**Journal of Optimization in Soft Computing (JOSC)**

Vol. 2, Issue 4, pp: (28-39), Winter-2024
Journal homepage: https://sanad.iau.ir/journal/josc

**Paper Type (Research paper)**

# Edge-based Object Detection using Optimized Tiny YOLO on Embedded Systems

Peyman Babaei

*Department of Computer Engineering, West Tehran Branch, Islamic Azad University, Tehran, Iran.*

**Article Info**

**Article History:**
*Received: 2024/12/29*
*Revised: 2025/01/30*
*Accepted: 2025/03/05*

*DOI:*

**Keywords:**
Tiny YOLO, Model optimization, Model Deployment, Quantization, Pruning, Weight Clustering, Embedded Systems.

**Corresponding Author's Email Address: Peyman.Babaei@IAU.ac.ir*

**Abstract**

Object detection at the edge has gained considerable attention for enabling real-time, low-latency, and privacy-preserving solutions by processing data locally on resource-constrained devices. This paper explores using Tiny YOLO, a lightweight variant of the YOLO architecture, for object detection on embedded systems. Tiny YOLO is specifically designed for edge devices to run efficiently on constrained devices by utilizing a reduced architecture with fewer parameters while maintaining good performance for real-time object detection. The study examines the deployment of optimized Tiny YOLO models on embedded systems, incorporating techniques like quantization, pruning, and clustering to reduce model size, enhance speed, and lower power consumption. Optimization methods show significant improvements, with quantization speeding up inference, pruning eliminating redundant parameters, and clustering enhancing accuracy. Specifically, the study compares the performance of Tiny YOLO under these optimization techniques, presenting results for both Pascal VOC and COCO datasets. The results demonstrate that optimized Tiny YOLO models are effective for real-time object detection on microcontrollers. These methods enable the efficient deployment of deep learning models for edge computing, without relying on cloud infrastructure.

## 1. Introduction

In recent years, edge computing has emerged as a key technology for processing data closer to where it is generated, offering distinct advantages over traditional cloud-based computing. At its core, edge computing allows devices to process and analyze data locally rather than sending it to centralized servers or the cloud [1]. This localized processing significantly reduces latency, decreases reliance on network bandwidth, improves privacy, and increases overall system efficiency, making it particularly valuable for real-time applications such as image classification and object detection.

Embedded systems, which are small, low-cost, and energy-efficient computing units, are a key enabler of edge computing and Internet of Things devices. They are commonly used in applications where space and power consumption are constrained, such as in smart home devices, wearable electronics, and industrial sensors. However, microcontrollers are typically limited in terms of computational power, memory, and storage, making it challenging to run complex machine learning models [2,3].

Traditional deep learning models require substantial computational resources, especially in terms of processing power and memory, which makes it difficult to deploy them on embedded systems. However, recent advancements in model optimization techniques, such as quantization, pruning, and the use of lightweight neural network architectures (e.g., Tiny YOLO), have made it possible to deploy deep learning-based object detection models even on microcontrollers. These optimization techniques help reduce the size of the models, increase their inference speed, and reduce power consumption, all while maintaining acceptable levels of accuracy.

Deploying deep learning models on embedded systems is a key step in bringing artificial intelligence to the edge, where real-time decision-making is critical [4,5]. While challenges such as limited computational power, memory, and energy resources remain, advancements in model optimization techniques, lightweight architectures, and specialized hardware accelerators are making AI deployment on small devices more feasible [6,7]. For example, Tiny YOLO, a compact version of the well-known YOLO (You Only Look Once) object detection model, has proven to be effective for edge deployment due to its small size and efficient performance. This is especially valuable in applications such as autonomous systems, security surveillance, and robotics, where real-time object detection is needed on resource-constrained devices. One of the key hurdles in deploying deep learning on embedded systems is ensuring that these models can operate efficiently while maintaining a balance between performance and resource consumption [8-10].

Model optimization methods like quantization, pruning, and clustering help in reducing the memory footprint, lowering computation requirements, and speeding up inference times, making these models more suitable for edge devices like ESP32 [11]. Tools such as TensorFlow Lite provide frameworks that make it easier to run AI models on these constrained platforms, optimizing them further for mobile and embedded applications [12].

The rise of AI-powered microcontrollers is transforming industries by enabling smarter, decentralized systems [13,14]. In smart homes, microcontrollers are being used for voice recognition in virtual assistants and object detection in security cameras. In healthcare, wearable devices equipped with AI can monitor vital signs and detect falls in real-time. In industrial IoT, microcontrollers power predictive maintenance systems that can analyze sensor data like vibration and temperature to prevent equipment failure. Additionally, environmental monitoring using microcontrollers allows for the processing of data to predict weather patterns, track pollution levels, and monitor wildlife. The agricultural sector benefits from AI-enabled microcontrollers by enabling crop health monitoring, soil condition analysis, and pest detection, ultimately advancing precision farming techniques [15,16]. These examples underscore the versatility of microcontroller-based AI, showcasing its potential to enhance various domains by making intelligent decisions at the edge [17,18].

This study conducted aimed to evaluate the performance of the Tiny YOLO model on various edge devices, including ESP32, ESP32-S3, Pico W, and Jetson Nano, across different optimization techniques such as quantization, weight pruning, and clustering. The experiment utilized the COCO [19] and Pascal VOC [20] datasets to assess the model's mean Average Precision (mAP), frames per second (FPS), model size, inference time. Results showed that while ESP32 and Pico W exhibited significant limitations in accuracy and real-time performance due to their limited computational power, applying optimizations did provide some improvements in terms of model size and inference speed. In contrast, Jetson Nano demonstrated superior performance, achieving high mAP values and fast inference times, even with optimized models. This highlighted the importance of hardware capabilities in achieving real-time object detection, with Jetson Nano proving to be the most suitable platform for running optimized models like Tiny YOLO efficiently on more complex datasets.

In the following, the Edge-based object detection is presented in section 2, the YOLO and Tiny YOLO architectures are presented in sections 3 and 4. The optimization techniques of learning models are presented in section 5, which also refers to the proposed approach. In section 6, the implementation of different scenarios of Tiny YOLO model optimization are presented, and then in section 7, the results of evaluation are compared. Finally, the conclusion is presented in section 8.

## 2. Edge-based object detection

Deploying object detection models on embedded systems for edge computing is a promising solution for a wide range of real-time applications. As optimization techniques improve, the ability to run sophisticated object detection algorithms on embedded systems will continue to advance, opening up new possibilities in fields such as healthcare, security, autonomous systems, and environmental monitoring. The ability to perform local image processing without relying on cloud infrastructure is transforming industries and enabling more intelligent, responsive, and energy-efficient systems.

This breakthrough allows for real-time object detection on devices with limited resources. The ability to process images and classify objects at the edge, without the need for cloud computing, opens up a wide range of possibilities for various

applications [21-23]. Below are some key use cases where microcontroller-based image processing is particularly beneficial:

- **Smart Home Automation:**
  - **Object Detection:** Embedded systems can be used to deploy object detection models to detect objects, faces, or gestures in smart home environments. For example, a security camera system could use a microcontroller to classify objects in real-time, identifying potential intruders or monitoring for specific actions.
  - **Gesture Recognition:** In a smart home, gesture recognition can be used to control lighting or appliances with simple hand movements, all processed on an embedded system.
- **Healthcare and Medical Devices:**
  - **Medical Imaging:** Embedded systems can assist in analyzing medical images such as X-rays, CT scans, or skin lesions directly on medical devices, facilitating faster diagnosis and reducing the need for data transmission to the cloud.
  - **Wearable Health Devices:** Image classification models deployed on wearable devices can monitor the health of individuals by identifying changes in skin tone, detecting the presence of medical conditions, or tracking movement patterns for rehabilitation purposes.
- **Industrial Automation and Monitoring:**
  - **Defect Detection in Manufacturing:** Embedded systems with object detection capabilities can be used in automated inspection systems to identify defects in products on an assembly line, improving quality control and reducing human error.
  - **Predictive Maintenance:** By analyzing visual data from sensors, embedded systems can help detect signs of wear or malfunction in machinery, enabling predictive maintenance and preventing downtime.
- **Autonomous Systems:**
  - **Robotics:** Autonomous robots, drones, and vehicles can leverage image classification at the edge to understand and interpret their environment, recognizing obstacles, people, or objects in real-time for navigation and decision-making.
  - **Agriculture and Environmental Monitoring:** Drones equipped with embedded systems can analyze images of crops or forests to monitor plant health, detect diseases, and evaluate environmental conditions without needing cloud-based processing.
- **Smart Cities and Surveillance:**
  - **Public Safety and Security:** Microcontrollers embedded in surveillance cameras can perform face recognition or detect unusual behaviors, enabling automated security systems that operate in real-time without relying on cloud servers.
  - **Traffic Monitoring:** Embedded systems can be used in traffic cameras to analyze road conditions, detect traffic congestion, or recognize vehicle types, all processed locally for faster decision-making.
- **Environmental Monitoring:**
  - **Wildlife Monitoring:** Edge devices equipped with embedded systems can monitor wildlife, detecting and identifying animals in remote areas through camera traps, without needing to transmit large image files to the cloud.
  - **Pollution Detection:** Image classification models can help detect pollution or other environmental hazards through cameras, enabling automated monitoring systems for air, water, or land quality.
- **Retail and Consumer Interaction:**
  - **Product Recognition:** Embedded systems can be used in point-of-sale systems or vending machines to recognize products through image classification, enabling automatic stock tracking or facilitating seamless customer interactions.
  - **Customer Behavior Analysis:** In retail settings, embedded systems can process visual data from in-store cameras to track customer behavior, optimize store layouts, or improve marketing strategies based on customer interaction patterns.

## 3. YOLO architectures

YOLO (You Only Look Once) is a popular series of deep learning models for object detection. It's known for its speed and efficiency, making it a best choice for real-time object detection tasks. Over the years, different versions of YOLO have been released, each with improvements in accuracy, speed, and architecture [24,25]. The summary of YOLO's evolution is shown in table 1. Below is an overview of the main versions and their key features:

- **YOLOv1**, introduced the idea of using a single convolutional neural network to predict bounding boxes and class probabilities in one pass, making it incredibly fast for real-time detection.
  - **Architecture:** A single convolutional neural network that simultaneously predicts bounding boxes and class probabilities for all objects in the image in one evaluation. The network

divides the image into a grid and for each grid cell, it predicts:
  - Bounding boxes (x, y, width, height)
  - Confidence score (how likely the box contains an object)
  - Class probabilities (which object class the box belongs to).
- **Strengths:** Very fast (real-time detection), unified approach (object localization and classification in one pass).
- **Weaknesses:** Struggles with detecting small objects and handling overlapping objects, less accurate in comparison to other models like Faster R-CNN.

❖ **YOLOv2**, released in 2017, brought significant improvements such as the introduction of anchor boxes, batch normalization, and multi-scale training, which increased both speed and accuracy, especially for larger objects.
- **Architecture:**
  - Introduced improvements like a new backbone network, Darknet-19, which was more powerful than YOLOv1's architecture.
  - Added anchor boxes for better bounding box prediction, addressing the issue of poor localization seen in YOLOv1.
  - Used multi-scale training, where the model was trained on different image sizes to improve generalization.
  - Introduced batch normalization to stabilize and speed up training.
- **Strengths:** Faster and more accurate than YOLOv1, improved handling of different object scales, better generalization, and more robust performance.
- **Weaknesses:** Still struggles with small object detection.

❖ **YOLOv3**, released in 2018, the model was further enhanced with a new backbone (Darknet-53), multi-label classification, and the use of three different scales for prediction, allowing it to better detect small objects. Despite these improvements, YOLOv3 still had limitations when compared to more complex models like Faster R-CNN.
- **Architecture:**
  - YOLOv3 used a new backbone called Darknet-53, which improved accuracy and allowed for better feature extraction.
  - Used multi-label classification to improve the detection of objects with more than one class.
  - Introduced three different scales for prediction (small, medium, and large), allowing the network to detect objects at various sizes.
  - Introduced Residual Connections to help deeper networks train better and avoid vanishing gradients.
  - The output layer was redesigned to use logistic regression for bounding box prediction.
- **Strengths:** Better detection of smaller objects, significant performance improvement over v2 in terms of both speed and accuracy.
- **Weaknesses:** Still not as accurate as more complex architectures like Faster R-CNN for certain tasks, especially in cases of very dense or small objects.

❖ **YOLOv4** released in 2020, focused on improving detection performance with a new backbone (CSPDarknet53) and techniques like Mosaic data augmentation and self-adversarial training, leading to better accuracy, especially for small and dense objects, while maintaining fast inference times.
- **Architecture:**
  - Built on the YOLOv3 model but incorporated several new techniques for better performance, including:
    - CSPDarknet53 as the backbone network, which balances accuracy and speed.
    - Mosaic Data Augmentation to improve generalization by combining multiple images during training.
    - Self-adversarial training for improved robustness.
    - DropBlock regularization for better bounding box predictions.
  - Improved performance on smaller objects with better feature pyramids.
- **Strengths:** Higher accuracy than YOLOv3, better at handling small and dense objects, faster inference times, state-of-the-art performance in real-time detection.
- **Weaknesses:** Larger model size compared to earlier versions, requiring more computational resources.

❖ **YOLOv5**, which was not developed by the original YOLO creators but became very popular due to its ease of use, modular design, and efficient performance on a range of hardware.
- **Architecture:**
  - YOLOv5 is a separate project developed by Ultralytics, which is not an official continuation of the YOLO series but has become very popular in the community.

- It focuses on speed and ease of use, and its codebase is built in PyTorch (as opposed to Darknet for the official YOLO models).
- YOLOv5 uses a modular design with different model sizes (small, medium, large, extra-large) to balance speed and accuracy.

- **Strengths:** Very easy to use, with a lot of built-in features like model training, testing, and deployment. Achieves state-of-the-art performance with relatively lightweight models.
- **Weaknesses:** It is not an official release from the original YOLO authors, so it may differ in implementation or long-term support compared to the official YOLO versions.

❖ **YOLOv6**, released in 2022, continued the trend of optimization, especially for edge devices, by focusing on speed and efficiency.

- **Architecture:**
  - YOLOv6 is optimized for both speed and accuracy with improvements over YOLOv5, particularly in handling dense and small objects.
  - Introduced a more efficient backbone (CSPResNet) and neck (PP-YOLO) to enhance detection performance.
  - Focused on optimizing inference speed for deployment on edge devices.
- **Strengths:** Real-time performance, better accuracy with fewer resources.
- **Weaknesses:** Like YOLOv5, it's not an official version, so community-driven development may lead to less consistency over time.

❖ **YOLOv7**, also released in 2022, utilized more advanced techniques such as efficient transformers and heterogeneous module fusion, further enhancing both speed and accuracy.

- **Architecture:**
  - YOLOv7 continues improving on YOLOv5 and YOLOv6, focusing on both accuracy and inference speed. It utilizes the efficient transformer architecture for better handling of spatial relationships in images.
  - Improved backbone for better feature extraction and information flow.
  - Introduced Heterogeneous Module fusion for better performance in terms of both accuracy and speed.
- **Strengths:** One of the fastest YOLO versions to date, highly optimized for real-time object detection.
- **Weaknesses:** Complexity in tuning for specific tasks, requires careful hyperparameter tuning for optimal performance.

❖ **YOLOv8**, introduced in 2023, offers cutting-edge performance with improvements in backbone architectures, better handling of various object detection tasks, and optimization for real-time and embedded systems.

- **Architecture:**
  - YOLOv8 aims to offer even better accuracy, speed, and efficiency than its predecessors. It is designed to perform well on various object detection tasks and includes newer backbone and neck architectures, as well as better loss functions for bounding box predictions.
  - It also focuses on fine-tuning for specific tasks like segmentation and key point detection.
- **Strengths:** Cutting-edge performance, high accuracy, and optimized for both real-time and edge devices.
- **Weaknesses:** Requires more computational resources than earlier versions but offers a significant boost in performance.

**Table 1: Summary of YOLO's evolution.**

| Version | Key Features |
|---|---|
| **YOLOv1** | First release; groundbreaking for real-time object detection using a single CNN for bounding box and classification predictions. |
| **YOLOv2** | Improved accuracy and speed; introduced anchor boxes, batch normalization, and multi-scale training. Better at handling larger objects. |
| **YOLOv3** | Significant improvements in architecture with Darknet-53 backbone; better at detecting small objects with multi-scale predictions and multi-label classification. |
| **YOLOv4** | Focused on speed, accuracy, and robustness, especially for real-time applications; introduced Mosaic data augmentation and CSPDarknet53. |
| **YOLOv5** | A community-driven model; emphasizes ease of use, modular design, and optimized for both speed and accuracy, with multiple model sizes. |
| **YOLO v6 & v7** | Optimized for edge devices and real-time applications; further enhancements in speed, accuracy, and performance, especially in dense or small object detection. |
| **YOLOv8** | The latest version with cutting-edge performance and optimizations for real-time and embedded devices; handles various detection tasks. |

The YOLO family continues to evolve with a stronger emphasis on speed, accuracy, and resource efficiency, making it a top choice for real-time object detection in areas like autonomous driving, surveillance and robotics. Each version of YOLO has brought improvements in terms of accuracy, speed, and efficiency, making it one of the top choices for real-time object detection in

fields such as autonomous driving, robotics, and surveillance.

**4. Tiny YOLO**

Tiny YOLO is a smaller, lighter version of the YOLO model, specifically designed for applications where computational resources are limited, such as on edge devices or in real-time systems that require fast processing speeds. It is a trade-off between performance and efficiency, sacrificing some accuracy for the sake of reduced size and faster inference time. Tiny YOLO simplifies the architecture of the original YOLO by reducing the number of layers and parameters. For example, in Tiny YOLO, the backbone network (typically Darknet) has fewer convolutional layers and a smaller number of filters. This results in faster processing speeds and reduced memory requirements, making it suitable for devices with limited computational power, such as embedded systems, mobile devices, and IoT applications.

**Faster Inference**: Tiny YOLO is much faster than the standard YOLO models due to its smaller size and fewer parameters. This makes it ideal for real-time object detection applications, especially on resource-constrained devices.

**Lower Computational Requirements**: The reduced architecture allows Tiny YOLO to run efficiently on devices with limited GPU or CPU capabilities. It's particularly useful for edge devices, mobile phones, and embedded systems where processing power is a concern.

**Smaller Model Size**: The smaller model size makes it easier to deploy Tiny YOLO on devices with limited storage capacity. This is important for applications where storage space is constrained, such as drones or IoT devices.

**Good for Low-Latency Applications**: Because of its faster processing, Tiny YOLO is suited for low-latency tasks where quick decision-making is necessary, such as autonomous vehicles or real-time video surveillance.

**Lower Accuracy**: Because of the simplified architecture, Tiny YOLO generally achieves lower accuracy compared to full YOLO versions (like YOLOv3, YOLOv4, or YOLOv5). It may struggle with detecting small objects or complex scenes with a high degree of clutter.

**Limited Detection Capabilities**: While Tiny YOLO is good for general object detection, its performance can degrade in challenging scenarios, such as detecting objects in high-density environments or cases where fine-grained classification is required.

**Less Robust in Difficult Conditions**: Tiny YOLO might not perform as well under varying conditions, such as different lighting, weather, or occlusion, compared to more complex models.

Tiny YOLO is a powerful tool when you need object detection on devices with limited resources, where speed and efficiency are more critical than achieving the highest possible accuracy. Its trade-off between performance and resource usage makes it suitable for real-time applications like autonomous vehicles, drones, and mobile devices.

Key Characteristics of Tiny YOLO's Architecture are:

- **Fewer layers and filters:** The network has fewer layers and smaller filter sizes compared to the full YOLO versions, making it faster but less accurate.
- **Simplified structure:** By reducing the depth of the network and the number of neurons in the fully connected layers, Tiny YOLO is optimized for speed and smaller model size.
- **Max Pooling:** Max pooling layers help reduce the spatial resolution of feature maps, aiding in faster processing and reducing overfitting by discarding irrelevant details.
- **Lower resolution input:** Tiny YOLO generally works with lower resolution input images, which reduces computation time but may decrease accuracy in detecting small objects.

Tiny YOLO sacrifices some complexity and accuracy from the standard YOLO architecture in exchange for faster processing and reduced computational requirements. This makes it suitable for real-time applications on edge devices and embedded systems, where speed and low resource consumption are prioritized over the highest possible accuracy. The Tiny YOLO architecture table is shown in table 2. The layers of this architecture are described below:

**Input Layer:** Takes images of size 224x224x3, commonly used for image classification and detection tasks.

**Convolutional Layers**: These layers progressively extract more abstract features from the image by applying convolution with 3x3 filters. The number of filters increases as the network deepens, allowing for more complex representations.

**Max Pooling Layers**: Reduce the spatial dimensions of the feature maps, making the model more efficient and helping to avoid overfitting.

**Fully Connected Layers**: Compress the features extracted from the convolutional layers and map them to a higher-dimensional space, enabling the prediction of object classes and bounding boxes.

**Output Layer**: Predicts both the class probabilities and bounding box positions (class + 4 for bounding box coordinates). The final output is structured to handle N classes and the corresponding bounding box for each object detected.

**Table 2: Tiny YOLO architecture.**

| Layer | Number of Filters | Filter Dimensions | Output Dimensions |
|---|---|---|---|
| Input Layer | --- | 224x224 | 224x224x3 |
| Convolutional 1 | 16 | 3x3 | 224x224x16 |
| MaxPooling 1 | --- | 2x2 | 112x112x16 |
| Convolutional 2 | 32 | 3x3 | 112x112x32 |
| MaxPooling 2 | --- | 2x2 | 56x56x32 |
| Convolutional 3 | 64 | 3x3 | 56x56x64 |
| MaxPooling 3 | --- | 2x2 | 28x28x64 |
| Convolutional 4 | 128 | 3x3 | 28x28x128 |
| MaxPooling 4 | --- | 2x2 | 14x14x128 |
| Convolutional 5 | 256 | 3x3 | 14x14x256 |
| MaxPooling 5 | --- | 2x2 | 7x7x256 |
| Fully Connected 1 | 4096 | N/A | 1x1x4096 |
| Fully Connected 2 | Classes + 4 | N/A | 1x1x(N+4) |
| Output | N/A | N/A | 1x1x(N+4) |

This structure is a simplified version of the YOLO architecture, designed for efficient image classification and object detection with reduced computational resources.

## 5. Model Optimization Techniques

Model optimization techniques aim to reduce the size and computational demands of machine learning models without compromising their performance. This is crucial for deploying models on small, resource-limited devices. Methods such as pruning, quantization, and weight clustering are commonly used to achieve this goal [26]. The main objective is to enable large models to run smoothly on edge devices with limited memory, processing power, and battery life. These optimizations are especially useful for applications requiring continuous operation. The benefits of using optimization techniques include:

**Inference Speed:** Large models take longer to make predictions, which can be problematic for real-time applications like video or audio processing. Optimization enhances inference speed, making models more suitable for time-sensitive tasks.

**Cost and Resource Efficiency:** Training and deploying large models demand substantial computational resources, often resulting in high costs. Optimization reduces these needs, enabling faster and more efficient training and deployment.

**Deployment Flexibility:** Large model sizes can hinder deployment on certain platforms or environments. Optimization makes models more portable and easier to deploy.

Quantization is a technique that reduces the size and computational complexity of machine learning models by using fewer bits to represent weights and activations. It is particularly useful for devices with limited memory and computational power, like edge and IoT devices. The technique involves reducing the precision of model weights, such as converting 32-bit floating-point numbers to 8-bit integers, which reduces model size and improves inference speed but may slightly affect accuracy. Quantization can be applied during or after training, with post-training quantization being simpler but potentially introducing errors, while quantization-aware training simulates quantization effects during training to preserve accuracy and improve performance. The main benefits include faster inference, reduced memory use, and lower energy consumption, but balancing model size and accuracy requires careful calibration [27,28].

Pruning is a method used to reduce model size by removing unnecessary parameters, lowering computational and storage needs, and improving generalization. It involves setting certain weights to zero, thus removing them from the model. Pruning can be done before, during, or after training and is effective for various models like deep neural networks and decision trees. The benefits of pruning include reduced size, simpler interpretation, and easier deployment. Weight pruning is commonly used, where less important weights are set to zero, creating sparsity in the model and reducing memory usage. While it speeds up inference, excessive pruning may degrade performance, requiring a balance between model size and accuracy [29,30].

Weight clustering is another optimization technique that reduces the number of unique weight values in a model. Instead of storing each individual weight, only unique values are saved, minimizing memory usage. The technique groups similar weights into clusters, often using the cluster centroid as the representative value for all weights in that group. By reducing the number of clusters, the model becomes more compact, saving memory and improving efficiency [31].

## 6. Implementation of Optimized Models

The objective of this experiment was to evaluate the deployment performance of the Tiny YOLO model on various embedded hardware platforms, including the ESP32, ESP32-S3, Pico W, and Jetson Nano. These platforms were chosen to

compare the feasibility of running a real-time object detection model like Tiny YOLO on resource-constrained devices, with a focus on the impact of optimization techniques such as quantization, weight pruning, and clustering.

The ESP32 and Pico W are microcontroller-based platforms known for their low power consumption and small form factors, making them suitable for simple edge applications. However, their limited computational power and memory impose constraints when running more complex deep learning models like Tiny YOLO. The ESP32-S3 variant was also included in the test, which offers enhanced AI capabilities compared to the basic ESP32 model, but still lacks the computational resources required for high-performance tasks. These microcontrollers were tested with optimizations to reduce the size of the model, improve inference time, and reduce latency. Quantization was used to reduce the precision of weights and activations, weight pruning removed less important parameters to decrease model size, and clustering grouped similar weights to further optimize the model.

The Jetson Nano, a more powerful platform equipped with a GPU and designed specifically for AI applications, was also tested. It provides significant computational power, making it better suited for real-time deep learning tasks. The Jetson Nano was used as a benchmark to compare the performance of the microcontroller-based platforms and to see how well Tiny YOLO can perform with more robust hardware. The same optimization methods were applied to the Jetson Nano to assess their impact on performance, although the higher computational power of the device meant that the benefits of optimization were less significant than on the microcontrollers.

The following metrics were measured across all devices: mean Average Precision, Frames Per Second, Model Size, Inference Time, and Latency. These metrics were used to evaluate the trade-offs between performance and computational efficiency after applying the optimization techniques. In the case of ESP32, ESP32-S3, and Pico W, the models were optimized to fit within the limited memory constraints of the devices. The resulting models were small in size but showed significant limitations in terms of accuracy, speed, and real-time performance, as the inference time remained high.

Overall, this experiment demonstrated that while optimizations such as quantization, pruning, and clustering can help make deep learning models more feasible for microcontroller-based platforms, the limited computational power of devices like ESP32 and Pico W remains a major bottleneck for real-time object detection tasks. On the other hand, the Jetson Nano proved to be a much more capable platform for deploying Tiny YOLO in real-time applications.

Quantization is first applied by converting the model’s 32-bit floating-point weights and activations to 8-bit integers. This reduces the model's size and boosts inference speed. The model is then assessed for memory savings, computational efficiency, and any slight loss in accuracy due to the reduction in numerical precision. Next, pruning is performed by eliminating weights that have little impact on the model’s performance during training, thus reducing both the model size and computational load. The pruned model is tested to evaluate the balance between efficiency improvements and any potential accuracy loss, which depends on the extent of pruning. Lastly, weight clustering is implemented, grouping similar weights into a predefined number of clusters and replacing them with shared centroids. This technique reduces memory usage without affecting numerical precision, and the clustered model is assessed for memory savings and any accuracy degradation caused by reduced weight granularity.

Deploying optimized models on hardware platforms like ESP32, ESP32-S3, Pico W, and Jetson Nano offers a range of possibilities, each suited to different use cases based on the computational power and application requirements. By applying techniques like quantization and pruning, the model's size and inference time can be reduced, making it more feasible for deployment on edge devices. Overall, selecting the appropriate platform depends on the balance between performance, power consumption, and the complexity of the task at hand.

## 7. Evaluation Results

Performance of each optimized model is compared to the base model to evaluate the benefits and trade-offs of each technique. The results of the combined optimization methods are also analyzed to find the best strategy for balancing performance and efficiency. This evaluation provides valuable insights for deploying Tiny YOLO in real-world scenarios with limited resources. The evaluation focuses on key metrics such as mean Average Precision (mAP), Frames Per Second (FPS), and Inference Time (ms), which collectively assess the models' performance and suitability for resource-

constrained environments. When deploying Tiny YOLO on embedded systems, it's essential to consider various metrics. These metrics help understand the trade-offs between efficiency and accuracy, guiding the optimization process.

Table 3 focuses only on the Pascal VOC dataset for the Tiny YOLO models deployed on ESP32, ESP32-S3, Pico W, and Jetson Nano, providing a comprehensive framework for evaluating the optimized Tiny YOLO models. The models balance high accuracy with smaller size, improved efficiency, and reduced inference time, making them suitable for image classification tasks in resource-limited environments.

**Table 3: Evaluation results for Pascal VOC dataset.**

| optimization Method | Device | mAP (%) | FPS | Inference Time (ms) |
|---|---|---|---|---|
| Base Model | ESP32 | 35.2 | 1.5 | 2750 |
| | ESP32-S3 | 27.3 | 2.5 | 1879 |
| | Pico W | 35.0 | 1.0 | 2940 |
| | Jetson Nano | 77.0 | 17.0 | 279 |
| Quantization | ESP32 | 34.9 | 1.5 | 947 |
| | ESP32-S3 | 27.1 | 2.5 | 738 |
| | Pico W | 33.8 | 1.0 | 1095 |
| | Jetson Nano | 76.7 | 17.0 | 127 |
| Pruning | ESP32 | 34.5 | 1.5 | 1030 |
| | ESP32-S3 | 26.8 | 2.5 | 712 |
| | Pico W | 33.6 | 1.0 | 1240 |
| | Jetson Nano | 76.5 | 17.0 | 145 |
| Clustering | ESP32 | 34.2 | 1.5 | 968 |
| | ESP32-S3 | 26.6 | 2.5 | 780 |
| | Pico W | 33.2 | 1.0 | 1155 |
| | Jetson Nano | 76.2 | 17.0 | 132 |

In terms of mean Average Precision (figure 1), ESP32 and Pico W show relatively low values, ranging from 34.2% to 35.2%, even after applying optimization techniques like quantization, pruning, and clustering. These platforms struggle to achieve high accuracy due to their limited processing power. On the other hand, Jetson Nano demonstrates significantly higher mAP values, ranging from 76.2% to 77%, which is a clear reflection of its superior computational capabilities. Despite optimizations, the Jetson Nano consistently maintains strong accuracy, making it a better choice for tasks requiring higher precision.

For inference time (figure 2), ESP32, ESP32-S3, and Pico W have high values, ranging from 712ms to 2940ms, due to their hardware constraints. This long inference time is detrimental to real-time object detection, as it introduces delays in processing. Conversely, Jetson Nano achieves much faster inference times, ranging from 127ms to 145ms, depending on the optimization method applied. This makes Jetson Nano an ideal platform for real-time object detection.

Jetson Nano outperforms ESP32 and Pico W across all evaluation metrics, including mAP, FPS, inference time, and latency, making it the best choice for real-time object detection tasks using Tiny YOLO. While ESP32 and Pico W offer low-cost and power-efficient solutions, their performance for complex models like Tiny YOLO is limited, making them unsuitable for real-time applications that require high accuracy and speed. Despite the modest improvements offered by optimization techniques such as quantization, pruning, and clustering, the hardware constraints of the microcontroller-based platforms continue to limit their ability to perform effectively for more demanding tasks.

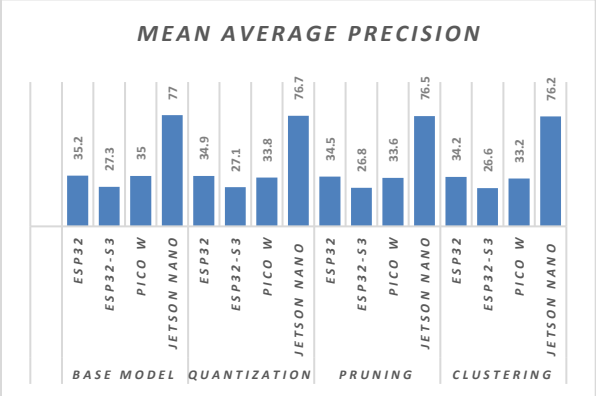


**Figure 1: The mAP for Pascal VOC.**

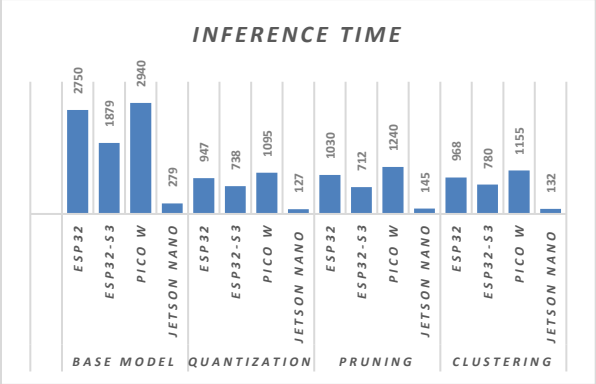


**Figure 2: Inference time for Pascal VOC.**

Table 4 focuses only on the COCO dataset for the Tiny YOLO models deployed on ESP32, ESP32-S3, Pico W, and Jetson Nano, providing a comprehensive framework for evaluating the optimized Tiny YOLO models. The models balance high accuracy with smaller size, improved efficiency, and reduced inference time, making them suitable for image classification tasks in resource-limited environments.

In terms of mean Average Precision (figure 3), ESP32 and Pico W show relatively low values, with the highest mAP reaching around 27.5% even after applying optimization techniques. The limited computational resources on these microcontrollers result in lower accuracy, which is a significant challenge despite the optimizations. In contrast, Jetson Nano consistently achieves much higher mAP values, ranging from 66.9% to 67.7%, demonstrating the platform's ability to handle more complex models like Tiny YOLO with greater precision due to its superior hardware capabilities.

**Table 4: Evaluation results for COCO dataset.**

| Optimization Method | Device | mAP (%) | FPS | Inference Time (ms) |
|---|---|---|---|---|
| Base Model | ESP32 | 29.1 | 1.5 | 3142 |
| | ESP32-S3 | 35.5 | 2.5 | 2057 |
| | Pico W | 27.2 | 1.0 | 3260 |
| | Jetson Nano | 67.7 | 17.0 | 325 |
| Quantization | ESP32 | 28.6 | 1.5 | 1180 |
| | ESP32-S3 | 34.6 | 2.5 | 875 |
| | Pico W | 26.4 | 1.0 | 1308 |
| | Jetson Nano | 67.5 | 17.0 | 117 |
| Pruning | ESP32 | 28.2 | 1.5 | 1270 |
| | ESP32-S3 | 33.9 | 2.5 | 913 |
| | Pico W | 25.8 | 1.0 | 1382 |
| | Jetson Nano | 66.9 | 17.0 | 166 |
| Clustering | ESP32 | 28.9 | 1.5 | 1195 |
| | ESP32-S3 | 35.1 | 2.5 | 897 |
| | Pico W | 26.8 | 1.0 | 1336 |
| | Jetson Nano | 67.6 | 17.0 | 132 |

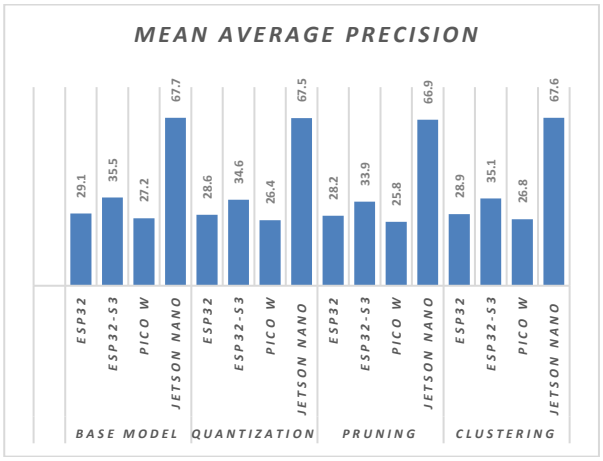


**Figure 3: The mAP for COCO.**

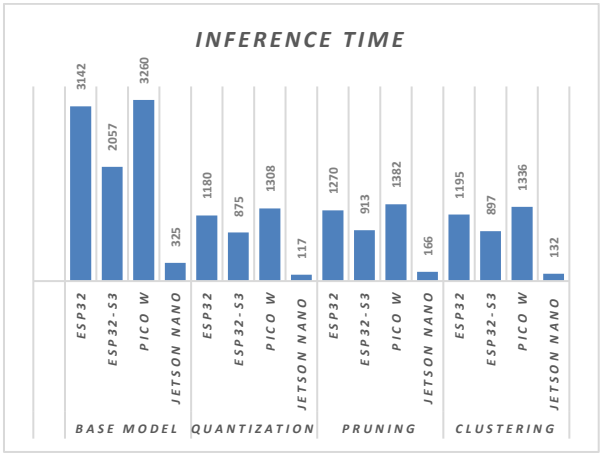


**Figure 4: Inference time for COCO.**

In terms of inference time (figure 4), ESP32, ESP32-S3, and Pico W exhibit high inference times ranging from 875ms to 3260ms, which makes these platforms unsuitable for real-time applications where speed is crucial. In contrast, Jetson Nano achieves much lower inference times, between 117ms and 166ms, making it well-suited for real-time tasks that demand faster processing.

Jetson Nano clearly outperforms both ESP32 and Pico W across all evaluation metrics, making it the optimal choice for real-time object detection with Tiny YOLO on the COCO dataset. The ESP32 and Pico W show significant limitations due to their hardware constraints, even after optimization, and are better suited for tasks of lower complexity or for applications where real-time performance is not as critical. These platforms can still be useful for simpler AI tasks, but when it comes to real-time detection requiring high accuracy, Jetson Nano is the clear leader.

## 8. Conclusion

The experiment conducted to evaluate the deployment of Tiny YOLO on a range of embedded systems, including ESP32, ESP32-S3, Pico W, and Jetson Nano, reveals key insights into the feasibility of running optimized deep learning models on resource-constrained devices. The evaluation was carried out on two popular object detection datasets, COCO and Pascal VOC, with the focus on the performance impact of three model optimization techniques: quantization, weight pruning, and clustering. The results, detailed in the tables, provide a comprehensive analysis of the trade-offs between mean Average Precision, frames per second, and inference time across different hardware platforms.

Jetson Nano, with its powerful GPU and higher computational resources, consistently outperformed the other platforms in terms of both

mAP and real-time performance. This was expected, as the Jetson Nano is designed for AI applications, offering substantial processing power and memory to handle complex models like Tiny YOLO. It demonstrated an impressive mAP of around 66.9% to 67.7% on the COCO dataset, which is a significant advantage for more computationally intensive tasks. The inference time was also much lower compared to the microcontroller-based platforms, further emphasizing its suitability for real-time applications. However, optimizations like quantization, pruning, and clustering did lead to slight improvements in inference time and latency, showing that resource-efficient techniques can make these platforms viable for simpler tasks.

One notable aspect of the experiment is the importance of model optimization. While the optimizations did not dramatically increase the mAP on these low-power platforms, they did make the models more feasible for deployment, balancing the trade-off between computational efficiency and accuracy.

The results underscore the importance of selecting the right hardware for edge AI deployment, where a balance between computational power, model size, inference time, and energy consumption must be considered. Future work could focus on further optimizing the Tiny YOLO model for even smaller and more power-efficient devices while maintaining reasonable accuracy for a broader range of real-world applications.

## References

[1] Kotha, H.D. and Gupta, V.M., 2018. IoT application: a survey. Int. J. Eng. Technol, 7(2.7), pp.891-896.

[2] Dian, F.J., Vahidnia, R. and Rahmati, A., 2020. Wearables and the Internet of Things (IoT), applications, opportunities, and challenges: A Survey. IEEE access, 8, pp.69200-69211.

[3] Asghari, P., Rahmani, A.M. and Javadi, H.H.S., 2019. Internet of Things applications: A systematic review. Computer Networks, 148, pp.241-261.

[4] Li, H., Ota, K. and Dong, M., 2018. Learning IoT in edge: Deep learning for the Internet of Things with edge computing. IEEE network, 32(1), pp.96-101.

[5] Liangzhen Lai and Naveen Suda. 2018. Enabling Deep Learning at the IoT Edge. In Proceedings of the International Conference on Computer-Aided Design (San Diego, California) (ICCAD '18). ACM, New York, NY, USA, Article 135, 6 pages.

[6] Singh, R. and Gill, S.S., 2023. Edge AI: a survey. Internet of Things and Cyber-Physical Systems, 3, pp.71-92.

[7] Wang, X., Han, Y., Leung, V.C., Niyato, D., Yan, X. and Chen, X., 2020. Edge AI: Convergence of edge computing and artificial intelligence (pp. 3-149). Singapore: Springer.

[8] David, R., Duke, J., Jain, A., Janapa Reddi, V., Jeffries, N., Li, J., Kreeger, N., Nappier, I., Natraj, M., Wang, T. and Warden, P., 2021. Tensorflow lite micro: Embedded machine learning for tinyml systems. Proceedings of Machine Learning and Systems, 3, pp.800-811.

[9] Rashidi, M., 2022. Application of TensorFlow lite on embedded devices: A hands-on practice of TensorFlow model conversion to TensorFlow Lite model and its deployment on Smartphone to compare model's performance.

[10] Mamtha, G.N., Sharma, S. and Sing, N., 2023, December. Embedded Machine Learning with Tensorflow Lite Micro. In 2023 International Conference on Power Energy, Environment & Intelligent Control (PEEIC) (pp. 1480-1483).

[11] Berthelier, A., Chateau, T., Duffner, S., Garcia, C. and Blanc, C., 2021. Deep model compression and architecture optimization for embedded systems: A survey. Journal of Signal Processing Systems, 93(8), pp.863-878.

[12] TensorFlow Lite, TensorFlow, 2021. Available online: https://www.tensorflow.org/lite

[13] Hua, H., Li, Y., Dong, N., Li, W. and Cao, J., 2023. Edge computing with artificial intelligence: A machine learning perspective. ACM Computing Surveys, 55(9), pp.1-35.

[14] Deng, S., Zhao, H., Fang, W., Yin, J., Dustdar, S. and Zomaya, A.Y., 2020. Edge intelligence: The confluence of edge computing and artificial intelligence. IEEE Internet of Things Journal, 7(8), pp.7457-7469.

[15] Grzesik, P. and Mrozek, D., 2024. Combining Machine Learning and Edge Computing: Opportunities, Challenges, Platforms, Frameworks, and Use Cases. Electronics, 13(3), p.640.

[16] Li, H., Ota, K. and Dong, M., 2018. Learning IoT in edge: Deep learning for the Internet of Things with edge computing. IEEE network, 32(1), pp.96-101.

[17] Chang, Z., Liu, S., Xiong, X., Cai, Z. and Tu, G., 2021. A survey of recent advances in edge-computing-powered artificial intelligence of things. IEEE Internet of Things Journal, 8(18), pp.13849-13875.

[18] Sivaganesan, D., 2019. Design and development ai-enabled edge computing for intelligent-iot applications. Journal of trends in Computer Science and Smart technology (TCSST), 1(02), pp.84-94.

[19] Jain, S., Dash, S. and Deorari, R., 2022, October. Object detection using coco dataset. In 2022 International Conference on Cyber Resilience (ICCR) (pp. 1-4). IEEE.

[20] Shetty, S., 2016. Application of convolutional neural network for image classification on Pascal VOC challenge 2012 dataset. arXiv preprint arXiv:1607.03785.

[21] Li, C., Wang, J., Wang, S. and Zhang, Y., 2024. A review of IoT applications in healthcare. Neurocomputing, 565, p.127017.

[22] Afzal, B., Umair, M., Shah, G.A. and Ahmed, E., 2019. Enabling IoT platforms for social IoT applications: Vision, feature mapping, and challenges. Future Generation Computer Systems, 92, pp.718-731.

[23] Dian, F.J., Vahidnia, R. and Rahmati, A., 2020. Wearables and the Internet of Things (IoT), applications, opportunities, and challenges: A Survey. IEEE access, 8, pp.69200-69211.

[24] Tripathi, A., Gupta, M.K., Srivastava, C., Dixit, P. and Pandey, S.K., 2022, December. Object detection using YOLO: A survey. In 2022 5th International Conference on Contemporary Computing and Informatics (IC3I) (pp. 747-752). IEEE.

[25] Hussain, M., 2024. Yolov1 to v8: Unveiling each variant–a comprehensive review of yolo. IEEE Access, 12, pp.42816-42833.

[26] Babaei, P., 2024, March. Convergence of Deep Learning and Edge Computing using Model Optimization. In 2024 13th Iranian/3rd International Machine Vision and Image Processing Conference (MVIP) (pp. 1-6). IEEE.

[27] Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M.W. and Keutzer, K., 2022. A survey of quantization methods for

efficient neural network inference. In Low-Power Computer Vision (pp. 291-326). Chapman and Hall/CRC.
[28] Rokh, B., Azarpeyvand, A. and Khanteymoori, A., 2023. A comprehensive survey on model quantization for deep neural networks in image classification. ACM Transactions on Intelligent Systems and Technology, 14(6), pp.1-50.
[29] Liang, T., Glossner, J., Wang, L., Shi, S. and Zhang, X., 2021. Pruning and quantization for deep neural network acceleration: A survey. Neurocomputing, 461, pp.370-403.
[30] Madnur, P.V., Dabade, S.H., Khanapure, A., Rodrigues, S., Hegde, S. and Kulkarni, U., 2023, November. Enhancing Deep Neural Networks through Pruning followed by Quantization Pipeline: A Comprehensive Review. In 2023 2nd International Conference on Futuristic Technologies (INCOFT) (pp. 1-8). IEEE.
[31] Choudhary, T., Mishra, V., Goswami, A. and Sarangapani, J., 2020. A comprehensive survey on model compression and acceleration. Artificial Intelligence Review, 53, pp.5113-5155.