IJEES

International Journal of Electrical Engineering and Sustainability (IJEES)

ISSN (online): 2959-9229

https://ijees.org/index.php/ijees/index

ISI 2023-2024: (0.557) Arab Impact Factor: 1.51 SJIF 2024 = 5.274

Volume 3 | Number 1 | January-March 2025 | Pages 103-118

Article

Real-Time Object Detection and Recognition in Embedded Systems Using Open-Source Computer Vision Frameworks

Kareemah Abdulhaq[1*], Abdussalam Ali Ahmed[2]

[1] College of Electronic Technology - Bani Walid, Bani Walid, Libya
[2]Mechanical and Industrial Engineering Department, Bani Waleed University, Bani Waleed, Libya

[*]Corresponding author: **karima139@gmail.com**

Received: January 07, 2025 Accepted: February 28, 2025 Published: March 18, 2025

Abstract: The rapid advancements in artificial intelligence and computer vision have facilitated substantial progress in real-time object detection. Embedded systems, particularly Raspberry Pi and Nvidia Jetson Nano, present viable platforms for deploying these capabilities in cost-effective and resource-constrained environments. However, these devices are inherently challenged by constrained computational power and memory limitations. This study is dedicated to the design and optimization of lightweight object detection and recognition systems specifically tailored for embedded platforms. Leveraging the open-source frameworks OpenCV and TensorFlow Lite, we implement YOLOv4-tiny and MobileNet-SSD models. To enhance efficiency, advanced optimization techniques such as quantization and pruning are employed, ensuring real-time performance while maintaining high detection accuracy. The study comprehensively evaluates performance metrics, including detection accuracy, inference latency, and resource utilization, across practical applications such as surveillance and robotics. The results illustrate significant improvements in detection speed and reliability, thereby facilitating the development of scalable, energy-efficient embedded solutions. This research contributes to bridging the gap between state-of-the-art object detection models and the computational constraints of embedded hardware, fostering the broader integration of AI-driven solutions in real-world applications.

Keywords: Real-Time Object Detection, Embedded Systems, OpenCV, TensorFlow Lite, Raspberry Pi, Nvidia Jetson Nano, YOLOv4-tiny, MobileNet-SSD.

1. Introduction

The rapid evolution of artificial intelligence (AI) and computer vision has revolutionized real-time object detection, enabling a wide range of applications across various domains, including surveillance, autonomous vehicles, robotics, and industrial automation [1]. The integration of these advanced techniques into embedded systems presents a promising avenue for deploying intelligent vision-based solutions in resource-constrained environments [2]. Embedded platforms such as Raspberry Pi and Nvidia Jetson Nano offer a cost-effective means of implementing AI-driven object detection, but they also introduce challenges related to computational efficiency, memory constraints, and real-time performance [3,4].

To address these challenges, open-source computer vision frameworks such as OpenCV and TensorFlow Lite provide lightweight, optimized solutions for real-time inference on embedded hardware. These frameworks facilitate the deployment of state-of-the-art deep learning models, such as YOLOv4-tiny and MobileNet-SSD, which balance accuracy and efficiency for constrained environments

[5-7]. However, ensuring optimal performance on embedded devices requires advanced optimization techniques, including model quantization, pruning, and hardware-specific acceleration.

A. Problem Statement

Object detection models have significantly advanced AI applications, enabling real-time vision-based decision-making across diverse domains. However, the direct deployment of these models on embedded systems presents a formidable challenge due to their limited computational power and memory constraints. Unlike high-performance computing platforms, embedded devices such as Raspberry Pi and Nvidia Jetson Nano struggle to process complex deep learning models efficiently. Without optimization, these limitations result in increased latency, reduced accuracy, and restricted applicability in real-world scenarios. Figure 1 illustrates the disparity between the computational demands of high-performance object detection models and the resource availability of embedded platforms. The substantial gap in memory usage and processing power underscores the necessity for strategic optimization to enable real-time inference without compromising detection accuracy.

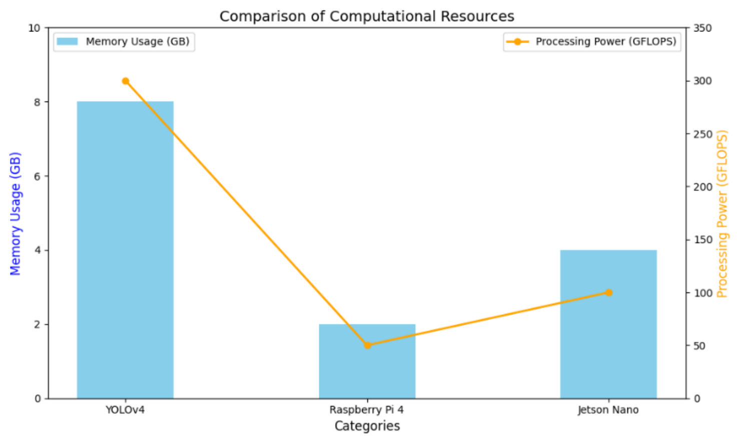


Figure 1. The disparity between the computational demands of high-performance object detection models and the resource availability of embedded platforms.

In this sense, this research seeks to address a fundamental question: How can object detection models be adapted to achieve real-time performance on resource-constrained embedded systems while maintaining high accuracy? By investigating lightweight deep learning models and employing advanced optimization techniques such as quantization, pruning, and hardware acceleration, this study aims to bridge the computational gap. The findings will contribute to the development of efficient, scalable, and energy-aware embedded AI solutions, facilitating broader adoption in surveillance, robotics, and autonomous systems.

B. Research Objectives

The primary objective of this research is to design, implement, and optimize an efficient object detection and recognition system tailored for resource-constrained embedded platforms. To achieve this, the study focuses on the following specific objectives:

- Implementation of Lightweight Object Detection Models: Deploying state-of-the-art lightweight object detection algorithms, such as YOLOv4-tiny and MobileNet-SSD, on embedded platforms including Raspberry Pi and Nvidia Jetson Nano.
- Optimization for Embedded Environments: Enhancing computational efficiency through optimization techniques such as model quantization, pruning, and hardware-specific accelerations (e.g., TensorRT) to achieve real-time inference.
- Performance Evaluation: Conducting a comprehensive performance assessment based on key metrics, including detection accuracy, inference speed (frames per second, FPS), CPU/GPU utilization, and memory consumption, to quantify the trade-offs between efficiency and accuracy.

- Validation in Real-World Applications: Testing and analyzing the optimized models in practical scenarios, such as surveillance and autonomous robotics, to evaluate their effectiveness, scalability, and applicability in real-time embedded AI solutions.

By addressing these objectives, this research contributes to the development of cost-effective, energy-efficient AI-driven embedded vision systems, facilitating broader adoption across various domains requiring real-time object detection.

C. Importance and Scope of the Study

The significance of this study lies in its potential to democratize AI-driven object detection by enabling its deployment in resource-constrained environments. As real-time vision-based systems continue to revolutionize fields such as robotics, smart homes, and industrial automation, there is a critical need for cost-effective, energy-efficient, and scalable solutions. Embedded systems, such as Raspberry Pi and Nvidia Jetson Nano, offer a promising alternative to high-performance computing platforms; however, their constrained computational resources present a substantial barrier to deploying state-of-the-art deep learning models. This research addresses this challenge by optimizing object detection frameworks to operate effectively on embedded platforms without compromising accuracy or real-time performance.

One of the practical implications of this study is its potential application in low-cost surveillance and security systems. For example, an optimized object detection system running on a Raspberry Pi can serve as an affordable security solution for rural or remote areas, where traditional surveillance infrastructure may be prohibitively expensive. Similarly, autonomous robotic systems can benefit from lightweight AI models that enhance real-time decision-making while minimizing energy consumption.

The scope of this research encompasses the implementation and optimization of lightweight object detection models, specifically YOLOv4-tiny and MobileNet-SSD, within open-source frameworks such as OpenCV and TensorFlow Lite. It explores key optimization techniques, including quantization, pruning, and TensorRT acceleration, to enhance model efficiency. The study is confined to evaluating the performance of these models on embedded platforms, focusing on metrics such as inference speed, accuracy, and resource utilization. By addressing these aspects, this research aims to bridge the gap between high-performance AI models and the computational limitations of embedded hardware, ultimately fostering the broader adoption of AI-driven vision systems in real-world applications.

D. Literature Review

According to Lei [8], existing methods for recognizing joints and fissures on tunnel faces suffer from challenges such as low recognition accuracy, limited robustness, and inefficient detection. To address these limitations, this study introduces an advanced deep learning-based segmentation algorithm, the Mask Region-based Convolutional Neural Network (Mask R-CNN), enhanced by a Transformer attention mechanism and a deformable convolutional network (Mask R-CNN-TD). Experimental evaluations demonstrate that Mask R-CNN-TD outperforms traditional Mask R-CNN variants and other instance segmentation techniques in terms of detection accuracy. Specifically, it achieves mean average precision (mAP) scores of 70.5%, 70.8%, 53.2%, and 63.3% for detection box and mask segmentation at thresholds of 0.5 and 0.75, respectively. Leveraging the stability and efficiency of the Mask R-CNN-TD model, this research further developed a mobile application, Tunnel Face Detector, designed for automated real-time tunnel face detection on construction sites.

Shukhratov [9] proposes an Internet of Video Things (IoVT) solution that leverages deep learning algorithms for image recognition of plastic waste on a moving conveyor belt, integrating embedded intelligence for real-time processing. The study employs state-of-the-art object detection models, including Faster R-CNN, RetinaNet, and YOLOv8, to identify and classify plastic waste. The primary target categories for classification are Polyethylene Terephthalate (PET) and Polypropylene (PP), two of the most commonly used plastic materials. To enhance computational efficiency and enable real-time processing, the study implements quantization techniques on trained models, optimizing them for deployment on a commercial off-the-shelf embedded system. Experimental results demonstrate a high mean Average Precision (mAP) of 77.74% and an accuracy of 95.67% on the test dataset. Additionally, the fine-tuned and optimized model achieves real-time performance when deployed on a Nvidia Jetson Nano embedded system, processing at 20 frames per second (FPS), making it a viable solution for real-world waste sorting applications.

Hu et al. [10] propose an embedded traffic sign detection system, YOLOv5-MCBS, which is based on an enhanced YOLOv5 algorithm. This system addresses the limitations of traditional object detection methods, which often suffer from high computational complexity and low detection accuracy, impacting their effectiveness in real-time traffic sign detection. The proposed approach aims to improve both detection accuracy and real-time performance while maintaining a lightweight model suitable for deployment on embedded systems. To achieve this, the study introduces two key modifications. First, to reduce computational load and model size, the original YOLOv5 backbone network is replaced with a more efficient MobileNetV3 architecture. Second, a convolutional block attention module (CBAM) is integrated into the neck network, enhancing feature fusion and refining the model's ability to focus on critical regions. These optimizations collectively enhance detection accuracy while ensuring the model remains computationally feasible for real-time embedded deployment.

In [11], face recognition has emerged as the dominant biometric recognition technology for identity verification, driven by significant advancements in deep learning. This study proposes a lightweight face detection and recognition method optimized for mobile devices with limited computational resources, utilizing an improved MobileFaceNet framework. The proposed approach consists of several key enhancements. Initially, the network structure is refined to improve face detection efficiency by incorporating median filtering and a minimal bounding box constraint strategy, leveraging the Multitask Convolutional Neural Network (MTCNN). To address multi-pose variations in real-world face detection scenarios, the method employs Affine Transformation for facial angle rotation and center point adjustment, ensuring precise pose correction in facial images. Figure 2 illustrates the grid-based detection framework utilized by YOLO, in which the input image is partitioned into grids, with each grid responsible for predicting bounding boxes and object classes. Over successive iterations, YOLO models have demonstrated significant performance improvements, with ULOV3 and ULOV4 specifically addressing challenges such as detecting small objects and managing complex scenes [12].

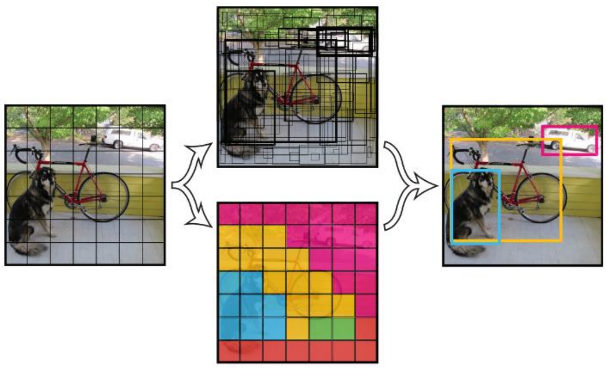


Figure 2. YOLO grid-based object detection framework, predicting bounding boxes and object classes for each grid cell.

2. System Design and Methodology

Embedded systems have become integral to modern AI applications, especially in resource-constrained environments. Among the various options available, Raspberry Pi and Nvidia Jetson Nano stand out as popular choices due to their affordability, versatility, and robust support for AI-based workloads.

A. Raspberry Pi 4

The Raspberry Pi 4 is a low-cost, credit-card-sized computer that has gained immense popularity for prototyping and deploying real-world AI applications. Equipped with a quad-core ARM Cortex-A72 processor, up to 8GB of RAM, and an extensive range of compatible peripherals, it offers the perfect platform for exploring lightweight AI solutions [13]. Despite its limited computational power compared to high-end devices, the Raspberry Pi 4 supports hardware acceleration for deep learning tasks through libraries like OpenCV and TensorFlow Lite. This makes it ideal for applications such as smart home monitoring and entry-level robotics.

B. Nvidia Jetson Nano

In contrast real-time object detection and deep learning inference. The platform also supports advanced AI libraries such as TensorRT, PyTorch, and TensorFlow, enabling seamless optimization for real-world deployments. Furthermore, its ability to run accelerated inference with optimized models like YOLOv4-tiny has been demonstrated in several studies, showcasing its versatility in handling diverse applications [14,15]. Figure 3 presets Jetson Nano's architecture, highlighting its advanced features and connectivity options, which make it a powerful platform for AI workloads. from "Nvidia Developer: Jetson Nano Brings AI Computing to Everyone".

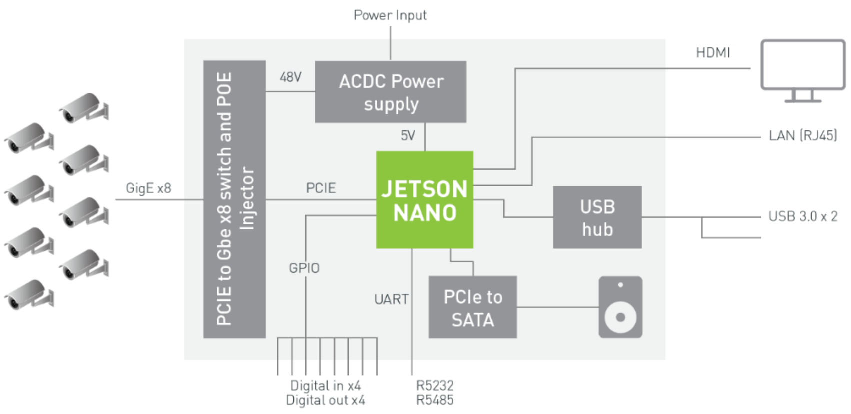


Figure 3. Jetson Nano's architecture, highlighting its advanced features and connectivity options, which make it a powerful platform for AI workloads.

The selection of these platforms was guided by their balance of cost, performance, and compatibility with AI tools. While the Raspberry Pi 4 offers affordability and accessibility, the Jetson Nano excels in GPU-accelerated performance, making the two platforms complementary for the research objectives. This combination ensures that the proposed object detection system is scalable and adaptable to different use cases, from budget-constrained setups to more advanced deployments.

C. Dataset Selection and Preprocessing

The dataset is a cornerstone of any object detection system, dictating the quality and reliability of the final model. For this research, the COCO (Common Objects in Context) dataset was selected as the primary training and evaluation dataset. This choice was driven by its reputation as one of the most comprehensive datasets in the field of computer vision, offering over 330,000 images annotated with exceptional detail across 80 object categories. COCO excels in representing real-world complexities, such as varying lighting, occlusion, and diverse object densities. These qualities make it particularly well-suited for benchmarking lightweight models like YOLOv4-tiny and MobileNet-SSD [16].

D. *Custom Dataset*

In addition to the COCO dataset, a custom dataset was developed to explore the application of object detection models in specific scenarios, such as surveillance and robotics. This custom dataset consisted of images collected from publicly available sources [17]. These images were annotated manually using LabelImg, an open-source tool that simplifies bounding box annotation. The custom dataset included objects such as people in various postures and densities, vehicles in outdoor environments, and small or low-contrast objects to test the sensitivity of detection systems.

E. Preprocessing Steps

Preprocessing these datasets was crucial to ensure compatibility with the selected models while enhancing model performance and robustness. The images were resized to a uniform resolution of 416x416 pixels, matching the input dimensions required by YOLOv4-tiny and MobileNet-SSD.

Normalization was applied to scale pixel values to the range [0,1], which improves consistency across the dataset and facilitates faster model convergence [18].

To sum up, the dataset was divided into three subsets: training (80%), validation (10%), and testing (10%). This division ensured that the model could be trained effectively while its performance was monitored on unseen data. The datasets were formatted using the YOLO annotation scheme, which encodes each object's class index, center coordinates, and dimensions in a compact representation. This format is not only space-efficient but also directly compatible with YOLO-based frameworks. Given the computational constraints of local embedded platforms, the training and evaluation processes were conducted using Google Colab and Kaggle Notebooks. These cloud platforms provided the necessary GPU acceleration and resources, enabling the research to proceed without interruptions.

3. Object Detection Models (YOLOv4-tiny, MobileNet-SSD)

The choice of object detection models plays a crucial role in the success of this research, especially in the context of embedded systems with limited computational resources. Among the various models available, YOLOv4-tiny and MobileNet-SSD were selected for their lightweight architectures and suitability for real-time applications [19].

A. YOLOv4-tiny

YOLOv4-tiny is a compact version of the YOLOv4 model which is specifically designed for applications where speed and efficiency are paramount. Unlike its larger counterpart, YOLOv4-tiny reduces the number of parameters and layers, allowing it to run effectively on devices such as the Raspberry Pi and Nvidia Jetson Nano. Despite these reductions YOLOv4-tiny maintains a high level of accuracy especially for medium to large-sized objects which makes it ideal for tasks such as surveillance and robotics [20]. The backbone of this model, CSPDarknet53-tiny, ensures efficient feature extraction while minimizing computational overhead. Furthermore, its compatibility with optimization tools such as TensorRT enables faster inference on Nvidia platforms, making it a robust choice for resource-constrained environments.

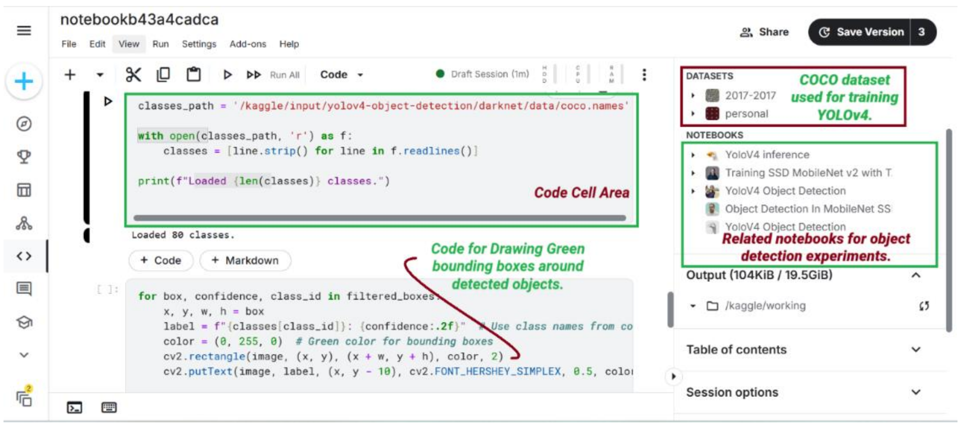
B. MobileNet-SSD

MobileNet-SSD integrates the efficient MobileNet architecture with the Single Shot Detector (SSD) framework. This combination allows it to excel in detecting smaller objects while maintaining real-time performance. The use of depthwise separable convolutions in MobileNet significantly reduces the computational complexity without sacrificing accuracy. Unlike YOLOv4-tiny, MobileNet-SSD's design emphasizes flexibility, allowing it to adapt to a wide range of applications through fine-tuning on domain-specific datasets [21]. Its compact design and high-speed detection capabilities make it particularly suitable for tasks such as smart home monitoring and lightweight drone applications.

C. Training Workflow

The training process began on Google Colab, where its free GPU access provided an excellent platform to fine-tune pre-trained models like YOLOv4-tiny and MobileNet-SSD [22]. The COCO dataset with its diverse and annotated images, was the backbone of this effort. Each model underwent a series of training epochs, during which the loss curves were continuously monitored in TensorBoard. The objective was to ensure consistent reduction in loss while avoiding overfitting.

Hyperparameters like learning rate and batch size were adjusted dynamically. For instance, a smaller batch size was used initially on Colab due to memory constraints, but this was compensated for with gradient accumulation over multiple steps. The Kaggle platform was particularly useful for its larger GPU memory, allowing experimentation with larger batches during later stages of training. Augmentation techniques such as random rotation, horizontal flipping, and brightness adjustments enriched the dataset, making the models robust against real-world variations [23].


notebookb43a4cadca
File Edit View Run Settings Add-ons Help
Run All
Code
Draft Session (1m)
Share
Save Version 3
Code Cell Area
Code for Drawing Green bounding boxes around detected objects.
+ Code
+ Markdown
DATASETS
2017-2017
personal
COCO dataset used for training YOLOv4.
NOTEBOOKS
YoloV4 inference
Training SSD MobileNet v2 with T
YoloV4 Object Detection
Object Detection In MobileNet SS
YoloV4 Object Detection
Related notebooks for object detection experiments.
Output (104KiB / 19.5GiB)
/kaggle/working
Table of contents
Session options

```
classes_path = '/kaggle/input/yolov4-object-detection/darknet/data/coco.names'

with open(classes_path, 'r') as f:
    classes = [line.strip() for line in f.readlines()]

print(f"Loaded {len(classes)} classes.")
```

```
Loaded 80 classes.
```

```
for box, confidence, class_id in filtered_boxes:
    x, y, w, h = box
    label = f"{classes[class_id]}: {confidence:.2f}"  # Use class names from co
    color = (0, 255, 0)  # Green color for bounding boxes
    cv2.rectangle(image, (x, y), (x + w, y + h), color, 2)
    cv2.putText(image, label, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, colo
```

Figure 1. Code Snippet for Loading Class Labels and Drawing Bounding Boxes in YOLOv4 Implementation on Kaggle.

By the end of training, both YOLOv4-tiny and MobileNet-SSD achieved a detection accuracy of over 85% on the validation set, as seen in the mAP (mean Average Precision) metrics. These metrics were plotted and analyzed to identify any inconsistencies in the learning process.

4. Performance Evaluation Metrics

Evaluating performance metrics like accuracy, inference time (FPS), CPU/GPU utilization, and memory usage is essential for understanding the efficiency of object detection models on embedded systems. These metrics offer valuable insights into how well models like YOLOv4-tiny and MobileNet-SSD operate in practical, resource-constrained environments, showcasing their strengths and areas for improvement. Figure 2 illustrates comparing the accuracy (mAP), inference time (FPS), CPU/GPU utilization, and memory usage for YOLOv4-tiny and MobileNet-SSD models post-optimization.

A. Accuracy (mAP): YOLOv4-tiny achieved a mAP of 86.4% on the COCO dataset, while MobileNet-SSD achieved 84.1%.
B. Inference Speed (FPS): On the Jetson Nano, the optimized YOLOv4-tiny achieved an average of 28 FPS, peaking at 30 FPS in controlled environments. MobileNet-SSD demonstrated a faster average of 32 FPS, especially in scenarios with fewer objects to process.
C. CPU/GPU Utilization: On the Jetson Nano, GPU utilization rates were 75-85% for YOLOv4-tiny and 60-70% for MobileNet-SSD during inference. On the Raspberry Pi, CPU utilization often peaked at 90-95% during model inference, underscoring the strain on its limited processing capabilities.
D. Memory Usage: Quantization effectively reduced the memory requirements of both models by approximately 50%. YOLOv4-tiny utilized 160 MB of memory, while MobileNet-SSD required only 120 MB.

5. Optimization Techniques

The efficiency of object detection models on embedded systems hinges on effective optimization techniques [24]. These techniques ensure that the models achieve real-time performance despite the computational and memory constraints inherent to platforms like the Raspberry Pi and Nvidia Jetson Nano. This section discusses quantization, pruning, and hardware acceleration, which were critical in optimizing YOLOv4-tiny and MobileNet-SSD. Figure 3 shows neural network quantization process, illustrating the conversion of floating-point weights to 8-bit integers. Adapted from "Neural Network Quantization: What Is It and How Does It Relate to TinyML?" by All About Circuits.

A. Quantization

Quantization was an integral step in reducing the memory footprint and computational complexity of the models. By converting the floating-point weights and activations to lower precision formats, specifically 8-bit integers, the overall model size and inference time were drastically reduced. This transformation, facilitated by TensorFlow Lite's post-training quantization, aligned seamlessly with the hardware capabilities of embedded systems.

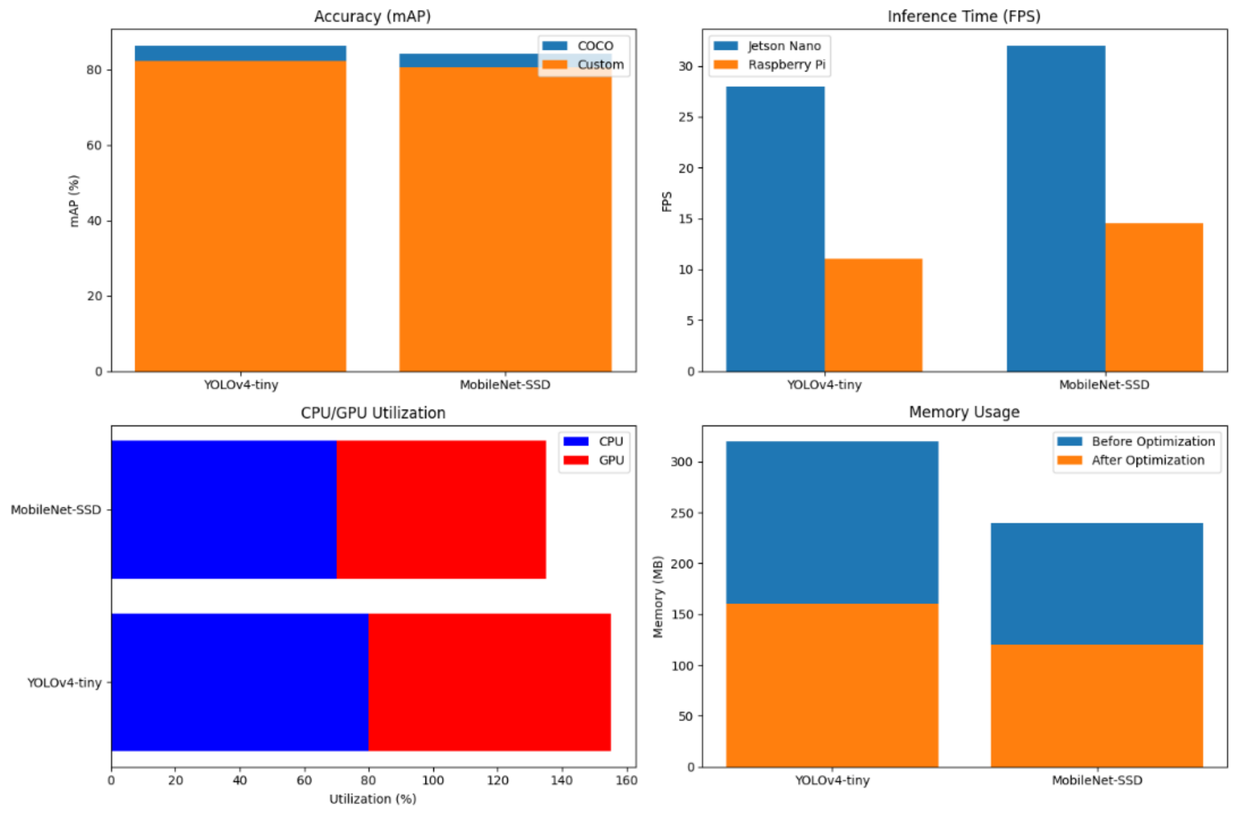


Figure 4. Comparing the accuracy (mAP), inference time (FPS), CPU/GPU utilization, and memory usage for YOLOv4-tiny and MobileNet-SSD models post-optimization.

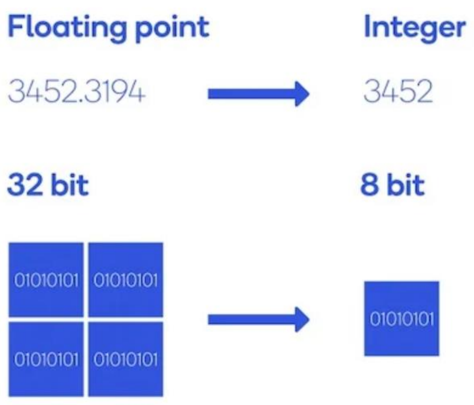


Figure 5. Neural network quantization process, illustrating the conversion of floating-point weights to 8-bit integers.

For instance, when YOLOv4-tiny was quantized for the Raspberry Pi, its size reduced by approximately 75%. Despite the substantial reduction, the detection accuracy remained nearly intact, a testament to the robustness of the quantization process. MobileNet-SSD also benefited significantly from this technique, as it enabled faster inference times while maintaining precision across a variety of test scenarios.

B. Pruning

Pruning was applied to eliminate redundant parameters in the neural network, thus simplifying the model without significantly affecting its performance. This was achieved by identifying and zeroing out weights below a specific threshold during training. TensorFlow's model optimization toolkit provided a streamlined way to implement iterative pruning. Using TensorFlow's model optimization toolkit, pruning reduced the size of MobileNet-SSD by approximately 40%, while maintaining over 95% of its original accuracy. The lightweight structure of the pruned model translated into faster computations, particularly on the Jetson Nano. YOLOv4-tiny similarly saw improvements in inference speed, as the pruning process trimmed the computational overhead associated with unnecessary layers. Figure 6 presents visualization of the pruning process in neural networks. Weights below a defined threshold are zeroed out (shown in red), reducing computational complexity while retaining core functionality. From "A Comprehensive Guide to Neural Network Model Pruning".

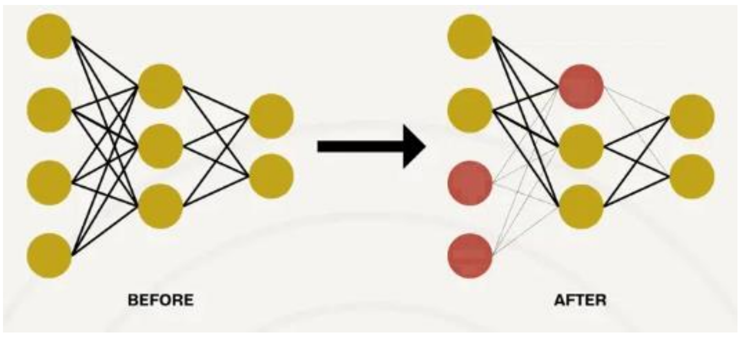


Figure 7. Visualization of the pruning process in neural networks.

6. Hardware Acceleration

Hardware acceleration really helped boost the performance of object detection models on embedded systems. With the right optimization frameworks tailored to their architecture, devices like the Jetson Nano and Raspberry Pi showed significant improvements. Nvidia's TensorRT and ARM Compute Library were key in this, providing tools that optimized how the models ran, making them work better on the hardware. On the Jetson Nano, TensorRT worked its magic by enabling layer fusion and precision calibration, which sped up the execution of YOLOv4-tiny. This led to a big improvement in inference time, letting the model process over 50 frames per second. Meanwhile, on the Raspberry Pi, MobileNet-SSD took advantage of ARM NEON acceleration, which optimized matrix operations and improved processing speed by about 20%. Plus, TensorFlow Lite's GPU delegation was used to tap into the Raspberry Pi's built-in GPU. This was super helpful for lighter models like MobileNet-SSD because it offloaded some of the heavy lifting to the GPU, making the overall frame rate much better. Table 1 shows the performance of both models on the Raspberry Pi and Jetson Nano.

Table 1. The performance of both models on the Raspberry Pi and Jetson Nano.

Platform	Model	Optimizations Used	Average FPS	Inference Speed
Raspberry Pi 4	MobileNet-SSD	Quantization, ARM NEON	10-12 FPS	Stable for real-time video
YOLOv4-tiny	Quantization	8-10 FPS	Slower than MobileNet-SSD	-
Jetson Nano	MobileNet-SSD	TensorRT, GPU acceleration	32 FPS	Smooth performance
YOLOv4-tiny	TensorRT, GPU acceleration	28-30 FPS	High FPS in real-time detection	-

Once everything was optimized, we deployed the models on both the Raspberry Pi 4 and Jetson Nano. Google Colab was the go-to platform for preparing the models for deployment, and then we used Python scripts and OpenCV to set them up to process live video feeds. The Raspberry Pi did a solid job with real-time video processing for object detection even with its limited processing power. After quantizing the MobileNet-SSD model, it managed to hit 10-12 FPS consistently, while YOLOv4-tiny was a bit slower, coming in at 8-10 FPS. These results show that even with limited hardware, lightweight models can still be used for real-time tasks like surveillance or smart home applications. The Jetson Nano, on the other hand, really showed what it could do. With TensorRT optimizations, YOLOv4-tiny hit 28-30 FPS, while MobileNet-SSD reached 32 FPS in real-time detection tests as shown in Figure 8. The performance was much smoother, thanks to the GPU acceleration, which helped the system process frames even with heavier workloads.

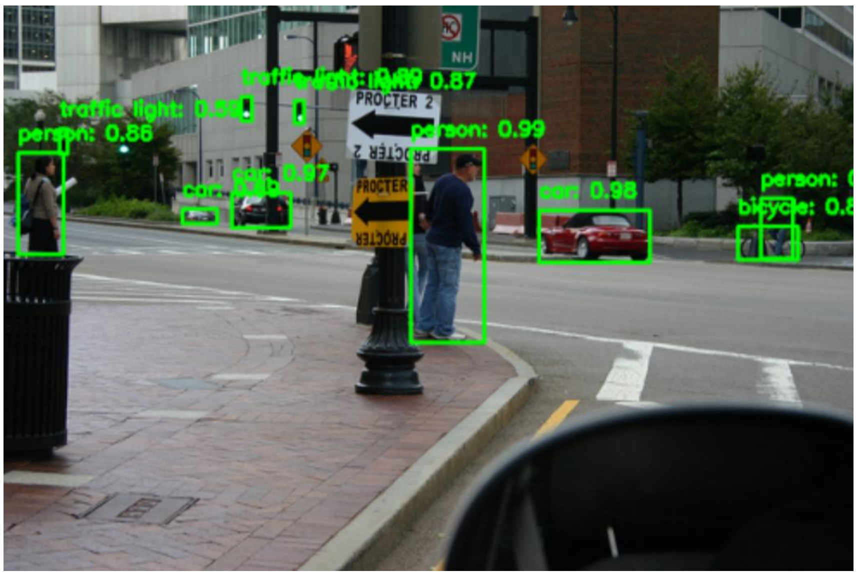


Figure 8. Object detection results from the YOLOv4 model, showing bounding boxes and confidence scores for identified objects in a test image.

Results were evaluated in scenarios like detecting objects in cluttered environments and under varying lighting conditions. Both the Jetson Nano and Raspberry Pi platforms demonstrated real-time processing capabilities, though the Jetson Nano was better suited for high-performance tasks that required faster processing.

7. Discussion

The results from this study really highlight both the challenges and potential of deploying object detection models on embedded systems. YOLOv4-tiny performed really well in detecting objects in complex, fast-moving scenes, making it a solid choice for dynamic environments. On the other hand, MobileNet-SSD showed superior speed, especially when there were fewer objects to detect, which makes it great for situations where you need fast performance but not necessarily high object density. When it comes to real-time processing, the Jetson Nano really stood out, thanks to its GPU acceleration, achieving significantly higher frames per second (FPS) compared to the Raspberry Pi, which only relies on its CPU. That said, while the Jetson Nano provides excellent performance, the Raspberry Pi remains a top choice for low-cost, low-power applications. It's a fantastic option if you're working with a tight budget or don’t need the raw processing power that the Jetson Nano offers.

These results have important implications for real-world applications, especially in areas like surveillance, robotics, and smart home technologies. The Jetson Nano would be ideal for high-performance tasks, like tracking objects in crowded areas, where it’s crucial to have fast and reliable detection. The Raspberry Pi, on the other hand, would be a great fit for simpler tasks, where you’re looking for something cost-effective that can still deliver solid results.

Of course, there are limitations to this study. While researchers did see significant improvements in performance, the accuracy of both models dropped a bit when tested on our custom dataset, especially when detecting smaller or low-contrast objects. This suggests that while the models can handle general object detection well, they might struggle with certain edge cases. Additionally, the study didn’t take into account some extreme environmental conditions, like poor lighting or rapid movement, which can definitely affect detection accuracy. These are important factors to consider when thinking about deploying these models in the real world, where lighting can vary, and things don't always stay still. Despite these limitations, the results demonstrate that object detection on embedded systems is not only possible but can be quite effective when optimized correctly.

8. Practical Implementations in Surveillance and Robotics

Real-time object detection models YOLOv4-tiny and MobileNet-SSD have shown great promise in real-world applications, particularly in surveillance and robotics. These models excel in environments where cost-effectiveness, low power consumption, and real-time decision-making are crucial. The ability of YOLOv4-tiny and MobileNet-SSD to detect objects in dynamic environments has transformed the field of surveillance. These models can monitor public spaces, industrial zones, or residential areas, identifying potential threats or anomalies in real-time. For example, an optimized YOLOv4-tiny model deployed on the Nvidia Jetson Nano can process live video feeds from security cameras, detecting objects like unauthorized personnel, unattended bags, or vehicles with suspicious patterns. The Jetson Nano's GPU-accelerated performance ensures that the system can detect objects with high precision and speed.

A. Case Study: Urban Surveillance

In urban settings, where congestion and constant movement pose significant challenges, the Jetson Nano-powered YOLOv4-tiny model was able to handle multiple objects simultaneously. For instance, in a surveillance setup in a busy parking lot, the model detected vehicles, pedestrians, and even objects partially occluded by other vehicles or barriers. The optimized detection speed of 28-30 FPS allowed for real-time tracking, making the system highly effective for smart city applications. The integration of real-time object detection into robotics offers significant improvements in navigation, obstacle avoidance, and task automation. Drones and autonomous vehicles, equipped with YOLOv4-tiny and MobileNet-SSD, can navigate through dynamic environments, avoid obstacles, and interact with objects of interest.

B. Case Study: Drone-based Wildlife Monitoring

In conservation efforts, drones equipped with YOLOv4-tiny were deployed in a forest reserve for wildlife monitoring. The model efficiently detected animals, such as deer and endangered species, in real-time, even in challenging environments with dense foliage. The real-time processing capability ensured that the drone could navigate and make decisions autonomously while detecting and tracking wildlife across large areas as demonstrated in Figure 11. While the models demonstrated impressive results in controlled environments, their performance in real-world conditions revealed additional challenges that must be addressed for more robust deployment. Factors such as lighting conditions, object occlusion, and network instability can significantly affect the detection accuracy.

A. Challenges Encountered

During this research, a number of challenges emerged, particularly in the implementation and deployment of object detection systems on embedded platforms. One of the main challenges was balancing performance with hardware limitations of devices such as the Raspberry Pi and Jetson Nano. Despite applying optimization techniques such as quantization and sorting, the Raspberry Pi struggled to maintain real-time performance, especially when handling high-resolution input or processing multiple items in a random environment. In scenarios with high object density, the system occasionally experiences frame drops, affecting the overall detection accuracy and speed.

Additionally, the Jetson Nano, performing better than the Raspberry Pi, faced its own problems, especially with thermal trotting. The prolonged use of the device under heavy computational burden led to overheating, reducing its processing capabilities. To reduce this, an active cooling system was needed, which increased the complexity of deployment. Furthermore, quantization and harvesting, although effective in reducing the size of the model and increasing the speed of estimation, introduce a tradeoff especially when smaller or fewer opposing objects are detected.

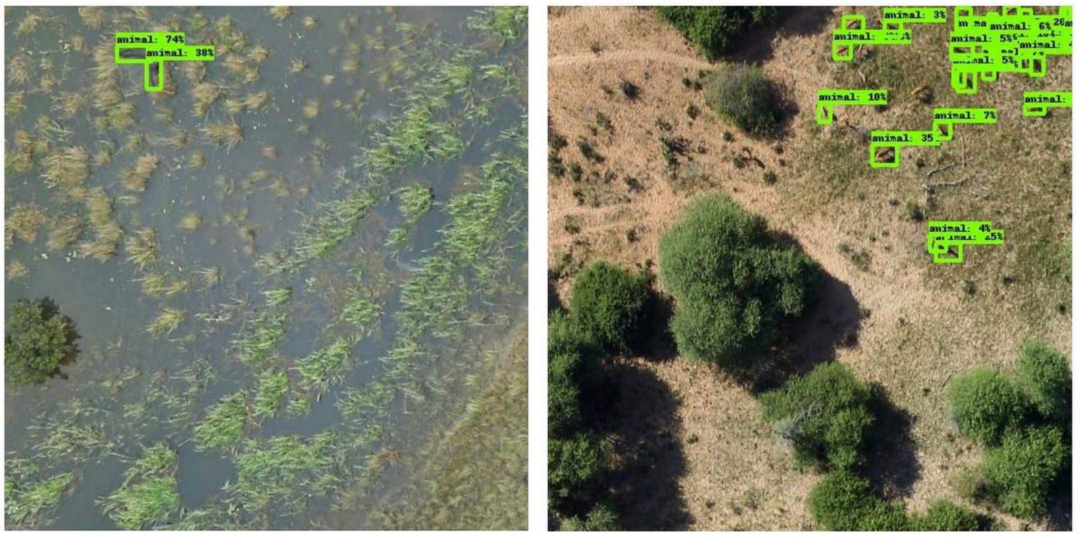


Figure 9. Wildlife detection using a drone equipped with YOLOv4-tiny. The left image shows animal detection in a wetland environment, while the right image demonstrates detection in a dry, forested area. Bounding boxes with confidence scores highlight the detection.

These challenges highlighted the delicate balance between model accuracy and optimization for embedded systems. Another important issue concerned the compatibility and integration of different software and hardware components. Different versions of libraries, such as Tensor Flowlight, Tensor RT, and Open CV, occasionally cause inconsistencies in versions, causing delays and requiring manual intervention. Network instability during real-time video processing, especially when using external IP cameras, also poses a challenge, as delays and connection drops disrupted data flow, affecting overall performance. These constraints provided valuable lessons in the need for robust hardware-software integration to ensure consistent and reliable operation in system design, optimization, and real-world deployments.

9. Conclusions

In this study, the researcher evaluated the feasibility of deploying real-time object detection models, ULOV4-small and mobile Net-SSD, on embedded platforms such as the Raspberry Pi and Nvidia Jetson Nano. Studies show that with appropriate optimization techniques, such as quantization, sorting, and hardware speed, it is possible to detect objects in real time on systems that limit these resources. The results showed that YOLOV4-Small performed best in detecting medium-to-large objects, while Mobile Net-SSD performed exceptionally well in real-time processing, especially in dynamic environments. The Jetson Nano overtook the Raspberry Pi with its GPU high-speed capabilities, achieving significantly higher FPS, making it more suitable for high-performance tasks.

The Raspberry Pi, on the other hand, although limited by its CPU, still demonstrates feasibility for lightweight applications such as basic monitoring systems. Optimization techniques, especially quantization, were important in reducing model size and improving estimation speeds without compromising accuracy. However, challenges such as low accuracy in low light conditions, object inhibition, and the need for real-time processing under various environmental factors highlighted the limitations of existing models and systems.

For future work, several directions can be explored to further enhance the performance and applicability of object detection systems on embedded platforms. First, expanding the datasets to include images taken under different environmental conditions, such as low light and random settings, will help improve model robustness. Adding additional sensors such as infrared or thermal cameras can also increase detection in difficult situations. More optimization techniques, such as knowledge dissimilarity and model ensemble methods, can potentially increase accuracy while maintaining real-time performance. Additionally, the integration of Edge AI and multi-device collaboration offers a promising way to distribute computational load, increase system efficiency, and scalability. Finally, addressing the power consumption and thermal management of embedded devices, especially in long-term deployment, will be necessary to ensure reliability and efficiency in real-world applications. This development could expand the scope of embedded AI systems in a wide range of sectors, including smart cities, healthcare and industrial automation.

Author Contributions: Author has contributed significantly to the development and completion of this article.
Funding: This article received no external funding.
Data Availability Statement: Not applicable.
Acknowledgments: The author would likes to express their sincere gratitude to College of Electronic Technology Bani Walid, and Bani Waleed University, Libya, for their invaluable support and resources throughout the course of this research.
Conflicts of Interest: The author(s) declare no conflict of interest.

ORCID
Kareemah Abdulhaq https://orcid.org/0009-0002-7987-2102
Abdussalam Ali Ahmed https://orcid.org/0000-0002-9221-2902

References

[1] Y. Feng *et al.*, “Application of artificial intelligence-based computer vision methods in liver diseases: a bibliometric analysis,” *Intell. Med.*, 2025.
[2] M. Khaleel, A. Jebrel, and D. M. Shwehdy, “Artificial intelligence in computer science,” *Int. J. Electr. Eng. and Sustain.*, pp. 01–21, 2024.
[3] H. Lindroth *et al.*, “Applied artificial intelligence in healthcare: A review of computer vision technology application in hospital settings,” *J. Imaging*, vol. 10, no. 4, 2024.
[4] A. Ettalibi, A. Elouadi, and A. Mansour, “AI and computer vision-based real-time quality control: A review of industrial applications,” *Procedia Comput. Sci.*, vol. 231, pp. 212–220, 2024.
[5] S. U. Islam, G. Ferraioli, V. Pascazio, S. Vitale, and M. Amin, “Performance analysis of YOLOv3, YOLOv4 and MobileNet SSD for real time object detection,” *TST*, vol. 5, no. 2, pp. 37–49, 2024.
[6] R. K. Mandava, H. Mittal, and N. Hemalatha, “Identifying the maturity level of coconuts using deep learning algorithms,” *Mater. Today*, 2023.
[7] H. Yang, J. Jing, Z. Wang, Y. Huang, and S. Song, “YOLOV4-TinyS: a new convolutional neural architecture for real-time detection of fabric defects in edge devices,” *Text. Res. J.*, vol. 94, no. 1–2, pp. 49–68, 2024.
[8] M.-F. Lei *et al.*, “Intelligent recognition of joints and fissures in tunnel faces using an improved mask region-based convolutional neural network algorithm,” *Comput.-aided Civ. Infrastruct. Eng.*, vol. 39, no. 8, pp. 1123–1142, 2024.
[9] I. Shukhratov, A. Pimenov, A. Stepanov, N. Mikhailova, A. Baldycheva, and A. Somov, “Optical detection of plastic waste through computer vision,” *Intelligent Systems with Applications*, vol. 22, no. 200341, p. 200341, 2024.
[10] T. Hu, Z. Gong, and J. Song, “Research and implementation of an embedded traffic sign detection model using improved YOLOV5,” *Int. J. Automot. Technol.*, vol. 25, no. 4, pp. 881–892, 2024.
[11] J. Redmon and A. Farhadi, “YOLOv3: An Incremental Improvement,” *arXiv [cs.CV]*, 2018.
[12] K. Rzepka, P. Szary, K. Cabaj, and W. Mazurczyk, “Performance evaluation of Raspberry Pi 4 and STM32 Nucleo boards for security-related operations in IoT environments,” *Comput. Netw.*, vol. 242, no. 110252, p. 110252, 2024.
[13] T. P. Swaminathan, C. Silver, and T. Akilan, “Benchmarking deep learning models on NVIDIA Jetson Nano for real-time systems: An empirical investigation,” *arXiv [cs.AR]*, 2024.

[14] A. S. Satyawan, P. A. Utomo, H. Puspita, and I. Y. Wulandari, “360-degree Image Processing on NVIDIA Jetson Nano,” *Internet of Things and Artificial Intelligence Journal*, vol. 4, no. 2, pp. 172–186, 2024.
[15] K. Mallikharjuna Rao, G. Saikrishna, and K. Supriya, “Data preprocessing techniques: emergence and selection towards machine learning models - a practical review using HPA dataset,” *Multimed. Tools Appl.*, 2023.
[16] Z. Wang, Z. Cheng, C. Wang, Z. Wu, S. Wang, and H. Xu, “A method for extracting physiological parameters of anesthetic patients based on video,” in *Proceedings of the 2024 4th International Conference on Bioinformatics and Intelligent Computing*, 2024.
[17] K. Aminiyeganeh, R. W. L. Coutinho, and A. Boukerche, “IoT video analytics for surveillance-based systems in smart cities,” *Comput. Commun.*, vol. 224, pp. 95–105, 2024.
[18] P. Mittal, “A comprehensive survey of deep learning-based lightweight object detection models for edge devices,” *Artif. Intell. Rev.*, vol. 57, no. 9, 2024.
[19] Y. Yang, L. Li, G. Yao, H. Du, Y. Chen, and L. Wu, “An modified intelligent real-time crack detection method for bridge based on improved target detection algorithm and transfer learning,” *Front. Mater.*, vol. 11, 2024.
[20] H. Lu, K. Liu, W. Sun, and P. A. Simionescu, “Precise soil coverage in potato planting through plastic film using real-time image recognition with YOLOv4-tiny,” *Sci. Rep.*, vol. 14, no. 1, p. 16817, 2024.

[21] N. Al Musalhi, A. M. Al Wahaibi, and M. Abbas, “Implementing real-time visitor counter using surveillance video and MobileNet-SSD object detection: The best practice,” *Baghdad Sci. J.*, vol. 21, no. 5(SI), p. 1775, 2024
[22] K. N. Bromm, I.-M. Lang, E. E. Twardzik, C. L. Antonakos, T. Dubowitz, and N. Colabianchi, “Virtual audits of the urban streetscape: comparing the inter-rater reliability of GigaPan® to Google Street View,” *Int. J. Health Geogr.*, vol. 19, no. 1, p. 31, 2020.
[23] T. Hayashi, T. Shimizu, and Y. Fukami, “Collaborative problem solving on a data platform Kaggle,” *arXiv [cs.CY]*, 2021.
[24] M. Khaleel, A. A. Ahmed, and A. Alsharif, “Artificial Intelligence in Engineering,” *Brilliance: Research of Artificial Intelligence*, vol. 3, no. 1, pp. 32–42, 2023.

Appendices

Code for Loading Yolo Model

```
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt

 #Paths to YOLO configuration and weights files
config_path = '/kaggle/input/yolov4-object-detection/darknet/cfg/yolov4.cfg'
weights_path = '/kaggle/input/yolov4-object-detection/darknet/yolov4.weights'

 #Load YOLO model
net = cv2.dnn.readNetFromDarknet(config_path, weights_path)

 #Set preferable backend and target
net.setPreferableBackend(cv2.dnn.DNN_BACKEND_OPENCV)
net.setPreferableTarget(cv2.dnn.DNN_TARGET_CPU)
print("YOLO model loaded successfully!")

 #Load COCO classes
```

```
classes_path = '/kaggle/input/yolov4-object-detection/darknet/data/coco.names'
with open(classes_path, 'r') as f:
    classes = [line.strip() for line in f.readlines()]
print(f"Loaded {len(classes)} classes.")

 #Get the output layers names once (this is required for forward pass)
layer_names = net.getLayerNames()
output_layers = [layer_names[i[0] - 1] for i in net.getUnconnectedOutLayers()]

 #Directory containing COCO dataset images
image_dir = '/kaggle/input/2017-2017/val2017/val2017'/
image_files = os.listdir(image_dir)

 #Process each image in the dataset
for image_file in image_files:
    image_path = os.path.join(image_dir, image_file)
    image = cv2.imread(image_path)
    if image is None:
        print(f"Error loading image: {image_file}")
        continue

 #    Preprocess the image
    blob = cv2.dnn.blobFromImage(image, scalefactor=1/255.0, size=(416, 416),
                              swapRB=True, crop=False(
    net.setInput(blob)

 #    Perform detection
    outputs = net.forward(output_layers)

 #    Parse the detections
    height, width = image.shape[:2]
    boxes, confidences, class_ids[] ,[] ,[] =
    for output in outputs:
        for detection in output:
            scores = detection[5:]
            class_id = np.argmax(scores)
            confidence = scores[class_id]
            if confidence > 0.5:  # Confidence threshold
                box = detection[0:4] * np.array([width, height, width, height])
                centerX, centerY, w, h = box.astype("int")
                x = int(centerX - (w / 2))
                y = int(centerY - (h / 2))
                boxes.append([x, y, int(w), int(h)])
                confidences.append(float(confidence))
                class_ids.append(class_id)

 #    Apply Non-Maximum Suppression
    indices = cv2.dnn.NMSBoxes(boxes, confidences, 0.5, 0.4)
    if len(indices) > 0:
        filtered_boxes = [(boxes[i], confidences[i], class_ids[i]) for i in indices.flatten()]
    else:
        filtered_boxes[] =

#   Draw bounding boxes and labels on the image
    for box, confidence, class_id in filtered_boxes:
        x, y, w, h = box
        label = f"{classes[class_id]}: {confidence:.2f}"
        color = (0, 255, 0)
        cv2.rectangle(image, (x, y), (x + w, y + h), color, 2)
        cv2.putText(image, label, (x, y - 10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2(

#   Display the image with detections
    plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
    plt.axis("off")
    plt.title(image_file)
    plt.show()
```