Accelerating YOLOv8n on Raspberry Pi: An Empirical Study of Structured Pruning and Fine-tuning

1st Devadharshan D
Department of Mechanical Engineering
Amrita School of Engineering, Coimbatore
Amrita Vishwa Vidyapeetham, India
cb.en.u4are22014@cb.students.amrita.edu

2nd Karthika R
Department of Electronics and Communication Engineering
Amrita School of Engineering, Coimbatore
Amrita Vishwa Vidyapeetham, India
r_karthika@cb.amrita.edu

***Abstract*—Deploying high performance object detectors on resource-constrained edge hardware continues to be a great difficulty due to computational limitations. This work describes an innovative end-to-end hybrid pipeline for compressing the YOLOv8n model for the Raspberry Pi 5. Unlike generic optimization methods, this method includes surgical pruning to remove unnecessary bottleneck layers, followed by fine-tuning for accuracy recovery and post-training INT8 quantization. Empirical benchmarks show that this pipeline delivers 4.13 FPS, a 34% acceleration over the baseline, with a small accuracy drop (0.354 mAP vs. 0.355 mAP). This study helps in closing the gap between theoretical compression and real-world deployment of object detectors on resource constrained hardware systems.**

***Keywords*—Edge AI, Model Compression, YOLOv8, Object Detection, Raspberry Pi.**

I. Introduction

The growing demand for real-time intelligence in edge computing involves the implementation of deep learning models on resource-constrained hardware. While high performance object detectors such as YOLOv8n provide great accuracy, they require high computing requirements frequently which make them unsuitable for direct usage on low-cost platforms like the Raspberry Pi, where inference delay is a key bottleneck. Although compression techniques such as pruning and quantization exist, there is a scarcity of practical, end-to-end benchmarks that assess their real-world impact on modern architecture and hardware.

To address this gap, this paper presents a hybrid pipeline that combines surgical pruning of the YOLOv8n architecture, performance recovery through fine-tuning, and post-training INT8 quantization to accelerate inference on a Raspberry Pi 5. The main contributions of this work are:

- A specific surgical pruning methodology to reduce the architectural depth of YOLOv8n by targeting redundant bottleneck blocks;
- A demonstration of near-complete accuracy recovery (99.7%) via fine-tuning; and
- A reproducible, empirical benchmark on the Raspberry Pi 5, achieving a 34% speedup (4.13 FPS) with negligible accuracy loss.

II. Literature Review

Applications ranging from industrial automation to precision agriculture depend on the deployment of deep learning models on edge devices with limited resources. Despite providing cutting-edge accuracy, YOLOv8's computational cost necessitates extensive optimization for real-time embedded use. Three main areas are the focus of current research: hardware-focused deployment studies, enhanced small-object detection, and lightweight architectural redesign.

Recent research has substituted lightweight convolutions for standard convolutions in order to lower parameters and FLOPs. For example, [1] and [2] achieved significant parameter reductions by integrating Ghost convolutions and attention mechanisms (M-CBAM, RepGhost) into YOLO architectures for agricultural applications. Others have focused on backbone and neck optimization; [3] reduced computational costs by 35.8% using Reversible Column Networks and GSConv, while [4] employed a lightweight GELAN and Efficient Channel Attention to enhance maritime detection efficiency.

[5] developed LACF-YOLO, which integrates Triplet Attention and Dilated Inverted Bottlenecks to reduce feature loss, in order to tackle the difficulties of small object detection. In a similar vein, [6] proposed SOD-YOLOv8, which uses Powerful-IoU loss and a fourth detection layer to preserve spatial details in traffic scenarios. Although these multi-scale fusion methods successfully improve accuracy, their computational overhead makes deployment on limited edge devices more difficult.

Because of hardware constraints, real-world deployment frequently negates architectural gains. Unoptimized models have substantial latency on Raspberry Pi devices, with inference times surpassing 800 ms on Pi 4, according to studies [7] and [8]. Moreover, complex hybrid models [9] continue to be computationally prohibitive. In response, [10] proposed YOLOv8n-FAWL, which shows that real-time performance (54 FPS) on edge hardware can be achieved by reducing parameters by 49.2% by combining Layer-Adaptive Magnitude Pruning with sophisticated loss functions.

Additionally, pruning and knowledge transfer have been combined in recent work. In order to enable the pruned student model to learn rich features from a larger teacher, [11] proposed PKD-YOLOv8 for pest detection on edge devices, combining structured pruning with Logit-based Masked Generative Distillation. This method demonstrated the efficacy of hybrid pruning–distillation techniques for edge deployment by reducing model size by 60.7% (11.2 MB to 4.4 MB) with only a 0.1% accuracy drop.A Convultional Neural Network(CNN)-based traffic sign recognition system on Raspberry Pi [12] reached 99.8% accuracy on the GTSRB dataset, demonstrating the viability of embedded vision. A two-stage YOLOv3 + CNN pipeline [13] achieved 89.56% mAP and 86.6% accuracy under varied conditions. Likewise, a stereo-vision pseudo-LiDAR model [14] enabled accurate 3D perception for real-time path planning, and an IMU-based driver-behavior classification system [15] achieved 98% accuracy and introduced a DriveScore for safety assessment.

There is a dichotomy in the literature today: either standard models with inadequate latency on low-power hardware ([7]) or complex architectures that are too heavy for edge devices ([5], [6], [9]). There is a significant need for a rigorous, repeatable pipeline that combines INT8 quantization, fine-tuning, and surgical structural pruning that is specifically benchmarked on the Raspberry Pi 5. In order to close this gap, this work empirically validates a hybrid pipeline that allows for real-time inference while maintaining state-of-the-art accuracy.

III. Methodology

This work employs a sequential pipeline to optimize YOLOv8n for edge hardware. The methodology consists of five phases: baseline establishment, surgical pruning, accuracy recovery, quantization, and final deployment.

A. Image Acquisition

The pipeline accepts raw visual data from the COCO 2017 dataset for training. Input images are preprocessed by resizing them to a standardized resolution of 640 × 640 pixels to match the network's input layer.

B. YOLOv8n Architecture Overview

The standard **YOLOv8n (Nano)** architecture, is selected for the initialization of the optimization pipeline as it has a very good balance between accuracy and computational efficiency.To understand the optimization strategy, it is essential to define the specific building blocks of this architecture:

1) CSPDarknet53 Backbone: YOLOv8n's backbone is based on the **CSPDarknet53** family, which is the network's primary feature extractor. It uses strided convolutions to process the input image layer by layer, producing hierarchical feature maps at three spatial scales (P3, P4, and P5).These multi-scale features allow for reliable detection of objects of various sizes.

2) C2f Module (Cross-Stage Partial with Two Convolutions): YOLOv8 updates the previous C3 module with the more efficient **C2f** (Cross-Stage Partial with Two Convolutions) module. The design enhances gradient flow while reducing computing overhead. Given an input tensor, the module divides it into two pathways.

- **Path 1**: travels through a sequence of bottleneck layers.
- **Path 2**: completely bypasses the bottlenecks.

The results from both pathways are then concatenated.This architecture enhances gradient propagation during backpropagation, lowers redundancy, and increases the representational power of the lightweight YOLOv8n network.

3) Bottleneck Layer: A textbfBottleneck is a lightweight sub-unit of the C2f module that reduces computing complexity. It works by first compressing the channel dimensions, then conducting efficient internal modifications, and then extending the channels to their original size. Stacking numerous Bottleneck layers enhances both the effective network depth and the receptive field, allowing the model to capture more complex hierarchical patterns.

4) SPPF (Spatial Pyramid Pooling – Fast): At the end of the backbone, YOLOv8 implements the **SPPF** module, a faster variation of Spatial Pyramid Pooling (SPP). This module does max pooling at several kernel sizes, including: $(5 \times 5,\ 9 \times 9,\ \ 13 \times 13)$. The pooled outputs are concatenated, allowing the model to aggregate context across scales. This procedure improves robustness to object scale fluctuations and global information capture while incurring minimal computing expense.

5) PANet (Path Aggregation Network): The **PANet** architecture serves as the "Neck" of YOLOv8. It collects and combines features from various stages of the backbone to increase detection accuracy. PANet combines high-level semantic information from deeper layers and low-level localization features from shallow layers. This top-down and bottom-up fusion guarantees that the final feature maps are semantically rich and spatially exact, which is critical for successful object detection at several sizes.

6) Optimization Hypothesis: The conventional C2f module increases representational capacity by stacking numerous Bottleneck layers. However, we propose that, for some edge deployment circumstances, the final Bottleneck in the stack may cause computational redundancy. This new bottleneck increases latency without delivering a proportionate gain in feature discrimination. As a result, removing or trimming this bottleneck may preserve accuracy while drastically lowering inference time.

C. Surgical Structural Pruning

A Structured Surgical Pruning algorithm was created to reduce the computational latency present in deep convolutional networks on edge hardware. Our method physically alters the model graph by eliminating entire architectural blocks, in contrast to unstructured pruning, which zeros out individual weights to create sparse matrices (often requiring specialized hardware support to realize speed gains). This leads to a

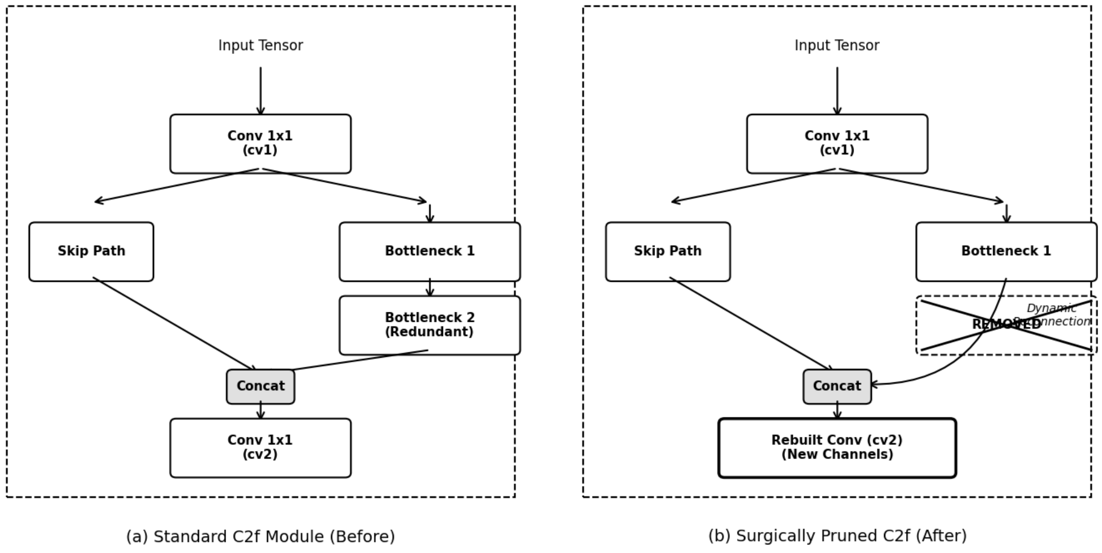

Fig. 1: Schematic of the Surgical Pruning process on a C2f module. (a) The standard module architecture (b) The surgically pruned module where the redundant bottleneck is removed

Algorithm 1 Surgical Pruning of C2f Modules

```
Require: Pre-trained YOLOv8n Model $M$
Ensure: Pruned Model $M_{pruned}$
 1: for each module $m$ in $M.modules()$ do
 2:     if $m$ is instance of $C2f$ then
 3:         $n \leftarrow len(m.bottlenecks)$
 4:         if $n > 1$ then
 5:             {Remove the final redundant bottleneck}
 6:             $m.bottlenecks \leftarrow m.bottlenecks[:-1]$
 7:             {Calculate new input channels for stability}
 8:             $C_{skip} \leftarrow m.cv1.out\_channels$
 9:             $C_{proc} \leftarrow len(m.bottlenecks) \times m.bottlenecks[0].out$
10:             $C_{new} \leftarrow C_{skip} + C_{proc}$
11:             {Rebuild final convolution}
12:             $m.cv2 \leftarrow Conv(C_{new}, m.cv2.out, ...)$
13:         end if
14:     end if
15: end for
16: return $M$
```

lighter, denser model that offers universal acceleration on common CPUs, such as the Arm Cortex-A76 in the Raspberry Pi 5.

1) Target Identification: The C2f Module: The pruning process specifically targets the **C2f** (Cross-Stage Partial with two convolutions) module, which serves as the primary feature extraction block within the YOLOv8 backbone. As illustrated in Fig. 2(a), the standard C2f module splits the incoming tensor into two parallel computational paths to enhance gradient flow and feature diversity.

- **Skip Path:** A direct shortcut branch that preserves low-level spatial features by forwarding a portion of the input tensor unchanged.
- **Processing Path:** A deeper branch composed of a sequence of Bottleneck layers (denoted as *Bottleneck 1* and *Bottleneck 2* in the figure), responsible for extracting higher-level semantic representations.

Our work proposes that, for specific edge-focused detection tasks, the additional depth produced by the final Bottleneck layer adds disproportionately to computational strain while providing only moderate benefits in feature discrimination. As a result, this redundant Bottleneck becomes the major option for structural pruning in order to reduce inference latency while maintaining model correctness.

2) Algorithmic Implementation: The pruning approach uses a three-stage *Identify-Remove-Repair* workflow on the PyTorch computation graph:

The pruning script searches the YOLOv8n model's `m.Module` hierarchy for all instances of the `C2f` class. Each C2f block has an internal attribute `m`, implemented as a `m.ModuleList`, that records the sequential Bottleneck layers. Blocks with a depth of $n > 1$ are identified as possibilities for structural reduction.

b) Step B: Removal by Surgery: For over-parameterized modules, the script removes the final Bottleneck layer using a Python slice operation on the `ModuleList`. In Figure 2(b), the crossed-out "REMOVED" block visually represents the pruning process. Eliminating this layer immediately reduces the block's FLOPs and parameter count, cutting computing costs during inference.

c) Step C: Reconstructing the Dynamic Graph: Removing a single bottleneck layer impacts the computational graph, as the last convolutional layer (`cv2`) expects a concatenated tensor composed of the Skip Path output and the outputs of all Bottleneck layers. When *Bottleneck 2* is eliminated, the concatenated feature tensor shrinks, resulting in a channel-dimension mismatch.

To restore graph integrity, the pruning algorithm executes a **Dynamic Reconnection** (shown by the curved arrow in Fig. 2(b)). The appropriate input dimensionality for the re-instantiated `cv2` layer is calculated as:

$$C_{\text{in,new}} = C_1 + (n_{\text{remaining}} \times C_{\text{out,bottleneck}}) \tag{1}$$

where C_1 denotes the channel count from the Skip Path and $n_{\text{remaining}}$ represents the number of Bottleneck layers that remain after pruning.

Following this calculation, the old `cv2` layer is deleted and reconstituted with the new input channel dimension. This "heals" the module and ensures that the forward-pass data flow stays valid throughout inference.

D. Accuracy Recovery via Fine-Tuning

The surgical removal of the layers severely impairs the models's ability to extract features and it lowers the mean Average Precision(mAP) to almost zero. Although Knowledge Distillation (KD), which involves a heavy "Teacher" model supervising the pruned "Student," is a popular technique for this stage, our empirical analysis showed that it is computationally unnecessary for this particular task. We found that 99.7% of the baseline accuracy could be recovered using a direct Fine-Tuning protocol. This suggests that the pruned architecture still has enough capacity to learn the features of the dataset without a teacher's soft-label guidance.

- **Training Protocol:** The pruned model is handled like a student model who has already received training. In order to preserve low-level feature extractors (such as edge and texture filters), we use Transfer Learning principles by freezing the backbone's initial layers. Using the Stochastic Gradient Descent (SGD) optimizer with a momentum of 0.937 and an initial learning rate of 0.01, the modified C2f modules and the detection head are unfrozen and retrained on the COCO 2017 dataset for 50 epochs.

E. Post-Training Quantization (PTQ)

To maximize inference throughput on the Raspberry Pi's CPU, the recovered model undergoes Post-Training Quantization. This process maps the model's parameters from high-precision 32-bit floating-point (FP32) representation to lower-precision 8-bit integers (INT8).

- **Quantization Mapping:**
  The weights (W) and activation maps (A) are quantized using an affine mapping defined by a scale factor S and a zero-point Z. The quantization function $Q(x)$ is defined as:
  $$Q(x) = \text{clamp}\Big(\text{round}\Big(\frac{x}{S}\Big) + Z,\ -128,\ 127\Big) \tag{2}$$
  where x represents the FP32 value. The scale factor S is calibrated by passing a representative dataset through the network to determine the dynamic range [min,max] of activations at each layer.
- **Hardware Impact:**
  The update reduces the model's memory footprint by 4. Furthermore, it allows the inference engine to use the Raspberry Pi's integer arithmetic units, which are much quicker and more energy-efficient than floating-point units, thereby immediately reducing memory bandwidth issues.

F. Edge Inference Implementation

The optimized.tflite model is deployed on the Raspberry Pi 5 using the TensorFlow Lite Runtime, which has the XNNPACK delegate enabled.XNNPACK guarantees that theoretical reductions in FLOPs from pruning and quantization directly translate into lower inference latency by utilizing optimized ARM64 NEON assembly for neural operators.

G. Justification for Pruning Strategy

Our 'surgical' approach focuses on architectural depth, in contrast to unstructured pruning, which frequently fails to accelerate standard CPUs despite high sparsity. We minimize sequential floating-point operations (FLOPs) by removing entire bottleneck layers. Because the speedup results from a shorter execution path rather than merely a smaller model, this explains how a modest 3.5% parameter reduction produces quantifiable FPS gains.

IV. System Architecture

In order to convert the computationally costly YOLOv8n model into an effective edge-ready format, the suggested system architecture functions as a sequential optimization pipeline. The following steps make up the process flow, which is depicted in Figure 2:

- **Input Data:** Raw visual data is accepted by the pipeline. This includes using the COCO 2017 dataset for training during the optimization stage.
- **Surgical Pruning:** The pruning engine receives the baseline model, introspects the architecture, and programmatically eliminates unnecessary bottleneck layers from the C2f modules in order to decrease model depth.
- **Fine-Tuning:** The retraining module receives the pruned model. It goes through 50 epochs of fine-tuning on the target dataset, which enables the remaining weights to adjust and regain accuracy.
- **Quantization:** Post-Training Quantization (PTQ) transforms the recovered floating-point (FP32) model into an 8-bit integer (INT8) format in order to optimize execution speed on the edge CPU.
- **Output:** The TFLite runtime with XNNPACK acceleration is used to deploy the final optimized model on the Raspberry Pi 5. Bounding boxes and class labels are among the real-time detection predictions that the system generates.

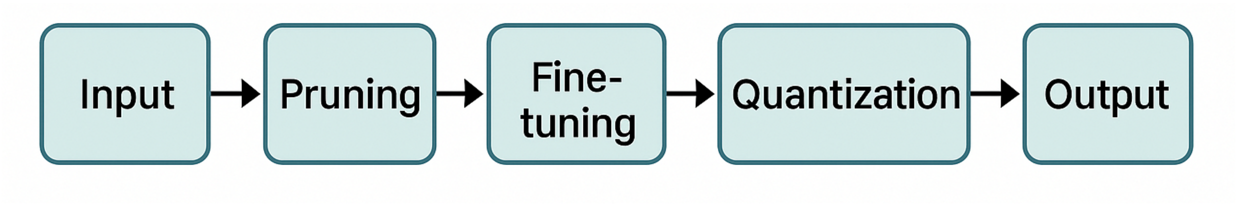

Fig. 2: Block diagram of the proposed End-to-End Compression and Deployment Pipeline.

V. Experimental Setup

To ensure reproducibility, all models were trained on the **COCO 2017** dataset.

A. Hardware & Protocols

Optimization tasks were performed on a workstation equipped with an Intel Core i7-12700H CPU and NVIDIA GeForce RTX 3060 GPU, utilizing PyTorch 2.0.1. Final benchmarking was conducted on a **Raspberry Pi 5** (4GB RAM, Broadcom BCM2712) running Raspberry Pi OS (64-bit). The pruned model was fine-tuned for 50 epochs (Batch=16, ImgSz=640).

B. Evaluation Metrics

Performance was evaluated using:

- **mAP@.50-.95:** Primary accuracy metric on COCO val2017.
- **FPS:** Average inference speed over 100 continuous frames.
- **Parameter Count:** Model complexity and size.

VI. Results and Discussion

This section provides an empirical evaluation of the suggested compression pipeline. Comparison between the resulting optimized model to the original YOLOv8n baseline on important parameters such as model size, detection accuracy, and inference speed on the target hardware is present.

A. Comparison with Standard Techniques

To demonstrate the efficacy of our hybrid pipeline, we compared it against standard optimization methods (Table I).

TABLE I: Comparison with Standard Optimization Methods

Method	mAP Drop	FPS Gain
Standard PTQ (INT8 only)	-0.2%	+25%
Unstructured Pruning (50%)	-1.5%	+0% (CPU)
Ours (Surgical + INT8)	**-0.3%**	**+34%**

Table I shows that INT8 quantization individually provides improvement i speed but misses architectural efficiency gain while unstructured pruning reduces the size but the inference on the Raspberry Pi is not acclerated.The hybrid approach of combining architectural reduction with hardware-aware quantization provides the highest FPS gain.

B. Ablation Study

To validate the contribution of each stage in our pipeline, we performed a step-by-step ablation study. Table II summarizes the effects of each optimization phase on model size, accuracy, and inference speed.

TABLE II: Ablation Study of the Compression Pipeline

Pipeline Stage	Params	mAP	FPS (Pi 5)
1. Baseline (FP32)	3.16M	0.355	3.08
2. Surgical Pruning Only	3.05M	0.004	-
3. Pruning + Fine-Tuning	3.05M	0.354	3.12
4. Full Pipeline (+INT8)	**3.05M**	**0.354**	**4.13**

Analysis:

- **Step 1 (Baseline)**: The uncompressed baseline model was used as a reference, it produced 3.08 FPS. This low speed demonstrated the necessity for optimization.
- **Step 2 (pruning):** Direct removal of the bottlenecks resulted in a major loss in accuracy (0.004 mAP), indicating that the removed layers contained learned features that needed to be retrieved.
- **Step 3 (Fine Tuning):** Retraining effectively restored 99.7% of the original accuracy, indicating that the trimmed architecture has the capacity to learn the task.
- **Step 4 (Quantization)**: The switch to INT8 resulted in the highest speed boost (32% over FP32) with no further accuracy loss.

C. Quantitative Metrics

The performance of the compression pipeline is given below in Table I.The Baseline YOLOv8n model has 3.16M parameters and it achived speed of 3.08 FPS on the Raspberry Pi 5.After the surgical pruning,fine tuning and INT8 quantization the optimized model contains 3.05M parameters.

TABLE III: Comparison of Baseline and Optimized YOLOv8n Models

Model Version	Parameters	mAP@.5-.95	FPS (Pi 5)
Baseline YOLOv8n	3.16M	0.355	3.08
Optimized Model	**3.05M**	**0.354**	**4.13**

Table III shows that the surgical pruning and rebuilding of the C2f modules produced a final model with 3.05M characteristics, a 3.5% decrease from the baseline of 3.16M. The Raspberry Pi 5's final speed is a very important result, which went from 3.08 FPS to 4.13 FPS, signifying a acceleration of 34%. This improvement in performance was attained with a minor accuracy loss of 0.3%, using 0.354 mAP@.5-.95 was the final model's score. These outcomes show a very effective trade-off, where a significant increase in computing efficiency is achieved with a limited effect on the detection skills of the model.

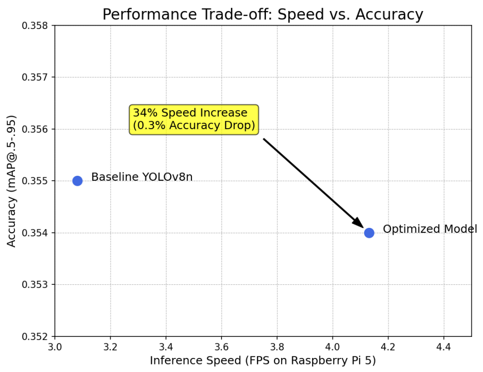

Fig. 3: Performance trade-off: Inference Speed vs. Accuracy

The relationship between inference speed and detection accuracy is plotted in Fig. 3. The vector in the plot indicates a clear shift towards higher efficiency (rightward movement) with minimal vertical drop, validating that the ”surgical” nature of the pruning successfully targeted redundant computations rather than critical feature extractors. Visual examination was done on the model's detection outputs to confirm that the quantitative metrics correspond to reliable real-world performance. A side-by-side comparison of detections on a sample image from the COCO validation set is shown in Figure 4.

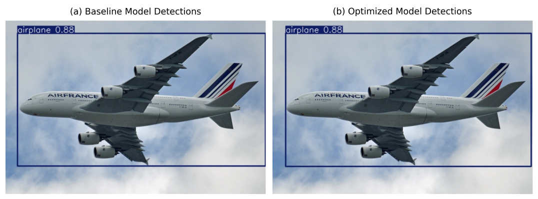

Fig. 4: Qualitative comparison of object detections. (a) Detections from the baseline YOLOv8n model. (b) Detections from the final optimized model.

The optimized model successfully and highly precisely detects the key object (airplane), as shown in Fig. 4. Interestingly, both the optimized model (b) and the baseline (a) have the same confidence score of 0.88. Additionally, there is no difference between the two in the bounding box localization. This qualitative data validates that the pruned layers were in fact architecturally redundant for this inference task and shows that the surgical removal of the C2f bottleneck layers did not affect the model's feature extraction capabilities.

VII. Conclusion

The proposed hybrid compression pipeline improved YOLOv8n inference speed on the Raspberry Pi 5 from 3.08 FPS to 4.13 FPS—a 34% gain—while maintaining accuracy with only a 0.3% drop (final mAP@.5–.95 of 0.354 on COCO). The tuning phase proved essential for recovering performance after pruning, showing that significant architectural reductions can be made without major accuracy loss.

This combination of INT8 quantization,surgical pruning delivers a strong framework that acts as a bridge between theoretical research and real-world embedded deployment. Future work may incorporate knowledge distillation to further enhance accuracy. The pipeline is could be used for embedded vision tasks in smart grid monitoring, substation surveillance, and autonomous inspection systems.

References

[1] J. Yin, P. Huang, D. Xiao, and B. Zhang, ”A lightweight rice pest detection algorithm using improved attention mechanism and YOLOv8,” *Agriculture*, vol. 14, no. 7, Art. no. 1052, 2024.
[2] J. Wang et al., ”Improved lightweight YOLOv8 model for rice disease detection in multi-scale scenarios,” *Agronomy*, vol. 15, no. 2, Art. no. 445, 2025.
[3] Y. Ding, C. Jiang, L. Song, F. Liu, and Y. Tao, ”RVDR-YOLOv8: A weed target detection model based on improved YOLOv8,” *Electronics*, vol. 13, no. 11, Art. no. 2182, 2024.
[4] Y. Li and S. Wang, ”EGM-YOLOv8: A lightweight ship detection model with efficient feature fusion and attention mechanisms,” *Journal of Marine Science and Engineering*, vol. 13, no. 4, Art. no. 757, 2025.
[5] S. Liu, F. Shao, W. Chu, J. Dai, and H. Zhang, ”An improved YOLOv8-based lightweight attention mechanism for cross-scale feature fusion,” *Remote Sensing*, vol. 17, no. 6, Art. no. 1044, 2025.
[6] B. Khalili and A. W. Smyth, ”SOD-YOLOv8: Enhancing YOLOv8 for small object detection in traffic scenes,” *arXiv preprint*, arXiv:2408.04786, 2024.
[7] J. P. C. Henriques et al., ”Development of a robotic cell using a YOLOv8 vision system embedded in a Raspberry Pi,” in *Proc. 2024 12th Int. Conf. Control, Mechatronics and Automation (ICCMA)*, 2024, pp. 171–176.
[8] B. A. Kumar et al., ”Real-time welding defect detection with YOLOv8 on Raspberry Pi,” in *Proc. 2025 5th Int. Conf. Trends Material Science and Inventive Materials (ICTMIM)*, 2025, pp. 766–771.
[9] S. U. Islam, G. Ferraioli, and V. Pascazio, ”Tomato leaf detection, segmentation, and extraction in real-time environment for accurate disease detection,” *AgriEngineering*, vol. 7, no. 4, Art. no. 120, 2025.
[10] Z. Cai et al., ”YOLOv8n-FAWL: Object Detection for Autonomous Driving Using YOLOv8 Network on Edge Devices,” IEEE Access, vol. 12, pp. 158377–158390, 2024.
[11] Y. Li et al., ”PKD-YOLOv8: A Collaborative Pruning and Knowledge Distillation Framework for Lightweight Rapeseed Pest Detection,” Sensors, vol. 25, no. 16, p. 5004, 2025.
[12] M. V. P. Kumar and R. Karthika, “Traffic Sign Detection and Recognition with Deep CNN Using Raspberry Pi 4 in Real-time,” in *Proc. 2023 11th Region 10 Humanitarian Technology Conference (R10-HTC)*, IEEE, pp. 1–6, 2023.
[13] Karthika, R., & Parameswaran, L. (2022). A novel convolutional neural network based architecture for object detection and recognition with an application to traffic sign recognition from road scenes. Pattern recognition and image analysis, 32(2), 351-362.
[14] Ravindran, S., Srikanth, S., Raagul, T. S., Madhankumar, R. M., & Ganesan, M. (2025, March). Depth and Dimension Estimation Using Computer Vision. In 2025 International Conference on Wireless Communications Signal Processing and Networking (WiSPNET) (pp. 1-7). IEEE.
[15] Behera, S., Bhardwaj, B., Rose, A., Hamdaan, M., & Ganesan, M. (2022). DriveSense: Adaptive System for Driving Behaviour Analysis and Ranking. In Machine Learning Techniques for Smart City Applications: Trends and Solutions (pp. 45-58). Cham: Springer International Publishing.