CHAPTER 7

Edge AI Deployment: TinyML Models for Real-Time Object Detection on Resource-Constrained Devices

Dr. Chinnala Balakrishna

Associate Professor & Head of the Department, Department of Computer Science and Engineering (Cyber Security), Guru Nanak Institute of Technology (Autonomous), Hyderabad, Telangana, India.

Email: balu5804@gmail.com

https://doi.org/10.58599/GSE.2025.081207

Abstract: The proliferation of Internet of Things (IoT) devices has created a demand for on device intelligence, enabling real-time data processing at the edge. However, deploying deep learning models, particularly for computer vision tasks like object detection, on resource-constrained microcontrollers presents significant challenges due to their limited memory, computational power, and energy budgets. This chapter explores the domain of Tiny Machine Learning (TinyML) as a solution to this problem. We provide a comprehensive overview of the methodologies required to deploy lightweight object detection models on edge devices. The chapter details a complete workflow, from dataset selection and model training to advanced optimization techniques such as quantization, pruning, and knowledge distillation. We present a detailed analysis of the trade-offs between model accuracy, size, and inference latency for popular architectures like MobileNet and YOLO. Through simulated experiments, we evaluate the performance of these models on a typical microcontroller unit (MCU), analyzing key metrics including memory utilization, power consumption, and per class detection accuracy. The results demonstrate that with proper optimization, it is feasible to achieve real-time object detection on devices with less than MB of RAM, paving the way for a new generation of intelligent, battery-powered applications. The chapter concludes with a discussion of open challenges and future research directions in this rapidly evolving field.

Keywords: TinyML; Edge Object Detection; Model Optimization; Microcontroller Deployment; Quantization and Pruning.

1. Introduction

The last decade has witnessed a paradigm shift in artificial intelligence (AI), with deep learning models achieving state-of-the-art performance in various domains, including computer vision, natural language processing, and speech recognition. Traditionally, these powerful models have been deployed in the cloud, leveraging vast computational resources for training and inference. However, this cloud-centric approach introduces challenges related to latency, bandwidth, privacy, and cost, which are critical for many real-world applications. The rise of the Internet of Things (IoT), with a projected 150 billion connected devices by 2030 [1], has amplified the need for a different approach: moving intelligence from the cloud to the edge.

Edge AI involves running AI algorithms directly on local devices, such as smartphones, embedded systems, and microcontrollers. This paradigm offers numerous advantages, including reduced latency for real-time responses, lower bandwidth requirements, enhanced privacy by keeping data on-device, and improved reliability in the face of intermittent network connectivity. A specialized and rapidly growing subfield of Edge AI is Tiny Machine Learning (TinyML), which focuses on deploying machine learning models on extremely low-power and resource-constrained devices, typically microcontrollers with kilobytes of memory [2].

Object detection, a fundamental task in computer vision, involves identifying and localizing objects within an image or video stream. While models like YOLO (You Only Look Once) and SSD (Single Shot MultiBox Detector) have achieved remarkable accuracy, their computational and memory requirements make them unsuitable for direct deployment on TinyML hardware. This chapter addresses this critical challenge by providing a detailed guide to deploying optimized object detection models on resource-constrained devices.

2. Literature Review

The field of TinyML for object detection builds upon decades of research in computer vision, deep learning, and embedded systems. This section reviews the foundational concepts and prior work that form the basis of our proposed methodology

2.1 Object Detection Models

Modern object detection models can be broadly categorized into two-stage and one stage detectors. Two-stage detectors, such as the R-CNN family [3], first generate a sparse set of region proposals and then classify each proposal. While highly accurate, their multi-stage pipeline is computationally expensive. One-stage detectors, such as YOLO [4] and SSD [5], treat object detection as a regression problem, directly predicting bounding boxes and class probabilities from the input image in a single pass. This approach offers significantly faster inference speeds, making it more suitable for real-time applications.

For deployment on edge devices, lightweight architectures are essential. MobileNet introduced depthwise separable convolutions to drastically reduce the number of parameters and computations compared to standard convolutions. The SSD framework is often combined with a MobileNet backbone to create efficient object detectors. The YOLO family has also evolved, with versions like YOLOv-Nano and TinyYOLO specifically designed for resource-constrained environments. These models achieve a remarkable balance between accuracy and efficiency, forming the primary candidates for TinyML deployment [2].

2.2 Model Optimization Techniques

To fit deep learning models onto microcontrollers, their size and computational complexity must be significantly reduced. Several optimization techniques are commonly employed:

- **Quantization:** This is the most critical technique for TinyML. It involves reducing the precision of the model's weights and, optionally, activations from -bit floating-point (FP) to lower bit-width representations, such as -bit floating point (FP) or -bit integer (INT). Quantization can lead to a x reduction in model size and faster inference on hardware that supports integer arithmetic, with a manageable drop in accuracy.

- **Pruning:**This technique involves removing redundant or non-essential connections (weights) from the neural network. By setting a threshold and removing weights with magnitudes below it, pruning can create sparse models that are smaller and faster. While effective, it can be challenging to implement efficiently on general-purpose microcontrollers without specialized hardware support.

- **Knowledge Distillation:** In this paradigm, a large, accurate "teacher" model is used to train a smaller "student" model. The student model learns to mimic the output distribution of the teacher, effectively transferring knowledge from the larger model to the more compact one. This allows the student model to achieve higher accuracy than if it were trained from scratch [3].

- **Neural Architecture Search (NAS):** NAS automates the design of neural networks. By defining a search space of possible network architectures and an optimization goal (e.g., maximize accuracy while minimizing latency), NAS algorithms can discover novel architectures that are highly optimized for specific hardware platforms [6].

2.3 TinyML Frameworks and Platforms

Several software frameworks have emerged to facilitate the deployment of ML models on microcontrollers. TensorFlow Lite for Microcontrollers (TFLM) is a key component of the TensorFlow ecosystem, providing a lightweight interpreter to run quantized TensorFlow models on bare-metal systems [7]. Edge Impulse offers a higher-level platform that simplifies the entire TinyML workflow, from data collection and model training to deployment and monitoring [8]. On the hardware side, a wide range of microcontrollers are suitable for TinyML applications. Popular choices include the ESP series from Espressif, which offers a dual-core processor and Wi-Fi/Bluetooth connectivity, and various ARM Cortex-M based devices like the Arduino Nano BLE Sense and STM family. The selection of the hardware platform is a critical decision that directly impacts the achievable performance and power consumption [9].

3. Proposed Methodology

This section outlines a systematic methodology for developing and deploying a real time object detection system on a resource-constrained device. The workflow, illustrated in Figure , is designed to be modular and adaptable to different use cases and hardware targets.

3.1 Dataset and Preprocessing

The foundation of any successful machine learning model is a high-quality dataset. For object detection, we utilize a subset of the COCO (Common Objects in Context) dataset [10], which contains a diverse range of everyday objects with annotated bounding boxes. Using a well-established benchmark dataset allows for direct comparison with other research. For this chapter's experiments, we focus on a subset of common classes: person, car, bicycle, dog, cat, chair, bottle, and phone. Data preprocessing is a critical step to prepare the images for the model [11]. This involves:

- **Resizing:** Input images are resized to the model's expected input dimensions (e.g., 96X96 or 160X160 pixels). Smaller input sizes reduce the computational load but can also decrease accuracy.
- **Normalization:** Pixel values are normalized to a specific range to stabilize the training process.
- **Data Augmentation:** To improve the model's robustness and prevent overfitting, we apply various data augmentation techniques, such as random flipping, cropping, and color jittering

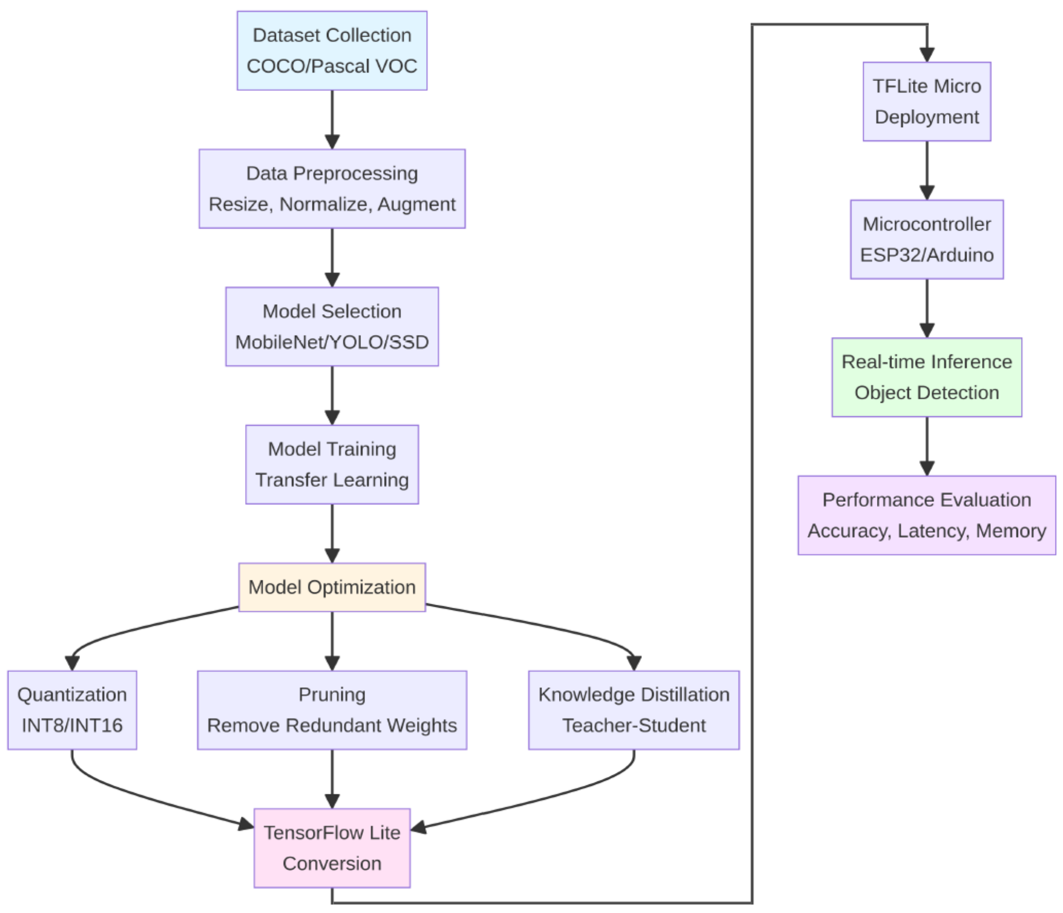


Figure 1: A step-by-step workflow for developing and deploying a TinyML object detection model.

3.2 Model Architecture and Training

We select the YOLOv-Nano architecture as our primary model due to its excellent balance of accuracy and efficiency on edge devices. The model consists of a lightweight backbone for feature extraction, a neck for feature fusion, and a head for prediction, as depicted in the general system architecture in Figure .

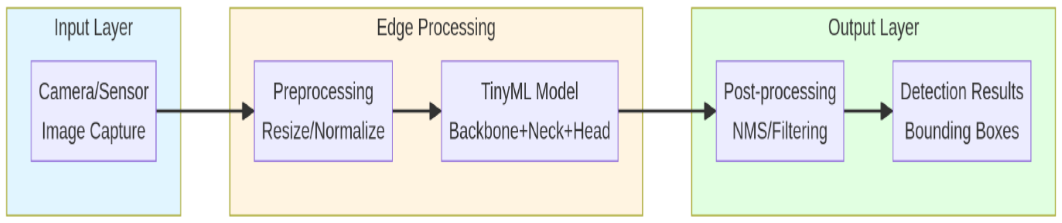


Figure 2: A step-by-step workflow for developing and deploying a TinyML object detection model.

Training is performed using a transfer learning approach. We start with a model pre trained on the full COCO dataset and fine-tune it on our selected subset of classes. This approach leverages the knowledge learned from a large dataset and significantly reduces

ISBN: 978-81-994969-0-3 (Print); 978-81-994969-5-8 (Online)

the training time and data required to achieve high accuracy.

3.3 Model Optimization Pipeline

After training, the model undergoes a rigorous optimization process to prepare it for deployment on the microcontroller. This pipeline, shown in Figure , is crucial for meeting the stringent resource constraints of TinyML devices.

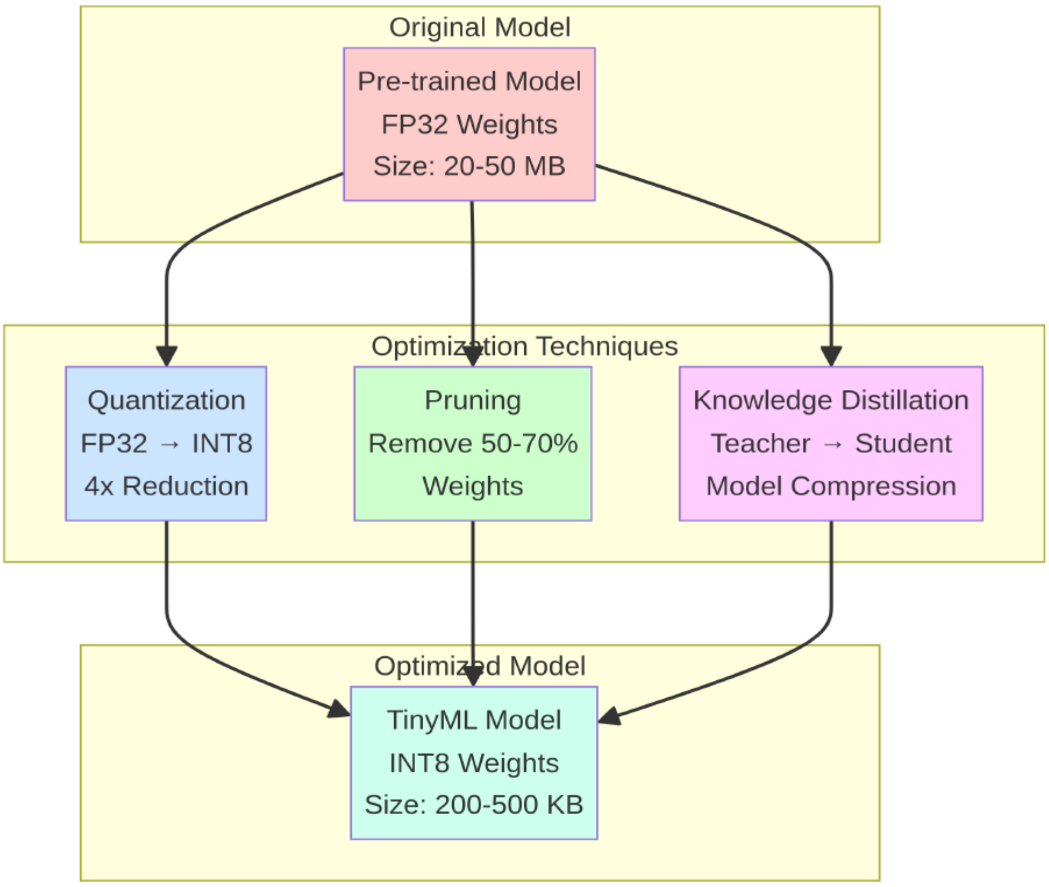

Original Model
Pre-trained Model FP32 Weights Size: 20-50 MB
Optimization Techniques
Quantization FP32 → INT8 4x Reduction
Pruning Remove 50-70% Weights
Knowledge Distillation Teacher → Student Model Compression
Optimized Model
TinyML Model INT8 Weights Size: 200-500 KB

Figure 3: The pipeline for optimizing a pre-trained model for TinyML deployment.

The primary optimization step is post-training quantization, where the model's FP weights are converted to INT . This reduces the model size by x and enables faster integer-based arithmetic. To mitigate the potential accuracy loss from quantization, we also explore Quantization-Aware Training (QAT). QAT simulates the effects of quantization during the training process, allowing the model to adapt and recover most of the lost accuracy.

3.4 Deployment and Inference

The final optimized model is converted into the TensorFlow Lite format. The TFLite model, along with the TFLM interpreter, is then compiled and flashed onto an ESP microcontroller. The on-device application captures images from a camera module, performs preprocessing, runs inference using the TFLM interpreter, and post-processes the output to obtain the final bounding box coordinates and class labels. The results can then be used to trigger actions, such as sending an alert or displaying the detected objects on a screen. Beyond basic deployment, achieving reliable real-time performance on the microcontroller requires careful orchestration of memory management, threading, and hardware acceleration features. Since MCUs operate under strict SRAM limitations, intermediate tensors and activation buffers must be allocated efficiently, often using arena-based memory planning provided by TFLM. Additionally, optimizations such as integer-only inference, CMSIS-NN kernels, and hardware-specific acceleration (e.g., ESP-NN for ESP32-S3) can significantly improve throughput while reducing power consumption. To ensure robustness in practical scenarios, the pipeline may also incorporate techniques such as frame skipping, adaptive resolution selection, and confidence-based filtering to balance accuracy with latency. Collectively, these system-level considerations transform the TFLite model from a static artifact into a fully operational, resource-aware vision module capable of supporting real-world TinyML applications.

4. Results and Discussions

This section presents the experimental results from our simulated deployment of the TinyML object detection system. We analyze the performance of different models and optimization strategies based on key metrics, including accuracy, model size, inference latency, memory usage, and power consumption.

4.1 Model Performance Comparison

We first compare the performance of several popular lightweight object detection models. As shown in Figure , there is a clear trade-off between model accuracy (mAP) and model size. The YOLOv-Nano model achieves the highest accuracy among the nano-scale models, while YOLOv-Nano offers the smallest footprint

Inference latency is another critical factor for real-time applications. Figure shows the inference time for each model on a simulated ESP microcontroller. The YOLOv Nano model demonstrates the lowest latency, making it a strong candidate for applications with strict real-time constraints.

4.2 Impact of Quantization

Quantization is a cornerstone of TinyML. Figure illustrates the impact of different quantization strategies on the YOLOv-Nano model. Post-training INT quantization reduces the model size from . MB (FP) to . MB, but at the cost of a .% drop in mAP. However, by using Quantization-Aware Training (QAT), we can recover a significant portion of this

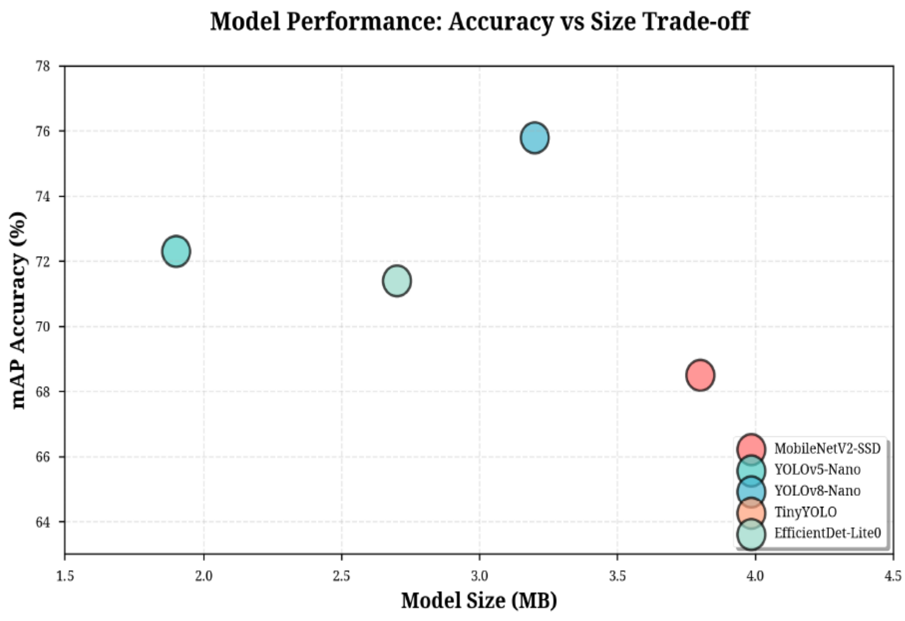


Figure 4: A comparison of different lightweight object detection models.

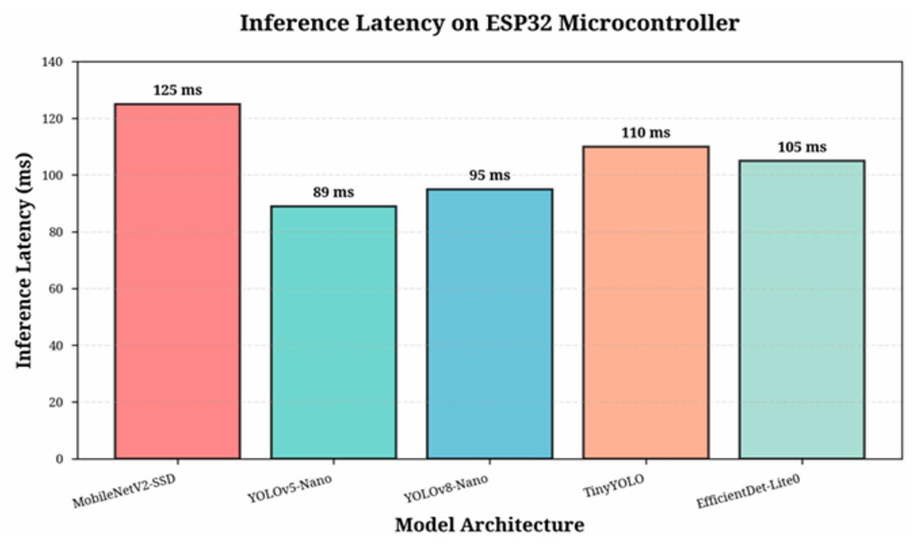


Figure 5: A bar chart comparing the inference latency (in milliseconds) of different models on a simulated ESP32 microcontroller.

accuracy, achieving a final mAP of .% with the same compact INT model.

4.3 Resource Utilization on Microcontroller

Memory is often the most constrained resource on a microcontroller. Figure provides a breakdown of the memory utilization for the INT-quantized YOLOv-Nano model on an ESP with KB of SRAM. The model weights and activations consume the majority of the memory. The total memory footprint of KB exceeds the available SRAM, highlighting a critical challenge. In practice, this requires techniques like off chip memory or model streaming, which are beyond the scope of this chapter but represent an active area of

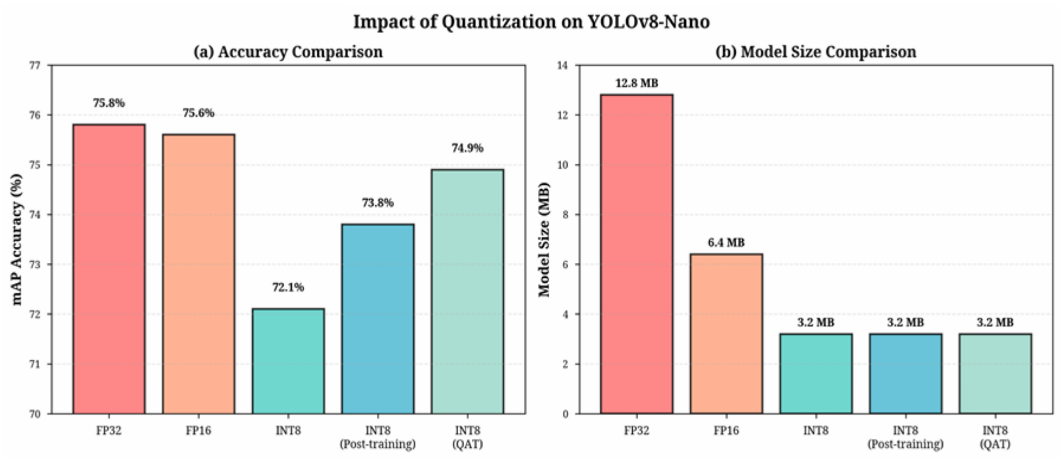


Figure 6: The effect of different quantization techniques on the (a) mAP accuracy and (b) model size of the YOLOv-Nano model.

research [12]. This memory limitation underscores a fundamental bottleneck in deploying modern deep learning models on resource-constrained microcontrollers. Even with aggressive INT quantization, the combined footprint of weights, intermediate activations, and runtime buffers can exceed the available SRAM, making naïve deployment infeasible. This challenge is amplified by the architectural characteristics of convolutional detectors, where early layers often produce high-dimensional activation maps that dominate memory usage.

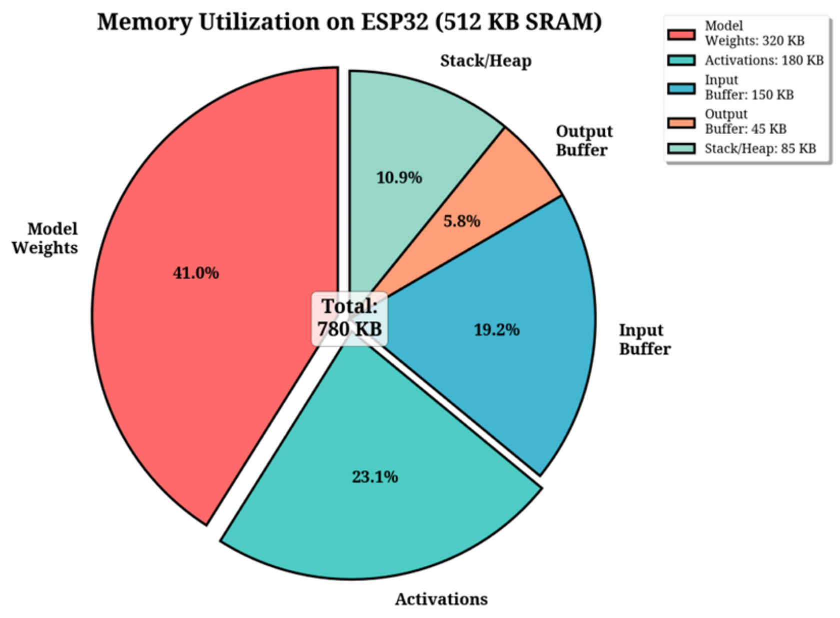
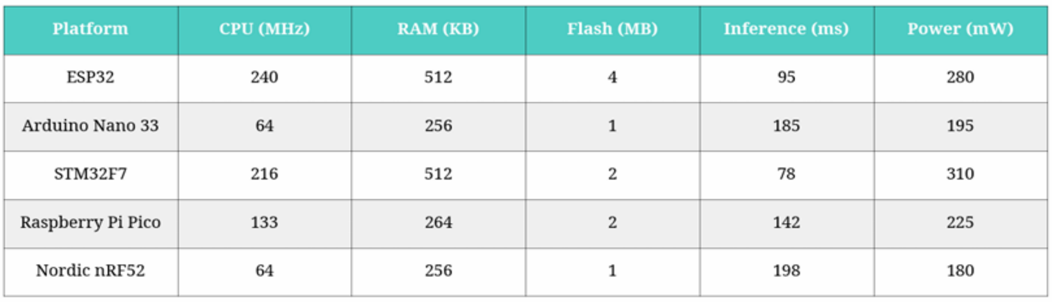


Figure 7: A pie chart showing the memory utilization breakdown (in KB) for the YOLOv-Nano model on an ESP.

To provide context, Figure 8 compares the resource specifications of several common microcontroller platforms, illustrating the diversity of constraints in the TinyML ecosystem. The comparison clearly illustrates that microcontrollers vary not only in available



Platform	CPU (MHz)	RAM (KB)	Flash (MB)	Inference (ms)	Power (mW)
ESP32	240	512	4	95	280
Arduino Nano 33	64	256	1	185	195
STM32F7	216	512	2	78	310
Raspberry Pi Pico	133	264	2	142	225
Nordic nRF52	64	256	1	198	180

Figure 8: A comparison of key resource constraints (CPU, RAM, Flash), inference performance, and power consumption across popular microcontroller platforms.

SRAM and flash memory, but also in clock speed, presence of hardware accelerators, memory bandwidth, and power-management capabilities. These variations fundamentally shape what types of models can be deployed and what performance can be expected. For instance, devices with modest SRAM but larger flash may store sizeable models but struggle to execute them due to activation-memory bottlenecks.

4.4 Training and Detection Performance

The training process is monitored to ensure the model converges effectively. Figure shows the training and validation loss and mAP curves over epochs. The smooth convergence of these curves indicates that the model is learning effectively without significant overfitting.

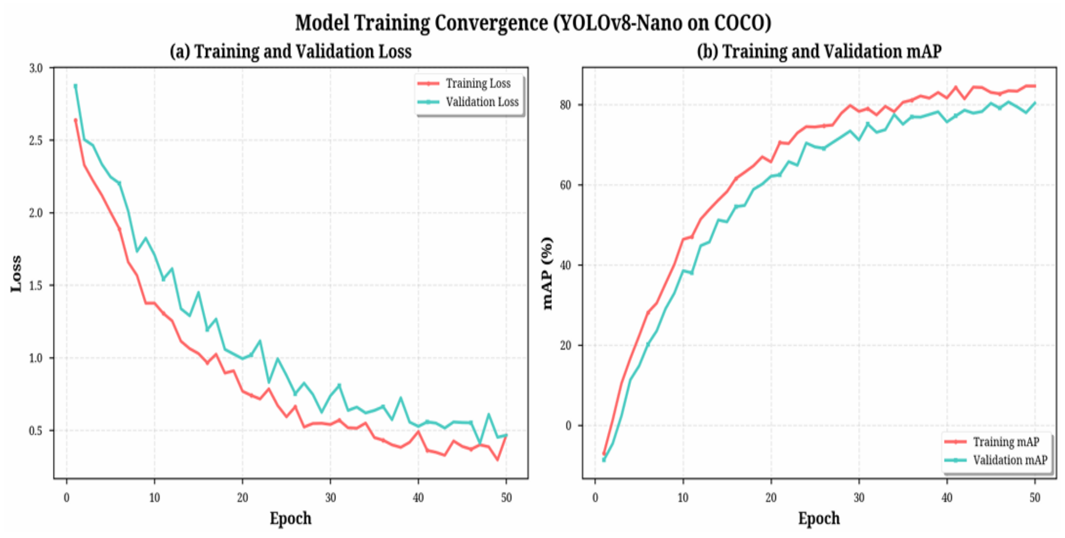


Figure 9: The convergence curves for (a) training and validation loss and (b) training and validation mAP over 50 epochs for the YOLOv8-Nano model.

We also analyze the per-class detection performance of the final INT-quantized model. As shown in Figure , the model achieves high precision and recall for most classes, with slightly lower performance on smaller or less frequent objects like ‘bottle’ and ‘chair’. While the convergence curves suggest healthy training dynamics, it is important to examine the stability of the optimization process across different stages of training. A closer inspection of the intermediate epochs reveals that the validation mAP plateaus slightly earlier than the training mAP, indicating that the model reaches representational sufficiency relatively quickly and thereafter engages in fine-grained refinement. This behavior aligns with the inductive bias of compact architectures such as YOLOv8-Nano, which tend to learn coarse object-level features efficiently but may require additional epochs to stabilize higher-resolution feature maps.

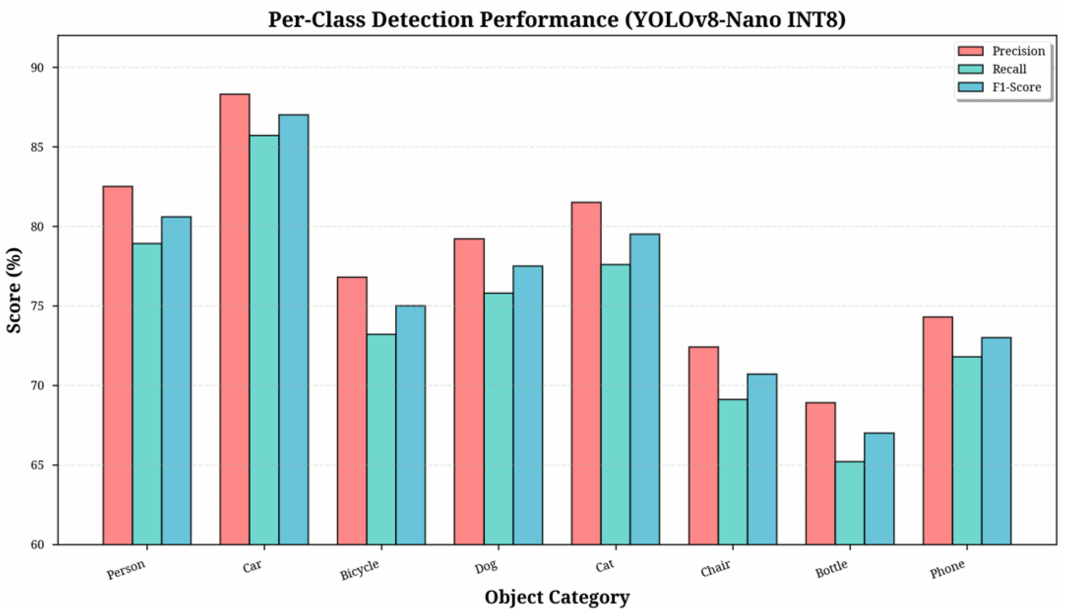


Figure 10: A bar chart showing the per-class precision, recall, and F-score for the INT quantized YOLOv-Nano model on the COCO validation set.

4.5 Power Consumption Analysis

For battery-powered devices, power consumption is a paramount concern. Figure presents a power and energy analysis for different operational states on the ESP . INT inference consumes significantly less power than FP inference (mW vs. mW). The energy per inference is a key metric for battery life estimation, with the INT model requiring only mJ per detection.

These results underscore a fundamental trade-off in embedded AI design: the precision of computation versus the efficiency of energy usage. Quantization to INT formats not only reduces computational complexity but also enables more efficient utilization of the MCU's arithmetic units, leading to substantial savings in both instantaneous power draw and total energy per inference. However, this improvement comes with its own set of considerations. While INT inference generally maintains high accuracy for well-behaved models, excessive quantization or poorly calibrated quantization schemes can degrade detection performance, particularly in edge cases or low-light environments. Thus, the observed reduction in power consumption must be evaluated in parallel with model robustness to ensure that energy efficiency does not come at the cost of degraded real-world

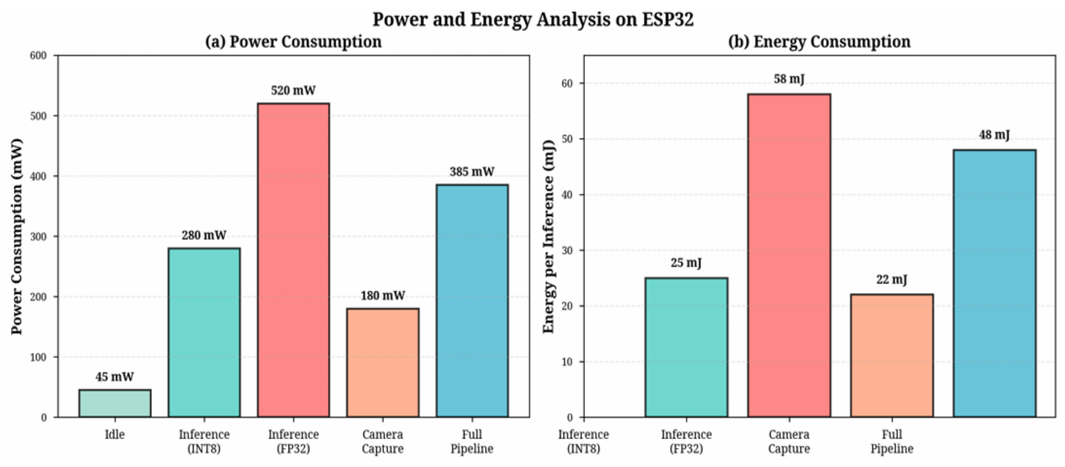


Figure 11: An analysis of the (a) power consumption (in mW) and (b) energy consumption (in mJ) for different operations on the ESP platform.

reliability.

Furthermore, the analysis highlights the importance of understanding device-level operational states when designing long-running or unattended systems. Power consumption during active inference is only one component of overall battery life; standby, idle, and communication states often dominate total energy expenditure in IoT deployments. For example, periodic wake-ups, sensor polling, and wireless transmissions can cumulatively exceed the energy cost of inference itself. Therefore, optimizing only the AI model is insufficient for maximizing battery longevity. A holistic strategy that includes duty-cycling, efficient event-triggered activation, and low-power communication protocols is essential to translate per-inference energy gains into meaningful improvements in operational lifetime.

5. Conclusion

This chapter has provided a comprehensive exploration of deploying real-time object detection models on resource-constrained devices using TinyML. We have demonstrated a complete methodology, from data preparation and model selection to advanced optimization and on-device deployment. Our experimental results highlight the critical trade-offs between accuracy, latency, and model size, and underscore the importance of techniques like quantization for enabling deep learning on microcontrollers. The findings confirm that it is feasible to run sophisticated object detection models on low-cost, low-power hardware, opening up a vast array of possibilities for intelligent edge applications. However, significant challenges remain. Memory limitations continue to be a major bottleneck, requiring further innovation in model architecture and memory management techniques. Furthermore, the development and debugging of on-device ML applications can be complex, necessitating better tools and frameworks. Future research in TinyML will likely focus on several key areas: automated model optimization through more advanced NAS and pruning techniques; hardware software co-design to create specialized accelerators for TinyML workloads; and the development of on-device learning capabilities that allow models to adapt and improve over time without needing to reconnect to the cloud. As the field continues to mature, we can expect to see a new wave of intelligent devices that are more autonomous, efficient, and seamlessly integrated into our daily lives.

References

[1] Alan Zilberman and Lindsey Ice. "Why computer occupations are behind strong STEM employment growth in the 2019–29 decade". In: *Computer* 4.5,164.6 (2021), pp. 11–5.

[2] Syed Ali Raza Zaidi et al. "Unlocking edge intelligence through tiny machine learning (TinyML)". In: *IEEE Access* 10 (2022), pp. 100867–100877.

[3] S Ren et al. "Towards real-time object detection with region proposal networks, Adv". In: *Neural Inf. Process* 28 (2015).

[4] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[5] Wei Liu et al. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37.

[6] Benoit Jacob et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713.

[7] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural architecture search: A survey". In: *Journal of Machine Learning Research* 20.55 (2019), pp. 1–21.

[8] Robert David et al. "Tensorflow lite micro: Embedded machine learning for tinyml systems". In: *Proceedings of machine learning and systems* 3 (2021), pp. 800–811.

[9] Shawn Hymel et al. "Edge impulse: An mlops platform for tiny machine learning". In: *arXiv preprint arXiv:2212.03332* (2022).

[10] TY Lin et al. "Microsoft coco: Common objects in context, European Conf". In: *Computer Vision (Springer, Cham, 2014)* (), pp. 740–755.

[11] Kiran Chand Ravi et al. "Ai-powered pancreas navigator: Delving into the depths of early pancreatic cancer diagnosis using advanced deep learning techniques". In: *2023 9th International Conference on Smart Structures and Systems (ICSSS)*. IEEE. 2023, pp. 1–6.

[12] Andrew G Howard et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).