Revista Eletrônica de Iniciação Científica em Computação, 2025, XX:1
ISSN: 1519-8219 • doi: 10.5753/reic.202X.XXXXXX

Research Paper

Quantization Strategies for Low-Resource Hardware in Deep Learning Applications

Caio Martins de Abreu [Instituto de Tecnologia e Liderança | *caio.abreu@sou.inteli.edu.br*]
Filipi Enzo Siqueira Kikuchi [Instituto de Tecnologia e Liderança |*filipi.kikuchi@sou.inteli.edu.br*]
Pablo Ruan Lana Viana [Instituto de Tecnologia e Liderança | *pablo.viana@sou.inteli.edu.br*]
Rodrigo Mangoni Nicola[Instituto de Tecnologia e Liderança | *nicola@prof.inteli.edu.br*]

Instituto de Tecnologia e Liderança, Av. Prof. Almeida Prado, 520, Butantã, São Paulo, SP, 05508-070, Brazil.

Abstract. This paper investigates quantization techniques applied to convolutional neural networks (CNN) on systems with limited hardware resources. Quantization is a technique that allows representation of numerical data with reduced precision, significantly decreasing memory usage and computational cost. Such technique is essential for embedded and energy-constrained devices. This paper explores and evaluates different quantization strategies to balance accuracy, efficiency for low-resources hardware.

Keywords: Quantization, Post Training Quantization (PTQ), Convolutional Neural Networks, YOLO, Raspberry Pi

Received: DD Month YYYY • **Accepted:** DD Month YYYY • **Published:** DD Month YYYY

1 Introduction

The usage of convolutional neural networks (CNNs) in the field of computer vision has become widespread, following their rapid progress in recent years [Zhao *et al.*, 2024]. CNNs have demonstrated remarkable performance in tasks such as image classification, object detection, and semantic segmentation, establishing themselves as the foundation of modern visual recognition systems. Their success is largely attributed to their hierarchical feature extraction capability, which allows for the automatic learning of increasingly abstract representations from raw image data [Lee *et al.*, 2016].

The exponential growth of CNN applications, especially in computer vision, has significantly increased the demand for deploying such models on embedded platforms and low-power devices. Although these platforms offer reasonable processing power and flexibility, implementing full-precision models — typically using 32-bit floating-point arithmetic — is often inefficient in terms of power consumption, logic utilization, and memory bandwidth [Dongarra *et al.*, 2024]. These limitations restrict the feasibility of real-time inference and hinder the integration of deep learning models into resource-constrained environments such as autonomous drones, IoT systems, and mobile robotics.

In this context, post-training quantization (PTQ) has emerged as a promising technique to reduce model complexity without requiring retraining. PTQ converts model parameters and activations from high-precision to lower-precision formats (e.g., 8-bit integers), effectively decreasing the computational and memory footprint. Among the PTQ techniques, methods like Accurate and Hardware-Compatible Post-Training Quantization (AHCPTQ) have shown the ability to maintain high levels of accuracy while ensuring compatibility with hardware-optimized arithmetic units [Zhang *et al.*, 2025]. This makes PTQ particularly valuable for real-world embedded implementations where energy efficiency and latency are critical factors.

Moreover, trainable fixed-point quantization approaches integrate quantization directly into the training process. This strategy allows the model to adapt to quantization-induced distortions during learning, often achieving better accuracy compared to purely post-training methods. By leveraging flexibility in defining data paths and word lengths, such approaches can produce hardware-aware models that are specifically optimized for the target architecture, as demonstrated in the HQOD: Harmonious Quantization for Object Detection framework [Huang *et al.*, 2024].

Beyond integer quantization, researchers have also explored the use of reduced-precision floating-point formats, including half-precision (FP16) and custom low-bitwidth floating-point types [Ortiz *et al.*, 2018]. These representations can offer a trade-off between numerical stability and efficiency, improving throughput and energy consumption while still supporting a broader dynamic range than integer quantization schemes.

Given these developments, this work proposes the implementation and evaluation of post-training quantization (PTQ) techniques on convolutional neural networks (CNNs), emphasizing strategies that balance model accuracy, computational efficiency, and hardware compatibility. By reducing numerical precision in model parameters and activations, PTQ enables more compact and energy-efficient deployments, a critical requirement for embedded platforms such as Raspberry Pi devices. This approach allows for substantial reductions in memory footprint and inference latency without necessitating additional retraining cycles or high-performance computing resources.

Building upon this motivation, the central objective of the present study is to assess the practical implications of applying PTQ to object detection networks operating under strict computational and memory constraints. Through a systematic evaluation of accuracy, inference latency, CPU utilization, and memory consumption, this work aims to characterize how quantization affects real-world performance when deployed on low-resource hardware. By demonstrating the feasibility of quantized models in embedded scenarios, the study underscores the importance of model optimization as a pathway to broaden the accessibility of deep learning technologies. In doing so, it positions quantization—and related compression techniques—as essential mechanisms for enabling scalable, efficient, and widely deployable artificial intelligence solutions.

2 Theoretical Foundation

Given the growing demand for efficient deployment of deep learning models in embedded environments, understanding the mathematical principles that govern model optimization techniques becomes essential. Among these techniques, quantization stands out as a key method for reducing computational complexity and memory consumption while maintaining acceptable accuracy levels. To properly evaluate its impact on convolutional neural networks, particularly in object detection tasks such as those performed by YOLO architectures, it is necessary to establish a solid theoretical foundation. The following section presents the fundamental concepts of quantization, including its mathematical formulation, types, and the inherent trade-offs between precision and efficiency that underlie its application in low-resource hardware systems.

2.1 Fundamentals of Quantization

In essence, quantization maps a continuous domain of floating-point values $x \in \mathbb{R}$ to a discrete set of integer levels $q \in \mathbb{Z}$ through a linear transformation [Jacob *et al.*, 2018]. This process, illustrated in **Figure 1**, demonstrates how continuous values are discretized into fixed integer representations, such as 8-bit quantization. Mathematically, the mapping can be expressed as:

$$q = \text{round}\left(\frac{x}{s}\right) + z \tag{1}$$

Where:

- s represents the *scale factor*, defining the step size between quantized values;
- z is the *zero-point*, aligning zero in the integer domain with zero in the real domain;
- q is the resulting quantized integer representation.

The inverse process, known as *dequantization*, restores an approximate floating-point value according to:

$$\hat{x} = s \cdot (q - z) \tag{2}$$

The parameters s and z are chosen to minimize the approximation error between x and $\hat{x}$. This optimization can be formalized as a mean squared error minimization problem:

$$\min_{s,z} \|x - \hat{x}\|_2^2 \tag{3}$$

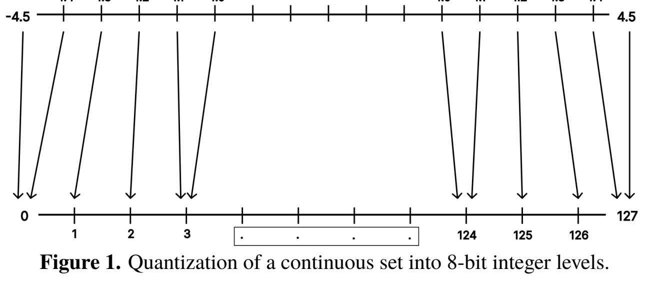

Figure 1. Quantization of a continuous set into 8-bit integer levels.

Through this formulation, quantization can be interpreted as a controlled approximation process in which numerical precision is deliberately traded for computational efficiency, a concept central to post-training quantization techniques.

2.2 Quantization Techniques

Quantization techniques differ mainly in how and when the quantization parameters are computed and applied. The two most common approaches are static quantization (also known as post-training quantization) and dynamic quantization.

2.2.1 *Static Quantization*

In static quantization both model weights and activations are quantized after the model has been trained using floating-point precision. This approach relies on a representative dataset—often referred to as a calibration dataset—to estimate the range of activations observed during inference.

Static quantization involves two key steps:

1. Range determination: Using the calibration data, the minimum and maximum values of activations are measured to define the appropriate scale s and zero-point z
2. Graph transformation: The floating-point operations in the model are replaced or augmented by quantized integer operations that operate on INT8 tensors.

By quantizing both weights and activations, static quantization significantly reduces memory bandwidth and computation cost, leading to faster inference. However, the fixed ranges determined during calibration can sometimes lead to small accuracy losses if the runtime activations differ significantly from the calibration statistics.

2.2.2 *Dynamic Quantization*

Dynamic quantization, on the other hand, stores the weights in quantized format but computes the activation quantization parameters dynamically at inference time, adapting to each input batch. Mathematically, this can be expressed as:

$$q = round\left(\frac{x}{s(x)}\right) + z(x) \tag{4}$$

where $s(x)$ and $z(x)$ vary according to the observed input distribution. Although this approach offers smaller latency gains compared to static quantization, it provides better robustness to input variability and does not require a calibration dataset.

2.3 Quantization Noise and Error Propagation

Quantization inherently introduces an approximation error, commonly referred to as quantization noise, which results from mapping a continuous-valued signal $x \in \mathbb{R}$ onto a set of discrete integer levels $z \in \mathbb{Z}$. This process inevitably leads to a loss of precision, as each continuous value is approximated by its nearest quantized representation. The quantization error can be modeled as an additive random variable, often assumed to follow a uniform distribution within the bounds of the quantization step size. This stochastic model, first introduced and analyzed by [Widrow and Kollár, 2008], serves as the foundation for understanding the statistical behavior of quantization noise and its impact on signal fidelity. The quantization error ε is thus expressed as:

$$\begin{aligned} \varepsilon &= x - \hat{x} && (5) \\ &= x - s \cdot (q - z) && (6) \end{aligned}$$

Where s represents the quantization step size, q the quantized value, and z the corresponding integer level.

When the rounding operation is unbiased and uniformly distributed, ε can be approximated as a random variable uniformly distributed in the interval:

$$\varepsilon \sim \mathcal{U}(-\frac{s}{2}, \frac{s}{2}) \qquad (7)$$

The expected value and variance of this noise are:

$$E[\varepsilon] = 0, \quad \mathrm{Var}[\varepsilon] = \frac{1}{12}s^2 \qquad (8)$$

This demonstrates that quantization does not systematically bias the data (zero-mean noise), and that the magnitude of the error grows quadratically with the quantization step s. Consequently, reducing the scale factor increases precision and decreases quantization noise.

The implications of this model extend directly to neural network quantization, where each operation—such as matrix multiplication or convolution—is affected by quantization-induced perturbations in both weights and activations. By interpreting these perturbations as structured noise, it becomes possible to analyze their statistical behavior and design strategies to mitigate their effects. Consequently, quantization can be viewed not only as a compression technique but also as a controlled approximation process that trades numerical precision for computational efficiency, a principle that underlies the post-training quantization methods explored in this study.

3 Related Work

Quantization has long been recognized as an essential technique for compressing deep neural networks (DNNs) to enable efficient inference on hardware-constrained platforms. Early research, as reviewed by [Santini *et al.*, 2025], explored the trade-offs between static and dynamic quantization strategies. Static quantization precomputes scaling parameters using calibration datasets, offering computational efficiency but limited robustness to distributional shifts. Dynamic quantization, in contrast, adapts parameters at runtime, improving generalization at the cost of higher computational overhead. The authors further discuss mixed-precision approaches that assign varying bit-widths to individual layers, optimized through sensitivity analyses or reinforcement learning, culminating in the Probabilistic Dynamic Quantization (PDQ) method — a technique that estimates activation statistics using probabilistic modeling to improve memory efficiency without retraining.

Building upon this foundation, [Zhang *et al.*, 2021] expand the discussion to include pruning and hardware co-design, emphasizing quantization's role in enabling energy-efficient inference. They introduce Term Quantization (TQ) and a multi-resolution inference framework, which reuse quantization terms across sub-models to reduce redundancy and dynamically adjust arithmetic precision. Their proposed multi-resolution multiplier-accumulator (mMAC) architecture exemplifies how algorithmic and hardware-level optimization can be integrated to achieve flexible precision control. This shift toward co-optimization reflects a growing recognition that effective quantization must jointly consider algorithmic structure and underlying hardware constraints.

Focusing on object detection, [Liberatori *et al.*, 2022] examine the application of compression techniques to YOLO-based architectures, emphasizing the difficulty of deploying such models on single-board computers like the Raspberry Pi. Their study combines structured filter pruning and post-training quantization (PTQ) on YOLOv4-Tiny, achieving nearly double the inference speed with minimal accuracy loss. Similarly, [Chen and Lou, 2022] refine PTQ for object detection through clipping-based optimization, mitigating outlier effects in activation distributions. Their method demonstrates that statistically tuned clipping thresholds can preserve mean average precision (mAP) while significantly reducing model size, extending the applicability of PTQ beyond classification tasks.

Addressing the persistent limitations of static quantization, [Kim and Kim, 2021] propose a zero-centered fixed-point quantization with iterative retraining for convolutional object detectors. By adjusting weight distributions toward zero and retraining iteratively, their method achieves near–full-precision accuracy on YOLOv3 and YOLOv4 while reducing bit usage to approximately 20% of the original representation. Finally, [Alqahtani *et al.*, 2025] present a benchmark study evaluating quantized object detection models such as YOLOv8, EfficientDet Lite, and SSD across Raspberry Pi and Nvideo Jetson, revealing the trade-offs between energy efficiency, inference latency, and accuracy across different hardware generations.

Collectively, these studies trace the evolution of quantization from theoretical optimization to practical deployment in embedded environments. Yet, most research focuses either on architectural efficiency or algorithmic compression, leaving limited analysis on the trade-offs among quantization strategies specifically tailored for low-resource hardware. This work aims to bridge that gap by evaluating post-training quantization techniques on convolutional object detection networks, highlighting the balance between precision, computational efficiency, and hardware adaptability in constrained embedded systems.

4 Methodology

For this analysis, the primary goal was to train and evaluate a computer vision model based on convolutional neural networks (CNNs) for a specific task of detecting people. In this work, the YOLOv8 (You Only Look Once) architecture was selected due to its proven efficiency and real-time detection capability [Maji *et al.*, 2022], making it a strong candidate for deployment on devices with limited computational resources.

The model was trained using a custom dataset designed to identify people in various environments and lighting conditions, allowing for more realistic evaluation in embedded scenarios. After the training process, the optimized model was deployed on a Raspberry Pi 5 platform to analyze its performance under constrained hardware conditions. The evaluation included key performance metrics such as accuracy, inference latency, CPU utilization, and memory consumption.

To enhance model efficiency, post-training quantization (PTQ) techniques were applied, reducing numerical precision to minimize computational load while preserving detection accuracy. This methodology enables a practical assessment of the trade-offs between model performance and resource efficiency in embedded artificial intelligence applications.

4.1 Model Selection

The YOLOv8 (You Only Look Once) model represents the latest evolution of the YOLO family of object detection architectures, developed by Ultralytics. It introduces several architectural and functional improvements over its predecessors, focusing on achieving higher accuracy, faster inference, and increased flexibility for deployment across various hardware platforms [Sohan *et al.*, 2024].

YOLOv8 follows an anchor-free detection paradigm, replacing the traditional anchor-based mechanism used in previous versions such as YOLOv5 and YOLOv7. This change simplifies the detection head and reduces the computational overhead, as the model directly predicts object centers, dimensions, and class probabilities without predefined anchor boxes. This approach leads to better generalization across different datasets and improves performance, particularly in small-object detection scenarios. The architecture of YOLOv8 can be conceptually divided into three main components:

1. Backbone: Responsible for extracting hierarchical features from input images. YOLOv8 employs a C2f module, an enhanced variant of the C3 block introduced in YOLOv5, which utilizes cross-stage partial connections to increase gradient flow and feature reuse while reducing parameter count.
2. Neck: The neck aggregates multi-scale features using a PAN (Path Aggregation Network) structure, allowing the model to maintain strong spatial and semantic representations at different resolutions.
3. Head: The detection head performs object localization and classification through decoupled branches for regression and classification tasks, which enhances convergence and improves detection accuracy.

Another important feature of YOLOv8 is its modular design, offering multiple model sizes (e.g., YOLOv8n, YOLOv8s, YOLOv8m, YOLOv8l, YOLOv8x) to balance accuracy and computational demand. For this study, the lightweight version, YOLOv8n, was selected to ensure compatibility with embedded devices such as the Raspberry Pi 5, where memory and processing resources are limited.

Training of the YOLOv8n model was performed using the Ultralytics YOLO framework. The configuration parameters were defined to balance accuracy and computational efficiency. Specifically, the training was executed for 50 epochs with an input image size of 640 × 640 pixels, a batch size of 16, and a custom dataset defined in an yaml file.

This configuration was chosen to ensure that the model converged effectively while maintaining feasible training times on the available hardware. The learning rate and optimizer parameters followed the default YOLOv8 settings, which apply adaptive adjustment through cosine annealing scheduling. These settings were found to offer a good balance between stability and performance for the person detection task.

YOLOv8's combination of computational efficiency, flexibility, and high detection performance makes it an ideal choice for evaluating post-training quantization techniques on embedded hardware platforms.

4.2 Custom Dataset

The dataset used in this study was created using the Roboflow platform, which provides an integrated environment for dataset management, annotation, and preprocessing in computer vision projects. The dataset consists of 3,737 images, including both scenes with and without people, captured under varying lighting conditions, locations, and contextual settings. This diversity was intentionally incorporated to improve the model's generalization capability and to simulate real-world environments where illumination and background complexity may vary.

To ensure proper evaluation, the dataset was divided into three subsets following standard machine learning practices:

- Training set: 3,060 images ($\approx 82\%$)
- Validation set: 526 images ($\approx 14\%$)
- Test set: 151 images ($\approx 4\%$)

All images were manually labeled for this research to guarantee annotation quality and consistency. Each label corresponds to a bounding box representing a person within the image. Among the 3,737 samples, 2,632 images contain between 2 and 15 annotated persons, resulting in a diverse distribution of object counts per frame. This variability allows the model to learn how to handle both sparse and dense detection scenarios.

The average image size in the dataset is approximately 0.92 megapixels, with a median resolution of 1280×720 pixels. These dimensions provide a balanced compromise between visual detail and computational efficiency during training.

Before training, Roboflow was also used to standardize the dataset format for the YOLO framework, automatically converting annotations to the required structure and resizing images to the selected training resolution (640×640). This ensured data uniformity and compatibility with the Ultralytics YOLOv8 training pipeline.

The resulting dataset offers a robust and representative foundation for evaluating the performance of quantized convolutional neural networks in person detection tasks under diverse real-world conditions.

4.3 Hardware

The Raspberry Pi platform has emerged as one of the most widely adopted single-board computers in embedded systems research, owing to its compact design, affordability, and versatility in supporting real-world machine learning applications. As discussed by [Ghael *et al.*, 2020], its balance between computational capability and energy efficiency has made it a practical platform for implementing artificial intelligence (AI) solutions in areas such as automation, computer vision, and the Internet of Things (IoT). Although its processing power remains limited compared to GPU-based systems, the Raspberry Pi's open hardware architecture and compatibility with lightweight deep learning frameworks make it an appropriate platform for studying model deployment under strict resource constraints.

Recent benchmarking efforts, such as YoloBench by [Lazarevich *et al.*, 2023], emphasize the importance of evaluating object detection models in realistic embedded environments. Their findings demonstrate that performance trade-offs between inference speed, energy consumption, and accuracy vary significantly across YOLO architectures when executed on low-power hardware. In this context, the Raspberry Pi serves as a representative edge platform for analyzing how post-training quantization (PTQ) strategies influence the computational efficiency and predictive performance of convolutional neural networks (CNNs).

In this study, the Raspberry Pi 5 is employed as the primary test environment for deploying quantized YOLO models. This setup provides a controlled and practical benchmark for assessing the effectiveness of PTQ techniques in embedded applications. The experiments aim to quantify the impact of quantization on inference latency, memory utilization, and detection accuracy, considering the hardware limitations typical of low-resource systems. This approach directly supports the study's objective of identifying quantization strategies that optimize performance without compromising accuracy in embedded object detection tasks.

4.4 Hardware Setup

The experimental methodology relied on two distinct computational environments: a high-performance workstation for model training and a low-power embedded device for deployment and evaluation. This separation allows for a clear analysis of quantization effects by contrasting training on unconstrained hardware with inference on resource-limited systems.

4.4.1 *Training Environment*

Model training was conducted on a high-performance notebook equipped with modern CPU and GPU architectures optimized for deep learning tasks. The configuration, summarized in **Table 1**, ensured stable training dynamics, rapid convergence, and accurate baseline model generation prior to quantization. This setup provided the computational foundation necessary to establish a reference model for subsequent performance comparison on embedded hardware.

Table 1. Hardware specifications used for model training.

Component	Specification
Architecture	x86 (Metero Lake 64-bit)
Processor	Intel® Core™ Ultra 9 185H 3.9GHz
Memory	32GB LPDDR5X
GPU	NVIDIA® RTX™ 4070, 8GB GDDR6
Operating System	Ubuntu 24.04.3 LTS

4.4.2 *Deployment Environment*

Deployment and performance evaluation were conducted on a Raspberry Pi 5, chosen for its representativeness as an embedded AI platform. The Raspberry Pi's combination of ARM-based processing, low power consumption, and compatibility with contemporary software frameworks makes it suitable for assessing real-world inference efficiency of quantized models.

The system was configured to execute quantized YOLO models using the ONNX Runtime backend, enabling cross-platform reproducibility and hardware-agnostic inference evaluation. The specifications of the deployment hardware are detailed in **Table 2**. This configuration was used to measure inference latency, memory utilization, and detection accuracy, enabling a comprehensive assessment of post-training quantization performance on constrained devices.

Table 2. Raspberry Pi 5 hardware specifications used for model deployment.

Component	Specification
Architecture	64-bit ARM (Cortex-A76)
Processor	Broadcom BCM2712
Memory	8GB LPDDR4X
GPU	VideoCore VII
Operating System	Ubuntu 24.04 (LTS)
Inference Backend	ONNX Runtime

The combination of these two computational environments reflects a realistic end-to-end AI workflow: training high-precision models on powerful hardware and deploying quantized versions to low-power embedded systems.

4.5 Quantization

Quantization was applied as a post-training optimization technique to reduce the computational complexity and memory footprint of the trained YOLO model, enabling efficient deployment on the Raspberry Pi 5. This process converts floating-point parameters and activations into lower-precision integer representations while attempting to preserve predictive performance. In particular, 8-bit integer quantization (INT8) was selected, as it offers a favorable balance between accuracy retention and inference efficiency on embedded hardware.

The quantization procedure followed the post-training quantization (PTQ) approach, implemented using the ONNX Runtime framework. Unlike quantization-aware training (QAT), PTQ does not require model retraining, making it suitable for fast and hardware-friendly deployment scenarios. The process involved calibrating activation ranges using representative samples to determine the optimal scaling factors and zero-points for each layer.

This study adopted the Quantization–Dequantization (QDQ) format, which explicitly inserts quantization and dequantization nodes in the computational graph. This structure facilitates compatibility with hardware accelerators and ensures more predictable inference behavior.

4.5.1 *Static*

Static quantization was employed to convert both weights and activations of the YOLO model from floating-point precision (FP32) to 8-bit integer representations prior to deployment. Unlike dynamic quantization, which determines activation scales at runtime, static quantization relies on a calibration phase that statistically analyzes representative input data to estimate the optimal quantization parameters for each tensor. This process ensures that activation distributions are adequately captured before inference, leading to more stable performance in convolutional and feature extraction layers.

The calibration procedure in ONNX Runtime is typically implemented through a *CalibrationDataReader* class, which iteratively feeds preprocessed image samples to the quantization function. These samples are used to compute the minimum and maximum activation values for each layer, which are then mapped into integer ranges according to the quantization scheme. In this study, the quantization process adopted the Quantization–Dequantization (QDQ) format, which explicitly inserts quantization and dequantization nodes into the computational graph. This structure enables precise control over quantized operations while maintaining compatibility with hardware accelerators.

```
quantize_static(
    model_fp32,
    model_quant,
    weight_type=QuantType.QInt8,
    activation_type=QuantType.QUInt8,
    calibration_data_reader=
        calibration_data_reader,
    quant_format=QuantFormat.QDQ,
    nodes_to_exclude=[...],
    per_channel=False,
    reduce_range=True
)
```

The code above represents the static quantization workflow applied to the YOLOv8 model. The calibration dataset consisted of a subset of validation images, preprocessed to match the model's input size and normalization format. During quantization, ONNX computed per-tensor scale and zero-point parameters for weights and activations, converting floating-point values to their corresponding integer representations. The resulting quantized model 'model_quant' exhibited substantially reduced memory consumption and improved inference efficiency.

An important refinement applied in this study was the selective exclusion of specific computational nodes from quantization, a process referred to as 'node pruning'. Certain layers, particularly non-linear operations such as concatenations, reshapes, softmax, and sigmoid functions, are highly sensitive to precision reduction and may cause instability when quantized. By omitting these nodes through the 'nodes_to_exclude' parameter, the quantization pipeline preserved the numerical integrity of critical components in the detection head, ensuring model stability without compromising overall efficiency.

This static quantization strategy allowed the YOLO model to retain near–full-precision accuracy while achieving significant reductions in inference time and model size, thereby demonstrating the practical effectiveness of post-training quantization.

4.5.2 *Dynamic*

Dynamic quantization was employed to enhance inference efficiency by converting model weights from 32-bit floating-point representation (FP32) to 8-bit unsigned integers (UINT8) during runtime. Unlike static quantization, which precomputes activation ranges through calibration, dynamic quantization determines the appropriate scaling factors and zero-points on demand as data flows through the model. This allows flexible adaptation to varying activation distributions while maintaining computational efficiency.

The ONNX Runtime framework provides native support for dynamic quantization through the following implementation:

```
quantized_model = quantize_dynamic(
    model_fp32,
    static_model_quant,
    weight_type=QuantType.QUInt8
)
```

In this implementation, 'model_fp32' denotes the original full-precision YOLO model, and 'static_model_quant' specifies the destination path for the quantized version. The parameter 'weight_type=QuantType.QUInt8' indicates that model weights are quantized to 8-bit unsigned integers, reducing memory requirements and accelerating inference on CPU-bound hardware.

Internally, ONNX applies dynamic quantization by analyzing weight tensors and assigning per-tensor scale and zero-point parameters that map floating-point values to the integer domain. During inference, activations are quantized dynamically, matrix multiplications are executed in integer precision, and results are subsequently dequantized to floating-point form. This approach avoids retraining or calibration stages, providing an effective balance between accuracy and runtime performance for deployment on embedded devices.

5 Results

This section presents the experimental results obtained from evaluating the post-training quantization (PTQ) techniques applied to the YOLO model. The objective of these experiments is to analyze the impact of quantization on inference performance, memory utilization, and detection accuracy when deploying convolutional neural networks (CNNs) in embedded environments. The evaluation was conducted using the Raspberry Pi 5 as the target hardware, representing a typical low-power and resource-constrained platform.

Results are organized to highlight three key aspects: (*i*) the accuracy trade-off introduced by quantization compared to the full-precision baseline model, (*ii*) the computational gains achieved in terms of inference latency and throughput, and (*iii*) the effect of quantization on memory consumption during model execution. Each analysis aims to assess the balance between efficiency and precision, illustrating how quantization enables practical deployment of object detection models on embedded systems. Quantitative results are complemented by comparative visual outputs to demonstrate the operational feasibility and robustness of the quantized YOLO model in real-world inference scenarios.

5.1 Accuracy Trade-Off Analysis

The evaluation of accuracy under quantization is essential to understanding the trade-offs introduced when reducing numerical precision in convolutional neural networks. Although post-training quantization aims to improve efficiency, it can also affect confidence calibration, detection thresholds, and overall prediction quality. To assess these effects, the performance of the full-precision, statically quantized, and dynamically quantized models was compared using standard object detection metrics, including precision, recall, and mean Average Precision (mAP). As summarized in **Table 3**, the dynamic model closely matches the accuracy of the full-precision baseline, while the static model exhibits a moderate reduction, particularly at higher IoU thresholds.

These quantitative differences are further reflected in the statistical behavior of the models' prediction confidences. To explore how quantization affects confidence distributions—which play a central role in bounding-box filtering and non-maximum suppression—the cumulative distribution of detection scores was analyzed, as shown in **Figure 2**. This visualization allows a direct comparison of how each quantization strategy influences the reliability and calibration of predicted confidence values.

Table 3. Model Metrics

Metric	Full Precision	Static	Dynamic
Precision	0.8909	0.8400	**0.8948**
Recall	**0.7729**	0.7163	0.7708
mAP@50	**0.8571**	0.8035	0.8542
mAP@50–95	**0.5668**	0.4855	0.5575
Fitness	**0.5668**	0.4855	0.5575

Figure 2 presents the cumulative distribution of detection confidences for the three evaluated models: the original full-precision model (FP32), the dynamically quantized version, and the statically quantized version. The x-axis represents the predicted confidence scores, while the y-axis corresponds to the cumulative probability of detections with confidence less than or equal to a given threshold. This visualization enables the comparison of how quantization affects the models' confidence calibration and overall detection reliability.

The distributions for the original and dynamically quantized models exhibit a close alignment across the entire confidence range, indicating that dynamic quantization preserves the statistical behavior of the original network with minimal deviation. In contrast, the statically quantized model shows a slightly steeper cumulative curve, suggesting a higher concentration of lower-confidence detections. This behavior reflects a modest degradation in confidence estimation, commonly attributed to precision loss in activation quantization and rounding effects during the calibration process.

Despite this shift, the static quantization model maintains a comparable global trend to the baseline, implying that the overall decision boundaries of the detector were preserved. The observed trade-off between quantization precision and predictive confidence demonstrates that both quantization strategies achieve a favorable balance between computational efficiency and model fidelity. These results confirm that post-training quantization, particularly in its dynamic form, can be applied to YOLO-based detectors with negligible loss of detection reliability on embedded hardware.

5.2 Latency and Throughput Evaluation

This subsection presents the comparative evaluation of inference latency and computational throughput for the original,

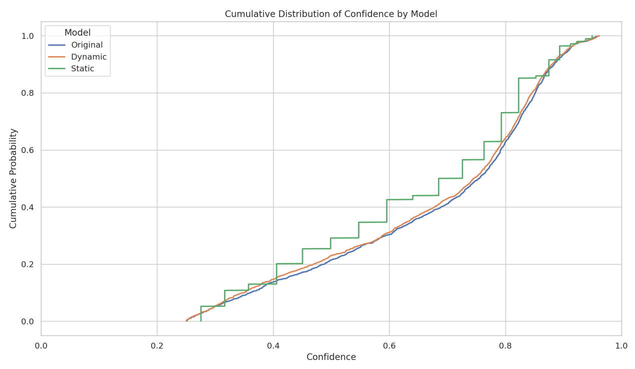

Figure 2. Cumulative confidence distribution for Original, Dynamic, and Static models.

static, and dynamically quantized YOLO models when executed on the Raspberry Pi 5. Latency measurements were obtained across 526 inference iterations for each model, with CPU utilization and memory consumption recorded to characterize hardware efficiency during runtime. **Figure 3**, illustrate series latency comparison for each model configuration.

The results indicate a clear performance advantage for the statically quantized model. As shown in **Table 4**, static quantization reduced the mean inference latency from **376.98 ms** in the original model to **143.38 ms**, representing a speedup of approximately **2.6**×. This improvement stems from the conversion of floating-point operations to low-precision integer arithmetic, which is more efficient on ARM-based processors. The dynamic quantization model achieved a mean latency of **305.31 ms**, demonstrating moderate acceleration relative to the baseline while preserving operational stability across varying input conditions.

Table 4. Mean performance metrics for each model evaluated on the Raspberry Pi 5.

Metric	Original	Static	Dynamic
Latency (ms)	376.98	**143.38**	305.31
CPU Utilization (%)	186.15	320.28	357.51
Memory Usage (MB)	377.09	**342.09**	366.35

While static quantization substantially decreased latency, it also led to higher CPU utilization, averaging **320.28%** compared to **186.15%** in the original model. This behavior reflects the increased computational parallelism resulting from integer-based operations executed across multiple CPU cores. Despite this higher CPU demand, the reduction in inference time contributes to a higher effective throughput, as more frames can be processed per second. Memory consumption remained stable across all configurations, with only a minor reduction observed in the statically quantized model due to compressed weight representation.

The dynamic quantization model presented intermediate results, with latency reductions of roughly **19%** relative to the baseline and similar memory utilization. Its advantage lies in the adaptability of scaling parameters at runtime, allowing more consistent performance across diverse input batches, albeit with less aggressive computational optimization than static quantization.

Overall, the results demonstrate that post-training quantization substantially enhances inference efficiency in embedded environments. Among the evaluated methods, static quantization yielded the most pronounced improvement in throughput and latency reduction, positioning it as the most effective strategy for deployment scenarios that prioritize real-time responsiveness and computational speed. Dynamic quantization, on the other hand, provides a balanced compromise between efficiency and adaptability, offering consistent performance without requiring prior calibration or dataset-dependent tuning—an advantage for rapidly deployable or variable-input applications.

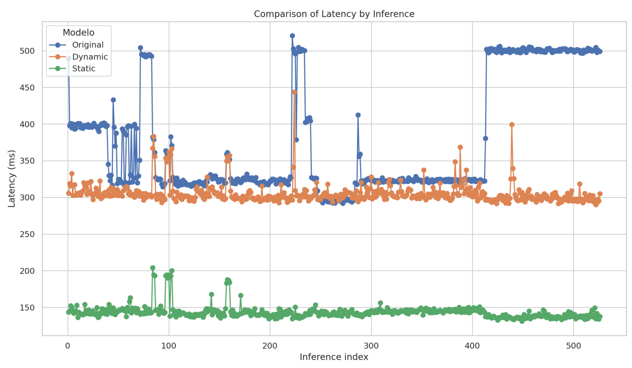

Figure 3. Inference latency per inference for Original, Static, and Dynamic models.

5.3 Memory Impact of Quantization

To complement the latency and throughput evaluation, this subsection analyzes the effect of post-training quantization on memory utilization during inference. Memory efficiency is a critical factor for deployment in embedded systems, where available RAM is often limited and shared among multiple concurrent processes. The average values of memory usage recorded for each model configuration are summarized in **Table 4**, while **Figure 4** illustrates the temporal distribution of memory consumption across inference iterations.

The results indicate that static quantization achieved the most consistent memory behavior among all tested models. The statically quantized YOLO model maintained an average memory usage of approximately **342 MB**, representing a reduction of roughly **9%** compared to the original full-precision network (**377 MB**). This reduction arises primarily from the compression of model weights from 32-bit floating-point to 8-bit integer representations, as well as the simplified arithmetic operations that reduce intermediate tensor allocation. Furthermore, the narrow variance between minimum and maximum memory values demonstrates the stability of the statically quantized inference process.

In contrast, the dynamically quantized model exhibited an average memory usage of **366 MB**, only slightly lower than the baseline. This limited reduction can be attributed to the dynamic allocation of temporary buffers used for activation quantization and dequantization at runtime. Despite its smaller memory advantage, dynamic quantization still offers a favorable trade-off between adaptability and footprint, especially when the application requires flexibility across variable input conditions.

Collectively, the memory analysis underscores that post-training quantization—especially in its static variant—optimizes memory allocation and runtime stability on constrained hardware. By compressing weight representations and minimizing intermediate tensor storage, static quantization effectively reduces the operational footprint of convolutional networks without imposing significant accuracy penalties. In contrast, dynamic quantization maintains moderate memory gains but offers flexibility across diverse runtime conditions. These findings highlight quantization as a critical enabler of scalable and sustainable deep learning deployment on compact embedded systems such as the Raspberry Pi.

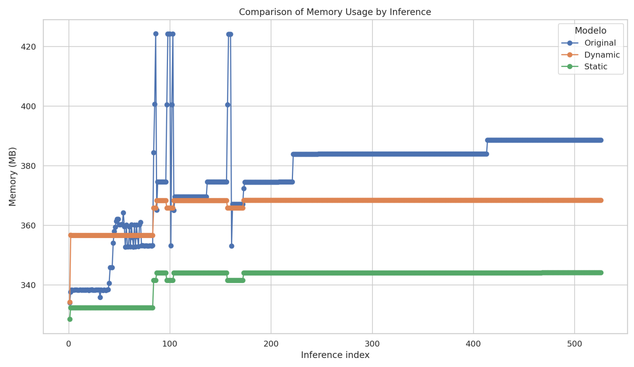

Figure 4. Memory utilization per inference for Original, Static, and Dynamic models.

5.4 Discussion

The experimental results presented across the previous sections provide a comprehensive understanding of how post-training quantization (PTQ) affects the performance, efficiency, and reliability of convolutional neural networks (CNNs) deployed on embedded hardware. By combining latency, memory, and confidence analyses, this study highlights the trade-offs and synergies between static and dynamic quantization techniques when applied to YOLO-based object detectors on low-resource devices such as the Raspberry Pi 5.

From the accuracy standpoint, the confidence distribution analysis revealed that both quantization strategies preserved the overall decision boundaries of the original model, with only minor deviations in prediction confidence. The dynamic quantization approach maintained a near-identical statistical behavior to the full-precision baseline, while static quantization exhibited a moderate shift toward lower confidence scores—a characteristic commonly associated with activation rounding and reduced numerical precision. Despite these effects, the global detection trend remained stable, confirming that post-training quantization can be applied without significant loss of reliability for real-world embedded inference tasks.

In terms of computational efficiency, the latency and throughput results demonstrated a substantial performance improvement enabled by quantization. Static quantization achieved the most pronounced acceleration, reducing inference time by approximately 2.6× compared to the full-precision model. This improvement stems from the transition to integer arithmetic, which is natively optimized on ARM architectures and reduces computational overhead. Although static quantization increased CPU utilization due to the higher degree of parallelism, the overall throughput benefit outweighs this cost, making it the preferred strategy for applications prioritizing real-time responsiveness. Dynamic quantization, by contrast, provided a balanced trade-off—offering moderate speed gains without requiring calibration or prior dataset knowledge, thus remaining advantageous for adaptive or rapidly deployable edge applications.

Regarding memory utilization, the results confirmed that quantization substantially improves runtime efficiency, particularly in the static case. By compressing 32-bit floating-point weights into 8-bit integers and simplifying intermediate tensor operations, static quantization reduced memory usage by approximately 9% while maintaining stable inference behavior. Dynamic quantization, though less effective in compression due to runtime activation scaling, still offered moderate reductions in memory footprint. These findings emphasize that quantization not only accelerates computation but also contributes to more predictable and resource-efficient memory management—an essential factor for embedded systems with strict hardware limitations.

6 Conclusion

The findings of this study substantiate post-training quantization as a viable and effective methodology for deploying deep learning models on resource-constrained hardware platforms. Among the evaluated approaches, static quantization demonstrated superior performance in terms of deterministic and high-throughput inference, establishing it as the most suitable configuration for real-time applications where latency and computational predictability are critical. Conversely, dynamic quantization exhibited greater flexibility and adaptability to varying input conditions, providing an advantageous balance between precision and efficiency for scenarios requiring minimal pre-processing and rapid deployment.

These results reinforce the role of quantization as a foundational technique for optimizing the trade-off between accuracy, performance, and hardware efficiency in embedded environments. By enabling compact model representation and accelerating inference without substantial accuracy degradation, quantization emerges as a key enabler of sustainable and scalable deep learning deployment, particularly for real-time computer vision tasks executed on edge devices.

7 Limitations and Future Work

Although the experimental results demonstrate the effectiveness of post-training quantization for deploying YOLO-based object detectors on embedded hardware, several limitations should be acknowledged. First, the evaluation was conducted exclusively on the Raspberry Pi 5 platform, which, while representative of edge computing environments, does not encompass the diversity of hardware accelerators and architectures commonly used in real-world applications. Future studies could extend this analysis to include heterogeneous environments, such as TPUs, NPUs, and FPGA-based accelerators, to better understand how quantization interacts with specialized hardware optimizations.

Second, the quantization methods explored in this study were limited to uniform 8-bit static and dynamic schemes. More advanced techniques—such as mixed-precision quantization, non-uniform quantization, or quantization-aware training (QAT)—may further improve performance and model fidelity without increasing computational overhead. Integrating these strategies into the training pipeline could reveal additional trade-offs between precision, robustness, and energy efficiency.

Lastly, the evaluation focused primarily on inference-level performance metrics, including latency, memory utilization, and confidence calibration. Future work could incorporate broader system-level analyses, such as energy consumption, thermal behavior, and end-to-end throughput in real-time applications. Expanding the benchmarking framework to include these dimensions would strengthen the understanding of quantization's role in sustainable and scalable deep learning deployment.

By addressing these limitations, future research can build upon the findings of this study to develop more generalizable quantization strategies, optimized for a wide range of hardware architectures and application domains.

Declarations

Authors' Contributions

Caio M. de Abreu contributed to this study with conceptualization, data curation, formal analysis, investigation, methodology, project administration, software, validation and writing — orginal draft. Filipi Enzo K. Siqueira contributed to this study with investigation, software, validation and writing — review & editing. Pablo R. L. Viana contributed to this study with formal analysis, resources, software and writing — review & editing. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests

Availability of data and materials

The dataset analyzed during the current study is available in [Hermes, 2025].

References

Alqahtani, D. K., Cheema, M. A., and Toosi, A. N. (2025). Benchmarking deep learning models for object detection on edge computing devices. *Lecture Notes in Computer Science*, 15404:176–191. DOI: 10.1007/978-981-96-0805-8_11.

Chen, L. and Lou, P. (2022). Clipping-based post training 8-bit quantization of convolution neural networks for object detection. *Applied Sciences*, 12(23). DOI: 10.3390/app122312405.

Dongarra, J., Gunnels, J., Bayraktar, H., Haidar, A., and Ernst, D. (2024). Hardware trends impacting floating-point computations in scientific applications. *arXiv preprint*. DOI: 10.48550/arXiv.2411.12090.

Ghael, H. D., Solanki, L., and Sahu, G. (2020). A review paper on raspberry pi and its applications. *International Journal of Advances in Engineering and Management (IJAEM)*, 2(12):4. DOI: 10.35629/5252-0212225227.

Hermes (2025). tcc dataset. `https://universe.roboflow.com/hermes-eaduu/tcc-j7on9`. visited on 2025-11-10.

Huang, L., Dong, Z., Song-Lu Chen, R. Z., Ti, S., Chen, F., and Yin, X.-C. (2024). Hqod: Harmonious quantization for object detection. *IEEE International Conference on Multimedia and Expo (ICME).*, pages 1–6. DOI: 10.48550/arXiv.2408.02561.

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. *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713. DOI: 10.1109/CVPR.2018.00286.

Kim, S. and Kim, H. (2021). Zero-centered fixed-point quantization with iterative retraining for deep convolutional neural network-based object detectors. *IEEE Access*, 9:20828–20839. DOI: 10.1109/ACCESS.2021.3054879.

Lazarevich, I., Grimaldi, M., Kumar, R., Mitra, S., Khan, S., and Sah, S. (2023). Yolobench: Benchmarking efficient object detectors on embedded systems. In *2023 IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*, pages 1161–1170. DOI: 10.1109/ICCVW60793.2023.00126.

Lee, B., Erdenee, E., Jin, S., and Rhee, P. K. (2016). Efficient object detection using convolutional neural network-based hierarchical feature modeling. *Spring Nature*, 10:1503–1510. DOI: 10.1007/s11760-016-0962-x.

Liberatori, B., Mami, C. A., Santacatterina, G., Zullich, M., and Pellegrino, F. A. (2022). Yolo-based face mask detection on low-end devices using pruning and quantization. *45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 900–905. DOI: 10.23919/MIPRO55190.2022.9803406.

Maji, D., Nagori, S., Mathew, M., and Poddar, D. (2022). Yolo-pose: Enhancing yolo for multi person pose estimation using object keypoint similarity loss. *IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. DOI: 10.1109/CVPRW56347.2022.00297.

Ortiz, M., Cristal, A., and Eduard Ayguadé, M. C. (2018). Low-precision floating-point schemes for neural network training. DOI: 10.48550/arXiv.1804.05267.

Santini, G., Paissan, F., and Farella, E. (2025). Probabilistic dynamic quantization for memory constrained devices. *IEEE/CVF International Conference on Computer Vision*, pages 3973–3982.

Sohan, M., Ram, T. S., and Reddy, C. V. R. (2024). A review on yolov8 and its advancements. *International Conference on Data Intelligence and Cognitive Informatics*, pages 529–545. DOI: 10.1088/1742-6596/1684/1/012028.

Widrow, B. and Kollár, I. (2008). *Quantization Noise: Round-off Error in Digital Computation, Signal Processing, Control, and Communications*. Cambridge University Press.

Zhang, S. Q., McDanel, B., Kung, H. T., and Dong, X. (2021). Training for multi-resolution inference using reusable quantization terms. *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 845–860. DOI: 10.1145/3445814.3446741.

Zhang, W., Zhong, Y., Ando, S., and Yoshioka, K. (2025). Ahcptq: Accurate and hardware-compatible post-training quantization for segment anything model. *IEEE/CVF International Conference on Computer Vision*. DOI: 10.48550/arXiv.2503.03088.

Zhao, X., Wang, L., and Zhang, Y. (2024). A review of convolutional neural networks in computer vision. *Artificial Intelligence Review*, 57(99). DOI: 10.1007/s10462-024-10721-6.