Abstract

This study delves into the development of an object identification technique, specifically focusing on implementing the YOLOv8n model for edge devices like the Raspberry Pi. The aim is to strike a balance between accuracy and inference time, highlighting the advantages of model quantization in terms of efficiency gains and reduced processing overhead. This paper emphasizes techniques such as data augmentation, model conversion, and performance optimization to improve efficiency, real-time performance and energy consumption. Through a comprehensive experimental setup involving a Raspberry Pi and Google Colab, the study investigates enhancements in object identification on edge devices. Implementation of YOLOv8n models on platforms like Raspberry Pi, coupled with quantization, demonstrated reduced inference time while maintaining accuracy. Specifically, the inference time of the main YOLOv8n model was 1534.2ms, whereas after applying quantization, multiple quantized models were obtained, with one model achieving an inference time reduction of approximately 77%, with an inference time of 356.91ms. The deployment on Google Colab with a T4 GPU showcased efficient processing capabilities. Utilizing a custom dataset of 90,000 images, the approach achieved a precision of 0.664, recall of 0.611, mAP50 of 0.668, and mAP50-95 of 0.446. The novelty of this study lies in the development of an object identification technique leveraging the YOLOv8n model for edge devices like Raspberry Pi. The implementation of model quantization to balance accuracy and inference time represents a significant advancement in efficient processing for resource-constrained devices. This approach not only improves efficiency but also maintains high accuracy levels, making the optimized models suitable for real-world deployment on edge devices.

Keywords: *Data Augmentation, Edge Devices, Model Quantization, Object Identification, Performance Optimization, Real-time Processing, Raspberry Pi, YOLOv8n Model,.*

Table of Contents

List of Figures

List of Tables

List of Abbreviations

CNN	Convolutional Neural Network
DFD	Data Flow Diagram
WBS	Work Breakdown System
YOLO	You Only Look Once

Chapter 1

Introduction

Object detection is the task of identifying and locating objects in images or videos. Traditional object detection methods are computationally expensive and not suitable for mobile devices. Deep learning has been used to develop more efficient and accurate object detection methods for edge devices. Some popular deep learning-based object detection algorithms for edge devices include YOLO, SSD, and Faster R-CNN. Deep learning-based object detection on edge devices is a promising technology with the potential to revolutionize the way we interact with the world around us. Convolutional neural networks, or CNNs, have shown remarkable performance in picture recognition tasks including object detection. The YOLO approach is one such CNN-based object identification framework that has become widely used in computer vision applications [1].

Although the YOLO and its variations are not as accurate as two-stage detectors, they function significantly faster than their rivals. YOLO performs effectively when faced with items of ordinary size; nevertheless, it is not able to identify small objects [2].

Object detection is a computer vision technique that identifies and locates objects in images or videos. It also needs to localize the objects, i.e., to determine their precise location in the image. There are many different object detection algorithms, but most of them work by first dividing the image into a grid of cells. Then, each cell is responsible for predicting whether or not an object is present in that cell. The most common algorithms for object detection are based on Deep learning. Deep learning-based object detection algorithms have been shown to be very accurate, but they can be computationally expensive.

1.1 Motivation

The necessity for effective and easily available solutions in the age of IoT and smart technology is the driving force behind the development of a deep learning model for object recognition on edge devices. Our methodology focuses on developing a small, light model that may be used on low-end devices. Numerous applications are made possible by this breakthrough, ranging from improving the capabilities of edge devices in remote, offline, or data-sensitive settings to real- time surveillance in contexts with limited resources. We enable these devices to execute object identification tasks and democratize the use of deep learning by optimizing for tiny size and resource efficiency, opening up cutting-edge technology to a wider user base.

1.2 Problem Statement

The main focus is on locating different object in a given image which is done by identifying the bounding boxes around each object. The other challenge faced is based on compression of model. Despite these challenges, deep learning has shown to be very effective. There are a number of deep learning models that havebeen developed for object detection that include various compression techniques like Quantization, Knowledge Distillation, Pruning. As Deep learning research continues to advance, it is likely that object detection will become more accurate and efficient. This will enable a wide range of applications such as self driving cars, video surveillance and medical imaging. Latency problems might arise when self-driving cars use cloud computing to predict objects through detection. This is due to the fact that data must be sent to and from the cloud, which cantake some time. Since the deep learning model's predictions are created on-board,reducing latency can be achieved by compressing the model and applying it tothe self-driving car.

1.3 Objectives

The objectives are as follows:

- Real-time object detection at high frame rates and minimal latency can be attained for practical uses.
- For edge devices with limited memory, processing power, and energy consumption, optimise object detection models.
- Sustain excellent detection robustness and accuracy in the face of changing lighting, occlusions, and complicated backdrops.
- By processing data locally on the edge device and reducing data transfer to the cloud, you may protect user privacy.
- Create models for adaptive object identification that may be adjusted to the capabilities and needs of certain edge devices.

1.4 Scope

Present a comprehensive overview of contemporary object detection models optimized for deployment on edge devices, including YOLO, SSD, MobileNet, and Tiny models. Emphasize the inherent trade-offs between computational efficiency and model precision within these frameworks. Elaborate on the data collection and preprocessing procedures required for training object detection models, while addressing the specific challenges associated with working with constrained data on edge devices. The project aims to provide an up-to-date examination of object detection models suitable for edgedeployment, focusing on key models like YOLO, SSD, MobileNet, and Tiny variants. Thisexamination will underscore the delicate balance between computational efficiency and model accuracy. Furthermore, the project will delve into the intricacies of data acquisition and preprocessing, especially relevant when dealing with limited data on edge devices."

Chapter 2

Review of Literature

2.1 Real-Time Flying Object Detection with YOLOv8

First, an improved model for the real-time identification of flying objects was proposed by Jordan Kupec *et al*. object detection study [8] by applying learning from a dataset representative of real-world situations to a model trained on a dataset with 40 different classes of flying objects. The lack of discussion of the models' abilities were to identify particular kinds of flying objects and tell them apart from other items in the picture is a negative. The research presented an improved transfer learning model that maintains speed and improved mAP50-95 to 0.835, while the extended model delivered real-time flying object identification with a mAP50-95 of 0.685 and 50 fps on 1080p films.

2.2. Real-Time Vehicle Accident Recognition from Traffic Video Surveillance using YOLOV8 and OpenCV

The study conducted by Deepak T. Mane *et al.* [9], using the YOLOv8 approach, the paper proposed an ensemble model for precise and effective event recognition in traffic video surveillance. The model was trained using the publicly available Crash Car Detection Dataset. It showed improvements in approaches in terms of mAP, accuracy, and recall. The effectiveness of the model in real-time traffic monitoring applications was assessed using traffic video data. An ensemble model with 91.3% precision, 93.8% mean average precision (mAP), and 96.1% overall accuracy that made use of OpenCV and YOLOv8 outperformed the findings.

2.3. Real-Time Onboard Object Detection for Augmented Reality: Enhancing Head-Mounted Display with YOLOv8

In the paper of Mikołaj Łysakowski *et al.* [10], the creation of real-time augmented reality object detection software is given. YOLOv8 on HoloLens 2 preserved accuracy and enhanced perception without relying on the cloud. The paper detailed the Microsoft HoloLens 2 onboard implementation, image processing, and evaluation using the COCO dataset for real-time object detection in an augmented reality setting.

2.4. Screening Autism Spectrum Disorder in childrens using Deep Learning Approach : Evaluating the classification model of YOLOv8 by comparing with other models.

The paper by Gautam S *et al.* [11], offered a practical method that used face images and the YOLOv8 deep learning model to assess children for autism spectrum disorder (ASD). Clinical findings of the variations in facial characteristics between photos with and without ASD are supported by this. By gathering sensitive spatial and contextual information with fewer parameters, the work demonstrated the promise of deep learning in ASD screening, and the versatility of YOLOv8 for classification is demonstrated.

2.5. American Sign Language Detection using YOLOv5 and YOLOv8

In the work by Shobhit Tyagi *et al.* [12], the YOLOv5 and YOLOv8 models were applied in a novel approach for American Sign Language (ASL) recognition. With the proposed custom model obtaining considerable mAP, recall, and accuracy, YOLOv8 outperformed earlier iterations. However, some of the drawbacks included limited assessment criteria, lack of thorough implementation data, and a singular emphasis on ASL without taking into account broader sign language applications.

2.6. A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS

The history of YOLO designs, from YOLOv1 to YOLOv8 and YOLO-NAS, is thoroughly evaluated in this study by Treven J *et al.* [13], which also examined modifications in network architecture and training methodologies. It highlighted significant insights, explained typical measures and postprocessing techniques, and provided a view on YOLO's future along with potential study fields. Its focus on YOLO is one of its drawbacks; it also lacks thorough comparative analyses and empirical backing for prospective advancements in the future.

2.7. Revolutionizing Target Detection in Intelligent Traffic Systems: YOLOv8-SnakeVision

In order to improve target recognition in intelligent traffic systems, Yang Liu *et al.* [14] provided YOLOv8SnakeVision in their article. It combined the Wise-IoU approach, Context Aggregation Attention Mechanisms, and Dynamic Snake Convolution for better performance, especially in complicated settings. Nevertheless, there were several drawbacks, such as incomplete comparisons, evaluations based on small datasets, unresolved processing complexity, and a lack of understanding of generalizability to a variety of contexts beyond traffic systems. The proprietary model presented in the research exhibited 95% accuracy, 97% recall, and 96% mAP @0.5 in real-time hand gesture detection.

2.8. YOLO-Drone: An Optimized YOLOv8 Network for Tiny UAV Object Detection

Xianxu Zhai *et al.* [15] introduced "YOLO-Drone," an improved version of YOLOv8 made especially for locating tiny UAV targets, in the article. Because it performs better than current methods in terms of accuracy, model size, and detection time, this innovative technique was ideal for engineering purposes. Among the assessment criteria were recall rates, model parameters, accuracy, frames per second (FPS), mean average precision (mAP), and average precision (AP). The study showed how UAV object identification had advanced was made possible by funding from the Natural Science Foundation of the Xinjiang Uygur Autonomous Region.

2.9. Detection and Classification via YOLOv8 and Deep Belief Network over Aerial Image Sequences

Using YOLOv8 and Deep Belief Network, Naif Al Mudawi *et al.* [16] presented a novel technique for vehicle detection and classification across aerial photo sequences. The model performed exceptionally well in preprocessing, segmentation, YOLOv8-based detection, feature extraction with SIFT, ORB, and KAZE, and DBN classification on the VEDAI and VAID datasets. However, there were a number of shortcomings, including the incapacity to measure up to modern methods of assessment on tiny datasets, an unclear degree of computing complexity, and the absence of a real-time implementation study.

Chapter 3

Requirements Analysis

3.1 Hardware Requirements

- Raspberry Pi 4 model b 8gb ram
- SanDisk 16gb memory card

3.2 Software Requirements

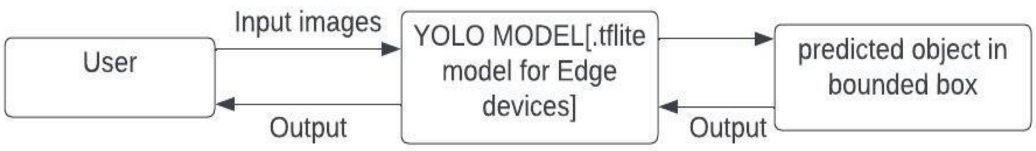
- Google Collab
- Roboflow
- Google drive
- Raspberry Pi OS

Chapter 4

Design

4.1 Dataflow Diagrams (DFDs)

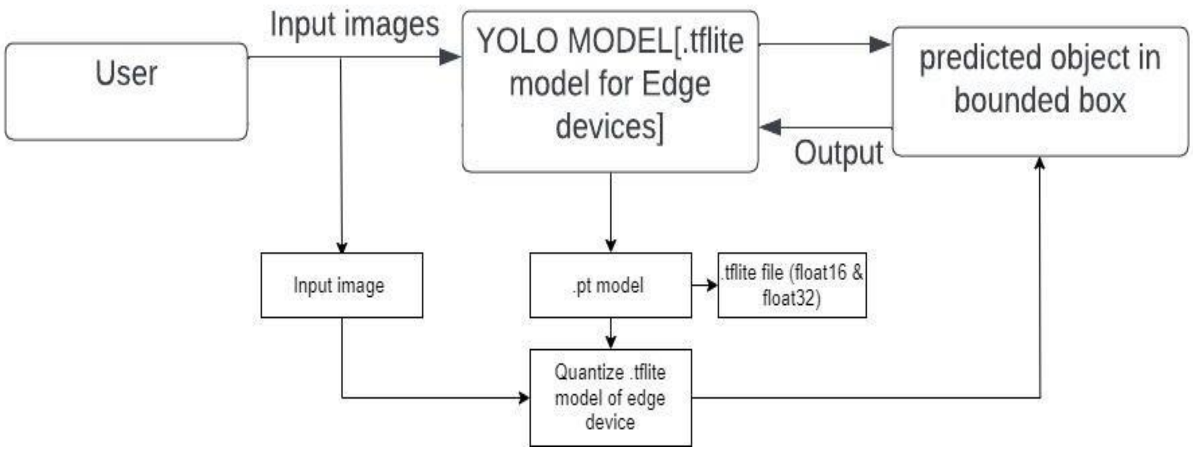
4.1.1 Level 0 DFD



User
Input images
YOLO MODEL[.tflite model for Edge devices]
predicted object in bounded box
Output
Output

Figure 4.1.1: Level 0 DFD diagram

4.1.2 Level 1 DFD



User
Input images
YOLO MODEL[.tflite model for Edge devices]
predicted object in bounded box
Output
Input image
.pt model
.tflite file (float16 & float32)
Quantize .tflite model of edge device

Figure 4.1.2: Level 1 DFD diagram

4.2 UML Diagrams

4.2.1 Use case Diagram

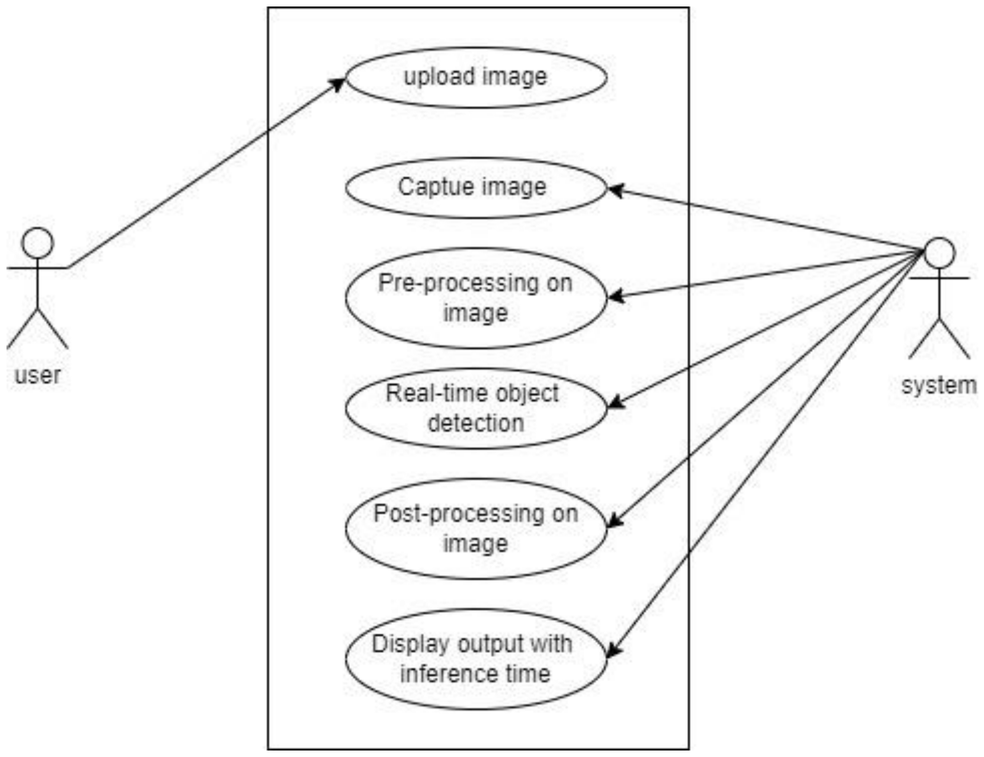


Figure 4.2.1: Use case diagram

4.2.2 Class Diagram

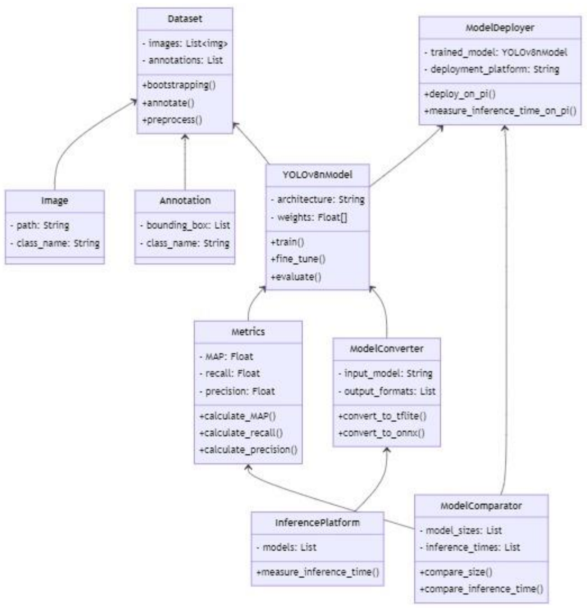


Figure 4.2.2: Class diagram

4.2.3 Activity Diagram

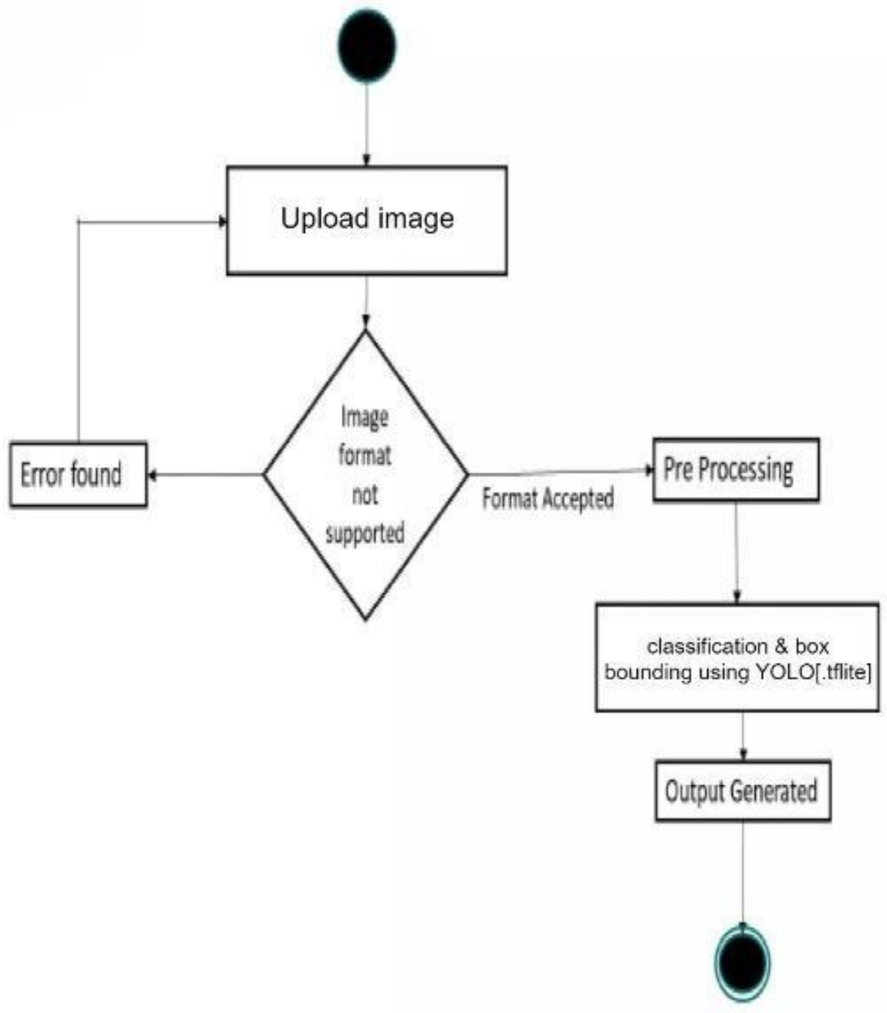

Upload image
Image format not supported
Error found
Format Accepted
Pre Processing
classification & box bounding using YOLO[.tflite]
Output Generated

Figure 4.2.3: Activity diagram

4.2.4 Sequence Diagram

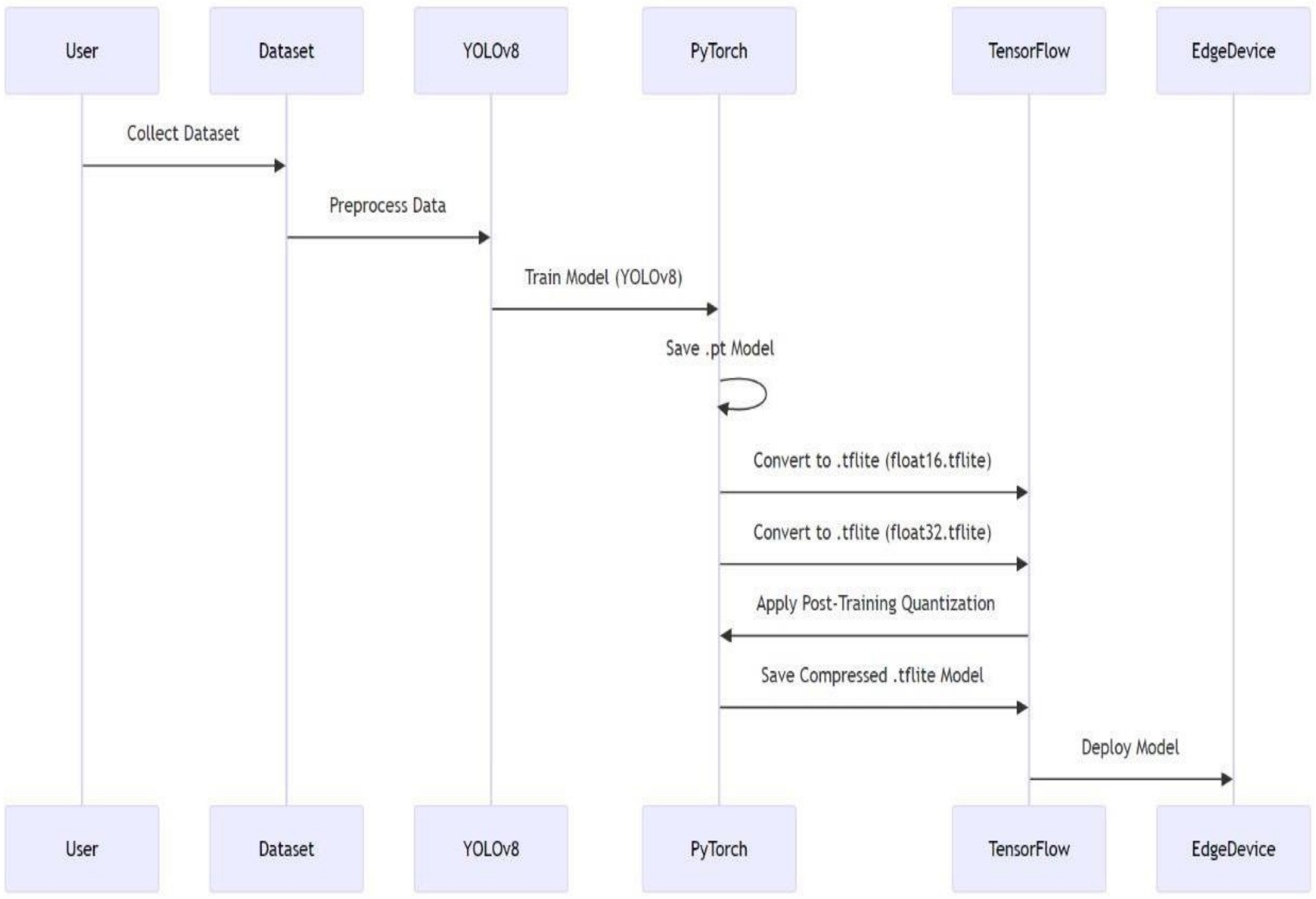


Figure 4.2.4: Sequence diagram

4.2.5 Timeline/Gantt Chart

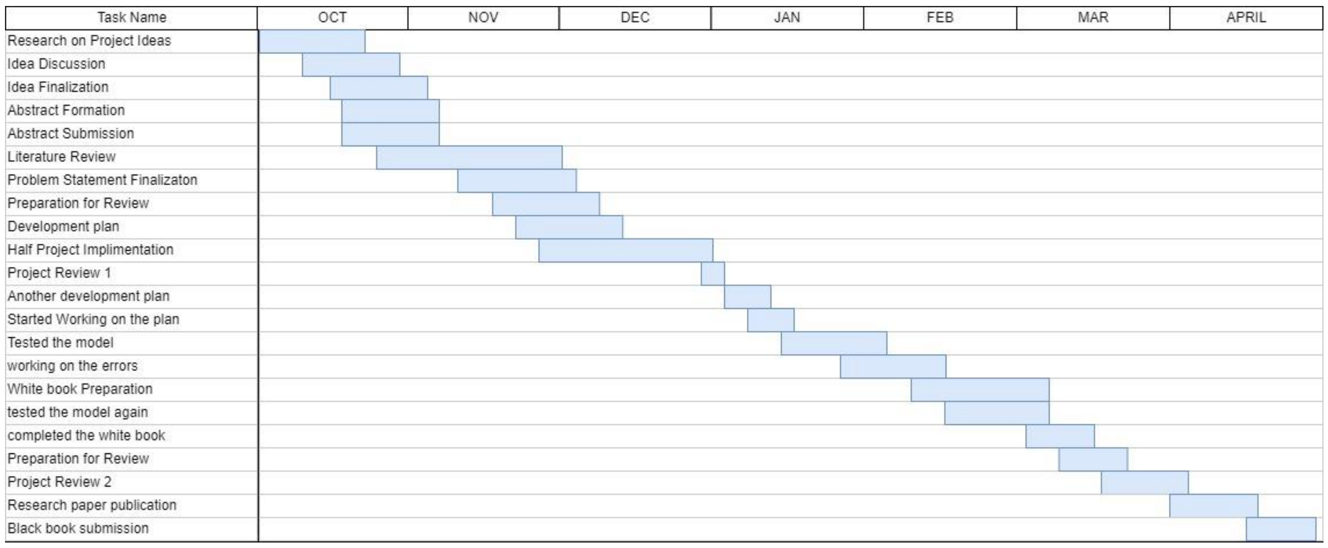


Figure 4.2.5: Timeline

4.2.6 Work Breakdown Structure (WBS) Chart

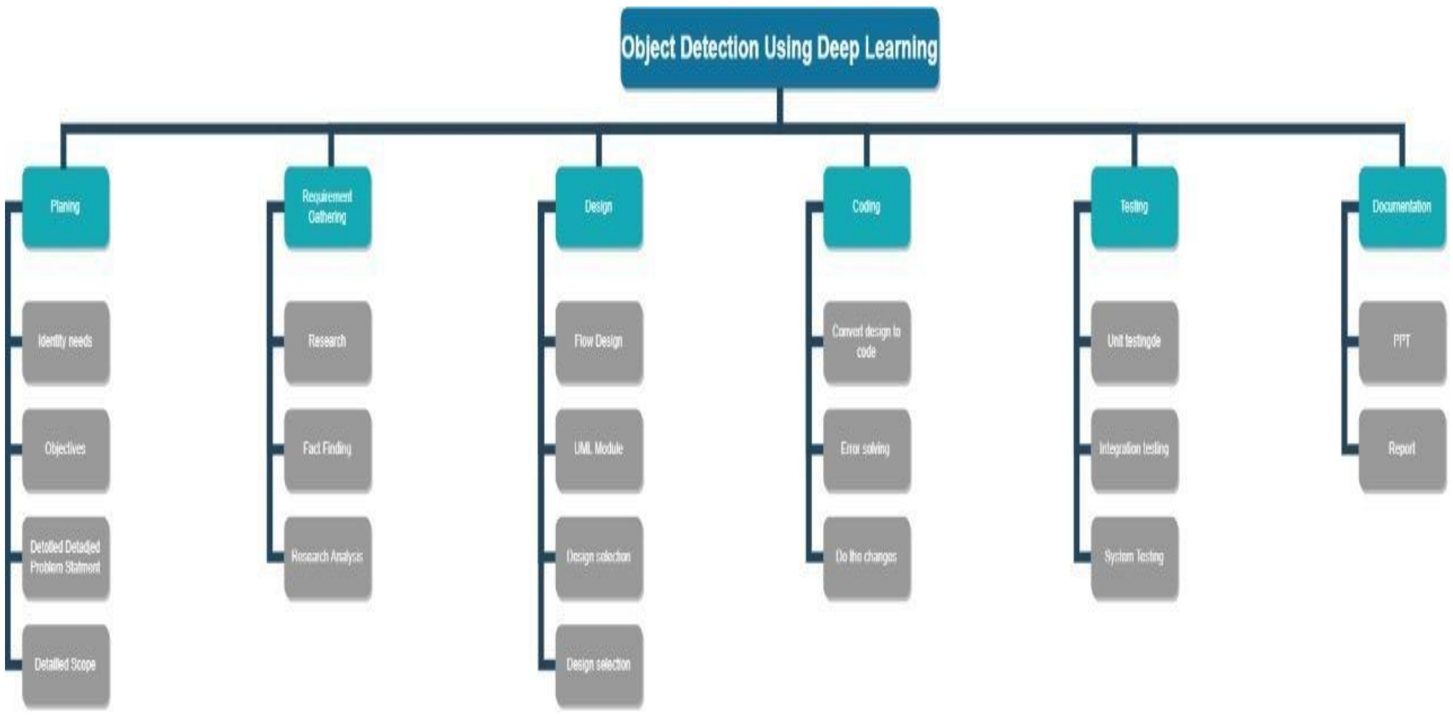


Figure 4.2.6: WBS Chart

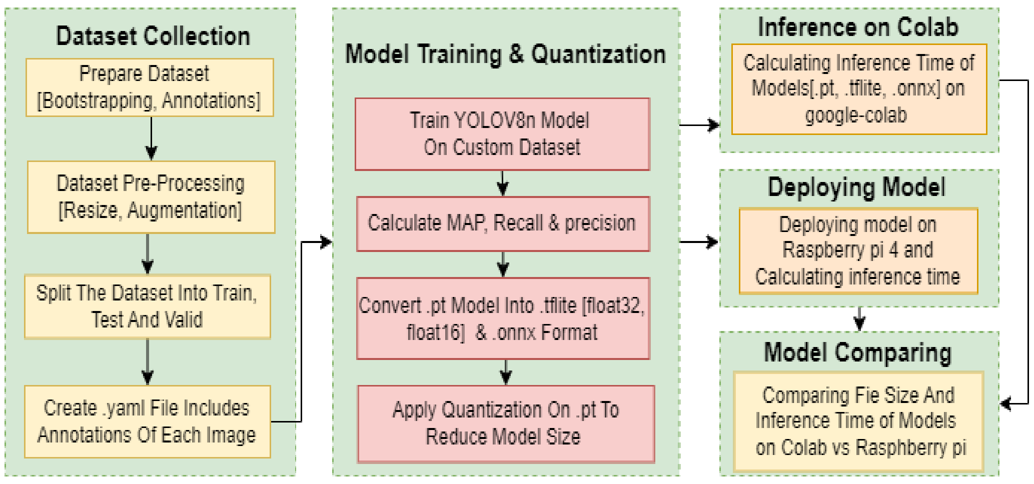
Chapter 5

Report on Present Investigation

5.1 Methodology/Proposed System

Through the use of deep learning on edge devices, the suggested approach seeks to transform object detection. Through the use of cutting-edge neural networks and their optimisation for effective inference, this system enables edge devices, such cameras, smartphones, and Internet ofThings sensors, to recognise and pinpoint things instantly. This innovation ensures data privacy and lessens the need for continuous cloud connectivity while improving security and surveillancesystems and creating new opportunities in industries like augmented reality and driverless cars. Itis a major step towards pushing sophisticated, on-device object detection to the forefront of technological innovation.

5.1.1 Block diagram of the System



Dataset Collection
Prepare Dataset [Bootstrapping, Annotations]
Dataset Pre-Processing [Resize, Augmentation]
Split The Dataset Into Train, Test And Valid
Create .yaml File Includes Annotations Of Each Image
Model Training & Quantization
Train YOLOV8n Model On Custom Dataset
Calculate MAP, Recall & precision
Convert .pt Model Into .tflite [float32, float16] & .onnx Format
Apply Quantization On .pt To Reduce Model Size
Inference on Colab
Calculating Inference Time of Models[.pt, .tflite, .onnx] on google-colab
Deploying Model
Deploying model on Raspberry pi 4 and Calculating inference time
Model Comparing
Comparing Fie Size And Inference Time of Models on Colab vs Rasphberry pi

Figure 5.1.1: Block Diagram

1) Dataset Collection:

(a) Prepare Dataset [Bootstrapping, Annotations]:

Collect a diverse set of images relevant to your object detection task, ensuring they cover different scenarios and variations. Use bootstrapping techniques to augment the dataset by combining existing data with new samples. To annotate the images, draw bounding boxes around the relevant elements and label them with the class names.

(b) Dataset Pre-Processing:

Resize, Augmentation: Resize all images to a uniform size compatible with the YOLOv8 model input. Apply data augmentation techniques like horizontal flips to increase dataset variability, improve model generalization, and prevent overfitting.

(c) Split The Dataset Into Train, Test, And Valid:

Divide the dataset into three subsets: training, testing, and validation. The training set is used to train the YOLOv8 model, the testing set evaluates its performance during development, and the validation set is used for final evaluation after training to assess generalization.

(d) Create .yaml File Includes Annotations Of Each Image:

Generate a .yaml file containing annotations for each image in the dataset. This file should include image paths, class names and other relevant metadata. The .yaml file serves as input to the YOLOv8 training pipeline, enabling seamless integration of images into the model training process.

2) Model Training & Quantization

(a) Train YOLOV8n Model On Custom Dataset:

Utilize the YOLOv8n architecture to train your custom object detection model on the prepared dataset. Fine-tune the model using techniques like transfer learning to adapt its weights to your specific task, ensuring high accuracy and performance on target objects.

(b) Calculate MAP, Recall & Precision:

Evaluate the trained YOLOv8n model using Mean Average Precision (MAP) to measure its overall object detection performance across different classes. Additionally, calculate recall (true positives divided by actual positives) and precision (true positives divided by predicted positives) to assess the model's ability to detect objects accurately while minimizing false positives and false negatives.

(c) Convert .pt Model Into .tflite [float32, float16] & .onnx Format:

Convert the trained YOLOv8n model stored in the .pt (PyTorch) format into optimized formats such as .tflite (TensorFlow Lite) in both float32 and float16 precision, and .onnx (Open Neural Network Exchange) format. These conversions enable compatibility with various deployment platforms and facilitate efficient inference on different hardware.

(d) Apply Quantization On .pt To Reduce Model Size:

Implement quantization techniques on the YOLOv8n .pt model to reduce its size and optimize inference performance. Quantization reduces the precision of model weights and activations, typically to 8-bit integers, without significant loss in accuracy, making the model more suitable for deployment on resource-constrained devices with limited storage and computational capabilities.

3. Inference on Colab:

Utilize Google Colab to measure the inference time of models in different formats such as .pt (PyTorch), .tflite (TensorFlow Lite), and .onnx (Open Neural Network Exchange). This step involves running inference tests on Colab's high-performance computing environment to evaluate the efficiency and speed of model predictions.

4. Deploying Model:

Deploy the trained model on a Raspberry Pi 4 to assess its inference performance in a real-world, edge computing scenario. Measure the inference time on the Raspberry Pi 4 to understand how the model performs in a resource-constrained environment, providing insights into its practical applicability and scalability.

5. Model Comparing:

Compare the file size and inference time of models between Google Colab and Raspberry Pi. Evaluate how the model's size and inference speed vary between the high-performance Colab environment and the resource-constrained Raspberry Pi setup, highlighting the trade-offs between computational power and edge deployment efficiency.

5.2 Implementation

5.2.1 Algorithm/Flowchart

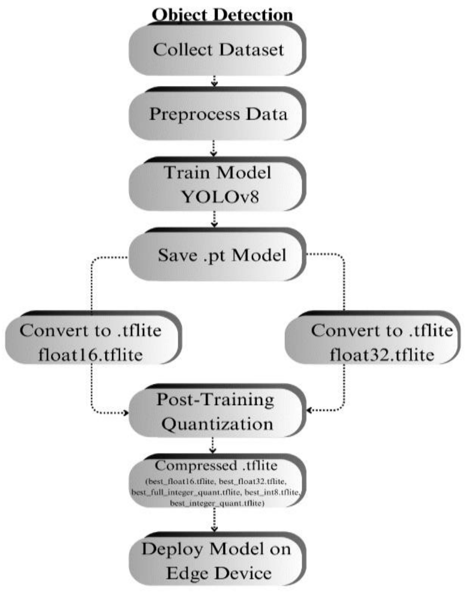

Object Detection
Collect Dataset
Preprocess Data
Train Model YOLOv8
Save .pt Model
Convert to .tflite float16.tflite
Convert to .tflite float32.tflite
Post-Training Quantization
Compressed .tflite (best_float16.tflite, best_float32.tflite, best_full_integer_quant.tflite, best_int8.tflite, best_integer_quant.tflite)
Deploy Model on Edge Device

Figure 5.2.1: Flowchart

The creation of an object recognition model like YOLOv8 involves several important processes. Initially, in the "Collect Dataset" stage, a comprehensive and extensive dataset made up of images tagged with bounding boxes surrounding the objects that require identification must be obtained. The quality and quantity of these data have a significant impact on the final model's accuracy. Inadequate and incorrectly labeled data will hinder the model's ability to identify things appropriately.

Following dataset collection, data preparation is necessary for the "Preprocess Data" stage. This calls for actions like resizing photos and using data augmentation techniques to increase the dataset's quality and diversity. Appropriate preprocessing is necessary to ensure that the model can effectively learn from the data during training. Ultimately, data preparation lays the foundation for successful model training and influences the model's ability to generalize to new, unknown data.

When the data is ready, the preprocessed dataset may be used to train the YOLOv8 model. YOLOv8 and other modern object recognition algorithms are well known for their accuracy and strength in identifying objects in images. The model is often saved in a PyTorch-compatible.pt file format after training. Applying post-training quantization is one potential method to reduce the model's size and ease its deployment on edge devices, such as IoT devices and smartphones. Since the model can be installed on edge devices and allows real-time inference without relying on cloud computing, it is suitable for a wide range of applications requiring on-device processing.

Chapter 6

Results and Discussion

Experimental Results

6.1 Data Collection to Model Deployment

6.1. 1. Bootstrapping and Data Collection:

The project uses bootstrapping, which is an effective way to collect and prepare data. Bootstrapping involves gathering data from several sources, as opposed to traditional datasets which are collected from a single source. This strategy has a number of advantages. Firstly, it increases the total amount of the dataset, which improves the model's durability and ability to be generalized Second, by including data from many sources, the model is exposed to a wider range of variations and complexity, thereby enhancing its ability to manage data that does not exist during real-world deployment.

30,000 images made up the initial dataset used in this research. A larger dataset can significantly improve the model's performance, especially when it comes to picture recognition, even though this can be sufficient for some applications. To overcome this, the project employed bootstrapping techniques. This most likely needed carefully combining new datasets with the old data that were relevant to the project's objective. Additionally, bounding boxes were drawn around objects of interest in the images using Roboflow, enhancing the dataset for object detection tasks and further refining the model's capabilities in spatial recognition.

6.1.2. Data augmentation and preparation:

The research was able to overcome the limitations of a collection of 30,000 pictures through the creative use of bootstrapping and data augmentation. By including data from various sources throughout the bootstrapping process, RoboFlow data augmentation techniques, artificially boosted the sample size to an amazing 90,000 photos. This variation allowed the software to perform well on unfamiliar input and let it learn complex patterns. Finally, all images were scaled to 512x512 to guarantee compatibility with the deep learning model's design. By carefully combining several techniques, the research created a richer training environment that led to a more reliable and adaptable model. Following the augmentation process resulting in 90,000 photos, the dataset was split into three subsets: 80,000 images for training, 2,000 images for testing, and 8,000 images for validation. This partitioning enabled comprehensive training, evaluation, and validation of the model's performance across diverse datasets, ensuring its reliability and effectiveness.

6.1.3. Model Training:

After training our model for 64 epochs using our prepared dataset, we ended up with a .pt model file. This file contains all the improvements and knowledge gained during our training sessions, ready for practical use.

6.1.4. Model Deployment on Resource-Constrained Hardware:

As part of our project, we had to deploy a reliable PyTorch model on new (Rasberry pi 4 model b) hardware with limited processing and storage power. To address this, we utilized model conversion. This meant reducing the model's size in order to fit it on the devices' constrained storage. Additionally, we converted the model from its original PyTorch file into the TensorFlow Lite (TFLite) and Open Neural Network Exchange (ONNX) formats. These smaller formats are specifically designed for efficient inference on edge computers with lower computing power. This translation process enables our robust model to run smoothly on these resource-constrained devices.

6.2. Model Performance Summary:

Table 1 : Performance Measurements

Class	**Images**	**Instances**	**P**	**R**	**mAP50**	**mAP50-95**
All	8000	88824	0.664	0.611	0.668	0.446
Bus	8000	4658	0.575	0.571	0.597	0.464
Car	8000	60857	0.789	0.777	0.846	0.593
Person	8000	7117	0.718	0.505	0.607	0.351
Traffic Light	8000	9592	0.704	0.548	0.647	0.308
Truck	8000	6600	0.535	0.656	0.64	0.515

6.2.1. Class Representation Metrics:

The provided table encapsulates crucial object detection metrics essential for evaluating the performance of a model across distinct classes within a dataset. Each row represents a specific object class, such as "bus," "car," "person," "traffic light," and "truck," while the columns delineate pertinent metrics utilized in assessing the model's detection efficiency.The "Images" column enumerates the instances of each class within the dataset, offering insights into their prevalence and representation. Concurrently, the

"Instances" column quantifies the total detected instances for each class, portraying the model's proficiency in identifying specific objects.

6.2.2. Detection Accuracy Metrics:

Metrics like "Box (P)" and "Box (R)" epitomize Precision and Recall, respectively, serving as benchmarks for the model's accuracy and completeness in object detection endeavors. Elevated values in these metrics indicate meticulous detection capabilities, encompassing both accuracy and comprehensiveness.

6.2.3. Average Precision Metrics:

The "mAP50" and "mAP50-95" metrics epitomize the mean Average Precision across diverse IoU thresholds, elucidating the model's adeptness across varying levels of object overlap. Heightened mAP values denote superior detection precision, particularly notable at higher IoU thresholds where object delineation can be more intricate. Additionally, the overall mAP50 is 0.668, highlighting the model's strong performance across all classes, while the mAP50-95 is 0.446, showcasing its consistency even under more stringent IoU conditions.

For instance, the "car" class showcases commendable Precision (Box (P) = 0.789) and Recall (Box (R) = 0.777), signifying precise and comprehensive car detection within the dataset. A notable mAP50 value of 0.846 underscores robust performance, especially at the 50% IoU threshold, while the mAP50-95 value of 0.593 underscores consistent accuracy even under more challenging detection scenarios.

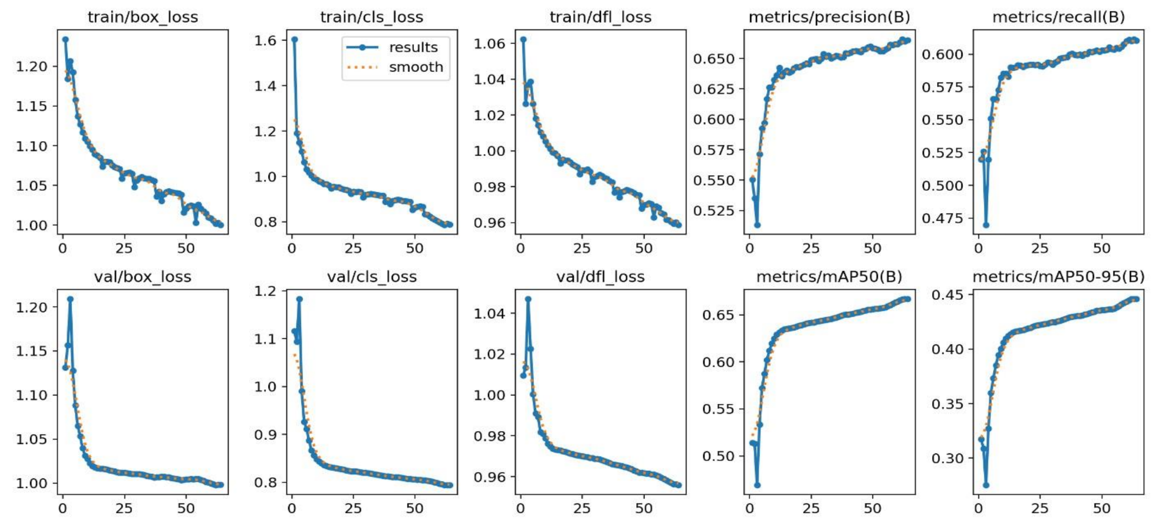
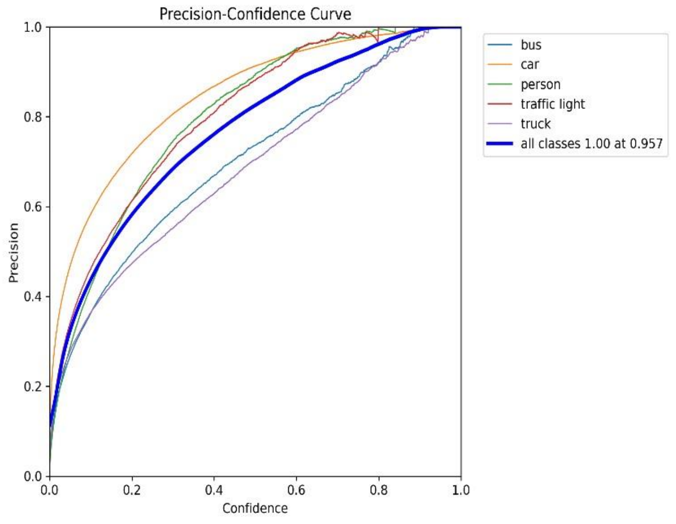
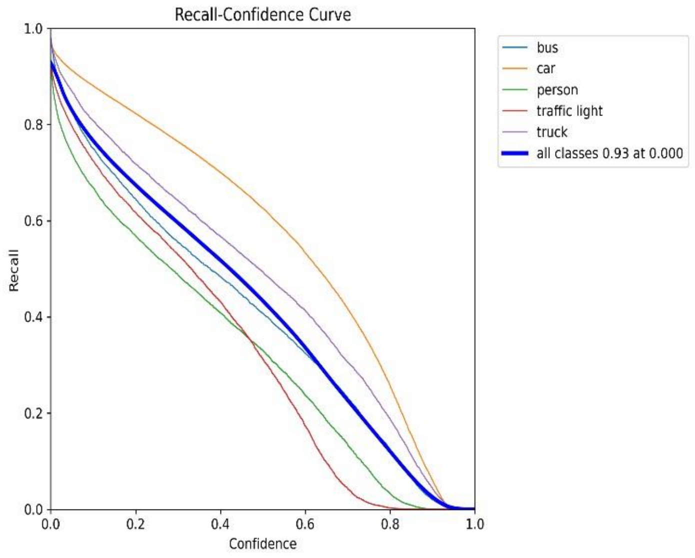


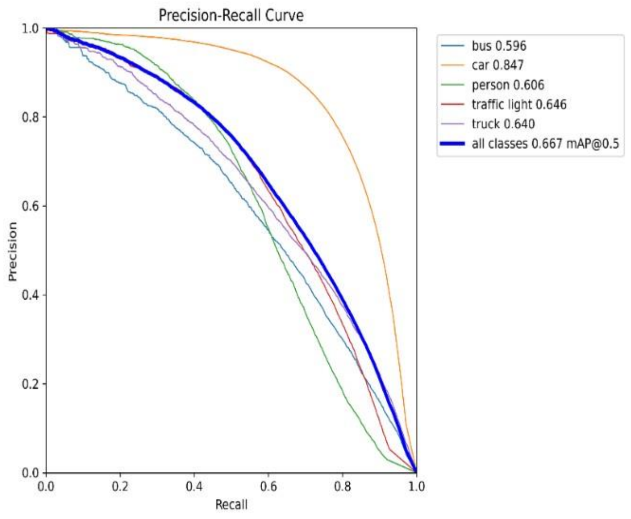
Fig.6.2.3(a).: The results of the proposed model



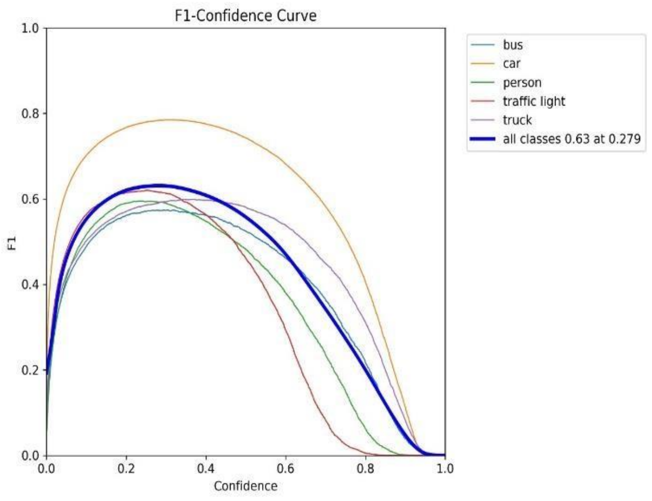
(A) Precision-Confidence Curve



(B) Recall-Confidence Curve



(C) Precision-Recall Curve



(D) F1-Confidence Curve

Fig. 6.2.3(b): Precision–recall-confidence curve of the proposed model

The weighted mean of the accuracy and recall percentages is known as the F1 score. Consequently, this score accounts for both false positives and false negatives. Compared to F1, precision is less common, but accuracy is not always apparent. When the costs of false negatives and incorrect positives are equal, accuracy is at its best. Recall should be considered in addition to accuracy if there are differences in the costs related to false positives and false negatives. The ratio of accurately predicted observations to all positively expected observations is known as precision when it comes to good outcomes.

A high precision and recall value in the model indicates an accurate identification and classification of all positive objects[17]. The percentage of genuine positive forecasts over all real positives is known as the recall. This may be computed using equation (1). The accuracy is defined as the proportion of true positive forecasts over all positive predictions. Equation (2) may be utilized to calculate it [1].

$$\text{Recall} = \frac{TP}{TP+FN} \qquad (1)$$

$$\text{Precision} = \frac{TP}{TP+FP} \qquad (2)$$

where the sign for True Positive (TP) is used. The symbols False Negative (FN), False Positive (FP), and True Negative (TN) stand for these concepts. The F1, accounts for accuracy and recall, is calculated using equation
(3).
.

$$F^1 = 2 * \frac{recall * precision}{recall + precision} \qquad (3)$$

6.3 Models Comparison

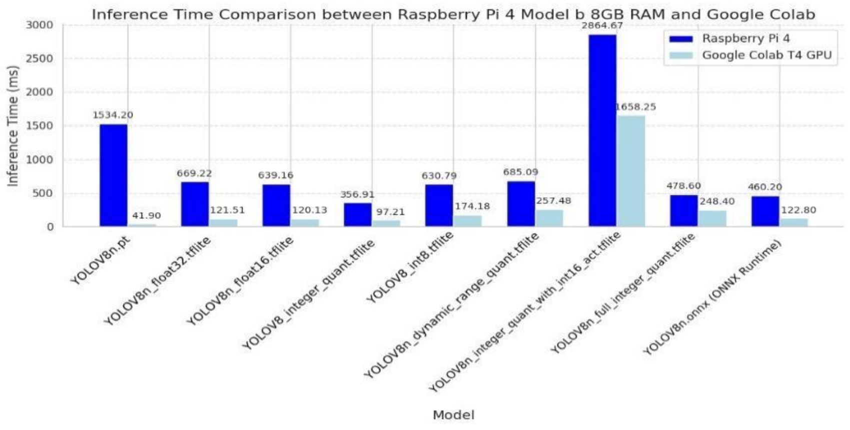


Fig.6.3.1: Comparison between Rasberry Pi and Google Colab

We started our experiments with the main file format, YOLOv8n.pt, which acted as the project's foundation. We converted this primary model into a TensorFlow Lite (TFLite) model using quantization methods, and the result was YOLOv8n_integer_quant.tflite. Furthermore, we created nine models in all, including the primary YOLOv8n.pt model, by converting the.pt model into the ONNX format. Following that, these models were carefully evaluated on the Raspberry Pi and Google Colab platforms, making use of the latter's robust T4 GPU for increased processing capability. As expected, the inference timings on the Google Colab T4 GPU were significantly shorter than those on the Raspberry Pi because of the significant difference in processing power between the two devices.

However, when the TFLite model was compressed before being deployed on the Raspberry Pi, there were still discernible disparities in inference times. In particular, the inference time of the uncompressed YOLOv8n.pt model on the Raspberry Pi was 1534.20 ms, but the inference time of the compressed YOLOv8n_integer_quant.tflite model was significantly shorter at 356.91 ms. These findings show that quantization is a useful technique for optimizing inference performance, particularly on low-resource systems like as the Raspberry Pi.

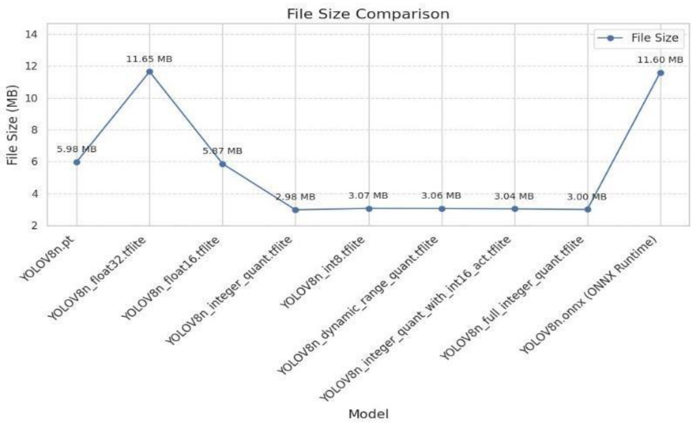


Fig.6.3.2: File Size Comparison

TensorFlow Lite (.tflite), ONNX Runtime (.onnx), and PyTorch (.pt) are the file formats present over here. YOLOV8N_float32.tflite appears to be the largest model on show, weighing in at 11.65 MB. A variety of other file formats, some of which are smaller than.tflite, are also provided by YOLOV8N. YOLOV8N_integer_quant.tflite appears to be the smallest file size, weighing in at 2.98 MB.

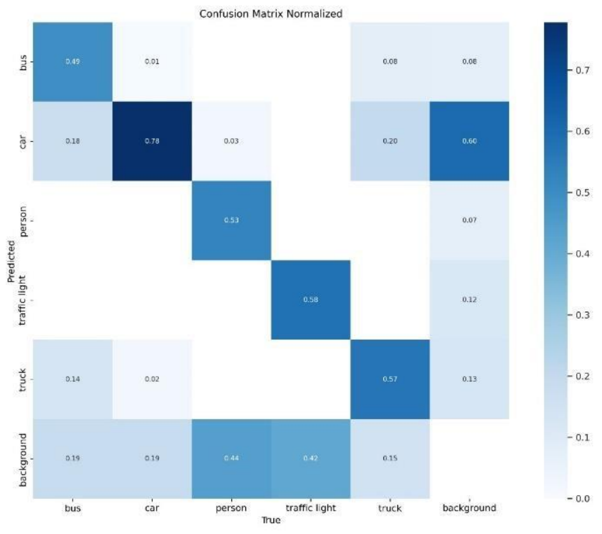


Fig.6.3.3: Confusion Matrix

The presented confusion matrix illustrates the performance of a classification model for five different classes: people, cars, buses, trucks, and traffic lights. The predicted classes are shown vertically in this matrix, whereas the actual classes are shown horizontally. The percentage of cases that were predicted to belong to a certain class relative to the instances that really do belong to that class is represented by each column in the matrix. The normalized version of these proportions shows the percentage of accurate predictions for each class in relation to the total number of cases in that class.

Table 2: Processing Power Comparison

GOOGLE COLLAB - HIGH PROCESSING POWER [T4 GPU]			
Model Name	Preprocessing Time (ms)	Inference Time (ms)	Postprocessing Time (ms)
YOLOV8n.pt	4.6	41.9	1206.1
YOLOV8n_float32.tflite	1.86	121.51	16.4
YOLOV8n_float16.tflite	0.9	120.13	16.28
YOLOV8n_integer_quant.tflite	1.91	97.21	15.64
YOLOV8n_int8.tflite	2.17	174.18	15.5
YOLOV8n_dynamic_range_quant.tflite	2.96	257.48	31.7
YOLOV8n_integer_quant_with_int16_act.tflite	0.68	1658.25	17.17
YOLOV8n_full_integer_quant.tflite	3.9	248.4	2.5
YOLOV8n.onnx (ONNX Runtime)	28.3	122.8	549.8

Raspberry pi 4 model b 8GB RAM- LOW PROCESSING POWER			
Model Name	Preprocess Time (ms)	Inference Time (ms)	Postprocess Time (ms)
YOLOV8n.pt	20.9	1534.2	52.4
YOLOV8Nn_float32.tflite	3.86	669.22	243.45
YOLOV8Nn_float16.tflite	3.62	639.16	244.77
YOLOV8Nn_integer_quant.tflite	4.47	356.91	255.34
YOLOV8Nn_int8.tflite	5.52	630.79	241.1
YOLOV8Nn_dynamic_range_quant.tflite	5.76	685.09	251.6
YOLOV8Nn_integer_quant_with_int16_act.tflite	6.33	2864.67	240.25
YOLOV8Nn_full_integer_quant.tflite	10.9	478.6	4.3
YOLOV8Nn.onnx (ONNX Runtime)	112	460.2	149.8

The inference time on the Raspberry Pi 4 Model B with 8GB RAM differs significantly between YOLOV8n.pt and YOLOV8n_integer_quant.tflite. YOLOV8n_integer_quant.tflite demonstrates a much shorter inference time of 356.91 ms compared to YOLOV8n.pt, which has a noticeably larger inference time of 1534.2 ms, indicating a reduction of approximately 76.74% in inference time when using YOLOV8n_integer_quant.tflite instead of YOLOV8n.pt. This improvement is due to quantization, which reduces the precision of the model's weights and activations, thereby optimizing its size and computational efficiency for deployment on resourceconstrained devices like the Raspberry Pi.

Additionally, the quantization process involves quantizing the model's parameters and activations to a lower bit precision, such as 8-bit integers, from their original floating-point precision. This not only reduces the model size but also speeds up inference by utilizing integer operations that are more efficient on hardware like the Raspberry Pi's Arm processor. Furthermore, quantization can lead to a slight loss in model accuracy, but for many applications, the trade-off between inference speed and accuracy is acceptable, especially on edge devices where computational resources are limited.

Among the various models tested on Google Colab with high processing power using a T4 GPU, YOLOV8Nn_integer_quant.tflite exhibited the shortest inference time of 97.21 ms, showcasing the impact of quantization on improving inference speed. This reduction in inference time further emphasizes the effectiveness of quantization in optimizing deep learning models for deployment in real-world scenarios with constrained hardware resources.

Results:

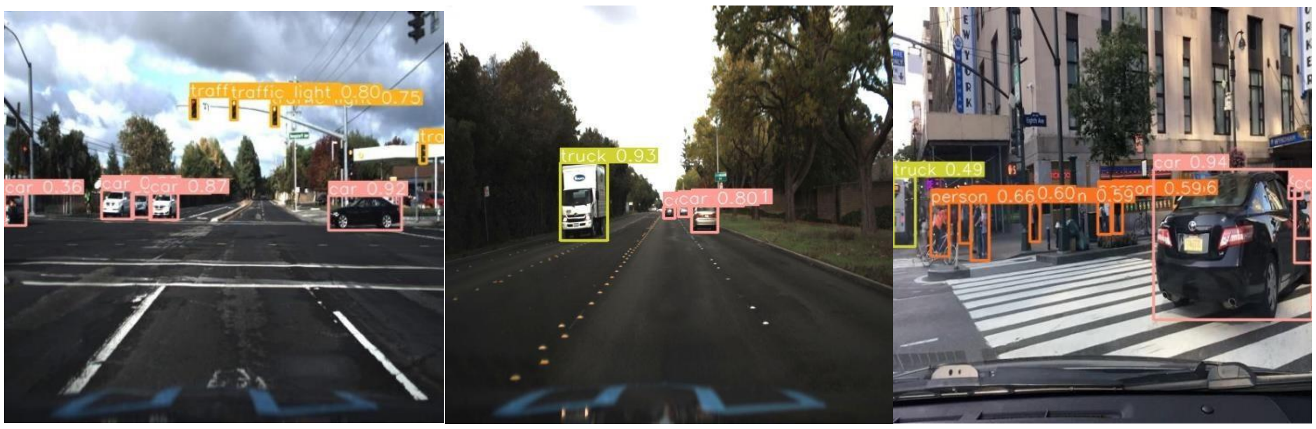


Fig 6.3.4: Image Detection

The YOLOv8n model has successfully identified items in a picture by correctly identifying their presence. In this method, bounding boxes with matching class names and confidence scores are provided to aid in the localization of objects. Class names indicate an object's category identity, while confidence ratings indicate how certain the model is about the accuracy of its predictions.

Robust predictions are shown by high confidence scores, which also show a strong alignment between the identified items and the appropriate classes. The model's capacity to identify things in pictures is demonstrated by its effective detection procedure, demonstrating its accuracy and dependability in object localization tasks.

Chapter 7

Conclusion

In conclusion, this work examines the advancements and applications of object detection on edge devices, focusing on the YOLO-based models' transition to YOLOv8n. By analyzing the YOLOv8n training and deployment process on edge devices like Raspberry Pi, the study highlights the significance of data augmentation, model conversion, and performance optimization measures. The experimental results demonstrate the trade-offs between model accuracy and inference time, as well as the efficiency gains brought about by model quantization. The study also looks at the implications of these findings, emphasizing future opportunities to enhance object recognition algorithms on edge devices through energy-efficient methods, flexible domains, and privacy-preserving techniques. All things considered, by enhancing object detection's accessibility, efficiency, and versatility, this study contributes to ongoing efforts to enhance it for useful applications on edge computing platforms.

Appendix

Technologies Used

1) Python

One well-liked programming language is Python. Guido van Rossum was the creator, and it was published in 1991. It is utilized in system programming, mathematics, software development, and server-side web development. Web applications may be developed on a server using Python. Workflows may be created with Python in conjunction with other programs. Database systems may be connected to in Python. It has the ability to read and edit files. Complex mathematics and large data handling may be accomplished using Python. Python may be used to produce software that is ready for production or for quick prototyping.

2) Python Libraries

- **TensorFlow:**

TensorFlow, our project's primary tool, allows building machine learning models quick and easy. Because of its flexible nature, developers may generate and use a wide range of models, from simple linear regressions to complex neural networks. TensorFlow's extensive ecosystem and plenty of documentation make development easier and enable quick iterations and experiments.

- **Matplotlib:**

Matplotlib is a vital tool for data visualization that transforms raw data into visually stunning graphs and charts. Its comprehensive plotting capabilities and user-friendly interface enable us to swiftly create visuals suitable for publication. We can visually depict patterns, distributions, or linkages in our data using Matplotlib, which makes it easier to communicate our findings in an interesting and impactful way.