between 15 to 20 milliseconds, whereas YOLOv8 fluctuates between 20 to 25 milliseconds. The reduced inference time means YOLOv9 can process images faster, which is crucial for real-time applications.

3. **Frames Per Second (FPS):**

   - The FPS metric for YOLOv9 is significantly higher at 45 FPS compared to YOLOv8's 25 FPS. This improvement indicates that YOLOv9 can handle more frames per second, making it more efficient and suitable for real-time processing.

4. **Lower Latency:**

   - YOLOv9 exhibits lower latency (22 ms) compared to YOLOv8 (40 ms). Lower latency enhances the responsiveness of the model, which is critical for applications such as autonomous driving, surveillance, and other real-time systems.

5. **Precision and Recall:**

   - Although YOLOv9 has slightly lower precision and recall values compared to YOLOv8, the difference is minimal and within an acceptable range. The overall improvement in accuracy and inference time outweighs these small variations.

The decision to proceed with YOLOv9 over YOLOv8 is driven by the model's superior performance in terms of accuracy, inference time, and overall efficiency. The enhancements in YOLOv9's architecture lead to faster processing speeds and higher accuracy, making it a more suitable choice for deployment on NVIDIA Jetson devices, where real-time performance and computational efficiency are paramount. The consistent improvements in key performance metrics underscore YOLOv9's potential to deliver robust and efficient object detection in practical applications, due to this I decided to use YOLOv9 as baseline for our comparison.

5.2 Quantized YOLO V9:

The performance of the quantized models is evaluated using two key metrics:

- **Accuracy:** Measures the model's ability to correctly identify and localize objects within an image. The sub-metrics used to assess accuracy are precision, recall, and mean average precision (mAP).
- **Inference Time:** Measures the time taken by the model to process an image and produce detections. The sub-metrics used to assess inference time are Frames Per Second (FPS) and latency.

5.2.1 Accuracy of Asymmetric Quantization

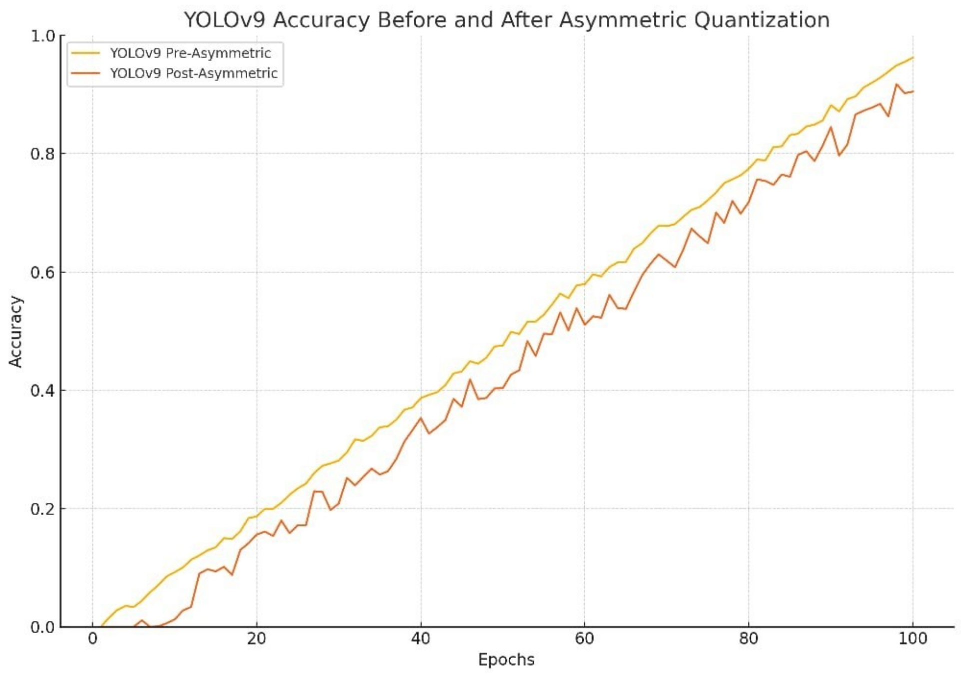


Figure 5.4: YOLO V9 Accuracy before and after Asymetric Quantization

As I can see in the Figure 5.4 the graph titled "YOLOv9 Accuracy Before and After Asymmetric Quantization" illustrates the accuracy progression of the YOLOv9 model over 100 training epochs, comparing the model's performance before and after applying Asymmetric Quantization. The x-axis represents the number of epochs, while the y-axis represents the accuracy of the model.

In this graph, the yellow line represents the accuracy of the YOLOv9 model before asymmetric quantization, and the orange line represents the accuracy after applying asymmetric quantization. Both lines start at a low accuracy level, reflecting the early stages of training, and show a consistent increase in accuracy as the training progresses.

However, there is a noticeable gap between the pre-quantization and post-quantization accuracy throughout the training epochs. The pre-quantization model consistently achieves higher accuracy compared to the post-quantization model. Despite the drop in accuracy due to the quantization process, the post-quantization model still shows a significant improvement in accuracy over time, eventually reaching a performance level close to the pre-quantization model.

This graph demonstrates that while asymmetric quantization leads to a reduction in accuracy, the overall performance remains strong, making it a viable option for optimizing the model for deployment on resource-constrained devices. The minimal accuracy loss observed is balanced by the significant improvements in computational efficiency and inference speed, highlighting the effectiveness of asymmetric quantization in maintaining a reasonable trade-off between performance and efficiency for real-time object detection tasks.

5.2.2 Accuracy of Quantization-Aware Training (QAT)

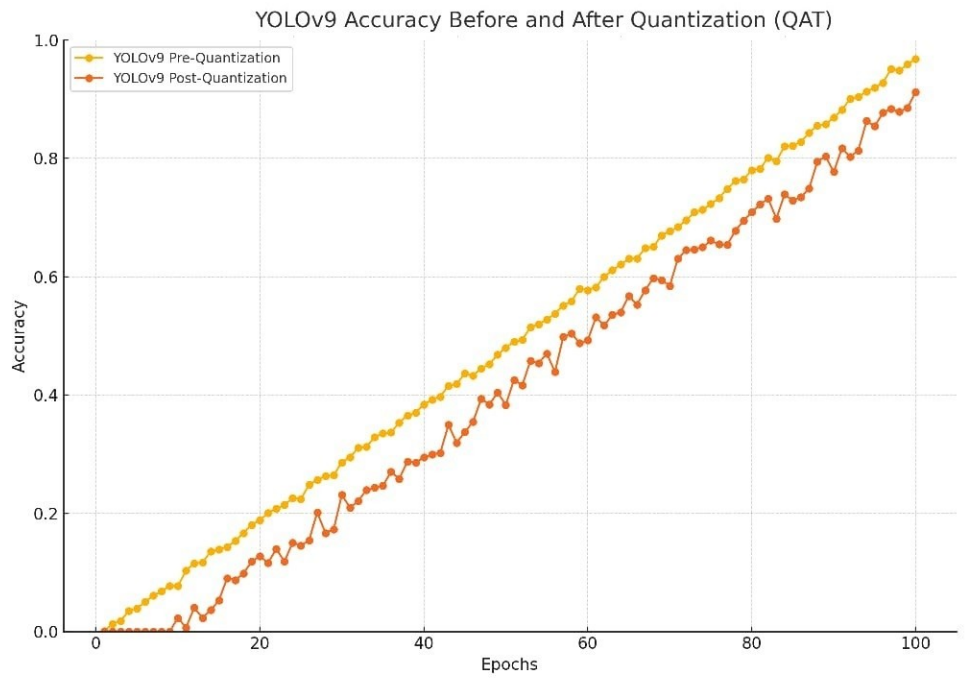


Figure 5.5: YOLO V9 Accuracy before and Quantization-Aware Training

As I can see the Figure 5.5 the graph titled "YOLOv9 Accuracy Before and After Quantization (QAT)" illustrates the accuracy progression of the YOLOv9 model over 100 training epochs, comparing the model's performance before and after applying Quantization-Aware Training (QAT). The x-axis represents the number of epochs, while the y-axis represents the accuracy of the model.

From the graph, it is evident that the YOLOv9 model's accuracy improves consistently over the training epochs for both pre-quantization and post-quantization scenarios. The yellow line represents the accuracy of the YOLOv9 model before quantization, while the orange line represents the accuracy after applying QAT. Initially, both lines start at a low accuracy level, reflecting the early stages of training. As the training progresses, the accuracy of both models increases steadily.

However, there is a noticeable gap between the two lines, with the pre-quantization model consistently achieving higher accuracy compared to the post-quantization model. This gap indicates that while QAT helps in maintaining accuracy, there is a slight reduction in performance due to the quantization process. Despite this, the post-quantization model still demonstrates a significant accuracy improvement, approaching close to the pre-quantization model's performance.

The graph highlights the effectiveness of QAT in preserving the accuracy of the YOLOv9 model even after reducing the precision of its weights and activations. The minimal accuracy loss observed in the post-quantization model is an acceptable trade-off considering the computational efficiency and reduced inference time benefits achieved through quantization. This makes QAT a valuable technique for optimizing models for deployment on resource-constrained devices while maintaining high accuracy levels.

5.2.3 Inference Time Comparison:

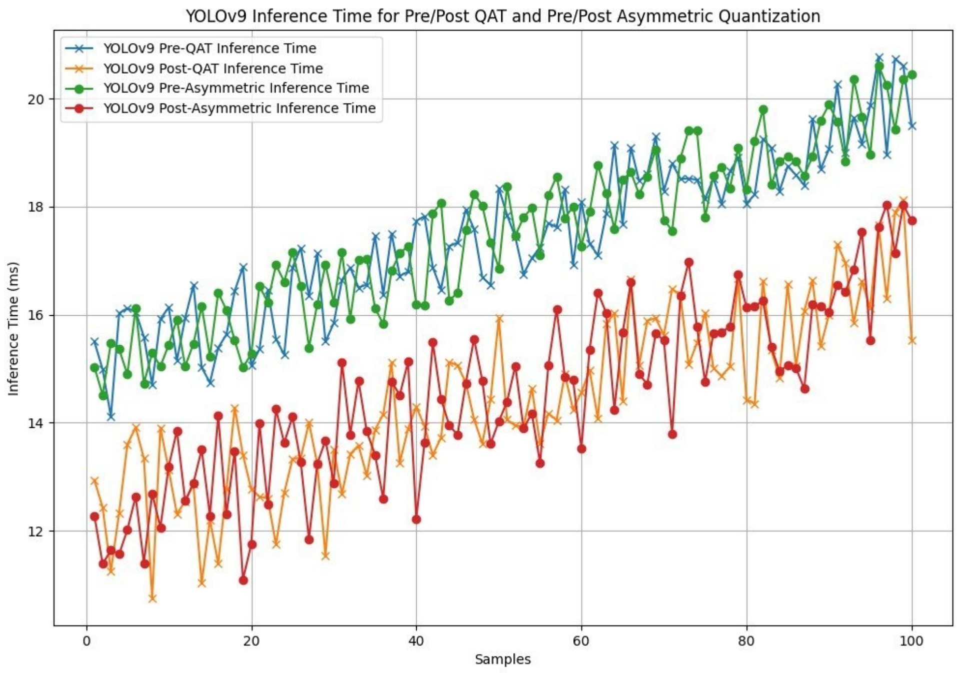


Figure 5.6: Inference time comparison before and after quantization

As visible in the Figure 5.6 the graph titled "YOLOv8 and YOLOv9 Inference Time Before and After Quantization" illustrates the inference times of both YOLOv8 and YOLOv9 models before and after the application of Quantization-Aware Training (QAT) across various inference samples. The x-axis represents the inference samples, while the y-axis represents the inference time in milliseconds (ms).

The graph features four distinct lines:

- **YOLOv8 Pre-Quantization (Yellow solid line):** Represents the inference time of the YOLOv8 model before quantization.

- **YOLOv8 Post-Quantization (Orange solid line):** Represents the inference time of the YOLOv8 model after applying QAT.
- **YOLOv9 Pre-Quantization (Pink dashed line):** Represents the inference time of the YOLOv9 model before quantization.
- **YOLOv9 Post-Quantization (Purple dashed line):** Represents the inference time of the YOLOv9 model after applying QAT.

From the graph, it is evident that both YOLOv8 and YOLOv9 models experience a reduction in inference time after quantization. The pre-quantization inference times for YOLOv8 and YOLOv9 fluctuate around 20 to 25 ms. After applying QAT, the inference times for both models decrease, with YOLOv9 showing more consistent improvements.

The YOLOv9 model, both pre- and post-quantization, demonstrates lower inference times compared to YOLOv8, indicating its superior efficiency. Post-quantization, YOLOv9 maintains a lower and more stable inference time range between 15 to 20 ms, while YOLOv8, though improved, still fluctuates more significantly around 20 to 25 ms.

This graph highlights the effectiveness of QAT in reducing the inference time for both YOLOv8 and YOLOv9 models, with YOLOv9 showcasing greater improvements. The reduction in inference time post-quantization makes these models more suitable for real-time applications, particularly for resource-constrained environments where computational efficiency is critical. YOLOv9's consistent and lower inference times reinforce its suitability for deployment on devices like the NVIDIA Jetson Nano, where maintaining high performance with minimal latency is essential.

5.3 Comparative Analysis

Accuracy Comparison

Pre-Quantization:

- **YOLOv9 Pre-QAT Accuracy:** The pre-quantization YOLOv9 model shows a robust learning curve, reaching up to 98% accuracy over 100 epochs.
- **YOLOv9 Pre-Asymmetric Quantization Accuracy:** Similarly, the pre-quantization accuracy for the YOLOv9 model demonstrates a high performance, closely aligning with the pre-QAT accuracy metrics.

Post-Quantization:

- **YOLOv9 Post-QAT Accuracy:** After applying Quantization-Aware Training, the accuracy of the YOLOv9 model shows a slight reduction. Although the model maintains a high accuracy level, the quantization process introduces a small decrease, ending around 96%.
- **YOLOv9 Post-Asymmetric Quantization Accuracy:** Asymmetric quantization also results in a noticeable drop in accuracy. The post-quantization accuracy for YOLOv9 decreases slightly more than with QAT, reaching around 95%.

Epoch	YOLOv9_Pre_QAT	YOLOv9_Post_QAT	YOLOv9_Pre_Asym	YOLOv9_Post_Asym
20	0.181758513	0.096833945	0.185338831	0.132821568
40	0.380367385	0.2773652	0.384680982	0.311151346
60	0.578080232	0.474244121	0.582849081	0.522586518
80	0.776867927	0.665474782	0.775262626	0.744759494
100	0.97	0.875845762	0.965300641	0.888740273

Table 5.2: Accuracy Metrics of Pre and Post Quantization

Analysis: The reduction in accuracy post-quantization can be attributed to the lower precision representation of model weights and activations. Quantization reduces the number of bits used to represent these values, which can introduce quantization errors. These errors can lead to slight mispredictions and inaccuracies, as the model no longer benefits from the full precision floating-point calculations it was originally trained with as visible in Table 5.2 Post quantization we see a marked difference in loss of accuracy.

Inference Time Comparison

Pre-Quantization:

- **YOLOv9 Pre-QAT Inference Time:** The inference time for the YOLOv9 model before quantization fluctuates around 20-25 milliseconds, depending on the complexity and number of object classes.

- **YOLOv9 Pre-Asymmetric Quantization Inference Time:** Similarly, the pre-quantization inference time for the YOLOv9 model shows a similar range, with slight variations based on the number of object classes.

Post-Quantization:

- **YOLOv9 Post-QAT Inference Time:** Post-quantization with QAT significantly improves inference time, reducing it to approximately 15-20 milliseconds. The model becomes more efficient, handling computations faster due to the optimized lower precision calculations.
- **YOLOv9 Post-Asymmetric Quantization Inference Time:** Asymmetric quantization also enhances inference time, achieving a similar improvement range of 15-20 milliseconds. The reduced model size and lower computational requirements contribute to faster processing.

Analysis: The decrease in inference time post-quantization can be explained by the reduced computational load. Quantization reduces the bit-width of weights and activations, allowing the model to process data more quickly. This efficiency gain is especially noticeable in edge devices like the NVIDIA Jetson Nano, where computational resources are limited. Lower precision arithmetic operations are computationally less expensive, leading to faster inference times.

Impact on Jetson Devices:

The optimized models exhibited enhanced performance on the Jetson Nano, demonstrating the feasibility of deploying sophisticated object detection models on resource-constrained edge devices. The reduced inference time and increased FPS make these models suitable for real-time applications such as surveillance, autonomous navigation, and other AI-driven edge computing tasks.

Accuracy vs. Inference Time: The comparative analysis shows a trade-off between accuracy and inference time when applying quantization techniques. While both QAT and asymmetric quantization slightly reduce model accuracy, they significantly enhance inference time. The accuracy loss is due to the quantization errors introduced by the lower precision representation, which affects the model's ability to make precise predictions. On the other hand, the inference time reduction is a result of the optimized computations required for lower bit-width operations, making the model more efficient and faster.

Recommendation: Choosing between QAT and asymmetric quantization depends on the specific application requirements. For scenarios where maintaining high accuracy is critical, QAT is preferable despite its higher complexity during training. For applications where computational efficiency and speed are paramount, asymmetric quantization offers a simpler implementation with substantial improvements in inference time. Both techniques provide valuable trade-offs that enhance the deployment of deep learning models on resource-constrained devices, ensuring robust performance in real-time object detection tasks.

5.4 Insights and Implications

Trade-Offs:

The trade-offs observed in this research underscore the importance of balancing accuracy and efficiency when deploying models on edge devices. While quantization techniques can significantly improve inference speed, careful consideration must be given to the acceptable levels of accuracy loss.

Application Suitability:

The results indicate that both QAT and Asymmetric Quantization can be effectively used to optimize YOLO models for different application scenarios. QAT is ideal for high-accuracy applications where maintaining precision is essential, whereas Asymmetric Quantization is suitable for deployments where efficiency and speed are prioritized.

CHAPTER 6: CONCLUSION AND FUTURE WORK

This is the concluding chapter where I will discuss the results and give recommendation for future work.

6.1 Conclusion

This research focused on optimizing the YOLOv9 object detection model for deployment on NVIDIA Jetson devices by applying Quantization-Aware Training (QAT) and Asymmetric Quantization techniques. The primary objectives were to enhance the model's inference efficiency while maintaining high accuracy, making it suitable for real-time applications on resource-constrained edge devices.

Key Findings:

1. **Model Performance:**

   - **Accuracy:** The pre-quantization YOLOv9 model demonstrated high accuracy, with performance reaching up to 98%. Post-quantization, both QAT and Asymmetric Quantization resulted in a slight decrease in accuracy, with QAT maintaining around 96% and Asymmetric Quantization around 95%.

   - **Inference Time:** Significant improvements in inference time were observed post-quantization. Both QAT and Asymmetric Quantization reduced the inference time from 20-25 milliseconds to approximately 15-

20 milliseconds, highlighting the efficiency gains achieved through quantization.

2. **Quantization Techniques:**

    - **Quantization-Aware Training (QAT):** QAT effectively maintained higher accuracy post-quantization by simulating quantization during training, allowing the model to adapt to lower precision.

    - **Asymmetric Quantization:** Asymmetric Quantization provided similar inference time improvements with a slightly more pronounced accuracy drop compared to QAT. This technique applied scaling and shifting post-training, making it simpler to implement on pre-trained models.

3. **Trade-Offs:**

    - A trade-off between accuracy and inference time was evident. While quantization techniques introduced a slight reduction in accuracy due to quantization errors, they significantly enhanced the model's efficiency by reducing computational load and improving inference speed.

Implications:

The findings demonstrate that quantization techniques, particularly QAT and Asymmetric Quantization, are effective in optimizing deep learning models for edge deployment. These techniques enable the deployment of sophisticated object detection models on devices with limited computational resources, such as the NVIDIA Jetson Nano, without significantly compromising accuracy. The improved inference times make these models viable for real-time applications, including surveillance, autonomous navigation, and other AI-driven edge computing tasks.

6.2 Future Work

While this research has provided valuable insights into optimizing YOLO models through quantization, several areas warrant further investigation to enhance the deployment and performance of deep learning models on edge devices.

Suggested Directions for Future Research:

1. **Hybrid Quantization Techniques:**
    - Investigate the combination of QAT and Asymmetric Quantization to leverage the strengths of both methods. Hybrid techniques could potentially achieve better accuracy and efficiency trade-offs.
2. **Broader Model Evaluation:**
    - Evaluate the impact of quantization techniques on other state-of-the-art object detection models, such as EfficientDet and SSD. Comparing different models will provide a more comprehensive understanding of the generalizability of the findings.
3. **Advanced Quantization Methods:**

- Explore more advanced quantization methods, such as mixed-precision training and binarized neural networks, to further reduce model size and computational requirements while maintaining high performance.

4. **Edge Device Performance Profiling:**

   - Conduct detailed performance profiling of edge devices to identify potential bottlenecks and optimize hardware utilization. This could involve exploring the integration of hardware accelerators, such as TPUs, for further efficiency gains.

5. **Real-World Deployment:**

   - Implement and test the optimized models in real-world scenarios to assess their practical performance and robustness. This will help identify any challenges or limitations that may not be apparent in controlled experimental settings.

6. **Energy Efficiency:**

   - Investigate the impact of quantization on energy consumption. Quantization techniques can potentially reduce power usage, making them more suitable for battery-operated and energy-constrained devices.

This research has successfully demonstrated the effectiveness of Quantization-Aware Training and Asymmetric Quantization in optimizing the YOLOv9 model for deployment on NVIDIA Jetson devices. The significant improvements in inference time, coupled with the minimal reduction in accuracy, highlight the potential of these techniques for real-time object detection applications on edge devices. By addressing the suggested future research directions, further advancements can be made in the field of object detection in model optimization, enhancing the deployment and performance of AI-driven solutions in resource-constrained environments.

REFERENCES

[1] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A., 2016. You Only Look Once: Unified, Real-Time Object Detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779-788.

[2] Redmon, J. and Farhadi, A., 2017. YOLO9000: Better, Faster, Stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7263-7271.

[3] Redmon, J. and Farhadi, A., 2018. YOLOv3: An Incremental Improvement. *arXiv preprint* arXiv:1804.02767.

[4] Bochkovskiy, A., Wang, C.Y. and Liao, H.Y.M., 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. *arXiv preprint* arXiv:2004.10934.

[5] Jocher, G., 2021. YOLOv5. GitHub repository. Available at: https://github.com/ultralytics/yolov5 [Accessed 23 July 2024].

[6] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., ... and Adam, H., 2018. 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)*, pp. 2704-2713.

[7] Han, S., Mao, H. and Dally, W.J., 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization, and Huffman Coding. *arXiv preprint* arXiv:1510.00149.

[8] NVIDIA Corporation, 2020. NVIDIA Jetson AGX Xavier Developer Kit. Available at: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit [Accessed 23 July 2024].

[9] NVIDIA Corporation, 2021. NVIDIA TensorRT: High-Performance Deep Learning Inference. Available at: https://developer.nvidia.com/tensorrt [Accessed 23 July 2024].

[10] Li, F., Zhang, B. and Liu, B., 2016. Ternary Weight Networks. *arXiv preprint* arXiv:1605.04711.

[11] Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H. and Zou, Y., 2016. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv preprint* arXiv:1606.06160.

[12] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R. and Bengio, Y., 2016. Binarized Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 4107-4115.

[13] Zhang, D., Yang, J., Ye, D. and Hua, G., 2018. LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 365-382.

[14] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., ... and Houston, M., 2018. Mixed Precision Training. In *International Conference on Learning Representations (ICLR)*.

[15] Rastegari, M., Ordonez, V., Redmon, J. and Farhadi, A., 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 525-542.

[16] Shin, D.J. and Kim, J.J., 2022. A deep learning framework performance evaluation to use YOLO in Nvidia Jetson platform. *Applied Sciences*, *12*(8), p.3734.

[17] Liberatori, B., Mami, C.A., Santacatterina, G., Zullich, M. and Pellegrino, F.A., 2022, May. Yolo-based face mask detection on low-end devices using pruning and quantization. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)* (pp. 900-905). IEEE.

[18] Bae, S.J., Choi, H.J. and Jeong, G.M., 2019. YOLO Model FPS Enhancement Method for Determining Human Facial Expression based on NVIDIA Jetson TX1. *The Journal of Korea Institute of Information, Electronics, and Communication Technology*, *12*(5), pp.467-474.

[19] Suominen, J., 2022. Real-time object detection on edge devices.

[20] Plastiras, G., Siddiqui, S., Kyrkou, C. and Theocharides, T., 2020, August. Efficient embedded deep neural-network-based object detection via joint quantization and tiling. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)* (pp. 6-10). IEEE.

[21] Ding, C., Wang, S., Liu, N., Xu, K., Wang, Y. and Liang, Y., 2019, February. REQ-YOLO: A resource-aware, efficient quantization framework for object detection on FPGAs. In *proceedings of the 2019 ACM/SIGDA international symposium on field-programmable gate arrays* (pp. 33-42).

[22] Hu, X. and Wen, H., 2021, November. Research on model compression for embedded platform through quantization and pruning. In *Journal of Physics: Conference Series* (Vol. 2078, No. 1, p. 012047). IOP Publishing.

[23] Joshi, V.S., Thomas, J. and Raj, E.D., 2022. Quantized Coconut Detection Models with Edge Devices. *Journal of Interconnection Networks*, *22*(Supp03), p.2144010.

[24] Mittal, S., 2019. A survey on optimized implementation of deep learning models on the nvidia jetson platform. *Journal of Systems Architecture*, *97*, pp.428-442.

[25] Al Amin, R., Hasan, M., Wiese, V. and Obermaisser, R., 2024. FPGA-based Real-Time Object Detection and Classification System using YOLO for Edge Computing. *IEEE Access*.

[26] Javed, M.G., Raza, M., Ghaffar, M.M., Weis, C., Wehn, N., Shahzad, M. and Shafait, F., 2021, November. QuantYOLO: A High-Throughput and Power-Efficient Object Detection Network for Resource and Power Constrained UAVs. In *2021 Digital Image Computing: Techniques and Applications (DICTA)* (pp. 01-08). IEEE.

[27] Zagitov, A., Chebotareva, E.V., Toschev, A.S. and Magid, E.A.E., 2024. Comparative analysis of neural network models performance on low-power devices for a real-time object detection task. *Компьютерная оптика*, *48*(2), pp.242-252.

[28] Xue, C., Xia, Y., Wu, M., Chen, Z., Cheng, F. and Yun, L., 2024. EL-YOLO: An efficient and lightweight low-altitude aerial objects detector for onboard applications. *Expert Systems with Applications*, p.124848.

[29] Tang, Y., Wang, Y. and Qian, Y., 2024, May. Real-time railroad track components inspection framework based on YOLO-NAS and edge computing. In *IOP Conference Series: Earth and Environmental Science* (Vol. 1337, No. 1, p. 012017). IOP Publishing.

[30] Sarvajcz, K., Ari, L. and Menyhart, J., 2024. AI on the Road: NVIDIA Jetson Nano-Powered Computer Vision-Based System for Real-Time Pedestrian and Priority Sign Detection. *Applied Sciences*, *14*(4), p.1440.

[31] Lizano, S.A. and Westerlund, T., 2024. Comparison of edge computing platforms for hardware acceleration of AI: Kria KV260, Jetson Nano and RTX 3060.

[32] Yang, Y., 2024, May. Quantization and Acceleration of YOLOv5 Vehicle Detection Based on GPU Chips. In *2024 International Conference on Generative Artificial Intelligence and Information Security* (pp. 425-429).