Performance Evaluation of the Object Detection Algorithms on Embedded Devices

Kasra Aminiyeganeh

A Thesis in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the

Requirements for the Degree of Master of Science

at Concordia University

Montreal, Quebec, Canada

August 2023

© Kasra Aminiyeganeh, 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis

prepared By: Kasra Aminiyeganeh

Entitled: Performance Evaluation of the Object Detection Algorithms on Embedded Devices

and submitted in partial fulfillment of the requirements for the degree of

Master of Science (Electrical and Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

________________________ Chair
Dr. Yan Liu

________________________ External Examiner
Dr. Jamal Bentahar

________________________ Internal Examiner
Dr. Yan Liu

________________________ Supervisor
Dr. Rodolfo Coutinho

Approved by ________________________
Dr. M. Zahangir Kabir, Graduate Program Director

October 15, 2023

____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Performance Evaluation of the Object Detection Algorithms on Embedded Devices

Kasra Aminiyeganeh

Edge computing has seen a dramatic rise in demand, driven by the necessity for real-time, low-latency applications across various domains from autonomous vehicles to surveillance systems. Among these, real-time object detection stands as a crucial technology. However, the inherent constraints of edge devices, including limited computational power, present significant challenges.

This thesis provides a comprehensive evaluation of several Convolutional Neural Networks based object detection models when deployed on resource-constrained edge devices, specifically Raspberry Pi and Google's Coral TPU. The models examined include EfficientDet, YOLO, and variants of the MobileNet family combined with SSD for object detection tasks.

We developed a novel benchmarking framework that allowed the evaluation of these models under different configurations, enabling an accurate assessment of their performance characteristics. The benchmarking framework and the metrics used for evaluation can provide a foundation for future work, focusing on the design and deployment of efficient real-time object detection models on edge devices.

The performance of these models was scrutinized based on an exhaustive set of metrics including processing speed (frames per second), model accuracy (F1 score), energy consumption, CPU utilization, memory footprint, and device temperature. A novel benchmarking framework was developed to evaluate these models under diverse configurations, providing a precise assessment of their respective performance characteristics.

This benchmarking framework, along with the evaluation metrics, sets the foundation for future research concentrating on the design and deployment of efficient real-time object detection models on edge devices. The findings of this study underscore the fact that no single model is a universal solution for all edge applications; instead, the choice of model is heavily dependent on the specific requirements and constraints of the given application.

By offering a detailed overview of the performance traits of each model, we aim to guide practitioners in making informed decisions when deploying object detection models in edge computing environments. This work sets the stage for future exploration in the development of more efficient and effective models for real-time object detection on edge devices.

Acknowledgements

First and foremost, I wish to express my profound gratitude to my supervisor, Prof. Rodolfo Coutinho, whose guidance, support, and expertise have been invaluable throughout the course of this research. His unwavering commitment to academic excellence and his insights have been instrumental in shaping this thesis.

I would also like to extend my deepest appreciation to my partner, who has provided me with unlimited support, encouragement, and understanding, allowing me to focus on my research and overcome the many challenges along the way. Thank you Sevda!

Special acknowledgment is reserved for my beloved mother, who, despite being abroad, went through tremendous hassles to provide for me. Her sacrifices ensured that I could remain focused on my studies.

I also wish to honor the memory of my late father, who remains a guiding light in my heart. His hard work and legacy have paved the way for my journey, and it is upon the foundation he laid that I stand today. His influence and love continue to inspire and drive me forward.

Contents

List of Figures ix

List of Tables x

List of Abbreviations xi

1 Introduction 1

1.1 Research Challenges . . . 4

1.2 Thesis Statement . . . 6

1.3 Thesis Outline . . . 7

2 Background 8

2.1 Introduction . . . 8

2.2 Embedded Devices . . . 10

2.2.1 Raspberry Pi: A Key Player in the World of Embedded Devices 11

2.2.2 Comparing Raspberry Pi and Jetson Nano: Two Powerhouses of the Embedded World . . . 12

2.3 Deep Learning and Convolutional Neural Networks . . . 13

2.3.1 Object Detection: Classifications, Bounding Boxes, and CNN-Based Detectors . . . 14

2.3.2 Classification of CNN Detectors . . . 16

2.3.3 Model Compression Techniques for CNN-based Object Detection on Embedded Devices . . . 18

2.4 Tensor Processing Units and Edge Accelerators . . . 20

2.4.1 Edge Accelerators . . . 20

2.4.2 TPUs 21

3 Literature Review 22

3.1 Introduction 22

3.2 Efficient CNN-Based Models for Embedded Devices 23

3.3 Model Benchmarking Literature 26

3.4 Conclusion 31

4 A Compendium of Cutting-edge CNN Models 33

4.1 Introduction 33

4.2 Object Detection Models Analysis 34

4.2.1 Criteria for Model Selection 34

4.2.2 MobileNet 35

4.2.3 SSD 38

4.2.4 MobileNet-SSD 39

4.2.5 EfficientDet 40

4.2.6 YOLO 43

4.2.7 MobileObjectLocalizer 45

4.3 Conclusion 45

5 Methodology 47

5.1 Introduction 47

5.2 Tools and Framework 48

5.2.1 Embedded Platform 48

5.2.2 Software Platform 54

5.3 Dataset 57

5.3.1 COCO Pre-Trained Models and Considerations for Accuracy Enhancement 58

5.4 Data Pipeline 59

5.5 Experiments 61

5.5.1 Metrics 62
5.5.2 Pi Configuration 64
5.5.3 Google Coral Config 65
5.6 Conclusion 65

6 Evaluation and Results 67
6.1 Introduction 67
6.2 Processing rate 68
6.2.1 Evaluation 68
6.2.2 Real-Time Processing of Live Camera Feed 72
6.3 Accuracy 73
6.3.1 Evaluation 74
6.4 Energy Consumption 76
6.4.1 Evaluation 77
6.5 CPU utilization and memory footprint 79
6.5.1 Evaluation 80
6.6 Device's temperature 82
6.6.1 Evaluation 83
6.7 Evaluating Trade-offs in Object Detection Systems 84
6.7.1 FPS - Energy - Temperature Trade-off 84
6.7.2 Accuracy - Memory - CPU - Temperature Trade-off 85
6.7.3 Conclusion 86
6.8 Conclusion 87

7 Conclusion and Future Work 89
7.1 Conclusion 89
7.2 Future Works 91

Bibliography 93

List of Figures

2.1 Various object detectors [41] 16

3.1 Sqeeznet model architecture. (A) illustrates the macroarchitecture and (B) shows the fire module [51]. 24

3.2 Simplistic architecture of a typical Tiny Yolo model. 25

3.3 Simplistic architecture of ShuffleNetV2 with type 1 shuffle unit. . . . 25

4.1 Comparative Architectures of Studied Models. 37

5.1 Model Conversion from TensorFlow to Edge TPU [73] 56

5.2 Data Pipeline for Object Detection 60

6.1 Processing rate in terms of frames per second: A) 1 CPU core. B) 2 CPU cores. C) 3 CPU cores. D) 4 CPU cores. 70

6.2 FPS of the models running on Google USB-CORAL 70

6.3 Box Plot of the Processing Rates on: A) 1 CPU core. B) 2 CPU cores. C) 3 CPU cores. D) 4 CPU cores. 71

6.4 Box Plot of the Processing Rates of the models running on Google USB-CORAL 72

6.5 Energy consumption. 77

6.6 CPU utilization. 80

6.7 Gauging temperature of Coral and Pi. 82

6.8 Processor's temperature. 83

6.9 FPS - Energy - Temperature Trade-off Triangle. 85

6.10 Accuracy - Memory - CPU - Temperature Trade-off Triangle. 86

List of Tables

3.1 Comparison of Literature Reviews on Embedded Deep Learning . . 30

6.1 Models F1 Score (%) 74
6.2 RAM usage (in MB) 81
6.3 Guideline for Model Selection Based on Various Criteria 88

List of Abbreviations

IoT	Internet of Things
CNN	Convolutional Neural Networks
MEC	Multi-access Edge Computing
HOG	Histogram of Oriented Gradients
YOLO	You Only Look Once
ROI	Regions of Interest
RPN	Region Proposal Network
ASICs	Application-Specific Integrated Circuits
SGD	Stochastic Gradient Descent
mAP	mean Average Precision
SoC	System-on-a-Chip
CPU	Central Processing Unit
GPU	Graphics Processing Unit
VGGNet	Very Deep Convolutional Networks for LargeScale Image Recognition
SSD	Single Shot MultiBox Detector
FPN	Feature Pyramid Network
TPUs	Tensor Processing Units
FPS	Frames-Per-Second
IOU	Intersection over Union
SBC	Single Board Computers
NCS2	Neural Compute Stick 2
DNN	Deep Neural Network
NAS	Neural Architecture Search

BiFPN Bi-directional Feature Pyramid Network

MOL MobileObjectLocalizer

TFLite TensorFlow Lite

Chapter 1

Introduction

The advancement of technology has brought about a new dawn of potent embedded devices, thus triggering the inception of an era marked by the Internet of Things (IoT) systems. These devices are increasingly sophisticated, offering users enriched experiences and unprecedented levels of interaction. They are designed with embedded video cameras and microphones, enabling gesture and voice-based user interactions with intelligent environments. This novel approach opens up possibilities for hands-free control of appliances in smart homes, machinery within industry 4.0 applications, and immersive interaction in IoT-based entertainment applications. The implications of this technological evolution are profound, shaping the landscape of IoT video-based applications, including transportation systems and smart surveillance.

IoT cameras, acting as the eyes of these systems, offload video frames to be processed on cloud computing facilities [1]. Convolutional Neural Networks (CNNs) are then utilized to learn spatial features from these images, thereby detecting, identifying, and tracking objects of interest from the received IoT video frames [2]. CNNs, with their capacity for high-level feature extraction, have proven to be indispensable for object detection tasks, leading to a revolutionary shift in the field. However, the latency and overhead incurred in IoT communication with distant cloud servers often make real-time IoT video analytics applications unfeasible, thus necessitating a shift in approach. Recently, multi-access edge computing (MEC) [3] has emerged as

a promising solution for time-sensitive IoT applications. MEC aims to deploy computation, storage, and computation resources at the network's edge, closer to the IoT devices. The idea is to allow machine learning models for object detection, identification, and tracking to be deployed at edge servers, with IoT devices offloading video frames to these servers for processing. While this approach reduces latency compared to cloud-based processing, it still does not guarantee a high frame processing rate due to the limited resources of edge servers and the latency incurred in communication. Several studies have proposed selective or partial offloading of IoT video frames to reduce latency and improve the frame processing rate [4]–[6]. Other studies have suggested distributed and collaborative edge servers to support IoT video analytics applications [7]–[9]. However, these solutions may not be sufficient to address the inherent challenges of real-time object detection in IoT systems.

Amidst these challenges, the concept of leveraging idle resources at IoT devices for performing computation-intensive tasks such as object detection and recognition has emerged as a promising approach. By performing a portion of the computations locally at the IoT device [10], [11], only part of the video frames need to be offloaded to edge and cloud servers. This approach can significantly reduce the overall latency and increase the frame processing rate. However, this approach comes with its own set of challenges. Specifically, the use of low-resolution video frames, often employed to reduce the resource demand at the embedded IoT device, can lead to decreased accuracy. Furthermore, the intensive computations can deplete the energy of energy-constrained IoT devices. Given these trade-offs, there is a pressing need for a comprehensive performance evaluation of CNN-based models on embedded devices. In this thesis, we propose a rigorous and practical methodology to benchmark CNN models on embedded devices, addressing this critical gap in the current understanding. We focus on the performance of six state-of-the-art object detection models on embedded devices, namely, Raspberry Pi and Google USB-Coral accelerator, providing a comprehensive evaluation of their performance.

Our attention is particularly directed towards the performance of CNN-based

object detection models in traffic surveillance applications. This focus is chosen considering the immense practical value and societal implications of such applications, which can significantly contribute to the design and deployment of such models in real-world applications. We evaluate the performance and trade-offs of the CNN models under varying conditions of frame resolution and video features, including day and night time, on the considered embedded devices. This thesis will establish a benchmarking system and framework to evaluate CNN-based models on embedded devices, catering specifically to real-life applications. Our comprehensive study considers the idiosyncrasies of embedded devices, which operate under substantial constraints in comparison to their high-powered counterparts. Their limitations include, but are not limited to, processing power, memory capacity, and energy consumption. Consequently, the effective deployment of complex object detection models on these devices presents a significant challenge. Therefore, our research places particular emphasis on these constraints, evaluating the trade-offs between accuracy, speed, and computational resources.

Another critical aspect our study encompasses is the role of video resolution in object detection. The use of low-resolution video frames is a common strategy to mitigate the resource demand on embedded IoT devices, but it can potentially compromise the detection accuracy. Our investigation delves into this conundrum, exploring the relationship between video resolution and detection performance.

Moreover, our research scrutinizes the influence of different video features, such as day and night time conditions, on the performance of object detection models. Environmental factors can significantly impact the effectiveness of these models, and our research aims to illuminate these dependencies. Given the diversity of available CNN architectures and their respective strengths and weaknesses, our study also undertakes a comparative analysis of multiple state-of-the-art object detection models. We implement and evaluate these models on embedded devices, thereby providing an extensive performance comparison that can guide future research and development efforts in this field.

The energy consumption of embedded devices is another crucial factor that our research takes into account. Given that many IoT devices operate on battery power, the energy efficiency of object detection models becomes a critical consideration. Our study assesses the power efficiency of the evaluated models, providing insights into their suitability for deployment on energy-constrained devices.

In summary, this thesis aims to provide a thorough, in-depth analysis of the performance of CNN-based object detection models on embedded devices. By navigating through the complexities and constraints inherent to these devices, this thesis offers a comprehensive guide for developers, researchers, and practitioners in the fields of computer vision and embedded systems. Our goal is to contribute to the ongoing effort to make intelligent applications on embedded devices more efficient, practical, and responsive, ultimately paving the way for a more interconnected and intelligent world.

1.1 Research Challenges

While the potential of CNN based object detection models on embedded devices is immense, this rapidly evolving field also faces a number of significant challenges that need to be addressed to fully realize its transformative potential. The challenges include [10], [12], [13]:

- **Limited Processing Power:** Embedded devices, due to their compact nature, often have limited processing power compared to full-fledged servers. Running complex CNN models on such devices can be computationally intensive and may exceed their processing capabilities. This may cause delays in object detection and recognition tasks, impacting the real-time performance of the system.

- **Energy Consumption:** Another critical challenge is the high energy consumption associated with running CNN models. Many embedded devices are battery-powered and have limited energy resources. The energy-intensive nature of CNNs could quickly deplete these resources, hindering the device's operation and limiting its practical deployment in various applications.
- **Model Accuracy:** The use of low-resolution video frames, often necessitated by the resource constraints of embedded devices, may result in decreased detection accuracy. While this strategy reduces the computational demand, it compromises the model's ability to accurately detect and identify objects, especially smaller or less distinguishable ones.
- **Data Privacy:** As embedded devices often process data locally, they could potentially handle sensitive data, especially in applications like smart surveillance and healthcare monitoring. Ensuring data privacy and security in these devices is a significant challenge that needs to be addressed.
- **Environmental Factors:** CNN models' performance can be affected by various environmental factors such as lighting conditions, weather, or camera angles. Designing models that are robust to these factors is a complex task, particularly when deploying them on embedded devices with limited processing capabilities.
- **Hardware-Software Co-Design:** Creating an effective hardware-software co-design can be challenging but is crucial for optimizing the performance of CNN models on embedded devices. The hardware configuration should be compatible with the software requirements of the CNN model to ensure efficient object detection and recognition tasks.
- **Lack of Standard Benchmarks:** Currently, there is a lack of standard benchmarks for evaluating the performance of CNN-based object detection models on embedded devices. Developing rigorous and comprehensive benchmarks

that consider various aspects such as detection accuracy, processing speed, energy consumption, and robustness to environmental factors is necessary to guide future research and development efforts in this field.

Addressing these challenges is crucial to harnessing the full potential of CNN-based object detection models on embedded devices. It requires a concerted effort from researchers, developers, and practitioners, spanning various disciplines including computer vision, embedded systems, machine learning, and data privacy. As we continue to make strides in this direction, we move closer to realizing a future where embedded devices with CNNs can effectively and efficiently enable intelligent, data-driven applications across various sectors.

1.2 Thesis Statement

This thesis proposes a methodology to benchmark CNN models on embedded devices. We provide a rigorous and practical framework for evaluating object detection models on embedded devices. We implement and evaluate the performance of six state-of-the-art object detection models on embedded devices, i.e., Raspberry Pi and Google USB-Coral accelerator. We focus on the performance of CNN-based object detection models in traffic surveillance, which can assist in guiding the design and deployment of such models in realworld applications. The performance and trade-offs of the CNN models are evaluated under different conditions of frame resolution and video features, i.e., day and night time, on the considered embedded devices. We believe that our evaluation framework can serve as a valuable resource for researchers and practitioners working in the fields of computer vision and embedded systems.

- The research outcomes presented in this thesis have been reviewed and accepted for presentation at the *13th ACM Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications (DIVANet'23)*.

- Further findings from this research will be submitted to the esteemed journal, *Internet of Things; Engineering Cyber Physical Human Systems*, and are currently under review.

1.3 Thesis Outline

The organization of this thesis, set into six main chapters. The initial chapter serves as an introduction, presenting the purpose and significance of this research, while also offering an overview of the problem statement. Subsequently, in Chapter 2, an exhaustive review of the current literature is conducted, surveying the landscape of advancements, contributions, and challenges in the realm of object detection using CNNs. This foundation paves the way for Chapter 3, which offers an in-depth examination of six state-of-the-art CNN models, dissecting their respective architectures, operations, and unique benefits. Following this, Chapter 4 outlines the methodologies used for benchmarking these models, detailing the implementation process on embedded devices, the nature of data employed for training and testing, as well as measures adopted to ensure the accuracy and reliability of our findings. The penultimate chapter, Chapter 5, delivers a comprehensive performance analysis of the models based on predefined metrics, allowing a comparison and appraisal of efficiency, accuracy, and practicality when running these models on embedded devices. Finally, Chapter 6 concludes the thesis by summarizing the research findings, highlighting potential avenues for future research and development. This systematic structure aims to present an exhaustive exploration of the topic and contribute valuable insights to the existing body of knowledge.

Chapter 2

Background

2.1 Introduction

In the current technological landscape, the application of CNN based object detection models on embedded devices has transformative potential across a myriad of fields. By enabling real-time processing and decision making at the edge, these models can significantly enhance the efficiency, responsiveness, and functionality of various systems. This section delves into five key application areas where embedded devices running CNNs can bring substantial value, transforming traditional operations into intelligent, data-driven processes [14]. These applications span across smart surveillance systems, autonomous vehicles, healthcare monitoring, industrial automation, and wildlife monitoring. Each of these applications underscores the significance of integrating powerful CNN models with compact, efficient embedded devices, highlighting the transformative potential of this technology:

1. **Smart Surveillance Systems:** The application of CNN-based object detection models on embedded devices revolutionizes the domain of surveillance. Security cameras in smart homes, businesses, or public spaces could identify and track objects of interest in real-time, enhancing security measures. Traditional surveillance systems suffer from the latency associated with sending video frames to distant servers for processing. However, by leveraging the power of embedded devices, object detection can occur locally, greatly reducing latency

and allowing real-time response. Moreover, the use of CNNs enables these devices to distinguish between different types of objects, providing a more nuanced understanding of the observed environment [15]–[17].

2. **Autonomous Vehicles:** Autonomous vehicles require real-time object detection to navigate safely through their environments. Embedded devices equipped with CNNs can detect, identify, and track other vehicles, pedestrians, traffic signs, and various other objects. The necessity for real-time analytics in this scenario is paramount as the safety of the vehicle and its surroundings depends on it. The use of CNN-based object detection models on embedded devices in these vehicles can provide the necessary speed and accuracy, allowing for safe, efficient autonomous navigation [18], [19].

3. **Healthcare Monitoring:** Embedded devices equipped with CNNs can also play a crucial role in healthcare monitoring systems. For instance, in elderly care, these devices can help monitor the patient's activities, detect any irregular behavior, or identify potential risks, such as falls. The real-time detection and rapid response facilitated by these devices could be critical in emergency situations. Furthermore, these devices can be used for remote patient monitoring, allowing healthcare providers to keep track of patients' health in real-time [20]–[22].

4. **Industrial Automation:** In the era of Industry 4.0, automated systems are integral to many manufacturing and assembly lines. Embedded devices running CNN-based object detection models can identify and sort different parts on a conveyor belt, detect defective products, or guide robotic arms. The accuracy and speed provided by these models allow for a high degree of precision and efficiency, which are paramount in industrial settings. The ability to process data locally on the device reduces latency, enabling real-time decision making, which can significantly enhance the overall productivity [23], [24].

5. **Wildlife Monitoring:** Embedded devices equipped with CNNs can be used in wildlife monitoring systems to track and study animal behavior without human intervention. Cameras placed in natural habitats can detect and identify different species, track their movements, and even monitor their behaviors. This allows researchers to gather data with minimal disturbance to the animals. The use of embedded devices ensures that the data can be processed locally, reducing the need for large data transmissions and lowering the energy consumption, which is crucial in remote or inaccessible locations [25]–[27].

In all these applications, the significance of embedded devices and CNNs is paramount. These devices provide the computational platform for CNNs to perform complex object detection tasks right at the edge, where the data is generated. This eliminates the need to transmit large amounts of data to distant servers, thus reducing latency and enabling real-time analytics. At the same time, the use of CNNs ensures high detection accuracy, which is crucial in all these applications. Therefore, the combination of embedded devices and CNNs promises to revolutionize various sectors, paving the way for smarter, more efficient, and responsive systems.

2.2 Embedded Devices

Embedded devices are specialized computing systems designed to perform dedicated functions within larger systems. They range from everyday devices such as smartphones and smartwatches to specialized equipment like IoT sensors and autonomous vehicles' onboard computers. These devices, due to their compact size, portability, low cost, and low power consumption, are increasingly being used for object detection tasks.

Implementing object detection models on embedded devices brings the power of deep learning to the edge [10], enabling real-time analytics and decision-making. However, this also presents a unique set of challenges. Embedded devices, due

to their small size, have limited computational resources and memory. Running complex CNN models on such devices requires careful consideration of these constraints, necessitating efficient model design and optimization techniques.

The synthesis of deep learning, particularly CNNs, and embedded devices in the realm of object detection is a dynamic and promising field. Despite the challenges, the successful integration of these technologies holds the potential to revolutionize a host of sectors, making them smarter, more efficient, and responsive.

2.2.1 Raspberry Pi: A Key Player in the World of Embedded Devices

The Raspberry Pi is a series of small single-board computers developed in the United Kingdom by the Raspberry Pi Foundation. Since its inception in 2012, it has become a significant player in the field of embedded devices due to its affordable price, low power consumption, and good computational capabilities [28].

- **Architecture and Performance:**

  The Raspberry Pi boasts a Broadcom system on a chip (SoC) with an integrated ARM compatible central processing unit (CPU) and on-chip graphics processing unit (GPU). The latest models, such as the Raspberry Pi 4 Model B, offer even more power, with a quad-core CPU, up to 8GB of RAM, and full gigabit Ethernet, making them capable of running a variety of complex tasks, including machine learning algorithms.

- **Applications of Raspberry Pi in Object Detection:**

  In the context of IoT, Raspberry Pi devices are used for smart traffic surveillance, enabling real-time vehicle and pedestrian detection, which can greatly contribute to traffic safety and management. Furthermore, with the right object detection models, Raspberry Pi can also find its application in autonomous

robotics and drones, helping these devices navigate their environment and interact with objects around them.

In industry applications, Raspberry Pi is used for quality control and fault detection. By implementing object detection algorithms, these devices can identify defective products or anomalies in a production line in real-time, thus reducing costs and improving efficiency.

- **Raspberry Pi and CNN-based Object Detection:**

  With the appropriate software libraries and tools, a Raspberry Pi can run various CNN-based object detection models. While these models may need to be optimized or compressed to run effectively on a Raspberry Pi due to its resource constraints, the availability of hardware accelerators and software tools such as TensorFlow Lite can further facilitate this process.

  In conclusion, the Raspberry Pi, with its balance of cost, power, and size, presents a compelling platform for deploying CNN-based object detection models in embedded devices, contributing significantly to the growth and proliferation of edge computing and IoT systems.

2.2.2 Comparing Raspberry Pi and Jetson Nano: Two Powerhouses of the Embedded World

While Raspberry Pi has been a long-standing favorite in the realm of single-board computers, NVIDIA's Jetson Nano is a newer player that has made a significant impact, especially in the world of AI and edge computing. These two platforms are often compared due to their shared market segment, but they cater to different needs and use cases due to their unique capabilities [29].

- **Architecture and Performance:**

  Jetson Nano, similar to Raspberry Pi, is a single-board computer, but it is specifically designed for AI and edge computing. It is equipped with a quad-core ARM A57 CPU, a 128-core NVIDIA Maxwell GPU, and 4GB of LPDDR4 RAM. In terms of raw performance, the Jetson Nano outpaces the Raspberry Pi, especially when it comes to tasks that can leverage the GPU, such as running deep learning models.

- **AI and Deep Learning Capabilities:**

  While both Raspberry Pi and Jetson Nano can run CNN-based object detection models, the Jetson Nano stands out with its superior deep learning capabilities. The Nano's GPU allows it to execute computationally intensive deep learning tasks much more efficiently than the Raspberry Pi. Moreover, it comes with the CUDA architecture, which enables parallel computing on the GPU. It also has native support for the NVIDIA's deep learning stack, including libraries and tools such as CUDA, cuDNN, and TensorRT.

- **Power Consumption and Cost:**

  The Raspberry Pi's main advantage over the Jetson Nano lies in its lower power consumption and cost. Raspberry Pi is more energy-efficient and can run on a smaller power supply. It is also significantly cheaper, making it a better option for applications where cost is a key consideration. However, for use-cases where AI and deep learning performance is critical, the Jetson Nano, despite its higher cost and power consumption, would be the preferred choice due to its superior computational capabilities.

2.3 Deep Learning and Convolutional Neural Networks

Deep learning [30], a subset of machine learning, is inspired by the structure and function of the human brain. It uses artificial neural networks with multiple layers

(hence the term 'deep') to model and understand complex patterns in data. Among various deep learning architectures, CNNs have become synonymous with image processing tasks due to their unique design tailored to handle grid-like data, including images.

A CNN [31] is composed of several layers, each playing a specific role in the transformation and extraction of features from the input data. The key components of a CNN include convolutional layers, pooling layers, and fully connected layers. Convolutional layers apply a series of filters to the input data, allowing the network to learn spatial hierarchies of features. Pooling layers, on the other hand, reduce the spatial size of the representation, controlling overfitting and reducing computational complexity. Finally, the fully connected layers perform high-level reasoning on the extracted features and make the final prediction.

2.3.1 Object Detection: Classifications, Bounding Boxes, and CNN-Based Detectors

The fundamental goal of object detection is to classify and spatially locate multiple objects belonging to different classes within an image simultaneously. The output of object detection typically includes class categories and a set of bounding boxes (Bboxes) that denote the object locations. This dual requirement of classification and localization makes object detection a more complex task than image classification.

CNN-based detectors have emerged as a powerful tool for object detection. These detectors generally comprise two major components: the feature extraction network, often referred to as the backbone, and the detection head [32].

Feature Extraction Backbone

The backbone of a CNN-based detector consists of stacked convolutional layers and pooling layers. The role of the backbone is to process the input image and extract

a series of feature maps. These feature maps capture the spatial hierarchies of features, with early layers capturing low-level features such as edges and textures, and deeper layers capturing high-level semantic features.

Several architectures have been proposed for the backbone, each with its own strengths and trade-offs. For instance, Very Deep Convolutional Networks for Large-Scale Image Recognition (VGGNet) [33], known for its simplicity and depth, was among the earliest backbones used in CNN-based detectors. However, due to its computational intensity, later models such as ResNet [34], with its innovative residual connections, and MobileNet [35], designed specifically for mobile and embedded vision applications, have been adopted. These backbones offer a balance between computational efficiency and detection performance, making them suitable for embedded devices.

Detection Framework and Detection Head

The detection framework includes the usage of the backbone and the process of manipulating feature maps before and within the detection head. The detection head takes the feature maps produced by the backbone and generates the final bounding boxes for object detection.

Early detection frameworks, such as R-CNN [36] and its successors Fast R-CNN [37] and Faster R-CNN [38], introduced the concept of region proposals. These methods first proposed potential object locations and then used a CNN to classify these regions and refine their bounding boxes. However, these methods were computationally intensive and relatively slow.

To address these limitations, single-stage detectors like YOLO (You Only Look Once) [39] and SSD (Single Shot MultiBox Detector) [40] were proposed. These methods eliminated the need for region proposals and directly predicted the class and bounding box coordinates from the feature maps, significantly improving the speed of detection.

The evolution of both the backbone and the detection framework has led to improvements in object detection performance. Despite these advancements, object detection on embedded devices remains challenging due to their limited computational resources. Therefore, the quest for more efficient and accurate object detection models continues, with research focusing on creating compact yet powerful models suitable for deployment on embedded devices.

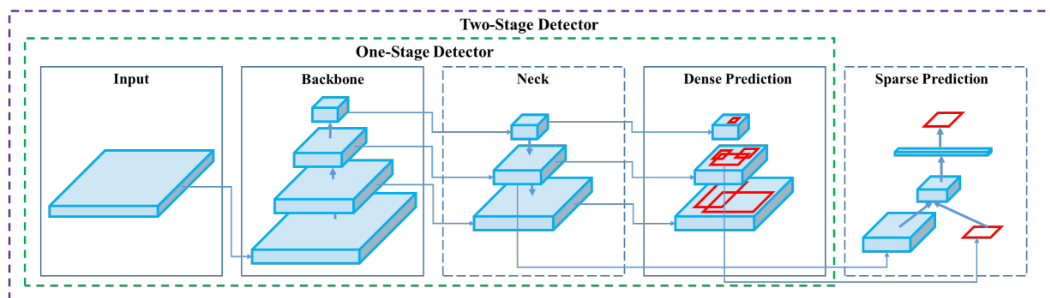


FIGURE 2.1: Various object detectors [41]

2.3.2 Classification of CNN Detectors

CNN-based detectors can be classified based on the number of stages in the detection process and the usage of anchors, a mechanism for predicting regions of interest (ROI) in the image [32], [42].

Number of Stages

- **Two-Stage Detectors:** The most classic and representative two-stage detectors include the R-CNN family [36]–[38]: R-CNN, Fast R-CNN, and Faster R-CNN. In the first stage, these detectors search for the ROI in the image.

  In the second stage, they perform classification and localization based on the results from the first stage. While these two-stage detectors demonstrate promising accuracy, they are slow in inference speed due to the two-step process.

- **One-Stage Detectors:** To improve the inference speed, one-stage detectors like YOLO were proposed [39]. These detectors predict the class and location simultaneously, eliminating the need to search for ROIs first.

  While the first version of YOLO achieved real-time detection, it suffered from lower accuracy due to the overwhelming negative samples. To achieve a balance between accuracy and speed, methods like RetinaNet were introduced, which incorporated the Feature Pyramid Network (FPN) and a novel loss function called Focal Loss to alleviate the class imbalance problem. Fig 2.1 illustrates various models.

Anchor Usage

- **Anchor-Based Detectors:** The concept of 'anchor' was introduced in the Region Proposal Network (RPN) of Faster R-CNN [38]. Anchors are pre-defined boxes with specific scales and aspect ratios that are used as references for predicting ROIs. Since its introduction, the anchor mechanism has been widely used in two-stage detectors like Faster R-CNN and Mask R-CNN [43], as well as in one-stage detectors like SSD and YOLO-v2.

- **Anchor-Free Detectors:** However, there are also detectors that avoid the use of anchors. For instance, YOLO-v1 [39] divided the image into grids to generate predictions, creating an anchor-free detection pipeline. Similarly, DeNet [44] predicted the four key points of the target to avoid the use of the anchor mechanism.

This classification of CNN-based detectors based on the number of stages and anchor usage provides a structured understanding of the various approaches taken in object detection. Each method has its strengths and trade-offs, and the choice of method depends on the specific requirements of the application, including the need for real-time detection, the acceptable level of accuracy, and the computational resources available, especially in the context of embedded devices.

2.3.3 Model Compression Techniques for CNN-based Object Detection on Embedded Devices

Integer Quantization

Integer quantization [45] is a technique that reduces the precision of weights and biases in the neural network to integer values. The significant advantage of integer quantization is the reduction in the model size and memory footprint, as well as faster inference times. It helps mitigate the memory and computational constraints of embedded devices while retaining high detection accuracy.

One downside of integer quantization is that it may slightly degrade the model's accuracy because of the reduced precision. In terms of implementation, it requires careful scaling to ensure the values fit within the new integer range without significant information loss.

In many use cases, the benefits of integer quantization outweigh the marginal accuracy loss. For example, in real-time object detection on embedded devices where low latency and high processing speed are crucial, integer quantization is highly beneficial. Furthermore, the lower memory footprint allows for deployment on devices with limited memory resources.

Pruning

Pruning is another technique for compressing CNN models [46]. It involves the removal of certain parts of a neural network, such as neurons, layers, or weights, that contribute little to the model's output. This process reduces the model's complexity and size.

Pruning's advantage lies in its ability to significantly reduce the model's size without drastically affecting the model's accuracy. However, the disadvantage is that it might lead to irregular memory access patterns, which could negate the computational benefits.

Knowledge Distillation

Knowledge distillation [47] involves training a smaller, student model to mimic the behavior of a larger, teacher model. The student model learns to approximate the function learned by the teacher model, thereby compressing the model while maintaining similar performance.

The primary advantage of knowledge distillation is that it can achieve a high degree of compression with minimal impact on model performance. However, the process can be computationally expensive and time-consuming as it involves training two models.

Weight Sharing

Weight sharing [48] is a technique that groups weights with similar values and assigns them a common value, reducing the number of unique weight values and hence the model's size. Weight sharing can significantly reduce the model's size, but it may lead to performance degradation due to the loss of unique weight values.

Comparison and Choice for this Thesis

Each of these techniques has its strengths and trade-offs. Integer quantization and pruning reduce model size and improve inference speed, but may slightly degrade accuracy. Knowledge distillation maintains performance but can be computationally expensive. Weight sharing reduces model size but might lead to performance degradation.

Given the specific requirements of this thesis, which include real-time object detection on embedded devices with limited computational resources and memory, integer quantization is the chosen compression technique. It offers the optimal trade-off between model size, computational efficiency, and detection accuracy. Moreover, the slight degradation in accuracy due to integer quantization is often acceptable in real-world applications where speed and efficiency are paramount.

2.4 Tensor Processing Units and Edge Accelerators

In the context of object detection on embedded devices, TPUs (Tensor Processing Units) and edge accelerators play a pivotal role. With their ability to perform high-speed, low-precision computations, TPUs can significantly accelerate the execution of CNN-based object detection models, enabling real-time object detection. On the other hand, edge accelerators enable the deployment of these models on embedded devices at the edge of the network. By performing computations locally, they not only reduce latency but also help conserve bandwidth, which is often a critical resource in IoT systems. Additionally, they help address privacy concerns by keeping data on the device instead of transmitting it to the cloud.

In conclusion, TPUs and edge accelerators provide the computational power and efficiency needed to run CNN-based object detection models on embedded devices, making them indispensable tools in the advancement of edge AI and IoT applications.

2.4.1 Edge Accelerators

Edge accelerators [49] are specialized hardware devices designed to accelerate machine learning tasks at the edge of the network, i.e., closer to the data source. These devices, which include Google's Edge TPU, Intel's Movidius Neural Compute Stick, NVIDIA's Jetson series, and others, are designed to process AI algorithms on-device, enabling real-time data processing without the need for constant connectivity.

Edge accelerators are critical for applications where low latency is crucial, such as autonomous vehicles, real-time video analytics, and robotics. By performing computation on the device, they eliminate the need to send data to the cloud for processing, thereby reducing latency and preserving user privacy. The primary advantage of edge accelerators is their ability to perform real-time, on-device AI processing while using minimal power, making them ideal for use in battery-powered

or energy-constrained devices. However, like TPUs, they are application-specific and may require specialized knowledge to program effectively.

2.4.2 TPUs

TPUs are Application-Specific Integrated Circuits (ASICs) developed by Google [50] specifically for accelerating machine learning tasks. They are designed to boost the speed and efficiency of tensor operations, which are fundamental to running deep learning models, including CNNs.

TPUs are highly optimized for high-volume, low-precision computation, offering a significant speedup for machine learning models while maintaining accuracy. They are capable of processing a large amount of data in parallel, making them highly efficient for large-scale machine learning applications.

There are two key advantages of TPUs. First, they provide high computational throughput, enabling faster training and inference times for machine learning models. Second, they are highly power-efficient, providing more computations per watt compared to traditional CPUs or GPUs.

However, TPUs are not without limitations. They are specialized hardware, meaning they are not as versatile as CPUs or GPUs. Furthermore, they can be challenging to program and may not be compatible with all machine learning frameworks.

Chapter 3

Literature Review

3.1 Introduction

The performance evaluation of CNN models on embedded devices has garnered significant research attention. As embedded platforms become increasingly prevalent, understanding the capabilities and limitations of deploying deep learning algorithms on resource-constrained devices is crucial. This literature review section aims to examine key studies that have investigated the performance of CNN models on various embedded systems, such as the Jetson TK1, Raspberry Pi 3, Jetson TX2, Jetson Nano, and Jetson Xavier NX. These studies shed light on the challenges encountered and explore optimization strategies to enhance the efficiency and accuracy of CNN models on embedded devices. By synthesizing the findings from these studies, this section aims to provide a comprehensive understanding of the performance evaluation of CNN models in the context of embedded devices, facilitating informed decision-making for future deployments.

Furthermore, we will also analyze several lightweight models that are prevalently employed in current times. However, it is essential to note that we will dedicate the subsequent chapter to a comprehensive discussion of the models that we have personally utilized in our study and that are deemed as state-of-the-art within this dynamic field.

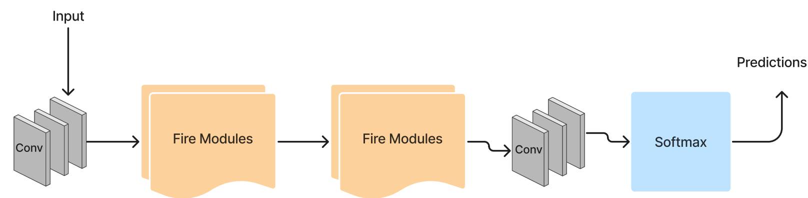
3.2 Efficient CNN-Based Models for Embedded Devices

As the scope and application of deep learning expand, the need for efficient models that can perform on devices with limited computational resources becomes increasingly critical. Embedded devices, while powerful and versatile, often lack the computing power and memory of more traditional computing systems. This necessitates the use of streamlined, efficient models that maintain a high level of performance while minimizing their computational footprint. Models like SqueezeNet and Tiny YOLO are particularly noteworthy in this context.

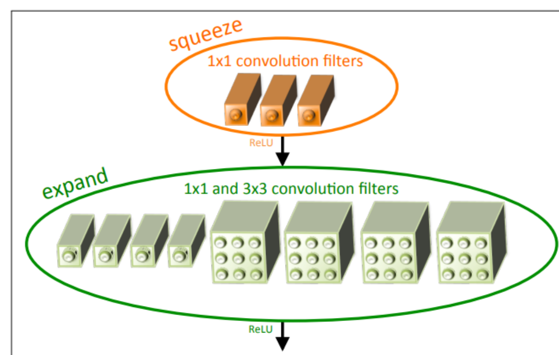
- **SqueezeNet:** SqueezeNet [51] is an innovative CNN architecture that was designed with the explicit goal of reducing model size while maintaining the level of accuracy achieved by traditional models like AlexNet [52]. The SqueezeNet architecture introduced the 'Fire Module' - a microarchitecture comprising of 'squeeze' and 'expand' layers. The squeeze layers, which consist of 1x1 filters, reduce the number of input data channels, and thus computational complexity. The expand layers use a mix of 1x1 and 3x3 filters to increase the channel depth, allowing SqueezeNet to maintain the representational capacity of the network.

  The compactness of SqueezeNet makes it particularly suitable for deployment on embedded devices where memory is often limited. With the use of deep compression techniques such as pruning, quantization, and Huffman coding, SqueezeNet can be compressed to a size less than 0.5MB, making it a very efficient choice for edge computing.

- **Tiny YOLO:** YOLO is a popular real-time object detection system that treats object detection as a single regression problem, making it faster than the two-stage approach used by other methods like R-CNN. Tiny YOLO [53] is a smaller version of the YOLO model, designed for systems with less computational power.



(A)



(B)

FIGURE 3.1: Sqeeznet model architecture. (A) illustrates the macroarchitecture and (B) shows the fire module [51].

The Tiny YOLO architecture uses fewer layers and less convolutional filters than the full YOLO model. Specifically, it consists of a few convolutional layers with leaky ReLU activations and Max-Pooling layers and a final output layer as illustrated in Fig 3.2. Despite its reduced complexity, Tiny YOLO still achieves remarkable results in real-time object detection.

Tiny YOLO is particularly suited for embedded devices due to its small size and fast inference time. While it may not achieve the same level of accuracy as larger models, it provides an excellent trade-off between size, speed, and accuracy, making it highly efficient for real-time applications on embedded devices.

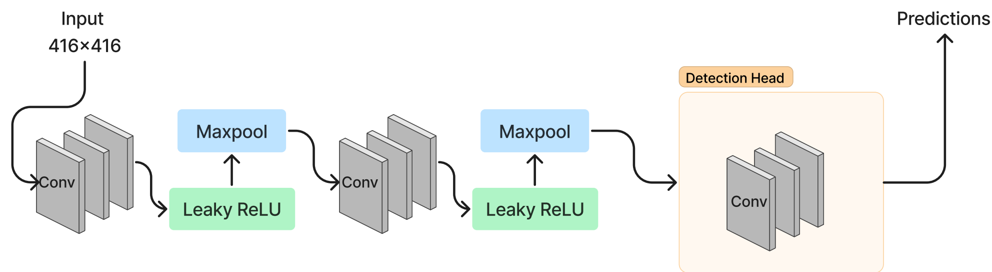


FIGURE 3.2: Simplistic architecture of a typical Tiny Yolo model.

- **ShuffleNet:** ShuffleNet [54] utilizes pointwise group convolutions and channel shuffling to greatly reduce computational cost while maintaining accuracy comparable to larger models. The use of group convolutions reduces the number of parameters, making the model smaller and more efficient.

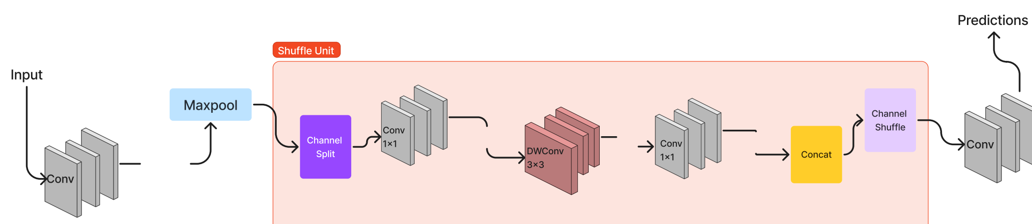


FIGURE 3.3: Simplistic architecture of ShuffleNetV2 with type 1 shuffle unit.

In conclusion, these streamlined and efficient models are pushing the boundaries of what is achievable with embedded devices, allowing for sophisticated deep learning applications in memory and compute-constrained environments.

3.3 Model Benchmarking Literature

A considerable body of research has been dedicated to the development of embedded platforms and the implementation of deep learning algorithms on these devices. One such notable study was conducted by Mao et al. [55], in which the authors successfully executed the Fast R-CNN algorithm on the Jetson TK1 platform. To accommodate the specific requirements of the TK1 platform, the Fast R-CNN algorithm underwent several alterations. Despite these modifications, the detection speed remained relatively low, achieving a rate of merely 1.85 Frames Per Second (FPS). This highlights the challenges and potential limitations of implementing complex deep learning algorithms on embedded devices.

Vidya et al. [56] investigated the performance of the EfficientDet family of CNNs on resource-constrained devices, specifically the Raspberry Pi 3. They found that integer quantization reduced the size of the model but came at the cost of decreased accuracy. Their analysis demonstrated that EfficientDet0 and EfficientDet1 were the most suitable models for deployment on a Raspberry Pi. When comparing the performance of the Stochastic Gradient Descent (SGD) and Adam optimizers, EfficientDet1 performed better with the SGD optimizer, while EfficientDet0 performed better with the Adam optimizer. The authors concluded that selecting EfficientDet1 with a moving average decay of 0.95 and the SGD optimizer is recommended when speed is not a critical factor. In contrast, when speed is essential, EfficientDet0 with a 0.9 moving average decay and the Adam optimizer is the optimal choice. The ideal Intersection over Union (IOU) threshold value for these models ranges from 0.5 to 0.95, depending on the specific application requirements. It is important to note that the findings of this study are limited to the Raspberry Pi 3 and may differ on other resource-constrained devices, such as mobile phones.

Süzen et al. [57] assessed the performance of CNNs for classifying traditional 2D images on three embedded platforms: Jetson TX2, Jetson Nano, and Raspberry

Pi. Utilizing the DeepFashion2 dataset, which contains 13 popular clothing categories and 45,000 images, the authors created five different sized data clusters to compare the performance of the CNN model on both GPU and CPU. Their results indicated that the Jetson TX2, despite having higher power consumption and cost, provided superior performance in terms of accuracy, shorter execution time, and handling larger datasets. The Raspberry Pi, although the most cost-effective option, was deemed unsuitable for deep learning applications due to its lack of NVIDIA GPU support and longer processing times. The authors concluded that selecting hardware with higher capabilities provides an advantage in accuracy and performance for deep learning applications but comes at a higher cost.

Zhu et al. [58] analyzed the performance of YOLO models, PPYOLO and YOLOv3, on embedded devices Jetson Nano and Jetson Xavier NX. PPYOLO showed better inference speed than YOLOv3 on both devices, with a slight advantage on the Xavier NX. CPU usage was higher for faster models, while memory usage varied with model input size on the Nano. Energy consumption was lower on the Xavier NX due to faster task completion. The lightweight versions, PPYOLO-tiny and YOLOv3-tiny, were about 4x faster and reduced power usage by around 20% on the Nano. The use of TensorRT for inference acceleration significantly improved performance on Jetson devices.

Liberatori et al. [59] studied the YOLOv4-Tiny based model for detecting face masks in images, optimized for low-end devices such as the Raspberry Pi 4. The model was trained on a publicly available "Mask-Detection-Dataset" with images manipulated to mimic low-fidelity environments. Techniques such as pruning and quantization were used to reduce the computational requirements of the model. A model pruned with a one-shot 60% pruning rate and statically quantized delivered the best performance, achieving a trade-off between mean Average Precision (mAP) of 0.574 and an increased frame rate of 1.97 FPS. The study's objective was to balance detection accuracy and inference speed, making it viable for real-world applications in resource-constrained environments. The study's results are comparable

with other models using YOLOv3 and YOLOv4 in terms of mAP, demonstrating its effectiveness despite its lighter footprint.

Feng et al. [60] analyzed the performance of accelerator-based Single Board Computers (SBCs) in running YOLO models for object detection, using different video frame sizes and two versions of the YOLO model. Their metrics include FPS, memory and CPU usage, and energy consumption. They find that more complex models and larger data sizes slow inference and increase memory and CPU usage, with GPU-based SBCs using significantly more memory than ASIC-based SBCs. Despite this, Jetson Xavier NX consistently outperforms other SBCs in terms of FPS, regardless of data size or YOLO model used, while Jetson Nano shows the worst performance. However, the mean confidence (average accuracy) of YOLOv3-tiny on the Raspberry Pi with Neural Compute Stick 2 (NCS2) is 0%, compared to 57.9% on the Jetson devices, suggesting the need for model adaptation to the SBC's architecture. Energy consumption mainly depends on the SBC itself; Jetson Xavier NX consumes the least energy due to its high performance and short inference time, despite its high average power usage. The authors conclude that ASIC accelerators allow low-performance SBCs to run high-performance CNN models while using minimal resources, but model adaptation is crucial. They also highlight the need for careful model selection on GPU-based SBCs to balance accuracy and speed.

Anggraini et al. [61] evaluated the performance of SSDLite MobileNetV2 and MobileNetV3 models for face mask detection on Raspberry Pi 4. Results show that the SSDLite MobileNetV2 model with fine-tuning performs best, detecting all inputs accurately. However, without fine-tuning, these models struggled to identify incorrectly used masks. The SSDLite MobileNetV3 Small model achieved the highest FPS equal to 10, indicating a faster detection speed than a two-stage mask detection system. Power consumption was found to be relatively consistent across models. In terms of accuracy, the SSDLite MobileNetV3 Small model achieved a 70% score, but struggled with detecting incorrect mask use and working under inadequate lighting conditions. Despite these limitations, the model had higher detection speed

and slightly lower power consumption compared to other models. Future research could explore different datasets, training model configurations, and camera types.

Sha et al. [42] used the CityPersons and ETH datasets to evaluate a variety of CNN-based pedestrian detectors, including FPN, RetinaNet, Faster R-CNN, FCOS, ALFNet, CSP, and BCNet, under the same computational environment. These datasets were chosen due to their diversity in image resolution, people density, and street view diversity, thus offering a comprehensive test of the detectors' generalization abilities. Experiments were conducted on a Nvidia GeForce GTX 1080 Ti GPU, without pre-training the models on any other datasets. Results indicated that the feature extraction capability of the detector's backbone, whether it be ResNet-50 or HRNet, significantly impacts accuracy. Reduced image resolution, while allowing compatibility with GPU memory, slightly decreased model accuracy. Furthermore, the study found a need to balance between hyper-parameter settings and network architectures to manage model complexity within GPU constraints. Despite some success, the study identified the generalization ability as an ongoing challenge in pedestrian detection, suggesting that future work should focus on improving this through increased dataset diversity.

Velasco et al. [62] evaluated the efficiency and effectiveness of several deep neural network (DNN) models specifically designed for embedded devices. These models included: Network in Network: This model significantly reduced the number of parameters compared to previous models like AlexNet, through the use of micro neural networks with 1x1 convolution kernels and global average pooling. GoogLeNet (Inception-v1): Winning the 2014 ImageNet Large Scale Visual Recognition Challenge, this model further reduced the number of parameters through the introduction of the Inception module, which used parallel convolutions with varying filter sizes. SqueezeNet: This model aimed to maintain the accuracy of AlexNet while drastically reducing network parameters. It achieved this through the Fire module, which used a squeeze convolution layer and an expand layer with 1x1 and

3x3 convolution filters. MobileNet: Designed for mobile and embedded vision applications, this model used depth-wise and point-wise convolutions to significantly reduce computational model parameters. It also incorporated a width multiplier and a resolution multiplier to further reduce computational cost. The performance of these models was evaluated on a Raspberry Pi, with metrics including accuracy, throughput, power consumption, and a composite Figure of Merit. The findings indicated that, while each model had its strengths, SqueezeNet provided the best balance between accuracy and efficiency with FPS of 4 and power usage of 4W.

TABLE 3.1: Comparison of Literature Reviews on Embedded Deep Learning

Authors	Devices Tested	Models	Key Findings	Metrics
Mao et al. [55]	Jetson TK1	Fast R-CNN	Successful execution on Jetson TK1 with alterations to Fast R-CNN	mAP, FPS, RAM Usage
Vidya et al. [56]	Raspberry Pi 3	EfficientDet family	Identified suitable models for Raspberry Pi 3. Recommendations for optimizer choice based on speed.	Input Size, AP
Süzen et al. [57]	Jetson TX2, Jetson Nano, Raspberry Pi	CNN (DeepFashion2 dataset)	Jetson TX2 showed superior performance in accuracy and execution time. Raspberry Pi deemed unsuitable due to long processing times.	AP, FPS, CPU/GPU Power, RAM
Zhu et al. [58]	Jetson Nano, Jetson Xavier NX	YOLO models (PPYOLO, YOLOv3)	PPYOLO showed better inference speed. Use of TensorRT improved performance on Jetson devices.	Power, Energy, Inference Time, CPU/RAM Usage
Liberatori et al. [59]	Raspberry Pi 4	YOLOv4-Tiny	Developed model for face mask detection in low-fidelity environments. Used pruning and quantization to reduce computational needs.	mAP, FPS
Feng et al. [60]	Various SBCs	YOLO models	Jetson Xavier NX outperformed others in FPS. Importance of model adaptation to SBC's architecture highlighted.	FPS, Mean confidence, CPU/RAM Usage, Power, Time

Continued on next page

Table 3.1 – continued from previous page

Authors	Devices Tested	Models	Key Findings	Metrics
Anggraini et al. [61]	Raspberry Pi 4	SSDLite MobileNetV2, MobileNetV3	Identified model with best performance for face mask detection. Noted challenges with detecting incorrect mask use.	Min/MAx FPS, Power, Fine Tuning, Accuracy
Sha et al. [42]	Nvidia GeForce GTX 1080 Ti GPU	Various CNN-based pedestrian detectors	Evaluated generalization abilities of several detectors. Emphasized the importance of feature extraction capability of the detector's backbone.	Miss Rate, Resolution, BackBones
Velasco et al. [62]	Raspberry Pi	Network in Network, GoogLeNet, SqueezeNet, MobileNet	SqueezeNet provided the best balance between accuracy and efficiency.	FPS, Power
This Work	Raspberry Pi 4, Google USB Coral Accelerator	MobileNetSSDv1, MobileNetSSDv2, EfficientDetv0, EfficientDetv2, YOLOv5, MOL	Established a through framework with five distinct metrics tailored for embedded devices.	Accuracy, FPS, Power Consumption, Temperature, CPU/RAM Usage

3.4 Conclusion

In conclusion, the performance evaluation of CNN models on embedded devices is a crucial topic in deep learning. The studies reviewed provide insights into implementing complex CNN algorithms on resource-constrained platforms, emphasizing the trade-offs between accuracy, speed, power consumption, and cost. Furthermore, hardware capabilities must be carefully considered when selecting embedded devices, and optimization techniques such as model quantization, input size, and inference acceleration play significant roles in achieving efficient and accurate inference.

The studies also highlight the potential of optimizing models for low-end devices, demonstrating the effectiveness of techniques like pruning, quantization, and model adaptation. The selection of hardware and careful model selection on GPU-based SBCs is important to balance accuracy and speed.

Evaluation of different CNN models, including specialized models like SqueezeNet,

sheds light on the impact of model architecture, feature extraction, and hyper-parameter settings on performance. Model complexity must be managed within GPU constraints, and generalization abilities remain a challenge.

The insights gained from these studies contribute to the advancement of embedded deep learning applications, providing valuable guidance for researchers and practitioners. Further research is needed to explore additional optimization strategies, evaluate emerging embedded systems, and address evolving challenges. The findings pave the way for future advancements in embedded deep learning.

Chapter 4

A Compendium of Cutting-edge CNN Models

4.1 Introduction

This chapter explores six distinct models that form the core of our investigation (i.e. MobileNetSSD V1, V2, EfficientDet V0, V2, MobileObjectLocalizer, and YOLOv5s). This next section, in essence, serves as an academic canvas where we elucidate the intricate details of each of these models, delving into their structure, functionality, and unique attributes. These models have been specifically chosen due to their notable presence in the domain of object detection, and their potential applicability within the constraints of embedded devices. We aim to provide a comprehensive understanding of these models before benchmarking their performance, a process that will be undertaken in the ensuing chapters. This structured and meticulous approach allows us to lay a solid foundation for the empirical analysis that is to follow, thereby providing a coherent and well-rounded narrative to our study.

4.2 Object Detection Models Analysis

4.2.1 Criteria for Model Selection

The process of selecting appropriate models for embedded object detection involves a multifaceted analysis of several key criteria. These criteria aim to ensure that the models not only offer high detection accuracy but are also computationally efficient, enabling their deployment in real-time applications on resource-constrained devices. Below, we outline and discuss the principal criteria that have informed our selection of the six models for this study.

Computational Efficiency

Computational efficiency is paramount, especially for embedded systems with limited processing capabilities. We evaluate the models based on model size, and inference speed. A model that offers high accuracy but is computationally intensive may not be suitable for real-time applications on embedded devices.

Detection Accuracy

Accuracy in detecting and classifying objects is another critical factor. It is measured using metrics like mean Average Precision (mAP) at various Intersection over Union (IoU) thresholds. While computational efficiency is essential, it should not come at a significant compromise to detection accuracy, as this would limit the model's practical applicability.

Versatility and Flexibility

The selected models should be versatile and flexible, capable of detecting a wide range of objects in various environmental conditions. This includes the ability to handle different lighting conditions, object orientations, and occlusions. The model's

adaptability to different application domains and scenarios is also a vital consideration.

Scalability

Scalability refers to the model's ability to maintain performance when scaled down to fit the constraints of embedded devices or scaled up to improve accuracy for more complex tasks. This involves analyzing how changes in model size, complexity, and computational requirements affect its detection performance.

Robustness

Robustness is the model's ability to maintain high performance in real-world scenarios, including noisy and dynamic environments. It also involves the model's resilience to adversarial attacks and its performance stability under various conditions.

Ease of Implementation

Practical implementation aspects, including the availability of pre-trained models, training data requirements, and ease of fine-tuning, are also crucial. Models that are easier to implement and adapt to specific tasks can expedite the development process and facilitate their deployment in real-world applications.

4.2.2 MobileNet

The MobileNet family of detectors, a well-regarded and widely implemented series of CNN models, are famed for their focus on providing high-performance object detection capabilities with reduced computational complexity, ideal for deployment on resource-constrained devices. These models, developed by Google researchers, have seen several iterations with the release of MobileNetV1, MobileNetV2, and

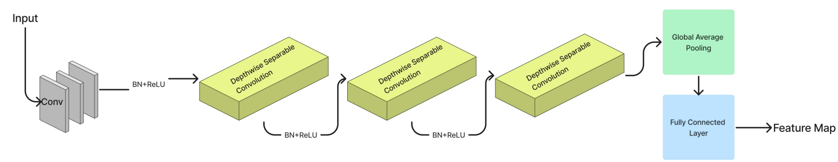
MobileNetV3, each improving upon the last in various aspects, particularly computational efficiency, accuracy, and speed.

MobileNetV1

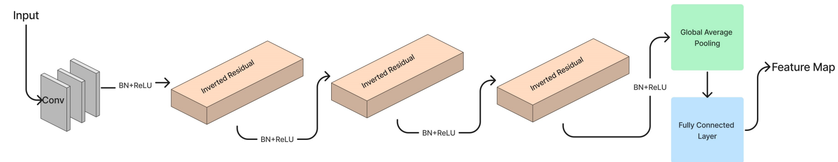
The primary architecture of MobileNetV1 [35] is characterized by the introduction of depthwise separable convolutions as illustrated in Fig. 4.1a, a significant departure from the traditional convolution operation used in most CNNs. This innovative design choice breaks down the standard convolution into two parts: a depthwise convolution and a pointwise convolution. The depthwise convolution applies a single filter per input channel, while the pointwise convolution uses a 1x1 convolution to build new features through computing combinations of input channels. This factorization into simpler operations significantly reduces the computational load, enabling the deployment of the model on devices with limited computational power while maintaining reasonable accuracy levels.

MobileNetV2

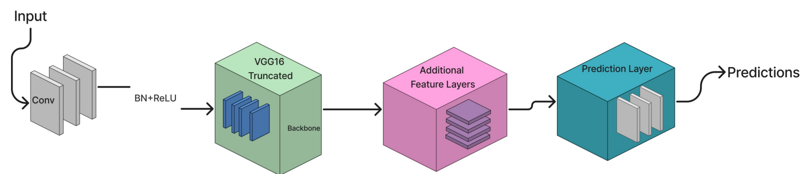
Building upon the architecture of MobileNetV1, MobileNetV2 [63] introduces two significant architectural elements – the inverted residual structure and the linear bottleneck as illustrated in Fig. 4.1b. The inverted residual structure essentially inverts the traditional residual block design, where the input undergoes a 1x1 expansion convolution to increase the number of channels, followed by a 3x3 depthwise convolution, and finally a 1x1 projection convolution to reduce the channel dimensions back down. The linear bottleneck, on the other hand, serves to preserve the information throughout the network. The authors argued that non-linearities in the final layers could destroy some critical information, hence a linear activation function was used instead. These two features lead to a substantial increase in efficiency and performance over MobileNetV1.



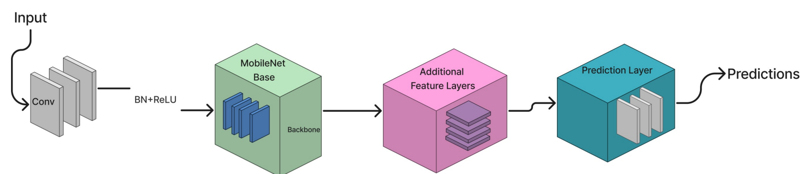
(A) MobileNetV1 Architecture.



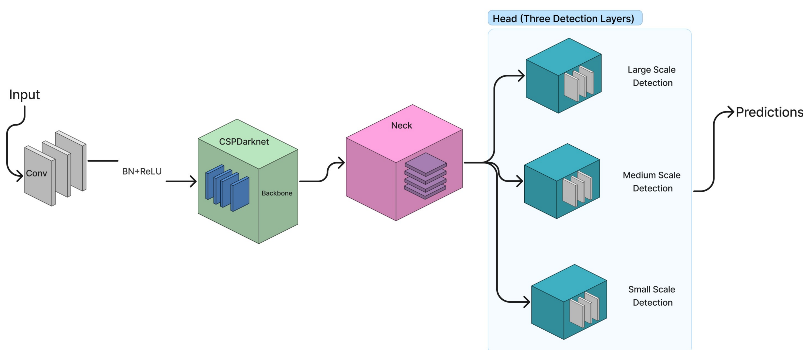
(B) MobileNetV2 Architecture.



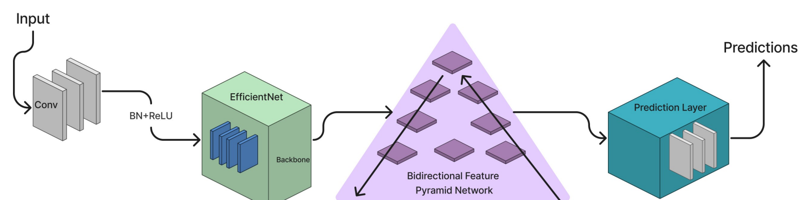
(C) SSD Architecture.



(D) MobileNetSSD Architecture.



(E) YOLO Architecture.



(F) EfficientDet Architecture.

FIGURE 4.1: Comparative Architectures of Studied Models.

4.2.3 SSD

SSD [40] is an object detection model that combines the advantages of both region proposal-based (two-stage) methods and regression/classification-based (one-stage) methods. Its structure includes a base network followed by several convolutional layers, making it one of the prominent single-shot object detectors.

Due to its balance between speed and accuracy, SSD has a wide range of use cases, including surveillance systems, autonomous vehicles, robotics, and any application requiring real-time object detection.

While SSD is not as lightweight as MobileNet or SqueezeNet, it still offers a reasonable compromise between accuracy and computational efficiency, making it suitable for deployment on resource-constrained devices, such as the Raspberry Pi. Furthermore, SSD can be combined with lightweight models (e.g., MobileNet) as the base network to form variants such as MobileNet-SSD, which can further optimize its performance on such devices.

Even on a Raspberry Pi, an SSD model can offer real-time object detection performance, although the exact frame rate may vary depending on the specific SSD variant and the complexity of the task at hand. Quantization and other model compression techniques can also be applied to the SSD model to improve its efficiency on resource-constrained devices further.

In conclusion, while SSD may not be as efficient as some other lightweight models, its balance of speed and accuracy, coupled with the potential for further optimizations, makes it a robust choice for object detection on resource-constrained devices.

Architecture

SSD employs a VGG16 model pretrained on ImageNet as its base network [40]. As illustrated in Fig. 4.1c it modifies the VGG16 architecture by removing some of the

fully connected layers, replacing them with a series of convolutional layers of decreasing sizes, thus forming a multi-scale feature map. The multi-scale feature maps at the top of the network predict the offsets to default boxes of different scales and aspect ratios and their associated confidences. This enables SSD to handle objects of various sizes and shapes, a significant advantage over models such as YOLO that use a fixed grid for detection.

Furthermore, SSD introduces the concept of default boxes (or anchor boxes), which are pre-computed boxes of various aspect ratios. During training, SSD determines the offsets and confidences for these default boxes. The application of these default boxes across multiple feature maps of different resolutions allows the detection of objects at various scales.

4.2.4 MobileNet-SSD

MobileNet-SSD is a highly efficient, single-shot object detection model that is a combination of MobileNet and SSD [64]. As the name suggests, it leverages the MobileNet architecture as a feature extractor (backbone) and the SSD framework as the object detection mechanism (detection head).

MobileNet-SSD combines the best of both worlds, where MobileNet provides a compact and efficient base network, and SSD offers a powerful detection mechanism. The resulting model is a fast and accurate object detector with a significantly reduced computational footprint. This makes MobileNet-SSD particularly well suited for real-time applications on resource-constrained devices such as the Raspberry Pi.

The model can deliver impressive performance even on these devices, managing to run object detection tasks in real-time while maintaining reasonable accuracy levels. Moreover, it can also be further optimized using techniques like quantization and pruning to reduce the memory and computation requirements, thereby improving the speed and efficiency without sacrificing much accuracy.

In conclusion, MobileNet-SSD is an excellent choice for object detection applications on resource-constrained devices due to its efficient architecture and strong performance. It is a practical solution for scenarios where both speed and accuracy are crucial, and computational resources are limited.

Architecture

The architecture of MobileNet-SSD combines the strengths of both models as illustrated in Fig. 4.1d. The MobileNet component is responsible for the extraction of high-level features from the input image [64]. It utilizes depthwise separable convolutions, significantly reducing the computational requirements while still delivering robust feature extraction capabilities. This makes MobileNet particularly useful for environments where computational resources are limited.

The SSD component is responsible for predicting the class and location of objects in the input image. It uses multiple feature maps from different layers of the network to detect objects at different scales. Each feature map cell gives predictions for a set of default boxes of different aspect ratios. This multi-scale, multi-aspect-ratio approach makes SSD highly effective for detecting objects of various sizes and shapes.

4.2.5 EfficientDet

EfficientDet, developed by Tan et al. [65], is a family of object detectors recognized for their excellent balance of accuracy and computational efficiency that come in different sizes, denoted by a scaling parameter d. The larger the value of d, the more complex and accurate the model is. They represent the next phase in the evolution of object detection models, incorporating novel architectural and methodological techniques. At the core of the EfficientDet family are EfficientDet-D0 and EfficientDet-D2, each reflecting an advancement over the preceding version.

The EfficientDet family of object detectors are designed to balance accuracy with computational efficiency, Fig. 4.1f shows a basic general architecture, making them attractive options for deployment on resource-constrained devices. However, their application in such scenarios demands thoughtful consideration. While these models are more efficient than many other object detectors of similar or higher accuracy, they are still computationally more intensive than lightweight models such as MobileNets or SqueezeNet.

EfficientDet-D0 and D2, despite being more streamlined than other detectors of comparable performance, may still be too resource-intensive for devices with stringent computational and power limitations. For instance, deploying these models on low-cost embedded devices such as Raspberry Pi may result in slower inference times and greater power consumption, compared to deploying lightweight models.

However, for higher-end edge devices such as Google's Edge TPU or NVIDIA's Jetson series, which possess more substantial computational resources while still maintaining a small footprint, EfficientDet models can deliver robust object detection performance with acceptable efficiency. Furthermore, the use of techniques such as model quantization or pruning could potentially optimize these models further for deployment on resource-constrained devices.

In summary, while EfficientDet models represent a significant stride towards creating efficient, high-performing object detectors, their deployment on resource-constrained devices requires careful consideration of the device's computational capabilities and the specific requirements of the task at hand. It underscores the necessity of detailed performance benchmarking across various devices and models, to facilitate informed decisions when choosing the most suitable model for deployment.

EfficientDet-D0

The EfficientDet-D0 model builds on the EfficientNet backbone, a CNN architecture optimized using compound scaling, which simultaneously scales up network

depth, width, and resolution. The principal innovation in EfficientDet-D0 lies in the introduction of a bi-directional feature pyramid network (BiFPN), a significant improvement over the single-directional FPN used in earlier models. BiFPN allows for multi-scale feature fusion, enabling the model to process information at different scales and resolutions, thereby providing a more robust representation of objects. Another significant contribution is the weighted feature fusion, which enables the model to assign weights to different features based on their importance, enhancing the model's ability to focus on more relevant features. It achieves similar accuracy as YOLOv3 with 28x fewer FLOPs

EfficientDet-D2

The EfficientDet-D2 model furthers the architectural developments of D1 by increasing the number of layers in the BiFPN and employing a more substantial EfficientNet backbone. The scaling up of the architecture leads to improved performance due to the increased capacity of the model. Further, EfficientDet-D2 includes a compound scaling method, which uniquely scales up the resolution, depth, and width of the network based on a fixed resource budget. This comprehensive scaling strategy results in improved model performance without exponentially increasing the computational complexity.

The EfficientDet family showcases a remarkable interplay of several novel techniques, including compound scaling, BiFPN, and weighted feature fusion, resulting in highly accurate and computationally efficient models. As with the MobileNet family, the choice between EfficientDet-D1 and EfficientDet-D2 would depend on the specific task at hand and the resource constraints of the device in use. The importance of thorough performance benchmarking, therefore, cannot be overstated in determining the optimal model for a given task. It achieves state-of-the-art accuracy on COCO test-dev dataset among single-model detectors, with 48.5 mAP.

4.2.6 YOLO

YOLO [66] is a real-time object detection system that has revolutionized the field of computer vision due to its superior speed and accuracy. Unlike the two-stage approach taken by other models such as R-CNN, YOLO frames object detection as a regression problem to spatially separated bounding boxes and associated class probabilities. A single pass through the network is enough to make predictions, making YOLO an exceptionally fast and efficient model.

YOLOv5s

YOLOv5s is a more compact variant of the YOLOv5 model [67]. Designed with edge devices in mind, its aim was to strike a balance between model size and detection performance while keeping computational constraints into consideration. However, real-world benchmarks, such as those in this thesis, show that while YOLOv5s exhibits commendable accuracy, its inference speed, especially on devices like the Raspberry Pi, might not always align with real-time expectations.

By employing a smaller backbone and fewer filters in the detection head, YOLOv5s boasts a reduced architecture and a smaller set of parameters. This results in less computational load in theory, suggesting its applicability in real-time scenarios, especially on devices with restricted computational capabilities and memory.

The YOLO series, and in particular YOLOv5s, have been proposed for applications necessitating real-time object detection, ranging from surveillance systems to autonomous vehicles and drones.

However, when deployed on truly resource-constrained platforms like the Raspberry Pi, while YOLOv5s exhibits a notable accuracy in object detection, it may not always meet the desired real-time speeds. This is a testament to the challenges of deploying complex models in constrained environments. Yet, its impressive accuracy does highlight its potential in applications where precision might be prioritized over pure speed.

In summary, while YOLO and its lightweight counterpart, YOLOv5s, are groundbreaking in their approach to object detection, practical implementations underscore the importance of holistic testing. Specifically, in real-world scenarios, there's a delicate balance to strike between speed, accuracy, and performance, especially on resource-limited devices.

Architecture

YOLO uses a single convolutional network to predict multiple bounding boxes and class probabilities for those boxes as illustrated in Fig. 4.1e. The model divides the input image into an SxS grid, and for each grid cell, it predicts B bounding boxes, confidence for those boxes, and C class probabilities. Each bounding box also has an associated confidence score indicating the probability that the box contains an object.

However, it should be noted that YOLOv5 was not released by the original YOLO authors but by a different group of researchers. The exact architecture of YOLOv5 may vary as it evolves, but the general structure and workflow based on the information available on their Github page, YOLOv5 is designed with the same basic principles as earlier versions of YOLO. The general workflow is as follows:

- **Image Input:**

  The input image is taken and resized to a square image of a size that's a multiple of 32 (like 320x320, 416x416, 608x608 pixels), as this is the requirement of the model to function properly.

- **Backbone / Feature Extraction:**

  The resized image is passed through a series of convolutional layers to generate a feature map. In the case of YOLOv5, the backbone network is a custom architecture that starts with a Focus layer, which is essentially a type of convolution layer that reduces the spatial dimensions of the image and increases the channel dimensions to extract features.

- **Neck / Additional Feature Extraction and Pyramid Scaling:**

  The feature map is then passed through several more layers. These layers include feature pyramid scaling to detect objects of different sizes and some additional convolutional layers to refine the features.

- **Head / Detection:**

  The final stage involves a detection head which outputs bounding box coordinates, class predictions, and objectness scores (how likely the predicted bounding box actually encloses an object). This is typically done using three different scales (small, medium, large objects) to capture objects of different sizes.

4.2.7 MobileObjectLocalizer

MobileObjectLocalizer (MOL), a lightweight object detection model developed by Google [68], can be deployed for recognizing a vast array of objects. Unlike models such as YOLO, which categorizes objects into 80 classes based on the COCO dataset, MOL refrains from assigning specific labels to the objects it identifies. Although its precision might not be top-notch, MOL is capable of identifying any object's bounding box without the need for learning. MOL employs MobileNetV2 and SSD-Lite, accepts input with a resolution of 192px × 192px, and can produce up to 100 bounding boxes.

4.3 Conclusion

In conclusion, We have selected these six state-of-the-art models – MobileNetSSD V1, V2, EfficientDet V0, V2, MobileObjectLocalizer, and YOLOv5s – for this study after careful consideration of their unique attributes, which make them well-suited for resource-constrained embedded devices and real-world applications.

The EfficientDet models, while larger, demonstrate remarkable accuracy and can be scaled down to operate effectively on embedded devices, making them appealing for applications where accuracy is paramount.

The SSD and MobileNet-SSD models demonstrate impressive speed in detecting objects, making them well-suited for real-time applications. Furthermore, their lightweight nature allows them to perform efficiently on devices with limited computational resources.

We have included YOLOv5s in our selection despite it being slightly heavier than the other models. The reason behind this is to demonstrate its robust capabilities in real-time object detection. Its architecture makes it suitable for applications where accuracy is of utmost importance. Furthermore, YOLOv5s has been optimized to work effectively on resource-constrained devices, and we aim to shed light on its limitations as well as its strengths.

We have chosen the MOL model for our use cases primarily because of its speed and lightweight architecture. Although its accuracy may not be at the highest level compared to other models, its strength lies in its efficiency and versatility. MOL is fast and lightweight, making it suitable for real-time applications and deployment on devices with limited computational resources. This model's ability to detect a broad range of objects without assigning specific labels further showcases its flexibility and wide applicability across various scenarios.

Chapter 5

Methodology

5.1 Introduction

In the forthcoming section of this thesis we will undertake a comprehensive assessment of methodology and tools that were used for selected CNN models operating on an embedded device, specifically, a Raspberry Pi 4 Model B with a quad-core 64-bit architecture and 8GB RAM, paired with the Google USB-Coral accelerator, a custom-made ASIC Edge TPU designed for AI edge computing. This setup is representative of many IoT gateways utilized in various applications.

The models chosen for this investigation include the Mobilenet series combined with SSD, EfficientDet series, YOLOv5, and MOL models. All these models were pre-trained on the COCO dataset [69] using TensorFlow Hub [70], then converted into TFLite and edgetpu formats through TensorFlow Lite Model Maker library APIs, enabling their execution on the Raspberry Pi with the CORAL accelerator. An additional step involved the application of a quantization technique to optimize the models for deployment on resource-limited devices.

For effective performance evaluation, a dataset of public traffic videos compiled by Xiao et al. [71] is employed, consisting of over 30 minutes of diverse traffic surveillance footage, previously annotated using the full SSDv2 trained on the COCO dataset. Original videos were shot in a range of conditions and from various sources, captured in high resolution (1280px × 720px at 30 frames per second). In order to establish a comparative benchmark between high-resolution and low-resolution data,

the video quality was reduced by 70%.

While this study particularly targets the performance of these models for traffic and pedestrian surveillance, the insights and conclusions drawn from the evaluation can have implications for other fields as well, underscoring the versatility and broad relevance of our findings.

The selected models are extensively evaluated over different video qualities (i.e., high quality of 1280px × 720px and low quality with a 70% reduction in quality) and conditions (daytime or night time). Performance metrics used to assess the models include F1 Score, frames per second, CPU and RAM usage, energy consumption, and temperature.

5.2 Tools and Framework

This section aims to do an exhaustive exploration and evaluation of the specific tools and theoretical frameworks employed throughout the course of this research. This will encompass a detailed analysis and comprehensive review of each, elucidating their unique features and capabilities that were instrumental in supporting this study. Additionally, an elaborate discussion will be presented, outlining the rationale behind the selection of these particular tools and frameworks. This will include a justification for their adoption, focused on how these choices have been strategically tailored to the objectives and requirements of this study, thereby enhancing the precision, efficiency, and overall validity of the research findings.

5.2.1 Embedded Platform

Raspberry Pi 4

The Raspberry Pi 4 Model B [72], outfitted with a quad-core 64-bit architecture and 8GB of RAM, serves as a widely adopted and potent representative within the broad

sphere of edge devices. Owing to its widespread use and versatility, this particular model offers a realistic and representative platform for investigating the performance of the CNN models selected for this study.

Several factors contribute to the suitability of the Raspberry Pi 4 Model B for this research. Primarily, its quad-core processor presents significant computational power, which is necessary for executing complex tasks like image processing, machine learning, and other computationally demanding operations. These calculations, it should be noted, are CPU-based, providing a standardized baseline for our tests.

The 64-bit architecture of the Raspberry Pi 4 Model B broadens the data path, facilitating swifter data processing—an essential attribute for managing data-heavy operations such as video streaming. Its generous 8GB of RAM enables the handling of larger datasets in memory, improving the efficacy of machine learning model execution.

The compact dimensions, low power requirements, and cost-effectiveness of the Raspberry Pi 4 Model B render it a preferred choice for edge computing applications, which include IoT gateways, home automation, and edge servers. Its capability to operate under these constraints without compromising on performance is emblematic of the prerequisites of genuine edge devices.

The robust and active community surrounding the Raspberry Pi platform furnishes an abundance of resources, support, and pre-compiled software, all of which ease the implementation and deployment of complex applications and machine learning models.

The Raspberry Pi series encompasses multiple models, each varying in performance capabilities and pricing, to cater to an extensive array of applications. The portfolio ranges from the lower-end Raspberry Pi Zero models, ideal for simpler control tasks, to the high-performance Raspberry Pi 4 Model B, capable of managing advanced computational tasks. This spectrum of solutions accommodates a diverse set of edge computing requirements.

Alternatives

While the Raspberry Pi 4 Model B serves as a representative example of general-purpose edge devices, there are also numerous SBCs specifically tailored for particular tasks. For instance, the NVIDIA Jetson series, such as the Jetson Nano, are purpose-built for machine learning and AI applications with powerful GPU acceleration capabilities. Similarly, the Google Coral Dev Board is specifically designed for fast on-device inferencing with its Edge TPU coprocessor.

Other notable SBCs include the BeagleBone Black, known for its extensive I/O capabilities, making it an excellent choice for robotics applications, and the Odroid series, which offers a range of high-performance boards with more robust CPU and GPU capabilities than many of their counterparts.

Motives

Despite these alternatives, we opted not to select these task-specific boards for several reasons. Firstly, while these boards offer improved performance for specific tasks, they are also significantly more expensive than the Raspberry Pi 4 Model B. Given the goal of this study is to assess the performance of selected CNN models on resource-constrained devices, cost-effectiveness is a critical consideration.

Secondly, the widespread adoption and popularity of the Raspberry Pi platform make it a representative example of edge devices in numerous real-world applications. Results obtained from Raspberry Pi 4 Model B will thus be more readily applicable and understandable to a wider audience.

Lastly, while some of the task-specific boards might offer superior performance for machine learning tasks, their specific hardware architectures may not be representative of the wide range of devices found at the network edge. By focusing on the Raspberry Pi 4 Model B, we aim to present insights that are broadly relevant across a variety of edge computing contexts.

Pi and Traffic Object Detection

In the context of traffic object detection, the Raspberry Pi presents an opportunity to implement low-cost, compact, and effective solutions that can be deployed in a wide range of scenarios. Here are some examples:

- **Ad-hoc Surveillance with Drones:** In situations where there is a need for temporary or flexible surveillance, such as monitoring traffic flow during major events, handling emergency situations, or managing temporary construction sites, drones equipped with Raspberry Pi can be deployed. These drones can perform real-time traffic object detection to provide valuable insights, enhancing situational awareness and decision-making. The small form factor and low power consumption of the Raspberry Pi make it an excellent choice for such battery-powered, mobile platforms.
- **Assisting Visually Impaired Individuals:** There's a significant potential for Raspberry Pi-based object detection systems in assistive technology for visually impaired individuals. A Raspberry Pi, paired with a camera, can be used to develop a wearable device that provides real-time object detection. Such a device can alert the user about nearby vehicles, potential obstacles, or other important environmental factors, increasing their confidence and safety while navigating urban environments.
- **Smart Traffic Management Systems:** In smart city applications, Raspberry Pi-based systems can be installed at intersections to monitor real-time traffic and detect vehicles, pedestrians, cyclists, and so on. This data can feed into a centralized traffic management system to adjust signal timings, detect traffic congestion, and generally optimize traffic flow in the city.

Peripherals for enhancement

Moreover, it is also worth acknowledging that these devices, like many general-purpose edge devices, may struggle with the high computational demands of running complex CNN models. To mitigate this performance issue, our study also incorporates the use of the Google Coral USB Accelerator, a device specifically designed to enhance machine learning inference on edge devices.

The Google Coral USB Accelerator integrates the Edge TPU, a small ASIC chip designed by Google to run TensorFlow Lite ML models at high speed, while still maintaining a low power draw. By pairing the Raspberry Pi with this accelerator, we can effectively demonstrate a common real-world scenario where a general-purpose edge device is augmented with dedicated machine learning hardware to handle the computational heavy lifting of advanced AI models.

Details about the exact role and benefits of the Coral USB Accelerator in our experimental setup, as well as its interaction with the selected CNN models, will be presented and discussed in the following sections of this thesis.

In conclusion, the Raspberry Pi 4 Model B was selected for this study owing to its computational capabilities, representative nature for edge devices, cost-effectiveness, and broad applicability. Moreover, to augment the computational performance, we employ different acceleration techniques, which will be elaborated on in subsequent sections. This investigation into its performance in the given context offers valuable insight into the viability and implications of deploying the selected CNN models on comparable edge devices.

Google Coral TPU

The Google Coral USB Accelerator [73] is a critical component in our study that significantly enhances the computational capabilities of the Raspberry Pi. As a specialized device for accelerating machine learning tasks, it allows us to perform complex

calculations more efficiently, speeding up the processing time of our selected CNN models significantly.

The Coral Accelerator is powered by the Edge TPU, a purpose-built ASIC designed to run AI at the edge. It is compatible with TensorFlow Lite, the lightweight solution of TensorFlow designed for mobile and embedded devices. The Edge TPU can execute state-of-the-art mobile vision models such as MobileNet v2 at 20+ FPS, in a power-efficient manner. This high-speed processing capability is particularly crucial for our study, where real-time video processing is needed.

One of the key advantages of using the Coral USB Accelerator is that it offloads the computational burden from the Raspberry Pi's CPU. This allows us to use the CPU for other tasks, while the Coral device takes care of the heavy lifting of the machine learning computations. This is particularly important in edge computing scenarios where resources are often limited.

When we consider real-world applications, the addition of a Coral USB Accelerator to a Raspberry Pi can turn a low-power edge device into a potent AI inference machine. This combination can be deployed in various contexts, from intelligent surveillance systems and autonomous robots to smart home applications. It offers a cost-effective way to implement AI applications on edge devices without needing a connection to the cloud.

It is worth noting, while the Edge TPU and GPUs both serve the purpose of accelerating machine learning tasks, they differ fundamentally in their design and best use scenarios.

A GPU, or Graphics Processing Unit, is a powerful and versatile piece of hardware capable of handling a wide variety of tasks. Its design, featuring hundreds to thousands of cores, enables it to perform large-scale parallel computations, making it ideal for training large and complex machine learning models. The downside, however, is that GPUs are often energy-intensive and may not be suitable for devices with strict power constraints.

On the other hand, the Edge TPU in the Coral USB Accelerator is an Application

Specific Integrated Circuit (ASIC) specifically designed for inference tasks of neural networks. It's much more energy-efficient than a typical GPU, making it more suitable for edge devices like the Raspberry Pi. Although its capabilities are not as wide-ranging as a GPU, the Edge TPU is highly optimized for the tasks it's designed for, delivering high performance at a fraction of the power usage.

In scenarios where power efficiency and footprint are of utmost concern, such as in embedded devices or mobile applications, the Edge TPU is a superior choice. However, for tasks that demand large-scale parallel computations or for the training phase of machine learning models, GPUs continue to be the more effective option.

Hence, the choice between a GPU and an Edge TPU depends on the specific requirements of the use case. In our case, the Edge TPU's high efficiency, compact form factor, and impressive inference speed make it an ideal choice for running our selected CNN models on a Raspberry Pi in edge computing scenarios.

In conclusion, the Google Coral USB Accelerator was incorporated into our study due to its ability to significantly enhance the performance of machine learning models on edge devices. Its compatibility with TensorFlow Lite, high processing speed, and power efficiency make it an excellent addition to the Raspberry Pi. Its integration into our research setup mirrors the trend of enhancing edge devices with specialized hardware accelerators to meet the growing computational demands of advanced AI models.

5.2.2 Software Platform

TensorFlow

TensorFlow is an open-source library developed by the Google Brain team that provides a platform for developing and running machine learning and deep learning models. It is popular for its flexible architecture, allowing users to deploy computations across multiple CPUs, GPUs, and even TPUs with a single API. This flexibility

extends to several platforms including desktops, servers, mobile and edge devices, and even cloud-based systems.

A significant feature of TensorFlow is its use of data flow graphs, where nodes represent mathematical operations and edges represent multidimensional data arrays or tensors. This approach allows for highly efficient and parallel computations, which is essential for training complex deep learning models.

Furthermore, TensorFlow comes with a suite of visualization tools known as TensorBoard, which allows users to interactively visualize their model's computational graphs, monitor training progress, and evaluate metrics.

TensorFlow Lite

While TensorFlow is comprehensive and powerful, its capabilities may be overkill for some applications, particularly for edge devices where resources are constrained. Here, TensorFlow Lite (TFLite) [74] comes into play.

TensorFlow Lite is a set of tools provided by TensorFlow to help developers run TensorFlow models on mobile, embedded, and IoT devices. It is designed to be lightweight and to provide fast performance with a small binary size.

TFLite works by converting full TensorFlow models into a simpler format which removes unnecessary information and optimizes the model for speed and size. It also supports hardware acceleration with the Android Neural Networks API and Apple's Core ML.

One key feature of TensorFlow Lite is the support for post-training quantization. This reduces the model size and increases inference speed while maintaining a high level of accuracy.

In our study, we used TensorFlow Lite for its suitability for resource-constrained edge devices like the Raspberry Pi. By converting our selected CNN models to the TensorFlow Lite format, we could maintain a high level of performance while reducing memory usage and power consumption. Moreover, TensorFlow Lite provides a

smooth path for deploying our models on the Edge TPU accelerator via the Coral USB Accelerator, enhancing their performance significantly.

To summarize, the combination of TensorFlow's robust capabilities for developing and training models and TensorFlow Lite's optimization for edge devices forms an ideal solution for our research.

Edge TPU

The Google Coral USB Accelerator is a device that houses the Edge TPU in a USB stick form factor. It can be used to add machine learning acceleration to existing systems by simply plugging into a USB port. This makes it a convenient tool for enhancing the machine learning capabilities of devices such as the Raspberry Pi.

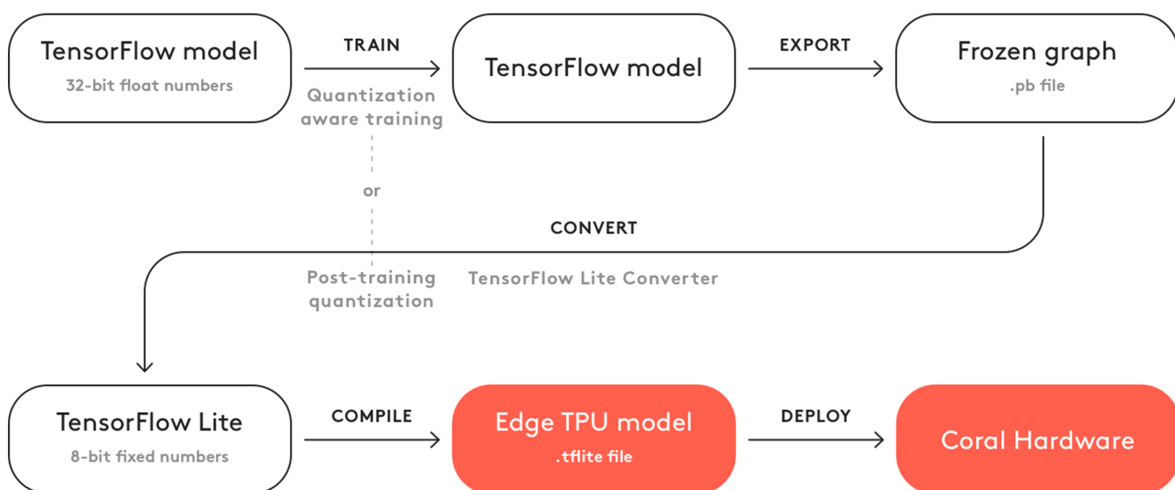


FIGURE 5.1: Model Conversion from TensorFlow to Edge TPU [73]

To use the Edge TPU, TensorFlow Lite models need to be converted to a specific format that the Edge TPU can understand based on Figure 5.1. This conversion process is facilitated by the Edge TPU Compiler, a tool provided by Google. The Edge TPU Compiler takes a TensorFlow Lite model as input and outputs a new TensorFlow Lite model that is compatible with the Edge TPU.

The conversion process primarily involves the mapping of certain operations in the TensorFlow Lite model to equivalent operations that can be run on the Edge TPU. Some operations that are not supported by the Edge TPU are left to run on the

CPU. This allows for a seamless execution of the model, leveraging the strengths of both the Edge TPU and the CPU.

In conclusion, the use of Edge TPU and Coral USB Accelerator in our research allows us to run state-of-the-art CNN models at high speeds, thereby making real-time object detection on resource-constrained devices like the Raspberry Pi a reality.

5.3 Dataset

A crucial component of our methodology was the selection of an appropriate dataset to conduct an effective and realistic performance evaluation of the selected models. In our research, we utilized the public traffic video dataset compiled by Xiao et al. This dataset is curated and consists of over 30 minutes of traffic surveillance data, offering a rich variety of scenarios and conditions which is highly representative of real-world applications.

The public traffic videos dataset was initially annotated using the full SSDv2 model trained on the COCO dataset, compiled by Xiao et al.[71]. This is noteworthy as finding a manually annotated dataset that closely mirrors real-world conditions can be quite challenging. A manually annotated dataset requires a significant amount of time and resources, which may not always be feasible. Therefore, it is common in such research to resort to using a dataset annotated by a fully trained model, despite the potential for some inherent bias towards the model used for annotation. It is essential to bear in mind that this procedure might introduce some degree of bias favoring the SSDv2 model. However, this approach is widely accepted in the research community due to the practical constraints of manual annotation.

The original videos were shot at a high resolution of 1280px × 720px, with a rate of 30 frames per second, from a multitude of sources including both moving and stationary traffic cameras. This variety is invaluable in offering a wide range of environmental and situational conditions, such as daytime or nighttime and crowded or vacant scenarios, across different weather conditions like sunny or cloudy.

To conduct a benchmarking comparison and evaluate the performance of models with both low-resolution data and high-resolution data, we manipulated the video quality by reducing it by 70%. Therefore, our research includes an extensive evaluation of the selected models over varying video qualities (high quality of 1280px × 720px and a lower quality representing a reduction of 70%) and different conditions (daytime or nighttime).

In summary, the use of this traffic video dataset allows us to evaluate the selected models under a range of conditions and resolutions that reflect real-world traffic surveillance scenarios, thereby providing a robust and practical assessment of their performance.

5.3.1 COCO Pre-Trained Models and Considerations for Accuracy Enhancement

In this study, we opted for using models pre-trained on the COCO dataset due to its widespread adoption in the research community and the robustness of the models trained on this large, diverse dataset. However, this choice is primarily guided by our objective of creating a benchmarking framework to evaluate the performance of object detection models running on resource-constrained environments, such as a Raspberry Pi.

Our primary focus is not necessarily on achieving the highest possible accuracy, but rather on gaining insights into the performance trade-offs associated with different models in real-world, low-resource scenarios. By utilizing COCO pre-trained models, we provide a consistent baseline for comparing different models, which is essential for an effective benchmarking framework.

That said, if increased accuracy is the primary goal, there are other methodologies that could be adopted. For instance, the application of transfer learning techniques could potentially enhance the performance of these models. A model pre-trained on COCO could be fine-tuned to identify a subset of objects under different

conditions (such as snow, rain, night-time, etc.), or to better detect objects in specific contexts (like posters or billboards).

Further, if there are objects of interest that are not covered by the COCO dataset, transfer learning can also be applied by fine-tuning the pre-trained models on another dataset that includes these objects. This can often yield better performance for these specific object detection tasks.

Another approach to improve accuracy would be to train a model from scratch on a completely different dataset more relevant to the specific use-case. However, this can be computationally intensive and may not be feasible for edge devices like the Raspberry Pi.

After such training, the models would then have to be converted into the TensorFlow Lite (TFLite) format for efficient execution on the Raspberry Pi. This adds an additional step to the process, but allows the deployment of the model in a format optimized for edge devices.

It is important to note that all these alternatives, while potentially leading to improved accuracy, would require additional computational resources, expertise, and time, and would shift the focus away from our main goal of understanding the performance trade-offs associated with different models in a real-world, edge computing scenario. As such, these considerations may form part of future works building upon this research.

5.4 Data Pipeline

The object detection system used in this study was realized through a carefully designed data pipeline, as visualized in Figure 5.2. This pipeline served as a sequence of processing steps to extract relevant information from our input data, prepare it for model inference, and then log and analyze the results. It's worth noting that this data pipeline is just one of many possible configurations and the exact design can be altered based on the specific needs and constraints of a given application.

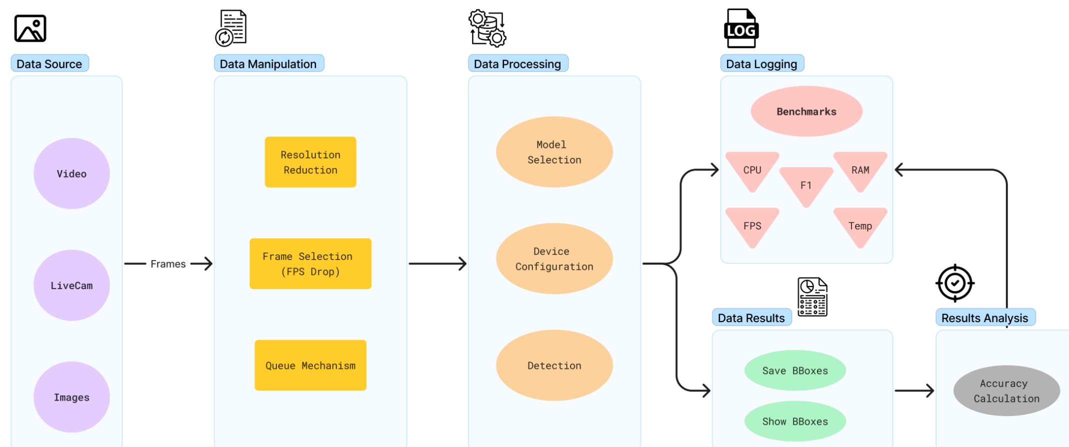


FIGURE 5.2: Data Pipeline for Object Detection

- **Input Data:** This study utilized pre-recorded video footage, but the source of input data could also be a live camera feed, a database of images, or even a stream of data from other sensors. The input data is the raw material for the object detection system and can come in various formats and from different sources.

- **Data Preprocessing:** The next step in the pipeline is preprocessing the input data. This could involve data manipulations such as resolution reduction, frame selection, or data augmentation techniques. These tasks could be performed on edge devices or in the cloud, depending on the system design and application requirements. This stage is crucial as it can significantly impact the performance of the system, especially in real-time scenarios where efficient processing techniques such as queue management or frame dropping may be required.

- **Model Inference:** Once the data is prepared, it's fed into the selected object detection model for processing. In this study, the Raspberry Pi device was set up with the TensorFlow Lite environment and one of the studied models, ready to perform inference on the preprocessed input data.

- **Logging and Result Processing:** Following the inference stage, two parallel tasks are performed: the device conditions (e.g., temperature, CPU usage, memory usage, and FPS) are logged for further analysis, and the object detection results are processed and saved. The processed results could be in the form of bounding box coordinates or visual representations of detected objects.
- **Result Analysis:** Finally, the logged data and detection results are analyzed. This could involve comparing the model's predictions with ground truth labels to calculate performance metrics such as the F1 score, which are then also logged.

In conclusion, the data pipeline used in this study represents a comprehensive workflow from raw data to result analysis. However, it should be seen as a foundation upon which additional modifications or improvements can be made, depending on the requirements of specific applications and system constraints. Our methodology is intended to serve as a flexible framework for evaluating different object detection models under various conditions, helping researchers and developers make informed decisions when designing and implementing their systems.

5.5 Experiments

In the ensuing section, we delve into the analytical process employed to evaluate the performance metrics of the selected CNN models. We have developed a systematic benchmarking protocol, constructed to not only assess the functioning of these models but also to provide a comparative framework to gauge their performance. The comprehensive assessment of our selected models is achieved through a series of meticulously conducted experiments using a variety of evaluation metrics, each designed to provide specific insights into the models' performance. The aim is to provide a holistic and unbiased evaluation, thus aiding future research and practical applications in the deployment of such models on edge devices. This

protocol incorporates measurements of various computational parameters and environmental variables, thereby ensuring that the results obtained are representative of a real-world scenario.

5.5.1 Metrics

To ensure a comprehensive evaluation of the selected models running on the embedded devices, we relied on a set of metrics encompassing both the performance of the models and the resource utilization of the devices. These metrics have been chosen carefully to provide a holistic view of the performance and efficiency of the models.

- **F1 Score:** The F1 score [75] is a well-established measure used in statistics and machine learning for evaluating the performance of binary classification models. It is especially useful when dealing with imbalanced datasets. The F1 score is the harmonic mean of precision and recall, both of which are crucial metrics in model evaluation. Precision gauges the proportion of correctly identified positive instances out of all instances labeled as positive, while recall (also known as sensitivity or true positive rate) assesses the proportion of correctly identified positive instances out of all actual positive instances. The F1 score, computed as

$$F1 = \frac{2 \times (precision \times recall)}{precision + recall}, \tag{5.1}$$

  offers a single, unified measure of a model's performance, balancing the trade-off between precision and recall.

  In evaluating the performance of our object detection models, it's important to note that we opted to use the F1 score rather than the commonly used metric, the mAP.

The decision to utilize F1 score over mAP was driven by several considerations. First, the F1 score provides a simple, yet robust, performance measure for binary classification tasks. It balances the trade-off between precision and recall, both of which are critical in our context where both false positives and false negatives have significant implications.

Furthermore, the F1 score is particularly useful when dealing with imbalanced datasets, which is often the case in object detection scenarios where the number of 'no object' instances vastly outnumber the 'object' instances. This characteristic of the F1 score can provide a more realistic performance assessment in our specific use case.

On the other hand, mAP, while offering a comprehensive measure of model performance across different recall levels, can be complex to calculate and interpret. This is especially true in our case where we aim to benchmark models on resource-constrained devices like Raspberry Pi, and the added computational complexity of calculating mAP could impact the overall evaluation process.

Therefore, for the reasons of simplicity, computational efficiency, and the specifics of our use case, we have chosen the F1 score as our primary performance metric. This does not diminish the usefulness of mAP for other scenarios and it remains a key metric in the broader field of object detection.

- **Frames per Second (FPS):** The frame rate, measured in frames per second, provides an indicator of the smoothness of the video processed by the model. The higher the frame rate, the smoother the video will appear. For our models, the FPS was calculated by averaging the time it took to infer 1000 frames. A frame's inference time was determined by measuring CPU time before and after it was processed. This allows us to assess the real-time capability of the models

- **CPU and RAM Usage:** These metrics measure the resource utilization of the Raspberry Pi when running the ML models. The CPU utilization and RAM footprint were calculated by averaging the usage of the CPU in percentile, and RAM in megabytes, over 30 minutes of surveillance footage. These metrics are vital in assessing the efficiency and practicality of deploying these models on resource-constrained devices.
- **Energy Consumption:** We also calculated the energy consumption of the models running on the Raspberry Pi. Using a multimeter, we determined the current and voltage of the device, and then calculated the power consumption by multiplying the current and voltage. This gives an idea of the power efficiency of the models, an important consideration for edge devices.
- **Temperature:** Lastly, using a thermal camera, we measured the temperature of the Raspberry Pi's CPU to assess the amount of heat it may generate when running a model. The temperature was calculated by averaging the readings in degrees Celsius taken every minute over 30 minutes of surveillance footage. This measurement provides insight into the thermal impact of running the models, a critical factor considering the compact size and lack of advanced cooling systems in many edge devices.

By considering these metrics, we can provide a thorough evaluation of the models' performance and their suitability for use on edge devices such as the Raspberry Pi.

5.5.2 Pi Configuration

In the course of our research, we configured and operated the Raspberry Pi 4 Model B under varying conditions to effectively simulate the broad range of scenarios that such edge devices might encounter in real-world applications. Our benchmarking protocol was designed to elucidate the performance characteristics of the selected CNN models under diverse load conditions on the Raspberry Pi platform.

One of the key configuration variables we considered was the utilization of the quad-core CPU available in the Raspberry Pi 4 Model B. We progressively engaged more cores to the task of executing the CNN models, starting from a single-core configuration, moving to dual and tri-core setups, and finally, utilizing all four cores in the quad-core setup. This staged approach allowed us to evaluate the impact of increased computational power on the performance of the CNN models, thereby providing valuable insights into the scalability and parallel processing capability of these models on multi-core platforms.

5.5.3 Google Coral Config

Simultaneously, we also explored the performance benefits of offloading the AI processing task to an external accelerator, specifically, the Google Coral USB accelerator. This scenario involved the Raspberry Pi functioning primarily as a task manager, handling lighter tasks such as input/output operations and system management, while the computationally heavy task of model inference was offloaded to the Coral TPU. This configuration served to demonstrate the performance benefits of task-specific accelerators in conjunction with general-purpose computing platforms, particularly in the context of executing resource-intensive AI models.

These experiments were designed to provide a holistic understanding of the performance capabilities and limitations of CNN models when deployed on resource-constrained edge devices. The insights derived from these different configuration scenarios inform us of how to effectively balance system resources to achieve optimal performance in diverse application settings.

5.6 Conclusion

In conclusion, the methodology chapter has laid the groundwork for the performance evaluation of our selected CNN models on embedded devices. We have established a comprehensive protocol for benchmarking, which includes a variety

of metrics encompassing model accuracy, computational resource utilization, energy consumption, and operational temperature. This range of parameters provides a holistic view of the model performance, particularly in the context of resource-constrained environments.

By choosing the Raspberry Pi 4 Model B and Google Coral USB accelerator as our test platforms, we aim to represent real-world scenarios, given these devices' prevalence in edge computing applications. Our choice of TensorFlow Lite and Edge TPU optimization aligns with the need for model compression and optimization for performance enhancement on such devices.

The use of a traffic surveillance dataset provides a real-world application setting, acknowledging the increasing relevance of AI in such applications. Despite the potential bias in the data annotation, the comprehensive nature of the data, encompassing various traffic conditions and video qualities, makes for a robust performance evaluation.

By employing these methodologies, we strive to provide a reliable, representative, and informative performance assessment of the selected CNN models on edge devices. The findings from our study will serve as a reference for researchers and developers in choosing the appropriate models for their specific applications.

In the next chapter, we will present and discuss the results obtained from the execution of our benchmarking protocol, highlighting the performance of each model under various operational conditions.

Chapter 6

Evaluation and Results

6.1 Introduction

As we move forward in this study, we now present an in-depth discussion of the results obtained from our methodological approach, as detailed in the preceding sections. This part of the research aims to provide a comprehensive understanding of the performance characteristics of the chosen CNN models under varied conditions on the Raspberry Pi 4 Model B platform, and when paired with the Google Coral USB accelerator.

Our evaluation primarily hinges on the metrics previously discussed, namely, the F1 Score, FPS, CPU and RAM usage, Energy Consumption, and Temperature. These metrics, used collectively, give a broad-based perspective on the efficacy and efficiency of the models, shedding light not only on their computational performance but also on their practical implications in terms of resource consumption and thermal management.

It is noteworthy to mention that all our results have been analysed with a confidence interval of 95%, where applicable. This means that the performance outcomes we present are statistically significant, and we can be 95% confident that the actual value lies within this specified range.

Through this evaluation and result analysis, we aim to illuminate the practical performance capabilities of these models in real-world edge device applications. This analysis will also provide crucial insights into how different factors such as

model choice, device configuration, and task distribution between CPU and TPU can impact overall performance in varied edge computing scenarios.

6.2 Processing rate

In this section, we focus our attention on the processing rate as measured by the FPS metric. The Frames per Second metric is critical in our study as it quantifies the processing rate of the selected CNN models when deployed on the Raspberry Pi 4 Model B platform and in conjunction with the Google Coral USB accelerator.

FPS is particularly pertinent in video processing tasks, where the capability to analyze a high number of frames per second can often directly correlate with the model's efficacy. A higher FPS indicates a higher processing speed, which in real-world surveillance applications translates into a smoother video feed and potentially more accurate object detection, particularly for fast-moving objects.

For our analysis, the FPS of the models is calculated based on the time it takes to infer 1000 frames. Given that FPS is the inverse of inference time, a frame's inference time is effectively measured by determining the CPU time before and after each frame is processed. This inverse relationship highlights the importance of quick inference times in achieving higher FPS. By using this measurement methodology, we aim to accurately gauge the real-world processing capabilities of our selected models under the varied conditions set for the experiments. This evaluation will directly inform us about the models' processing speed, which is a crucial aspect of real-time applications.

6.2.1 Evaluation

Figure 6.1 illustrates the processing rate, measured in FPS, for the evaluated CNN models under varied frame resolution conditions and differing numbers of CPU cores. As can be observed, the frame processing rate is generally low when the CPU

of the Raspberry Pi is utilized for running the models. This outcome can be attributed to the relatively limited computational resources of the Raspberry Pi compared to high-performance desktop CPUs.

Among the models studied, the MOL model consistently outperformed the others in terms of processing rate across all experiments. This might be due to the efficient design of the MOL model, which aims at maximizing performance under constrained computational resources.

In terms of comparison within the SSD and EfficientDet families, we noticed that SSDv1 outpaced both SSDv2 and the EfficientDet series, demonstrating its superior speed efficiency. Interestingly, SSDv2 and EfficientDetv0 showed similar processing rates, while EfficientDetv2 lagged behind. This could be attributed to the increasing complexity in the design of newer versions, which although offers improved accuracy, comes at the expense of processing speed.

The processing rate showed variation with the number of cores utilized. Generally, the lowest FPS was achieved when running on a single core, and the highest when two cores were used. Some models, like MOL and SSDv1, performed better with three cores as compared to four. Conversely, models like SSDv2 and EfficientDet exhibited better performance with four cores. This may be attributed to the models' individual designs, with some being better suited for multi-threading than others.

When we shifted the processing from the Raspberry Pi's CPU to the Google Coral accelerator (as illustrated in Figure 6.2), we observed a significant increase in the processing rate. In particular, SSDv1 and SSDv2 achieved high processing rates, especially with low-resolution frames. The processing rates exceeded 40 FPS, demonstrating the potential for near real-time object detection on edge devices. This significant increase in performance underscores the effectiveness of the Coral accelerator in improving the processing speed of CNN models on resource-constrained edge devices.

While speed in processing is a significant factor in the performance evaluation of

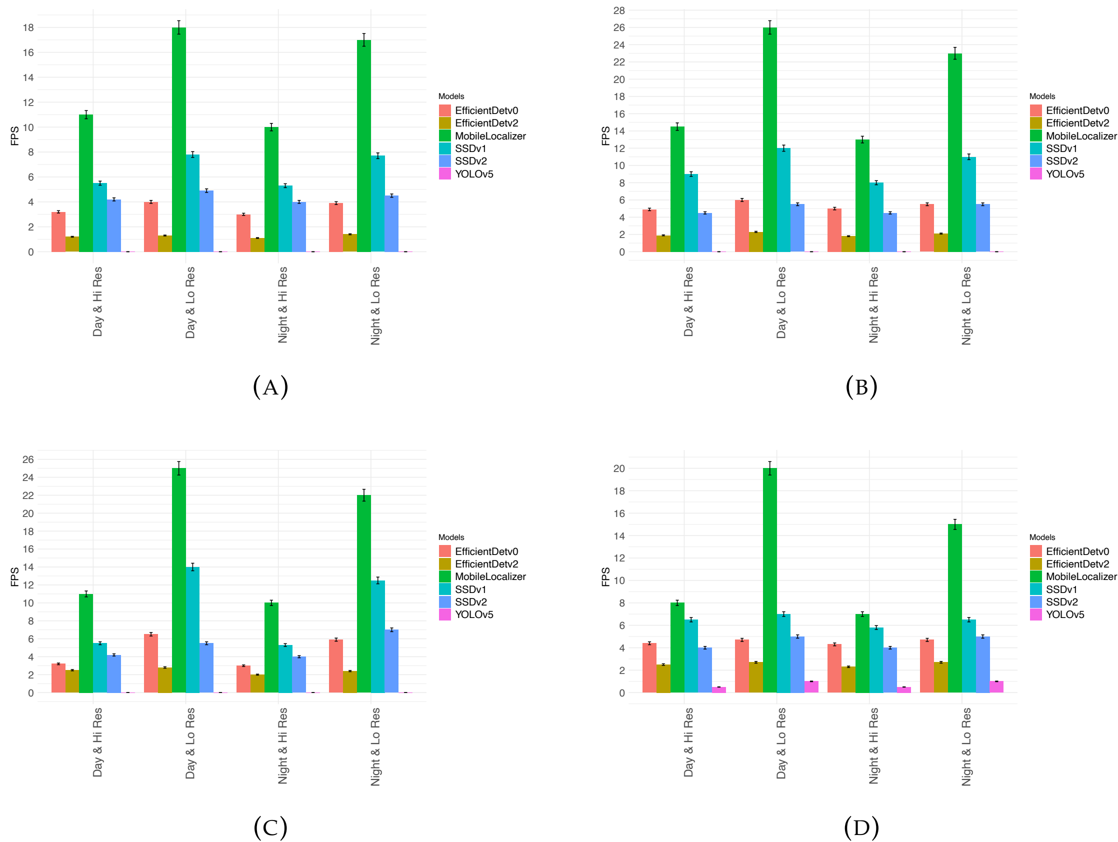


FIGURE 6.1: Processing rate in terms of frames per second: A) 1 CPU core. B) 2 CPU cores. C) 3 CPU cores. D) 4 CPU cores.

the models, it's worth noting that a higher processing rate (FPS) does not automatically equate to better overall model performance. A model that processes frames at

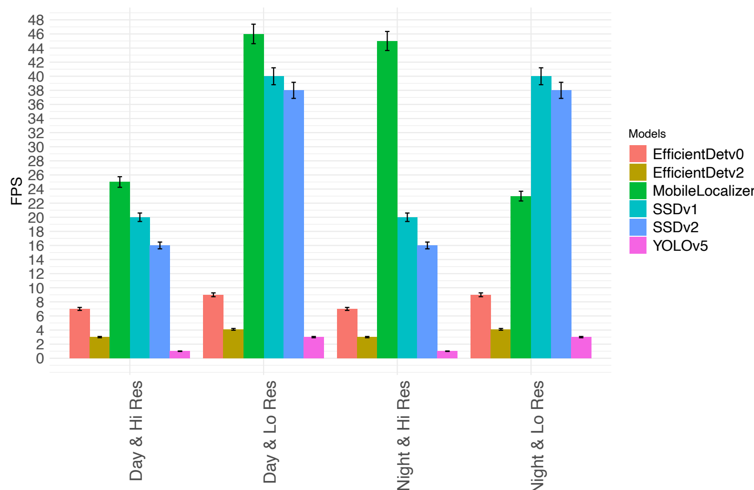


FIGURE 6.2: FPS of the models running on Google USB-CORAL

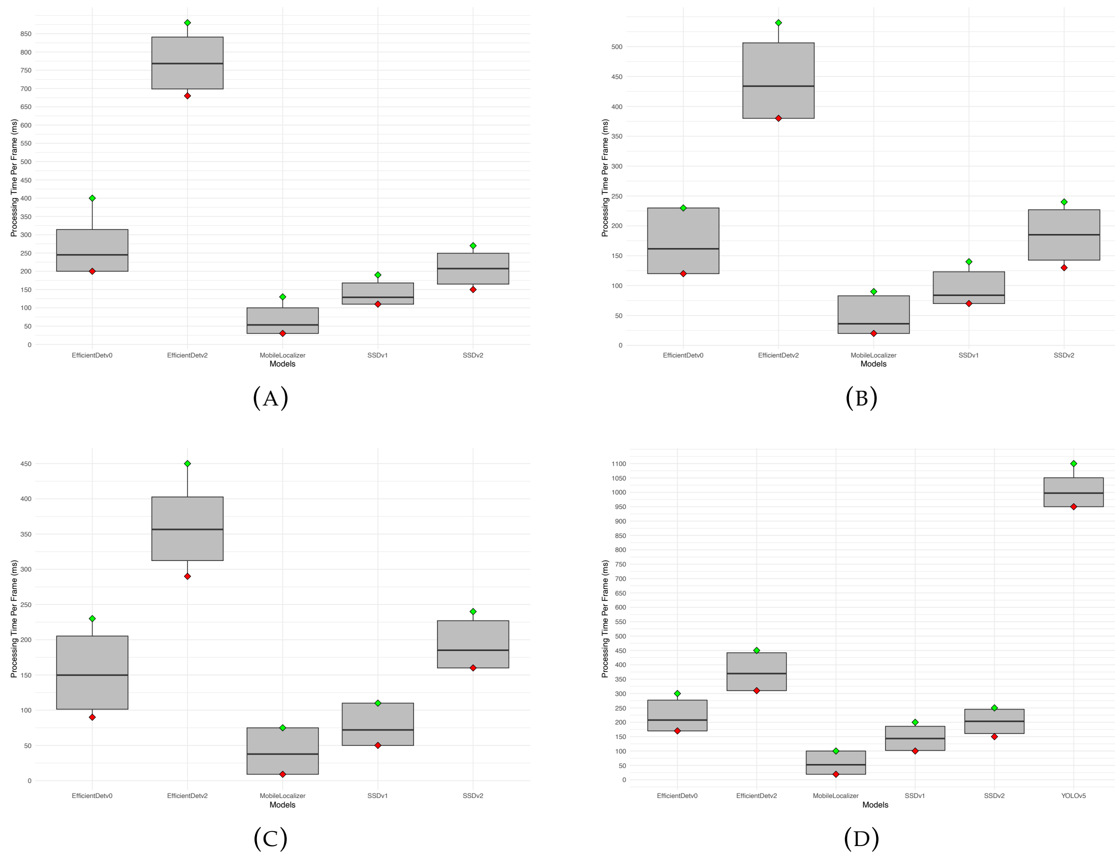


FIGURE 6.3: Box Plot of the Processing Rates on: A) 1 CPU core. B) 2 CPU cores. C) 3 CPU cores. D) 4 CPU cores.

a higher speed may not necessarily be more accurate in its detections or predictions. Accuracy is another vital aspect to consider when evaluating the performance of a model, and in some use cases, it may be even more critical than speed.

Therefore, while the observed faster frame processing of some models like MOL, SSDv1, and SSDv2 might initially seem advantageous, these models' real-world effectiveness would depend on their ability to accurately detect and predict objects within the frames they process. This highlights the importance of evaluating these models on multiple performance metrics, such as the F1 score and the model's resource utilization (CPU, RAM usage, energy consumption, and temperature), and not just their processing speed.

As part of the comprehensive evaluation of the studied object detection models, we have compiled the Frames Per Second (FPS) data for each model when run on varying number of CPU cores on Raspberry Pi and Google's Coral TPU. This data

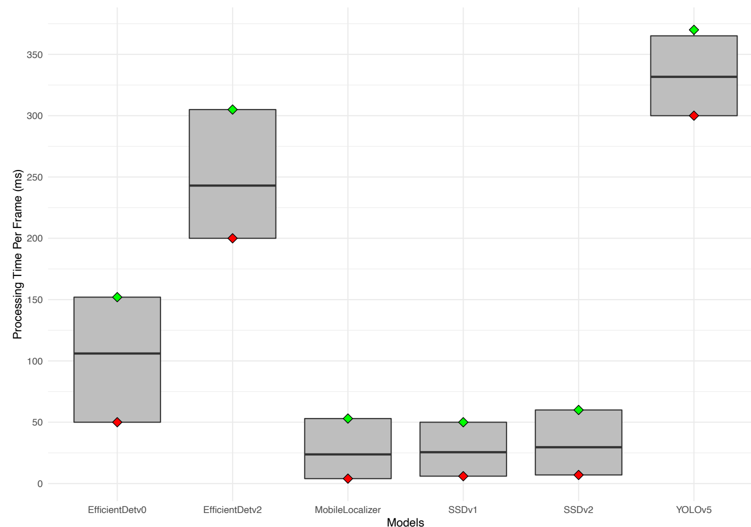


FIGURE 6.4: Box Plot of the Processing Rates of the models running on Google USB-CORAL

is depicted in a series of box plots for further analysis, as seen in Figure 6.3 and 6.4.

The box plots illustrate the spread and skewness of the FPS values for each configuration of cores or hardware accelerator, providing a graphical representation of the central tendency and variability of these measurements. Each box plot represents the FPS of a particular model running on 1, 2, 3, or 4 cores, and the Coral TPU.

From these plots, developers and researchers can discern not only the median FPS for each configuration but also any potential outliers, the range of FPS values, and any trends or patterns in the data. This provides additional depth to the analysis of the performance of these models in various configurations, aiding in the holistic understanding of their real-world applicability and efficiency.

6.2.2 Real-Time Processing of Live Camera Feed

In this study, we focused on the FPS metric as a measure of the processing rate of each model, using pre-recorded video footage for our evaluations. The pre-recorded video allowed us to maintain a consistent benchmark across all models, as each

model was evaluated against the exact same dataset. This, however, does not perfectly represent the complexities and challenges of real-time object detection systems which rely on live camera feeds.

In real-world applications, object detection is typically performed on a continuous stream of live video data. This introduces an additional layer of complexity because the video feed needs to be captured, preprocessed, and fed into the model in real-time. Furthermore, these systems must be designed to handle variations in the incoming frame rate, lighting conditions, and scene dynamics, among other factors.

Designing a real-time object detection system entails not just model selection but also the development of a robust data pipeline. This pipeline would need to handle tasks such as frame capture, preprocessing, model inference, post-processing, and potentially, action initiation based on detection results. This might involve strategies such as queuing mechanisms to buffer incoming frames, or frame dropping techniques to ensure the system can keep pace with the incoming video stream under limited computational resources.

Consequently, while our benchmarking framework and results provide valuable insights into the relative performance and limitations of various object detection models on embedded devices, they represent only one aspect of the overall system design. Practical application of these results should take into account the broader system requirements and constraints, such as how to handle real-time camera feeds. We hope our work can be a foundation for future works, helping researchers and developers make informed decisions when designing real-time object detection systems for edge devices.

6.3 Accuracy

In the forthcoming analysis, our primary focus is on evaluating the accuracy of the selected Convolutional Neural Network models using the F1 score as the primary

metric.

By leveraging this evaluation metric, we aim to understand the true accuracy of our chosen models in object detection tasks under various experimental conditions. It's crucial to remember that a higher F1 score indicates better performance in terms of precision and recall, thus implying a more accurate model.

Let's proceed to examine the F1 score outcomes for our selected models, bearing in mind that while accuracy is a pivotal criterion in model evaluation, it is not the sole factor determining a model's overall efficacy, particularly in edge computing scenarios where computational resources and efficiency play an equally significant role.

TABLE 6.1: Models F1 Score (%)

Objects	SSDv1	SSDv2	Efficient Detv0	Efficient Detv2	YOLOv5	Mobile Localizer
Vehicle	48	67	47	55	60	20
Motorcycle	35	45	40	43	30	20
Person	20	48	22	30	33	20

6.3.1 Evaluation

Table 6.1 provides a detailed comparison of the F1 scores achieved by the various models evaluated on the Raspberry Pi, which serves as an indication of their accuracy. In our study, we grouped the detection accuracies of cars, buses, and trucks under a single 'vehicle' category to facilitate a more coherent comparison.

In terms of F1 scores, SSDv2 yielded the highest accuracy, followed closely by YOLOv5, while the MOL model reported the lowest accuracy. Interestingly, the evaluation showed little difference in accuracy between low-resolution and high-resolution footage. This suggests that reducing the resolution could be a practical approach to boosting processing rates (as substantiated by Figure 1) without compromising the accuracy of detections significantly.

However, it's important to highlight that these models underwent modifications and transformations to the Tensorflow Lite version, which might have affected their

performance. The accuracy results were obtained by averaging the F1 scores over 10,000 frames.

Despite SSDv2 achieving the highest F1 score, it's worth noting that the ground truth was labeled using a full model of SSDv2, which may have introduced some bias in the results. Therefore, while SSDv2 ranked first in terms of accuracy, followed by YOLOv5, EfficientDetv2, SSDv1, EfficientDetv0, and MOL, these rankings should be understood within the context of potential biases and transformational impacts on the models.

The ranking of model accuracy is quite intriguing, given the variety of architectures and the specific strengths each model brings.

SSDv2 emerging as the most accurate model can be attributed to several factors. First, the ground truth labels were generated using a full SSDv2 model, which could inherently favor SSDv2 due to potential similarities in the learning and detection patterns. Beyond this, SSDv2 represents a refined version of the original SSD model, with improvements aimed at better handling of object scales and aspect ratios. This could potentially give it an advantage in a traffic surveillance scenario where objects (vehicles) vary greatly in size depending on their distance from the camera.

YOLOv5's high ranking is particularly noteworthy. Despite being a new and relatively unproven model in the research community as of my knowledge cut-off in September 2021, YOLOv5 is designed to be a lean and efficient model that offers a good balance between speed and accuracy. Its architecture is optimized for real-time object detection tasks, and it has proven quite effective in handling various object scales and classes, which can explain its strong performance in this study.

EfficientDetv2, the third-ranking model, is a part of the EfficientDet series, which is known for its compound scaling method that jointly scales the resolution, depth, and width for all backbone, feature network, and box/class prediction networks. This holistic scaling approach could help it maintain a competitive balance between accuracy and computational resources, even after conversion to Tensorflow Lite format.

SSDv1's higher ranking compared to EfficientDetv0 might seem surprising, given that EfficientDet models are theoretically more efficient and powerful. However, it's worth noting that the conversion of these complex models to Tensorflow Lite might impact their performance, with some features or optimizations potentially not translating well. Furthermore, the SSD architecture, while older, is robust and well-tested, particularly in scenarios involving multiple object classes and varying scales, which could explain its higher ranking.

Lastly, the MOL model ranking at the bottom could be due to its simplicity and minimalistic design, which might limit its ability to accurately detect and classify objects, particularly in complex or variable environments like traffic surveillance.

In conclusion, while these rankings provide useful insights, the overall performance of a model in a given task will depend on a multitude of factors, including the specifics of the task, the quality and nature of the training data, and the computational resources available for model deployment.

6.4 Energy Consumption

In the forthcoming section, we will be diving into an essential aspect of our research, gauging the energy consumption associated with running our machine learning models on the Raspberry Pi and Google Coral TPU. We understand that in real-world applications, especially for edge devices, energy efficiency is a paramount factor that can significantly influence the overall effectiveness and feasibility of a solution. Thus, our focus will not only be on the performance metrics of these models, like processing rate and accuracy, but also on their energy requirements. This detailed analysis will assist us in identifying the most sustainable and resource-friendly models and configurations for our study's context.

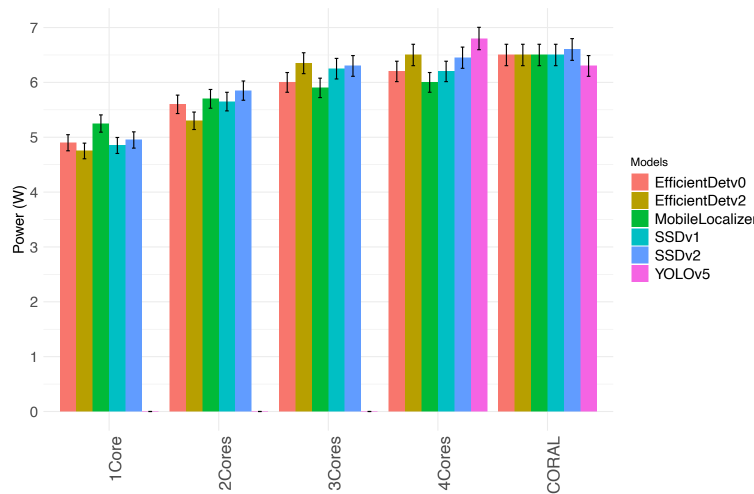


FIGURE 6.5: Energy consumption.

6.4.1 Evaluation

The results portrayed in Figure 6.5 provide us with a comprehensive understanding of the energy consumption of our chosen models when deployed on the Raspberry Pi under varying conditions of CPU core utilization and the usage of Google Coral accelerator. As predicted, the energy usage rises with an increase in the number of CPU cores engaged, which aligns with our understanding of the power consumption characteristics of multi-core processors.

Interestingly, when using the Coral accelerator, the energy consumption is roughly equivalent to the usage when three CPU cores of the Raspberry Pi are active. This balance reflects the efficiency of Coral's Edge TPU in performing computations while maintaining a lower power profile, allowing for energy-efficient, high-speed machine learning inference.

It's also noteworthy that a slight but distinct difference in energy consumption is observed when running models on one core versus four cores, particularly evident with SSDv1 and SSDv2. This variance could be due to the different complexity levels of these models, or the fact that running on four cores involves higher operational overheads and coordination, increasing power consumption.

Among all the models, YOLOv5, when run purely on the Raspberry Pi CPU,

appears to be the most energy-intensive. This may be because of its architecture, which, while being capable of high performance, is not optimized for energy efficiency on low-power devices like the Raspberry Pi.

An essential observation from these results is that when deployed on the Coral accelerator, the energy costs of all the models appear to be roughly the same. This can be attributed to the Coral accelerator's design, which is optimized for low-power, high-performance computations, thus maintaining a consistent energy footprint regardless of the model complexity.

Consequently, these findings offer valuable insights for engineers and practitioners planning to deploy CNN models on edge devices. The trade-off between speed and accuracy becomes a crucial factor in model selection, especially considering that the energy cost of running these models on an accelerator is almost uniform. Hence, a model can be chosen based on its speed and accuracy characteristics rather than its energy consumption when deployed with the Coral accelerator.

In our analysis of the energy consumption of various machine learning models, we took meticulous measures to ensure accuracy in our results. Using a precise USB-C volt meter, we monitored the power consumption of the Raspberry Pi while it processed the inferencing tasks for different models. Each reading represents the average power usage over a span of 10 minutes of continuous inference.

The power consumption is reported in watts, a standard unit for power measurement, which provides a straightforward way to compare the energy efficiency of different models and configurations. An important note in this context is that the idle power consumption of the Raspberry Pi, with no significant computation tasks running, stands around 2 watts. This is a vital baseline to consider when assessing the energy costs added by each model during operation.

This rigorous and methodical approach to capturing and analyzing energy consumption ensures that our findings are robust, reliable, and reflective of real-world scenarios, thereby offering valuable insights for designing and deploying energy-efficient AI solutions on edge devices such as the Raspberry Pi.

6.5 CPU utilization and memory footprint

In the upcoming section, we will be expanding our evaluation of the models by examining two critical operational metrics - CPU utilization and memory footprint. As the computational resources available on edge devices are often constrained, understanding how our selected machine learning models perform under such constraints is fundamental to our study.

To further detail our measurement methodology for CPU utilization and memory footprint, we recorded and averaged these metrics over the course of processing 10,000 frames for each model. This extensive frame processing was employed to ensure a broad and representative sample for the evaluation of our models. For the collection of the Raspberry Pi's performance metrics, we leveraged built-in Raspberry Pi commands that provide real-time and accurate data on CPU utilization and memory usage. This approach allowed us to observe and capture the evolving system resource usage as the models were actively inferring across the large sample of frames.

CPU utilization will allow us to understand the computational demand each model imposes on the Raspberry Pi. A model with high CPU utilization might provide good performance, but it could also limit the device's ability to handle other tasks simultaneously.

Similarly, the memory footprint of each model will shed light on the model's efficiency in using the available memory. Given the limited RAM capacity of devices like Raspberry Pi, a model with a lower memory footprint is likely to be more suitable for deployment.

Together, these metrics will provide valuable insights into the efficiency and practicability of deploying these models on devices with limited computational resources. By understanding these aspects, we can further refine our selection of the most appropriate models for edge computing applications.

6.5.1 Evaluation

CPU utilization

The computational resources demanded by machine learning models during execution can significantly vary based on their architecture, complexity, and the number of parameters they encompass. This variation can be observed in the CPU utilization and memory footprint results.

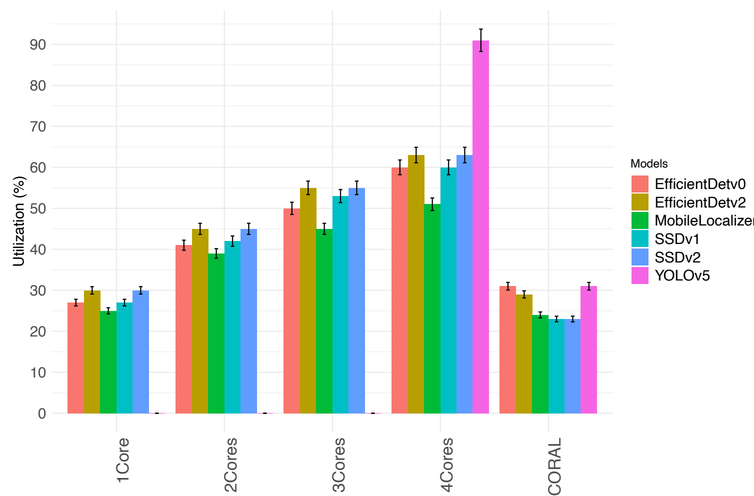


FIGURE 6.6: CPU utilization.

As depicted in Figure 6.6, YOLOv5's CPU utilization is noticeably higher than the other models. This higher utilization could be attributed to YOLOv5's more complex architecture and larger number of parameters compared to the other models. The model's higher computational needs consequently lead to a higher demand on the Raspberry Pi's CPU. This also explains why YOLOv5 can only run efficiently when all cores are utilized. If lesser cores are employed, the performance of YOLOv5 significantly degrades, rendering it impractical for use.

In contrast, the CPU utilization for other models does not display a stark disparity across varying core usage scenarios. All models experience an approximate 15% increase in CPU utilization as the number of cores is incremented. This similarity in CPU utilization across different core configurations suggests that the models are

almost equally efficient in their computational requirements under similar conditions. They are capable of maintaining a consistent performance output even when fewer cores are utilized, which underscores their efficiency and adaptability in diverse hardware settings.

RAM usage

As we shift our focus to memory footprint, shown in Table 6.2, it's evident that employing a Google Coral accelerator substantially reduces the RAM usage of all models by about 180 MB. The reasoning behind this is that the Coral accelerator offloads a portion of the computational workload from the Raspberry Pi, leading to lower memory demands.

TABLE 6.2: RAM usage (in MB)

Models	Raspberry Pi	Google CORAL
SSDv1	490	323
SSDv2	510	324
EfficientDetv0	504	335
EfficientDetv2	520	365
YOLOv5	610	507
MobileLocalizer	490	317

Among all the models, YOLOv5 has the highest RAM usage. This is consistent with the previously observed higher CPU utilization, indicating that YOLOv5 is more resource-intensive overall.

The MobileNet and EfficientDet series, however, have approximately equal RAM usage. This suggests that these models, despite differences in their architectures, have comparable efficiency in memory usage. This could be attributed to their similar design philosophy of balancing efficiency and performance, which leads to optimized memory utilization.

It's important to consider that the size of the models, the training procedures, and the number of parameters involved can all heavily influence the memory usage. Nevertheless, these results provide a valuable overview of the computational

efficiency and practicality of deploying these models on resource-constrained devices like the Raspberry Pi.

6.6 Device's temperature

As part of our comprehensive evaluation methodology, the following section will present our approach to gauge the operating temperature of our Raspberry Pi and Google Coral accelerator devices under different experiment conditions. Considering that excessive heat can be detrimental to both the performance and lifespan of these devices, understanding the thermal behavior under load is a crucial aspect of edge-based deep learning applications. Therefore, we will be discussing the temperature measurements taken during the model inference and explain the possible reasons and impacts behind the variations observed.

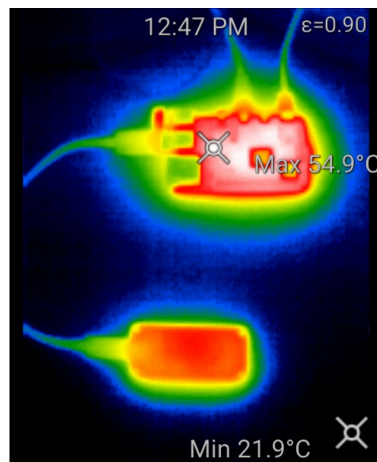


FIGURE 6.7: Gauging temperature of Coral and Pi.

To measure the operating temperature, we utilized a thermal camera as previously discussed like in Figure 6.7. Prior to each model inference, we allowed the Raspberry Pi and the Google Coral accelerator to reach their idle temperature, ensuring a fair and consistent starting point for each test. In order to capture a reliable indication of the temperature under sustained load, we didn't initiate the temperature measurements immediately after starting the model inferences. Instead, we allowed each model to run for at least 10 minutes before beginning the temperature

data collection. This approach was designed to bypass the initial thermal ramp-up period and capture more consistent measurements, representative of long-term operation. Our temperature readings are the average values over the aforementioned duration of running the models. This approach provides a realistic and representative understanding of how these models affect the operating temperature of edge devices over prolonged periods of use.

6.6.1 Evaluation

Figure 6.8b presents the thermal temperature measurements obtained during the execution of various models across different configurations of Raspberry Pi CPU cores and the CORAL accelerator. From these findings, it is evident that the models' inherent characteristics did not substantially influence the device temperature. Instead, the temperature seemed to be dictated more by the number of active CPU cores.

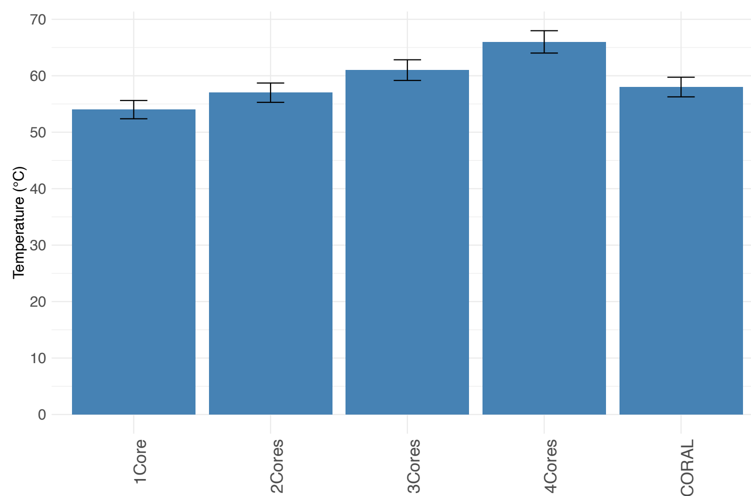


FIGURE 6.8: Processor's temperature.

This behavior can be attributed to the CPU's role in executing computational tasks. A higher number of active cores means more concurrent operations, and thus

more heat is generated due to increased CPU activity. This finding provides valuable insight into system management; to maintain a cooler operating temperature, one might consider employing fewer cores where performance permits.

Interestingly, when the detection tasks were offloaded to the CORAL accelerator, the CPU's temperature response resembled that of a two-core operation. This similarity may be due to the supporting role the CPU plays in managing and interfacing with the accelerator, even when the primary computational task is offloaded. In other words, although the detection task is delegated to the CORAL accelerator, the CPU still undertakes auxiliary activities such as data routing, which may account for the comparable heat generation.

However, it's worth noting that the exact thermal response will depend on various factors such as the model architecture, workload characteristics, and the thermal design of the particular edge device. Thus, these observations should be interpreted in the context of this specific experimental setup. Nonetheless, these results provide valuable insights into the thermal management strategies that could be employed during the deployment of ML models on edge devices.

6.7 Evaluating Trade-offs in Object Detection Systems

In the process of evaluating object detection models on edge devices, it becomes evident that system performance is subject to several trade-offs. As illustrated in Figure 6.9, these trade-offs form a triangle between frames per second (FPS), energy consumption, and temperature.

6.7.1 FPS - Energy - Temperature Trade-off

- If a high FPS and low temperature are desired, this typically results in increased energy consumption. This might be acceptable in scenarios where there is a reliable power source and the priority is on real-time detection and maintaining the device's longevity.

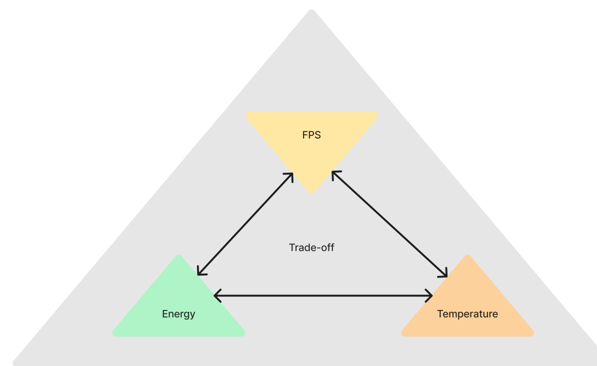


FIGURE 6.9: FPS - Energy - Temperature Trade-off Triangle.

- Conversely, if the objective is to achieve high FPS and conserve energy, the system will likely operate at higher temperatures. This could be a concern in scenarios where the device's temperature can impact its performance, longevity, or the safety of its surroundings.
- Lastly, if conserving energy and keeping the device cool are priorities, this will generally result in lower FPS. This trade-off might be acceptable in non-real-time applications or when the device operates under tight energy constraints.

6.7.2 Accuracy - Memory - CPU - Temperature Trade-off

Accuracy, while not directly in the FPS-Energy-Temperature trade-off triangle, is involved in another trade-off triangle of its own with memory usage, CPU utilization, and temperature Figure 6.10.

- Higher accuracy typically requires more complex models which tend to consume more memory and CPU, leading to increased temperature.
- If keeping the device cool is a priority, it might be necessary to use less complex models that are less accurate but have lower memory and CPU requirements.

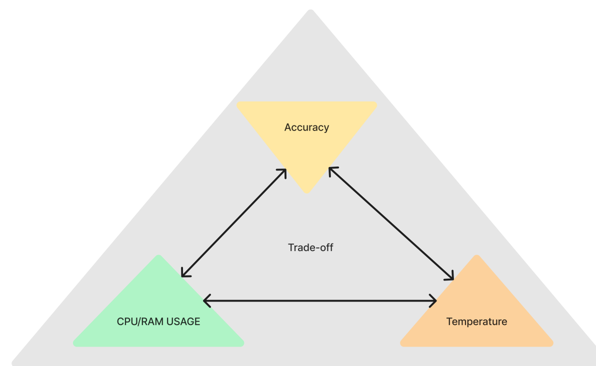


FIGURE 6.10: Accuracy - Memory - CPU - Temperature Trade-off Triangle.

- And if accuracy is the priority, one should be prepared for higher memory usage, increased CPU utilization, and possibly higher device temperatures.

Notably, we found that the trade-off between video resolution and accuracy was not a significant concern in our tests, as accuracy was not substantially impacted until video resolution was reduced by 70%. This observation suggests that resolution reduction can be a viable strategy for improving real-time performance without significant loss of accuracy.

6.7.3 Conclusion

In conclusion, the choice of the most suitable object detection model and system configuration will depend on the specific requirements of the application and the constraints of the operating environment. For instance, if the end product is a wearable device designed to assist visually impaired individuals with real-time object detection, high FPS and low temperature would likely be the primary concerns to ensure user comfort and instant feedback. This would mean developers should anticipate higher energy consumption and select components accordingly.

Our results provide valuable insights into these trade-offs, enabling developers to make informed decisions when designing their real-time object detection systems.

These considerations are vital in the development of practical, effective, and efficient solutions, like the aforementioned wearable, for real-world challenges.

6.8 Conclusion

As we conclude this evaluation chapter, we have observed that each of the machine learning models—MOL, SSDv1, SSDv2, EfficientDetv0, EfficientDetv2, and YOLOv5—brings its unique set of strengths and weaknesses when deployed on the Raspberry Pi and Google Coral accelerator platform. Our analysis was focused on key performance indicators such as frames per second, F1 score, energy consumption, CPU and RAM utilization, and temperature under different operating conditions.

The results revealed a range of performance characteristics, reflecting the diversity of model designs. In terms of frame processing rate (FPS), the MOL model exhibited the highest performance across all experimental conditions. On the other hand, SSDv2 scored highest in terms of accuracy (F1 score), closely followed by YOLOv5. As for energy consumption and CPU utilization, these metrics showed a general trend of increasing as more CPU cores were used, but with YOLOv5 consuming more power and CPU resources than the other models. Lastly, in terms of temperature, our results indicated that the number of active cores was the most influential factor, rather than the model's inherent characteristics.

However, it is essential to note that these metrics cannot be considered in isolation when deciding on the "best" model. The choice will depend on the specific needs of the application, the constraints of the operating environment, and the trade-offs one is willing to make between speed, accuracy, power consumption, resource utilization, and thermal constraints. A unified metric, incorporating all these factors, could provide a more holistic measure of a model's suitability for a given scenario. That being said, defining such a metric is beyond the scope of this current study but offers a fascinating avenue for future exploration.

TABLE 6.3: Guideline for Model Selection Based on Various Criteria

Criteria	**Recommended Model**	**Justification**	**Considerations for Application**
Accuracy	SSDv2	Achieved the highest F1 score	Ideal for applications where detection accuracy is paramount
Speed	SSDv1	Balance between speed and accuracy	Suitable for real-time applications where quick responses are essential
Energy Efficiency	EfficientDet	Lower energy consumption	Beneficial for battery-operated devices or where power usage is a constraint
Low Temperature	Models with Coral TPU	Lower operational temperature	Crucial for wearable technologies and prolonging device lifespan
Memory Efficiency	MOL	Lower RAM usage	Preferred in scenarios where memory resources are limited

Each selection should consider the specific requirements and constraints of the intended application, ensuring that the chosen model aligns with the operational, environmental, and performance needs. Also, remember that real-world deployment might require further optimizations and customizations to achieve the desired performance and efficiency.

With this comprehensive evaluation of our chosen models complete, we are nearing the end of our experimental journey. In the following chapter, we will discuss potential future work that could build upon these findings, and provide a concluding summary of our research, articulating the implications and applications of our study.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The primary motivation for our study was the ever-growing need for efficient and effective machine learning models suitable for edge computing devices, particularly for object detection tasks. Our journey in this field was driven by the understanding that the use of smaller, low-power devices such as the Raspberry Pi and Google Coral accelerator could offer substantial benefits in terms of cost, portability, privacy, and latency for many applications.

Our research began with an exploration of the theoretical underpinnings and historical evolution of CNNs in the background study. We investigated how these technologies have grown and evolved over time, delving into the principles behind CNNs, transfer learning, object detection, and edge computing.

In the literature review chapter, we provided a comprehensive examination of existing studies and technologies related to object detection on edge devices. The literature revealed a wide array of approaches and models, but it also highlighted the need for comparative studies that directly assess the trade-offs between different models when deployed on edge computing platforms.

To address this research gap, we embarked on an evaluation of six prominent CNN-based models: MOL, SSDv1, SSDv2, EfficientDetv0, EfficientDetv2, and YOLOv5.

In the models' overview chapter, we provided a detailed description of the architecture and functioning of these models, enabling a clear understanding of the methodologies used in each case.

Subsequently, in the methodology chapter, we described our experimental design, detailing how we adapted and optimized these models for deployment on a Raspberry Pi and Google Coral accelerator. We also elaborated on our metric selection, explaining why we chose the FPS, F1 score, energy consumption, CPU and RAM utilization, and temperature as our evaluation metrics.

In the evaluation results chapter, we presented our findings from the various experiments conducted under different operating conditions. Our study revealed the unique performance characteristics of each model across the chosen metrics. It underscored that while some models excel in certain aspects like processing speed (MOL) or accuracy (SSDv2), they might not be as proficient in others, such as energy consumption or resource utilization.

The results of our research, however, are not intended to provide a definitive "best" model, but rather to offer comprehensive insights into the strengths and weaknesses of each model under a variety of conditions. Our primary emphasis was to design and implement a systematic benchmarking framework that allows for thorough evaluation and comparison of these models. The framework, we believe, is a critical component to this study as it provides a structured methodology to evaluate the trade-offs between accuracy, speed, resource consumption, and other factors crucial in an edge computing context.

The most suitable model ultimately depends largely on the specific requirements of a given application and the constraints of the operating environment. By considering the detailed insights from our benchmarking framework, practitioners can make more informed decisions tailored to their specific needs.

7.2 Future Works

This research, although extensive, opens up a multitude of avenues for future explorations. Some potential directions for future work are as follows:

- **Unified Evaluation Metric:** Throughout our research, we evaluated various performance metrics independently. A possible improvement could involve the development of a unified evaluation metric that holistically combines accuracy, processing speed, energy consumption, CPU utilization, memory footprint, and device temperature. Such a metric would allow for a more comprehensive comparison of models, providing a singular measure that balances the trade-offs among the various factors.

- **Expanding Model Evaluation:** We focused on specific CNN models in our study, but the landscape of object detection models is vast and continuously evolving. Future work could incorporate a wider variety of models, including newer versions of existing models or entirely different architectures. This expansion would provide a more complete picture of the performance spectrum of object detection models.

- **Exploring Different Accelerators:** This study focused on the use of Google's Coral TPU. However, there are other accelerators available, such as NVIDIA's Jetson series or Intel's Movidius series, which offer different performance characteristics. Future research could explore the effects of these accelerators on the performance of object detection models.

- **Custom Model Training:** We used pre-trained models for our study. For more targeted applications, models could be trained to detect specific objects of interest. This customization might improve the performance of the models due to the reduced complexity of the detection task.

- **Hybrid Cloud-Edge Computing:** In our work, we focused entirely on edge computing. However, hybrid models that leverage both edge and cloud computing could be explored. In such a setup, some frames could be processed on the edge device, while others are offloaded to the cloud, depending on factors like network availability and the complexity of the scene. This approach could balance the strengths of edge and cloud computing, potentially leading to even more efficient real-time object detection systems.

By following these directions, future work could continue to advance the field of real-time object detection on edge devices, providing increasingly effective solutions for a wide range of applications.

Bibliography

[1] A. Anjum, T. Abdullah, M. Tariq, Y. Baltaci, and N. Antonopoulos, "Video stream analysis in clouds: An object detection and classification framework for high performance video analytics," *IEEE Transactions on Cloud Computing*, vol. 7, pp. 1–1, Jan. 2016. DOI: 10.1109/TCC.2016.2517653.

[2] A. R. Pathak, M. Pandey, and S. Rautaray, "Application of deep learning for object detection," *Procedia Computer Science*, vol. 132, pp. 1706–1717, 2018, International Conference on Computational Intelligence and Data Science, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2018.05.144. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050918308767.

[3] S. Shahzadi, M. Iqbal, T. Dagiuklas, and Z. U. Qayyum, "Multi-access edge computing: Open issues, challenges and future perspectives," *Journal of Cloud Computing*, vol. 6, no. 1, p. 30, 2017, ISSN: 2192-113X. DOI: 10.1186/s13677-017-0097-9. [Online]. Available: https://doi.org/10.1186/s13677-017-0097-9.

[4] J. Ren, G. Yu, Y. Cai, Y. He, and F. Qu, "Partial offloading for latency minimization in mobile-edge computing," Dec. 2017, pp. 1–6. DOI: 10.1109/GLOCOM.2017.8254550.

[5] Y. Li *et al.*, "Reducto: On-camera filtering for resource-efficient real-time video analytics," in *Proc. of the ACM SIGCOMM*, 2020, 359–376.

[6] W. Qian and R. W. L. Coutinho, "On the design of edge-assisted mobile iot augmented and mixed reality applications," in *Proc. of the 17th ACM Q2SWinet*, 2021, 131–136.

[7] Q. Liu *et al.*, "An edge network orchestrator for mobile augmented reality," in *Proc. of the IEEE INFOCOM*, 2018, pp. 756–764.

[8] P. P. Shahrbabaki *et al.*, "A novel sdn-enabled edge computing load balancing scheme for iot video analytics," in *Proc. of the IEEE GLOBECOM*, 2022, pp. 5025–5030.

[9] W. Qian and R. W. L. Coutinho, "Performance evaluation of edge computing-aided iot augmented reality systems," in *Proc. of the 18th ACM Q2SWinet*, 2022, 79–86.

[10] E. Saeed and R. W. L. Coutinho, "Performance evaluation of edge computing models for internet of things," in *Proc. of the 12th ACM DIVANet*, 2022, 63–69.

[11] R. W. L. Coutinho and A. Boukerche, "Modeling and performance evaluation of collaborative IoT cross-camera video analytics," in *Proc. of the IEEE Int'l Conf. on Communications (ICC)*, 2023, pp. 1–6.

[12] S. Hijazi, R. Kumar, C. Rowen, *et al.*, "Using convolutional neural networks for image recognition," *Cadence Design Systems Inc.: San Jose, CA, USA*, vol. 9, p. 1, 2015.

[13] Y. Wang, L. Xia, T. Tang, *et al.*, "Low power convolutional neural networks on a chip," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2016, pp. 129–132.

[14] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, "Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions," *ACM Comput. Surv.*, vol. 53, no. 4, 2020, ISSN: 0360-0300. DOI: 10.1145/3398209. [Online]. Available: https://doi.org/10.1145/3398209.

[15] S. Saponara, A. Elhanashi, and A. Gagliardi, “Real-time video fire/smoke detection based on cnn in antifire surveillance systems,” *Journal of Real-Time Image Processing*, vol. 18, pp. 889–900, 2021.

[16] A. C. Cob-Parro, C. Losada-Gutiérrez, M. Marrón-Romera, A. Gardel-Vicente, and I. Bravo-Muñoz, “Smart video surveillance system based on edge computing,” *Sensors*, vol. 21, no. 9, p. 2958, 2021.

[17] A. B. Khudhair and R. F. Ghani, “Iot based smart video surveillance system using convolutional neural network,” in *2020 6th International Engineering Conference “Sustainable Technology and Development”(IEC)*, IEEE, 2020, pp. 163–168.

[18] L. Chen, S. Lin, X. Lu, *et al.*, “Deep neural network based vehicle and pedestrian detection for autonomous driving: A survey,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 6, pp. 3234–3246, 2021.

[19] M. G. Bechtel, E. McEllhiney, M. Kim, and H. Yun, “Deeppicar: A low-cost deep neural network-based autonomous car,” in *2018 IEEE 24th international conference on embedded and real-time computing systems and applications (RTCSA)*, IEEE, 2018, pp. 11–21.

[20] A. Burger, C. Qian, G. Schiele, and D. Helms, “An embedded cnn implementation for on-device ecg analysis,” in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, IEEE, 2020, pp. 1–6.

[21] S Vimal, Y. H. Robinson, S. Kadry, H. V. Long, and Y. Nam, “Iot based smart health monitoring with cnn using edge computing,” *Journal of Internet Technology*, vol. 22, no. 1, pp. 173–185, 2021.

[22] W. Caesarendra, T. A. Hishamuddin, D. T. C. Lai, *et al.*, “An embedded system using convolutional neural network model for online and real-time ecg signal classification and prediction,” *Diagnostics*, vol. 12, no. 4, p. 795, 2022.

[23] Y. Wang, M. Liu, P. Zheng, H. Yang, and J. Zou, "A smart surface inspection system using faster r-cnn in cloud-edge computing environment," *Advanced Engineering Informatics*, vol. 43, p. 101 037, 2020.

[24] S. Han, F. Yang, G. Yang, B. Gao, N. Zhang, and D. Wang, "Electrical equipment identification in infrared images based on roi-selected cnn method," *Electric Power Systems Research*, vol. 188, p. 106 534, 2020.

[25] I. Zualkernan, J. Judas, T. Mahbub, A. Bhagwagar, and P. Chand, "A tiny cnn architecture for identifying bat species from echolocation calls," in *2020 IEEE/ITU International Conference on Artificial Intelligence for Good (AI4G)*, IEEE, 2020, pp. 81–86.

[26] X. Liu, Z. Jia, X. Hou, M. Fu, L. Ma, and Q. Sun, "Real-time marine animal images classification by embedded system based on mobilenet and transfer learning," in *OCEANS 2019-Marseille*, IEEE, 2019, pp. 1–5.

[27] S. CK *et al.*, "Automated wildlife monitoring using deep learning," in *proceedings of the International Conference on Systems, Energy & Environment (ICSEE)*, 2019.

[28] W. Gay, *Raspberry Pi Hardware Reference*. Jan. 2014, ISBN: 978-1-4842-0800-7. DOI: 10.1007/978-1-4842-0799-4.

[29] A. A. Süzen, B. Duman, and B. Şen, "Benchmark analysis of jetson tx2, jetson nano and raspberry pi using deep-cnn," in *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, 2020, pp. 1–5. DOI: 10.1109/HORA49412.2020.9152915.

[30] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[31] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *ArXiv e-prints*, Nov. 2015.

[32] Z. Zou, K. Chen, Z. Shi, Y. Guo, and J. Ye, "Object detection in 20 years: A survey," *Proceedings of the IEEE*, vol. 111, no. 3, pp. 257–276, 2023. DOI: 10.1109/JPROC.2023.3238524.

[33] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2015. arXiv: 1409.1556 [cs.CV].

[34] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[35] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. arXiv: 1704.04861.

[36] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, 2014, pp. 580–587. DOI: 10.1109/CVPR.2014.81.

[37] R. Girshick, "Fast r-cnn," in *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1440–1448. DOI: 10.1109/ICCV.2015.169.

[38] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, 2017. DOI: 10.1109/TPAMI.2016.2577031.

[39] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[40] W. Liu, D. Anguelov, D. Erhan, *et al.*, "Ssd: Single shot multibox detector," in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling,

Eds., Cham: Springer International Publishing, 2016, pp. 21–37, ISBN: 978-3-319-46448-0.

[41] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *ArXiv*, vol. abs/2004.10934, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:216080778.

[42] M. Sha and A. Boukerche, "Performance evaluation of cnn-based pedestrian detectors for autonomous vehicles," *Ad Hoc Networks*, vol. 128, p. 102 784, 2022, ISSN: 1570-8705. DOI: https://doi.org/10.1016/j.adhoc.2022.102784. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S157087052200004X.

[43] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2980–2988. DOI: 10.1109/ICCV.2017.322.

[44] L. Tychsen-Smith and L. Petersson, "Denet: Scalable real-time object detection with directed sparse sampling," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 428–436. DOI: 10.1109/ICCV.2017.54.

[45] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, *Integer quantization for deep learning inference: Principles and empirical evaluation*, 2020. arXiv: 2004.09602 [cs.LG].

[46] Z. Wang, C. Li, and X. Wang, "Convolutional neural network pruning with structural redundancy reduction," in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021, pp. 14 908–14 917. DOI: 10.1109/CVPR46437.2021.01467.

[47] G. Hinton, O. Vinyals, and J. Dean, *Distilling the knowledge in a neural network*, 2015. arXiv: 1503.02531 [stat.ML].

[48] L. Alzubaidi *et al.*, "Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions," *Journal of Big Data*, vol. 8, no. 1,

p. 53, 2021, ISSN: 2196-1115. DOI: 10.1186/s40537-021-00444-8. [Online]. Available: https://doi.org/10.1186/s40537-021-00444-8.

[49] W. Li and M. Liewig, "A survey of ai accelerators for edge environment," in *Trends and Innovations in Information Systems and Technologies*, Á. Rocha, H. Adeli, L. P. Reis, S. Costanzo, I. Orovic, and F. Moreira, Eds., Cham: Springer International Publishing, 2020, pp. 35–44, ISBN: 978-3-030-45691-7.

[50] K. Sato, *An in-depth look at google's first tensor processing unit (tpu) | google cloud blog*. [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

[51] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, *Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size*, 2016. arXiv: 1602.07360 [cs.CV].

[52] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[53] P. Adarsh, P. Rathi, and M. Kumar, "Yolo v3-tiny: Object detection and recognition using one stage improved model," in *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2020, pp. 687–694. DOI: 10.1109/ICACCS48705.2020.9074315.

[54] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," Jun. 2018, pp. 6848–6856. DOI: 10.1109/CVPR.2018.00716.

[55] H. Mao, S. Yao, T. Tang, B. Li, J. Yao, and Y. Wang, "Towards real-time object detection on embedded systems," *IEEE Transactions on Emerging Topics in Computing*, vol. 6, no. 3, pp. 417–431, 2018. DOI: 10.1109/TETC.2016.2593643.

[56] "Performance analysis of the pretrained efficientdet for real-time object detection on raspberry pi," English, in *2021 International Conference on Circuits, Controls and Communications, CCUBE 2021*, ser. 2021 International Conference on Circuits, Controls and Communications, CCUBE 2021, United States: Institute of Electrical and Electronics Engineers Inc., 2021. DOI: 10.1109/CCUBE53681.2021.9702741.

[57] A. A. Süzen, B. Duman, and B. Şen, "Benchmark analysis of jetson tx2, jetson nano and raspberry pi using deep-cnn," in *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, 2020, pp. 1–5. DOI: 10.1109/HORA49412.2020.9152915.

[58] J. Zhu, H. Feng, S. Zhong, and T. Yuan, "Performance analysis of real-time object detection on jetson device," in *2022 IEEE/ACIS 22nd International Conference on Computer and Information Science (ICIS)*, 2022, pp. 156–161. DOI: 10.1109/ICIS54925.2022.9882480.

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

[60] H. Feng, G. Mu, S. Zhong, P. Zhang, and T. Yuan, "Benchmark analysis of yolo performance on edge intelligence devices," in *2021 Cross Strait Radio Science and Wireless Technology Conference (CSRSWTC)*, 2021, pp. 319–321. DOI: 10.1109/CSRSWTC52801.2021.9631594.

[61] N. Anggraini, S. H. Ramadhani, L. K. Wardhani, N. Hakiem, I. M. Shofi, and M. T. Rosyadi, "Development of face mask detection using ssdlite mobilenetv3

small on raspberry pi 4," in *2022 5th International Conference of Computer and Informatics Engineering (IC2IE)*, 2022, pp. 209–214. DOI: 10.1109/IC2IE56416.2022.9970078.

[62] D. Velasco-Montero, J. Fernández-Berni, R. Carmona-Galan, and Rodríguez-Vázquez, "Performance analysis of real-time dnn inference on raspberry pi," May 2018, p. 14. DOI: 10.1117/12.2309763.

[63] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[64] Y.-C. Chiu, C.-Y. Tsai, M.-D. Ruan, G.-Y. Shen, and T.-T. Lee, "Mobilenet-ssdv2: An improved object detection model for embedded systems," in *2020 International conference on system science and engineering (ICSSE)*, IEEE, 2020, pp. 1–5.

[65] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 10 781–10 790.

[66] J. Redmon *et al.*, "You only look once: Unified, real-time object detection," in *Proc. of the IEEE CVPR*, 2016, pp. 779–788.

[67] J. Glenn, *Ultralytics/yolov5*, Accessed on 3/11/2023, 2020. [Online]. Available: https://github.com/ultralytics/yolov5.

[68] Google, *A class-agnostic mobile object detector*, Accessed on 3/11/2023, 2021. [Online]. Available: https://tfhub.dev/google/object_detection/mobile_object_localizer_v1/1.

[69] T.-Y. Lin *et al.*, "Microsoft coco: Common objects in context," in *Computer Vision – ECCV 2014*, Springer International Publishing, 2014, pp. 740–755.

[70] Google, *Tensorflow hub*. [Online]. Available: https://www.tensorflow.org/hub.

[71] Z. Xiao *et al.*, "Towards performance clarity of edge video analytics," in *Proc. of the IEEE/ACM Symposium on Edge Computing (SEC)*, 2021, pp. 148–164.

[72] R. P. Foundation, *Raspberry pi 4 model b*. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/?variant=raspberry-pi-4-model-b-8gb.

[73] Google, *Tensorflow models on the edge tpu*. [Online]. Available: https://coral.ai/docs/edgetpu/models-intro/.

[74] Google, *Tensorflowlite*. [Online]. Available: https://www.tensorflow.org/lite.

[75] M. Sokolova, N. Japkowicz, and S. Szpakowicz, "Beyond accuracy, f-score and roc: A family of discriminant measures for performance evaluation," vol. Vol. 4304, Jan. 2006, pp. 1015–1021.