Utilizing light-weight models for Object detection

Dissertation submitted to

Shri Ramdeobaba College of Engineering & Management, Nagpur

in partial fulfillment of requirement for the award of degree of

Bachelor of Technology (B.Tech)

In

COMPUTER SCIENCE AND ENGINEERING

By

Devesh Borkar (B - 39)

Hardik Thakkar (B - 43)

Om Bhagat (B - 52)

Of

VI Semester

Guide

Dr Khushboo Khurana

Department of Computer Science and Engineering

Shri Ramdeobaba College of Engineering & Management, Nagpur 440 013

(An Autonomous Institute affiliated to Rashtrasant Tukdoji Maharaj Nagpur University Nagpur)

April 2024

April 2024

SHRI RAMDEOBABA COLLEGE OF ENGINEERING MANAGEMENT, NAGPUR

(An Autonomous Institute affiliated to Rashtrasant Tukdoji Maharaj Nagpur University Nagpur)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Thesis on **"Utilizing light-weight models for Object detection"** is a Bonafide work of Devesh Borkar, Hardik Thakkar, Om Bhagat submitted to the Rashtrasant Tukdoji Maharaj Nagpur University, Nagpur in partial fulfillment of the award of a Degree of Bachelor of Technology (B.Tech), in Computer Science and Engineering. It has been carried out at the Department of Computer Science and Engineering, Shri Ramdeobaba College of Engineering and Management, Nagpur during the academic year 2023-2024.

Date:

Place: Nagpur

Dr Khushboo Khurana Project Guide Department of Computer Science and Engineering	Dr. R. Hablani H.O.D Department of Computer Science and Engineering

Dr. R. S. Pande
Principal

DECLARATION

We hereby declare that the thesis titled "**Utilizing light-weight models for Object detection**" submitted herein, has been carried out in the Department of Computer Science and Engineering of Shri Ramdeobaba College of Engineering and Management, Nagpur. The work is original and has not been submitted earlier as a whole or part for the award of any degree/diploma at this or any other institution / University.

Date:

Place: Nagpur

Name of the Student	Roll no.	Signature
Devesh Borkar	39	
Hardik Thakkar	43	
Om Bhagat	52	

APPROVAL SHEET

This report entitled "**Utilizing light-weight models for Object detection**" by Devesh Borkar, Hardik Thakkar, Om Bhagat is approved for the degree of Bachelor of Technology (B.Tech).

Dr Khushboo Khurana Project Guide	External Examiner

Dr. R. Hablani
H.O.D, CSE

Date:

Place: Nagpur

ACKNOWLEDGEMENTS

The project is a combined effort of a group of individuals who synergize to contribute towards the desired objectives. Apart from our efforts by us, the success of the project shares an equal proportion with the engagement and guidance of many others. We take this opportunity to express our gratitude to the people who have been instrumental in the successful completion of this project.

We would like to show our greatest appreciation towards Dr. Khushboo Khurana for providing us with the facilities for completing this project and for their constant guidance.

We would like to express our deepest gratitude to Dr. Ramchand Hablani, Head, Department of Computer Science and Engineering, RCOEM, Nagpur for providing us the opportunity to embark on this project.

Finally, extend our gratitude to all the faculty members of the CSE department who have always been so supportive and provided resources needed in our project development. We are very grateful to all of them who have unconditionally supported us throughout the project.

Date :

--Projectees
Devesh Borkar

Hardik Thakkar
Om Bhagat

ABSTRACT

Object detection is a cornerstone task within computer vision, facilitating diverse applications across industries such as security, retail, and environmental monitoring. However, traditional object detection methods often face challenges related to computational complexity and real-time performance, especially when deployed on resource-constrained devices or in scenarios requiring rapid decision-making. This project addresses these issues by investigating the application of lightweight deep learning models, specifically MobileNet, for object detection tasks. MobileNet stands out for its compact architecture and efficiency, making it well-suited for deployment on devices with limited computational resources without compromising on detection accuracy. By harnessing the capabilities of MobileNet, this research endeavors to enable real-time object detection for both indoor and outdoor environments. Through the utilization of MobileNet's streamlined architecture, the project aims to achieve a balance between computational efficiency and detection performance.

The proposed approach undergoes rigorous evaluation using benchmark datasets, showcasing its ability to accurately detect predefined objects while maintaining high processing speed, even in challenging environmental conditions. The findings of this study hold significant promise for a range of practical applications, including but not limited to smart surveillance systems, inventory management in retail settings, and environmental monitoring for conservation efforts. By leveraging lightweight deep learning models like MobileNet, this project paves the way for efficient and scalable

object detection solutions across various domains, ultimately contributing to advancements in real-time decision-making and automation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLE

CHAPTER 1
INTRODUCTION

Object detection is a crucial task in computer vision with applications spanning surveillance, autonomous driving, and industrial automation. Traditional approaches to object detection often rely on complex convolutional neural network (CNN) architectures or the state-of-the-art YOLO (You Only Look Once) v5 model. While effective, these models can be computationally intensive, limiting their deployment on resource-constrained devices and hindering real-time performance in dynamic environments.

In response to these challenges, this project explores the utilization of lightweight deep learning models, particularly MobileNet, for object detection tasks. MobileNet stands out for its efficiency and compact architecture, offering a viable solution for real-time object detection on devices with limited computational resources. By leveraging MobileNet's streamlined architecture, this research aims to achieve efficient and accurate detection of selected indoor and outdoor objects in real-world scenarios.

MobileNet differs from traditional CNN models and YOLO v5 by employing depthwise separable convolutions, which significantly reduce the computational burden while maintaining satisfactory performance levels. This makes MobileNet well-suited for deployment on edge devices such as smartphones, drones, or embedded systems. By harnessing MobileNet for object detection, this project seeks to strike a balance between detection accuracy and computational efficiency, enabling real-time object detection capabilities across diverse environments.

The proposed approach involves training MobileNet on annotated datasets to detect predefined objects of interest. By fine-tuning the model and optimizing its architecture for object detection tasks, the project aims to achieve high detection accuracy while minimizing computational overhead. The ultimate objective is to develop a robust and efficient object detection system capable of operating in real-time across various indoor and outdoor scenarios.

Through rigorous experimentation and evaluation on benchmark datasets, the effectiveness of the proposed approach will be assessed, focusing on its ability to accurately detect objects in diverse environments while maintaining real-time performance. The outcomes of this research hold promise for a wide range of practical applications, including smart surveillance, industrial automation, and augmented reality, where real-time object detection is essential for decision-making and interaction.

1.1 Problem Definition

The problem addressed in this project is the need for efficient object detection methodologies in various domains, such as computer vision, robotics, and surveillance.

The challenge is to develop and implement lightweight models for object detection that maintain high accuracy while minimizing computational complexity and memory usage.

1.2 Motivation

The motivation for utilizing lightweight models such as MobileNet for object detection lies in the need for efficient and real-time solutions in various applications. Traditional deep learning models for object detection, while effective, often come with high computational costs and memory requirements, making them impractical for deployment on resource-constrained devices or in scenarios requiring rapid decision-making.

MobileNet, with its lightweight architecture and efficient depthwise separable convolutions, offers a compelling alternative. By leveraging MobileNet, we can achieve real-time object detection capabilities without sacrificing accuracy. This is particularly advantageous in scenarios where swift detection of selected indoor or outdoor objects is crucial, such as in smart surveillance systems, industrial automation, or augmented reality applications.

Furthermore, MobileNet's compact design enables deployment on edge devices like smartphones, drones, or embedded systems, expanding the scope of object detection to diverse environments and use cases. Additionally, its ease of fine-tuning allows for customization to specific object detection tasks, ensuring optimal performance for detecting predefined objects of interest. As the demand for real-time object detection continues to grow across various industries, the adoption of lightweight models like MobileNet holds immense potential. By embracing these models, we can unlock new possibilities for enhancing safety, efficiency, and decision-making in applications where timely detection of objects is paramount.

1.3 Overview

Utilizing lightweight models like MobileNet for object detection streamlines the process into several key stages, each contributing to efficient and accurate detection of selected indoor and outdoor objects in real-time scenarios, ultimately culminating in deployment on the Jetson Xavier platform. The first stage involves leveraging MobileNet's architecture for feature extraction. MobileNet employs depthwise separable convolutions to efficiently capture relevant features from input images. This step ensures that crucial object characteristics are identified with minimal computational overhead, ideal for real-time applications.

Next, the extracted features are processed through the object detection pipeline. MobileNet's lightweight design enables swift processing, facilitating rapid identification and localization of objects of interest. This stage is pivotal for ensuring timely detection, particularly in dynamic environments where objects may appear or move quickly.

Furthermore, the object detection process is optimized for deployment on the Jetson Xavier platform, leveraging its computational capabilities and compatibility with lightweight models like MobileNet. This ensures seamless integration into existing systems and facilitates efficient utilization of resources.

Overall, the utilization of lightweight models like MobileNet for object detection offers a streamlined approach to real-time detection of indoor and outdoor objects. By leveraging MobileNet's efficiency and compatibility with edge devices like Jetson Xavier, this project aims to enhance object detection capabilities for a wide range of applications, from smart surveillance systems to industrial automation, with real-time performance and precision.

1.4 Objectives

1) *Develop lightweight object detection models that can perform with high accuracy:*

   The project aims to explore and develop lightweight deep learning models tailored specifically for object detection tasks. These models will prioritize computational efficiency without compromising on detection accuracy.

2) *Detection of various indoor/outdoor objects:*

   The project focuses on enabling the detection of a wide range of indoor and outdoor objects across diverse environments. This includes common objects found in indoor settings such as furniture, electronic devices, and household items, as well as outdoor objects like vehicles, pedestrians, and natural landmarks.

3) *Deployment of the model on Jetson Xavier for increased processing and compactness:*

   Once the lightweight object detection models are developed and trained, the next step involves deploying them on the Jetson Xavier platform. Jetson Xavier offers significant computational power in a compact form factor, making it well-suited for edge computing applications.

1.5 Proposed Plan of Work

-> Experimentation on existing models of object detection. (Eg: CNN/YOLO models)

-> Research existing lightweight object detection models and methodologies.

-> Define the requirements and constraints for the proposed lightweight models.

-> Design and implement lightweight object detection models based on the identified methodologies.

-> Train and optimize the models using appropriate datasets.

-> Fine-tune and optimize the lightweight models based on evaluation results.

CHAPTER 2
LITERATURE SURVEY

2.1 Introduction

The literature review focuses on exploring the utilization of lightweight deep learning models, particularly MobileNet, for real-time object detection tasks. Object detection plays a crucial role in computer vision applications, and the emergence of lightweight models offers efficient solutions for deployment on resource-constrained devices and in dynamic environments. The review aims to synthesize key findings from existing research literature, addressing methodologies, challenges, benchmark datasets, and recent advancements in object detection using MobileNet. It examines how MobileNet's compact architecture and efficient depthwise separable convolutions enable accurate detection of indoor and outdoor objects while ensuring real-time performance. Additionally, it highlights the deployment of lightweight object detection models on platforms like Jetson Xavier, emphasizing the significance of edge computing in modern applications. Through this analysis, the review provides insights into state-of-the-art techniques and future directions in leveraging lightweight models for object detection.

2.2 Literature Review

Table 2.1 Literature Review Table

Sr No.	Paper	Reference	Method
1.	Towards lightweight convolutional neural networks for object detection	[1]	Maintains feature map size Eliminates last two spatial reductions Adds dilations of 2 and 4 Employs channel samplingBalances FLOPs with accuracy
2.	Mobilenet-SSDv2: An Improved Object DetectionModel for Embedded Systems	[2]	Built on Mobilenet-v2 Incorporates Feature Pyramid Network (FPN) Improves accuracy and stabilityAddresses computational complexity challenges Suitable for embedded systemsEspecially for ADAS in autonomous driving
3.	Light-Weight RetinaNet for Object Detection on Edge Devices	[3]	Enhances practical deployment of RetinaNet Validates optimal FLOP-mAP trade-off Proposes lightweight structure Outperforms traditional methods like input image scaling Achieves 0.3% mAP improvement
4.	Lightdet: A Lightweight and Accurate Object Detection Network	[4]	Introduces LightDet with Detail-Preserving Module (DPM) Captures rich low-level features Employs lightweight prediction head Achieves 75.5% mAP on PASCAL VOC 2007 at 250 FPS 24.0% mAP on MS COCO dataset
5.	Object Detection Using Convolutional Neural Networks(TENCON)	[5]	Compares SSD with MobileNetV1 and Faster-RCNN with InceptionV2 Evaluates effectiveness in real-time application and accuracy

			SSD with MobileNetV1 offers high-speed detection Faster-RCNN with InceptionV2 provides more accurate results Showcases trade-off between speed and accuracy in object detection Lays groundwork for practical applications like bomb disposal robots for detecting IEDs

In **[1]** , the research paper investigates lightweight convolutional neural networks (CNNs) optimized for object detection, particularly focusing on vehicle detection within the DETRAC dataset. Through a comprehensive analysis, the study evaluates multiple popular lightweight network architectures and explores the efficacy of channel reduction algorithms to enhance computational efficiency while maintaining accuracy. Notably, the proposed models achieve remarkable performance, with the top-performing model achieving a validation subset AP of 95.13, while even the smallest model achieves real-time inference on CPU with a minor accuracy drop to 91.72 AP. A key emphasis is placed on preserving high spatial resolution in feature extraction, crucial for detailed scene understanding in real-world applications like surveillance and autonomous driving. The research findings suggest that the proposed network design principles are transferable to other practical tasks where intricate scene understanding is paramount. Additionally, the study highlights the adaptability of the network to compression and quantization techniques, further enhancing its performance in real-world scenarios. The simplicity and effectiveness of pruning techniques underscore the feasibility of achieving real-time performance with minimal accuracy compromise, rendering the models highly suitable for deployment in embedded visual applications requiring both speed and precision.

In **[1]** there is a reference to another paper (9) Speed/accuracy trade-offs for modern convolutional object detectors, that delves into the intricate balance between speed, memory utilization, and accuracy within modern convolutional object detection frameworks, offering insights crucial for selecting the optimal detection architecture tailored to diverse application scenarios and platform constraints. Through meticulous investigation, the study navigates the complexities of trading accuracy for speed and memory efficiency, navigating variations in fundamental components like base feature extractors such as VGG and Residual Networks, alongside considerations of image resolutions and hardware/software setups. A unified implementation of prominent detection systems like Faster R-CNN, R-FCN, and SSD is meticulously crafted, enabling a comprehensive exploration of trade-offs across varying architectures and critical parameters. By delineating a spectrum of trade-offs, ranging from achieving real-time inference speeds suitable for mobile deployment to attaining cutting-edge accuracy benchmarks on datasets like COCO, the paper provides valuable insights for practitioners. Furthermore, the experimental analyses identify novel techniques for enhancing speed without substantial sacrifices in accuracy, including strategies like reducing the number of proposals in Faster R-CNN. Ultimately, the paper aspires to equip practitioners with actionable guidance for selecting and optimizing object detection methods, facilitating informed decision-making for real-world deployment scenarios with diverse performance requirements and resource constraints.

In **[2]**, the research paper introduces Mobilenet-SSDv2 as an improved object detection model specifically tailored for deployment on embedded systems. It emphasizes the importance of larger spatial feature maps in enhancing detection performance, with a thorough comparison of lightweight networks to identify the most suitable feature extraction architecture. The study further explores the efficacy of channel reduction algorithms to accelerate model execution while maintaining accuracy. Notably, the proposed vehicle detection models demonstrate exceptional accuracy and speed,

rendering them well-suited for applications in embedded visual processing. Despite prioritizing efficiency, the models uphold high spatial resolution in feature extraction to preserve detailed scene information crucial for accurate detection. While initially designed for vehicle detection, the paper posits the potential transferability of the model's design principles to other tasks necessitating comprehensive scene understanding. Furthermore, the network design remains agnostic to network compression and quantization methods, facilitating further performance enhancements for real-world applications. The study showcases the efficacy of relatively straightforward pruning techniques in achieving real-time inference speeds on CPU with negligible accuracy trade-offs, underscoring the practicality and efficiency of the proposed approach for embedded object detection tasks across diverse domains.

In **[2]** there is a reference to another paper (3) Scalable Object Detection using Deep Neural Networks, that introduces a novel approach for object detection in images using deep convolutional neural networks. Unlike previous methods that predict a single bounding box for each object category, the proposed method predicts multiple class-agnostic bounding boxes along with confidence scores, allowing for handling variable instances of each class. The model leverages saliency-inspired concepts to predict locations likely to contain objects of interest, enabling cross-class generalization and scalability. Competitive performance is demonstrated on challenging benchmarks like VOC2007 and ILSVRC-2012, with the method capable of predicting locations with high accuracy while using only a small number of evaluations. The approach, termed DeepMultiBox, exhibits scalability and the ability to generalize across datasets and object categories, including instances of the same class. Future directions include integrating localization and recognition into a single network, reducing the need for multiple evaluations. Despite the two-pass procedure (localization followed by categorization), the approach remains competitive with other methods and scales efficiently with the number of classes. Overall, the proposed method presents a promising direction for scalable object detection using deep neural networks, with potential applications in various domains requiring accurate and efficient detection algorithms.

In **[3]**, the paper introduces a light-weight RetinaNet model designed to enable practical deployment on edge devices for IoT-based object detection services. Initially, the study validates RetinaNet as having the best FLOP-mAP trade-off among all mAP-30-tier networks. Subsequently, a light-weight RetinaNet structure is proposed, achieving effective computation-accuracy trade-offs by reducing FLOPs in computationally intensive layers. Compared to conventional methods of trading computation with accuracy through input image scaling, the proposed solution consistently offers a superior FLOPs-mAP trade-off curve. The light-weight RetinaNet model demonstrates a 0.3% mAP improvement at a 1.8x FLOPs reduction point compared to the original RetinaNet and achieves 1.8x more energy efficiency on an Intel Arria 10 FPGA accelerator for edge computing. The method holds promise for various object detection applications, allowing them to optimize runtime and accuracy while enjoying more energy-efficient inference at the edge. Overall, the paper provides insights into enhancing the FLOP-mAP trade-off in FPN-based detection networks, facilitating more efficient edge-based object detection with improved performance.

In **[4]**, the paper introduces LightDet, a lightweight object detector tailored for resource-constrained scenarios. It tackles the trade-off between accuracy and computational burden by proposing innovative modules. The Detail-Preserving Module enhances the backbone network to capture intricate low-level features crucial for accurate detection. Additionally, the Feature-Preserving and Refinement Module efficiently aggregates bottom-up and top-down features, outperforming existing methods like FPN. A lightweight prediction head further reduces network complexity. Experimental results on PASCAL VOC 2007 demonstrate LightDet's superiority, achieving a 75.5% mean average precision (mAP) at 250 frames per second (FPS). It also achieves a 24.0% mAP on the MS COCO dataset. The study emphasizes LightDet's effectiveness in resource-constrained environments, offering a balance between accuracy and computational efficiency. The proposed modules, particularly the Detail-Preserving and Feature-Preserving ones, significantly contribute to LightDet's performance improvement. This research sheds light on the importance of preserving

low-level information and refining feature aggregation for lightweight object detection. The findings challenge previous approaches and provide a more effective solution for one-stage object detection methods. LightDet's superior performance underscores its potential for practical applications where computational resources are limited.

In **[5]**, the research paper explores the utilization of Convolutional Neural Networks (CNNs) for object detection, crucial for applications like mobile robot navigation, surveillance, and explosive ordnance disposal (EOD). Two state-of-the-art models, Single Shot Multi-Box Detector (SSD) with MobileNetV1 and Faster Region-based Convolutional Neural Network (Faster-RCNN) with InceptionV2, are compared for their object detection performance. Results indicate that SSD with MobileNetV1 excels in real-time applications due to its high-speed detection, albeit with lower accuracy. Conversely, Faster-RCNN with InceptionV2 offers greater accuracy at the expense of speed. The conclusion highlights the trade-off between accuracy and speed in object detection tasks, suggesting SSD with MobileNetV1 for fast detection in real-time scenarios and Faster-RCNN with InceptionV2 for more precise detection needs. Future research aims to implement these models as part of a vision system for bomb disposal robots to detect improvised explosive devices (IEDs). This study underscores the importance of selecting the appropriate model based on the specific requirements of the application, balancing between speed and accuracy in object detection tasks.

In **[5]** there is a reference to another paper (1) Human Detection and Tracking for Video Surveillance, that addresses the escalating need for effective video surveillance systems amidst increasing crime rates worldwide. It introduces a novel method integrating Histograms of Oriented Gradients (HOG), Visual Saliency theory, and Deep Multi-Level Network to detect and track human beings in video sequences. The approach leverages Visual Saliency for region proposal, enhancing human detection accuracy compared to traditional methods. By incorporating the Deep Multi-Level Network for saliency prediction, computational efficiency is improved without

compromising precision and recall. Additionally, the utilization of the k-Means algorithm aids in clustering feature vectors, enabling the determination of human motion patterns and tracking paths within the video sequence. Experimental results demonstrate a detection precision of 83.11% and a recall of 41.27%, achieved 76.866 times faster than classification on normal images. Overall, the proposed method offers a promising solution for efficient human detection and tracking in video surveillance systems, with implications for enhancing security measures and crime prevention efforts.

In **[5]** there is a reference to another paper (2) Statistical-Based Tracking Technique for Linear Structures Detection, that proposes a novel tracking-based segmentation method for detecting linear structures, particularly blood vessels, in medical images, with a focus on retinal images for diagnostic purposes. The method utilizes Bayesian segmentation with the Maximum a posteriori (MAP) Probability criterion to achieve accurate vessel detection. Tests conducted on both simulated and retinal images demonstrate superior performance compared to existing vessel detection techniques. The proposed method offers advantages such as the ability to segment vessels based on their diameter, allowing for the detection of both main vessels and smaller ones. The study concludes with promising initial results, suggesting the potential of the proposed technique for improving blood vessel detection in medical imaging. Future research directions include enhancing the method with more advanced models, such as mixtures of Gaussians and additional geometric parameters, and further evaluation with a larger image database to validate its effectiveness comprehensively. Overall, this work contributes to advancing computer-aided diagnosis and disease monitoring through improved segmentation techniques for medical images.

2.3 Performance Analysis of existing Models

Table 2.2 given below highlights mAP calculated on different models when using the Detrac dataset, we observe that ResNet10 SSD gives the highest mAP.

Table 2.2 mAP of different models on Detrac dataset

DETRAC		
Model Name	Reference	mAP
MobileNet SSD	[1]	89.21
MobileNet light SSD	[1]	91.41
PVANet SSD	[1]	91.66
SqueezeNet1.0 SSD	[1]	92.05
ResNet10 SSD	[1]	92.52

In table 2.3 we calculated the mAP on the Pascal dataset where MobileNet SSdv2 outperforms and secures the maximum mAP.

Table 2.3 mAP of different models on Pascal dataset

PASCAL VOC		
Model Name	Reference	mAP
SqueezeNet1.0 SSD	[1]	38.45
ResNet10 SSD	[1]	64.83
PVANet SSD	[1]	67.69
MobileNet SSD	[1]	70.04
SSDM 7.5	[1]	73.08
MobileNet-SSD	[2]	72.4
MobileNet-SSDv2	[2]	75.6

In table 2.4 we calculated the mAP on the COCO dataset where Yolo v2 gave maximum mAP.

Table 2.4 mAP of different models on Coco dataset

COCO		
Model Name	Reference	mAP
RetinaNet	[3]	35.7
LW-RetinaNet-v1	[3]	35.4
LW-RetinaNet-v2	[3]	35.1
LW-RetinaNet-v3	[3]	34.6
Yolov2	[4]	76.8
SSD	[4]	77.2
MobileNet-SSD	[4]	68.0
Pelee	[4]	70.9

2.4 Background Models

1) Convolutional Neural Network Model (CNN)

It is a type of deep neural network that is particularly well-suited for image recognition and computer vision tasks. Below is a simplified explanation of how a CNN works:
The Fig.2.1 and Fig 2.2 describes an overview of the architecture.

Architecture Overview:

1. Input Layer:
   - The input layer of a CNN takes the raw input, typically an image.

- Each pixel value in the image is treated as a neuron in this layer.

2. Convolutional Layer:
- The convolutional layer is the core building block of a CNN.
- It consists of filters (also called kernels) that slide over the input image to detect patterns or features.
- Each filter is responsible for learning a different feature, such as edges, textures, or more complex patterns.
- The result of this convolution is a feature map, highlighting the presence of different features in the input.

3. Activation Function:
- After convolution, an activation function (commonly ReLU - Rectified Linear Unit) is applied element-wise to introduce non-linearity.
- The purpose is to capture complex relationships and make the network capable of learning from the data.

4. Pooling Layer:
- Pooling layers are used to reduce the spatial dimensions of the input volume.
- Max pooling is a common technique where the maximum value in a group of neighboring pixels is selected, effectively downsampling the input.

5. Flattening:
- The output from the convolutional and pooling layers is then flattened into a one-dimensional vector.
- This vector serves as the input for the fully connected layers.

6. Fully Connected (Dense) Layers:
- These layers connect every neuron from one layer to every neuron in the next layer.
- The fully connected layers combine the features learned by the convolutional layers to make predictions.

- The last fully connected layer typically produces the final output, and an activation function (e.g., softmax for classification) is applied to obtain probabilities for different classes.

7. Output Layer:
- The output layer provides the final prediction based on the learned features.
- The number of neurons in this layer depends on the task, with softmax often used for classification problems.

8. Training:
- The model is trained using a labeled dataset through a process called backpropagation.
- During training, the model adjusts its internal parameters (weights and biases) to minimize the difference between its predictions and the actual labels.

9. Loss Function:
- The loss function measures the difference between the predicted values and the actual labels.
- The goal during training is to minimize this loss.

10. Optimization:
- Optimization algorithms (e.g., SGD, Adam) are used to update the model parameters in the direction that reduces the loss.

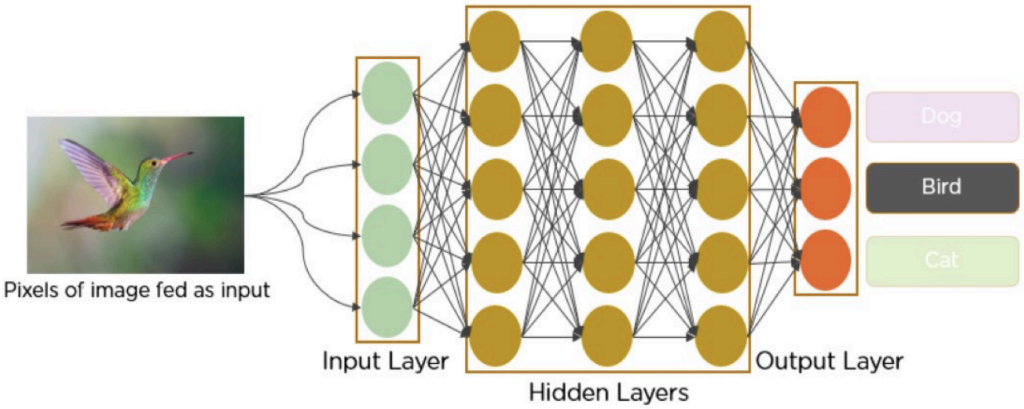

Fig 2.1 CNN Model

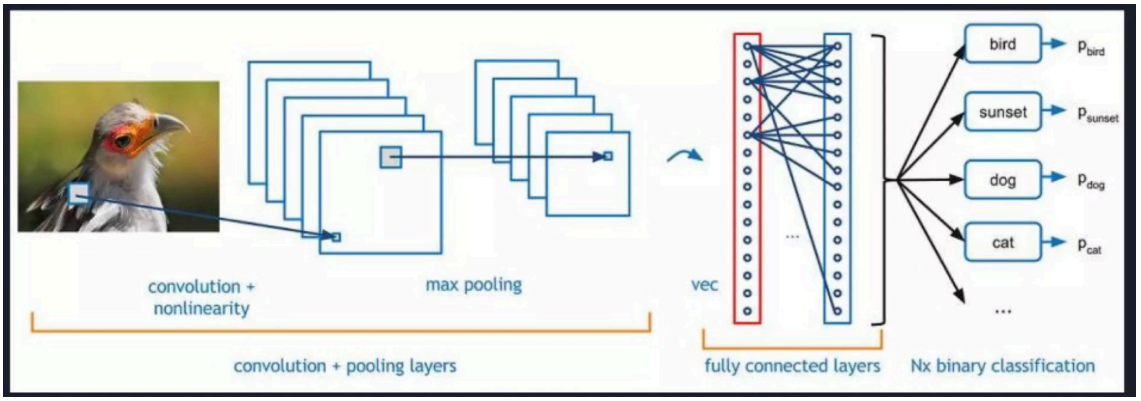

Fig 2.2 CNN layers

2) YOLO v5 Model:

YOLO (You Only Look Once) is an object detection algorithm that divides an image into a grid and directly predicts bounding boxes and class probabilities for each grid cell. YOLOv5 is one of the versions of the YOLO algorithm. Here's a simplified explanation of how YOLOv5 works:

The Fig.2.3 describes an overview of the architecture.

Architecture Overview:

1. Input Layer:

- Similar to other neural networks, YOLOv5 begins with an input layer that takes the raw input image.

2. Backbone Network:
- YOLOv5 typically employs a backbone network (e.g., CSPDarknet53) to extract hierarchical features from the input image.
- The backbone network consists of convolutional layers and other operations to capture both low-level and high-level features.

3. Neck:
- The neck of YOLOv5 connects the backbone to the detection head. It may involve additional convolutional layers to enhance feature representations.

4. Detection Head:
- The detection head is responsible for predicting bounding boxes and class probabilities for objects.
- YOLOv5 predicts bounding boxes directly, as opposed to anchor-based methods.
- The head consists of convolutional layers that produce predictions for object class, confidence score, and bounding box coordinates for each grid cell.

5. Output Layer:
- The output layer of YOLOv5 provides the final predictions in the form of bounding boxes and associated class probabilities.
- The predictions are often post-processed to filter out low-confidence detections and perform non-maximum suppression to keep only the most confident and non-overlapping boxes.

6. Training:
- YOLOv5 is trained using a labeled dataset, and the training involves optimizing the model parameters to minimize a specific loss function.

- The loss function considers both localization (bounding box coordinates) and classification (object class) errors.

7. Loss Function:

- YOLOv5 typically uses a combination of localization loss, confidence loss, and class loss in the form of the YOLO loss function.

8. Optimization:

- During training, optimization algorithms like SGD or Adam are used to update the model's weights and biases to minimize the loss.

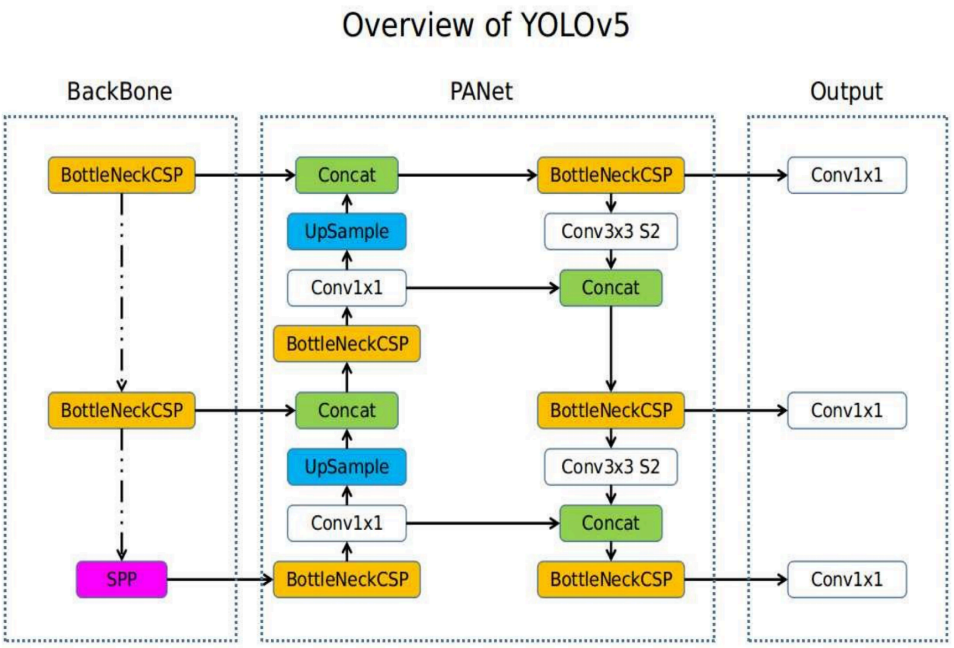

Fig 2.3 YOLO v5 Model

CHAPTER 3
METHODOLOGY

3.1 Dataset

Dataset Name : Indoor Objects Detection

Description: It contains Indoor objects dataset taken from Roboflow. This labeled dataset contains 10 classes:

1. Backpack
2. Book
3. Cup
4. Key
5. Laptop
6. Mouse
7. Phone
8. Remote
9. Wallet
10. Bottle

There are total 4945 images in the dataset.

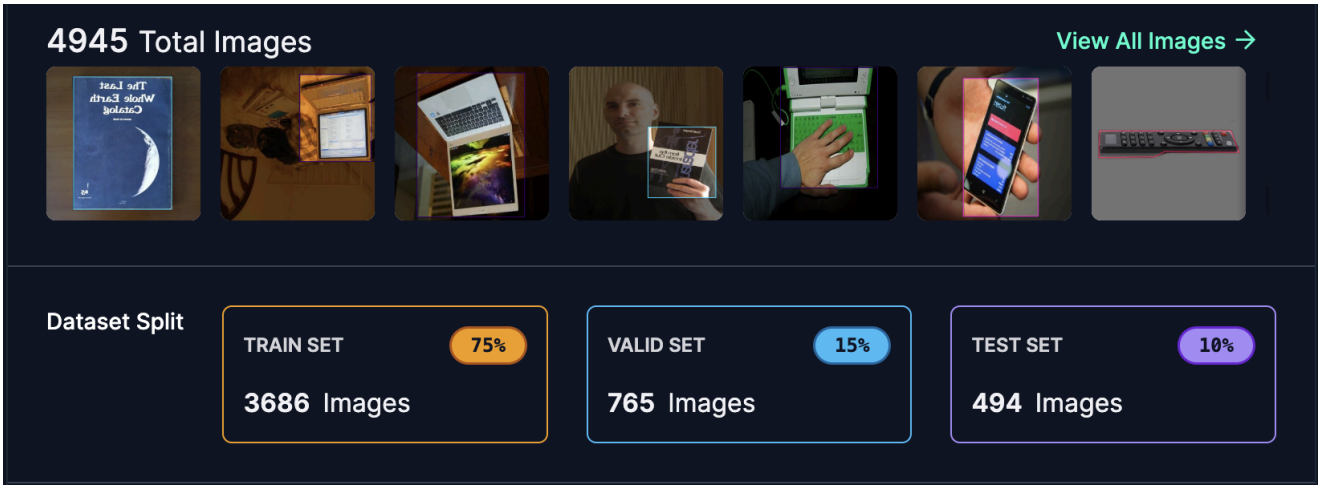

Fig 3.1: Dataset description

Dataset Details

We have downloaded the dataset in TFRecord format and Pascal Voc XML format.
TFR format contains .pbtxt files and the Voc files contains .xml files

Purpose of TFR and VOC format

The purposes of .pbtxt files and XML files in the context of object detection are to provide structured annotations for training and evaluating object detection models. Both file types play crucial roles in preparing and organizing datasets for training and evaluating object detection models.
Here's how each file type is typically used:

1. Protocol Buffers Text Format (.pbtxt):

- Purpose: .pbtxt files are often used to define the schema or structure of datasets, especially in the context of object detection tasks.
- Usage:
    - Label Mapping: They define the mapping between class names and their corresponding numerical IDs. For instance, in the provided example, "book" is mapped to ID 1, and "bottle" to ID 2.
    - Dataset Annotation Schema: In some cases, .pbtxt files might be used to define additional metadata or attributes associated with each class, such as color, shape, or additional descriptive information.
- Structure:
    Here's a breakdown of the structure and elements in the provided .pbtxt file:

Indoor-object_label_map.pbtxt

```
item {
    name: "book",
    id: 1,
    display_name: "book"
}
item {
    name: "bottle",
    id: 2,
    display_name: "bottle"
}
item {
    name: "cabinet",
    id: 3,
    display_name: "cabinet"
```

Fig 3.2: Sample .pbtxt file

-> item: This keyword defines a new item in the Protocol Buffers message definition. Each item block represents a distinct entity or object with its attributes.

-> name: This field specifies the internal name or identifier for the item. It's typically used to reference the item programmatically. For example, "book" and "bottle" are the names of the items defined in this file.

-> id: This field assigns a unique numerical identifier to the item. It's often used for efficient referencing and indexing of items within a dataset or system. In the provided example, "book" has an ID of 1, and "bottle" has an ID of 2.

-> display_name: This field specifies the human-readable display name for the item. It's used for presenting the item's name to users or for visualization purposes. In the provided example, both "book" and "bottle" have the same display names as their internal names.

2. XML Files (.xml):

- Purpose: XML files are used to provide detailed annotations for individual objects within images.
- Usage:
    - Bounding Box Annotation: XML files typically contain information about the bounding boxes of objects within images. This includes the coordinates of the bounding box and the class label of the object.
    - Object Attributes: They may also contain additional information about each object, such as pose, truncation, occlusion, and difficulty, which can be useful for training and evaluation.

- Structure: Each <object> element represents one annotated object in the image, and there can be multiple <object> elements in the file for multiple objects. These annotations are crucial for training and evaluating object detection algorithms.
  Let's break down the structure and elements of this XML file:

    1. <annotation>: This is the root element of the XML file, containing all the other elements within it.

2. <filename>: This element contains the filename of the image being annotated.

3. <path>: This element contains the file path of the image being annotated.

4. <source>: This element specifies the source of the data. In this case, it indicates that the data is sourced from roboflow.com.

5. <size>: This element contains information about the size of the image.

    a. <width>: Width of the image in pixels.
    b. <height>: Height of the image in pixels.
    c. <depth>: Number of channels in the image (e.g., 3 for RGB images).
6. <segmented>: This element indicates whether the image has been segmented. A value of '0' typically means the image is not segmented.

7. <object>: This element represents an object present in the image.

    a. <name>: Name of the object (e.g., "chair" in this case).

    b. <pose>: Pose of the object (e.g., "Unspecified").

    c. <truncated>: Indicates whether the object is truncated in the image (0 for no, 1 for yes).

    d. <difficult>: Indicates whether detecting the object is difficult (0 for no, 1 for yes).

    e. <occluded>: Indicates whether the object is occluded in the image (0 for no, 1 for yes).

    f. <bndbox>: Bounding box coordinates of the object.

        - <xmin>: Minimum x-coordinate of the bounding box.
        - <xmax>: Maximum x-coordinate of the bounding box.
        - <ymin>: Minimum y-coordinate of the bounding box.
        - <ymax>: Maximum y-coordinate of the bounding box.

Fig 3.3: Sample .xml file

3.2 Proposed Methodology

MobileNet Model

MobileNet is a family of lightweight deep learning models designed for efficient use on mobile and edge devices. These models are known for their small size and low computational

requirements while maintaining reasonably good performance. Fig 2.4 describes an overview of MobileNet architecture.

Architecture Overview:

1. Input Layer:

- The model starts with an input layer that takes the raw input, typically an image.

2. Convolutional Layers:

- MobileNetV2 primarily uses depth wise separable convolutions, which consist of two separate operations: depthwise convolutions and pointwise convolutions.

- Depthwise Convolution: Applies a single filter per input channel.

- Pointwise Convolution:Combines the outputs of the depthwise convolutions using 1x1 convolutions.

3. Inverted Residual Blocks:

- MobileNetV2 introduces inverted residuals, which consist of a lightweight shortcut connection across the depthwise separable convolution.

- The inverted residuals help in capturing and propagating information through the network.

4. Bottleneck Architecture:

- Each layer of MobileNetV2 utilizes a bottleneck architecture, which involves reducing the number of channels (dimensions) with a 1x1 convolution, performing the depthwise separable convolution, and then expanding the dimensions again with another 1x1 convolution.

- This bottleneck architecture reduces the computational cost while maintaining expressiveness.

5. Linear Bottleneck:

- MobileNetV2 introduces a linear bottleneck design, where the output channels of the layer are linearly scaled between the input and output.

- This helps in maintaining a balance between the low-level and high-level features.

6. Activation Function:

- Rectified Linear Unit (ReLU) is commonly used as the activation function between layers to introduce non-linearity.

7. Global Average Pooling:

- Instead of using fully connected layers at the end, MobileNetV2 typically employs global average pooling.

- Global average pooling reduces the spatial dimensions of the data to produce a fixed-size tensor, which is then fed to the output layer.

8. Output Layer:

- The output layer produces the final predictions based on the features learned by the previous layers.

- The number of neurons in this layer depends on the specific task (e.g., classification, object detection).

9. Training and Optimization:

- MobileNetV2 is trained using labeled datasets through a process called backpropagation.

- The model's parameters (weights and biases) are optimized using optimization algorithms like SGD (Stochastic Gradient Descent) or Adam.

- The training involves minimizing a loss function that measures the difference between the predicted values and the actual labels.

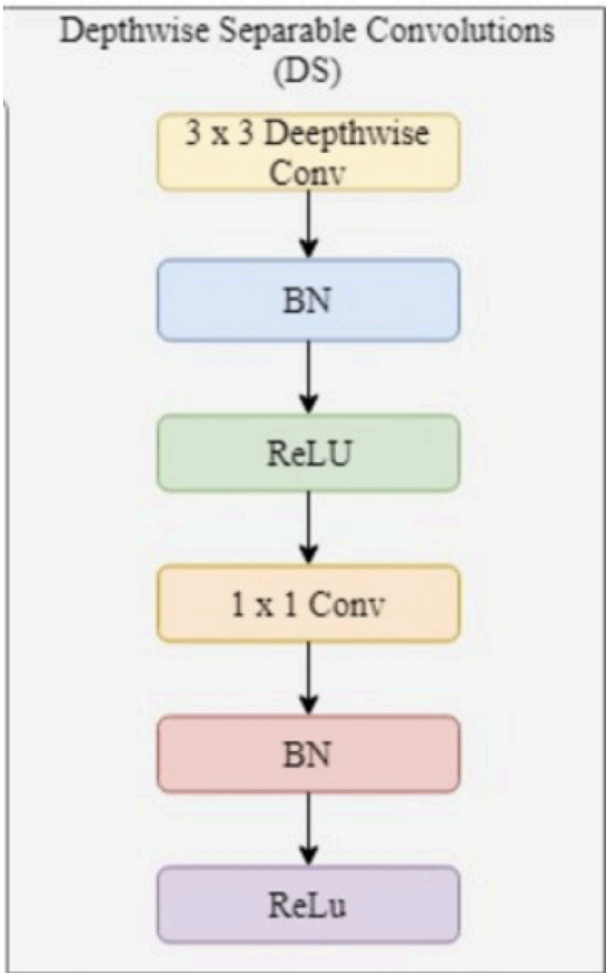

Fig 3.4 MobileNet Architecture

We use the idea of learning anchor boxes based on the distribution of bounding boxes in the custom dataset with K-means and genetic learning algorithms. This is very important for custom tasks, because the distribution of bounding box sizes and locations may be dramatically different than the preset bounding box anchors in the Roboflow dataset.

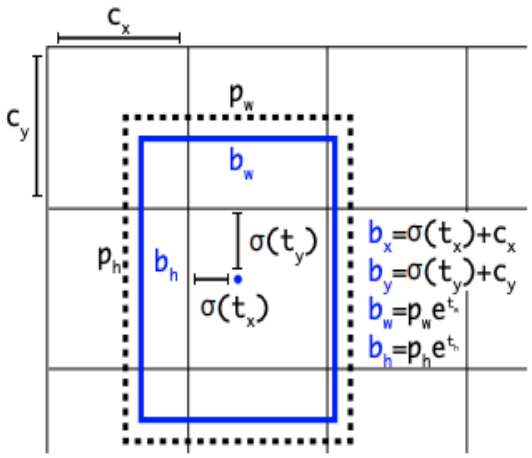

Fig 3.5 Bounding Boxes to determine objects

Equations used to compute the target bounding boxes :

$$b_x = \left(2 \cdot \sigma\left(t_x\right) - 0.5\right) + c_x$$
$$b_y = \left(2 \cdot \sigma\left(t_y\right) - 0.5\right) + c_y$$
$$b_w = p_w \cdot \left(2 \cdot \sigma\left(t_w\right)\right)^2$$
$$b_h = p_h \cdot \left(2 \cdot \sigma\left(t_h\right)\right)^2$$

3.3 Process Flow of Model

1) Model Selection:

The methodology begins with selecting a suitable lightweight deep learning model for object detection, focusing on MobileNet due to its efficiency and compact architecture.

2) Preprocessing:

Input images undergo preprocessing to standardize dimensions and enhance features, ensuring compatibility with the MobileNet architecture.

3) Feature Extraction:

MobileNet's depth wise separable convolutions are utilized to extract features from input images efficiently, capturing both low-level and high-level features relevant to object detection. Following are the major features in Feature Extraction:

- Input Layer: The model starts with an input layer that takes the raw input, typically an image.
- Convolutional Layers: Utilizes depthwise separable convolutions, consisting of depthwise convolutions and pointwise convolutions, to extract features efficiently from input images.
- Inverted Residual Blocks: These blocks help capture and propagate information through the network, enhancing feature representation.
- Bottleneck Architecture: MobileNet employs a bottleneck architecture to reduce computational complexity while maintaining performance.
- Activation Function: Rectified Linear Unit (ReLU) is commonly used as the activation function between layers to introduce non-linearity.
- Global Average Pooling: The feature maps are subjected to global average pooling, reducing their spatial dimensions and summarizing feature information.

4) Training:

The preprocessed images and corresponding object labels are used to train the MobileNet-based object detection model, fine-tuning its parameters for the specific task.

5) Real-time Detection:
    - Output Layer: The model concludes with an output layer that generates predictions for object detection tasks, typically using softmax activation to produce class probabilities.
    - Upon training completion, the model is deployed for real-time object detection, leveraging MobileNet's lightweight design to enable swift processing and accurate detection of selected indoor and outdoor objects.

6) Deployment on Jetson Xavier:
The trained model is optimized and deployed on the Jetson Xavier platform to leverage its processing power and compactness, ensuring efficient object detection in edge computing scenarios.

7) Performance Evaluation:
The performance of the deployed object detection system is evaluated using benchmark datasets and real-world scenarios, assessing factors such as detection accuracy, processing speed, and resource utilization.

8) Iterative Improvement:
The methodology allows for iterative refinement and optimization of the object detection model based on performance feedback, aiming to continuously enhance accuracy and efficiency in detecting indoor and outdoor objects in real-time scenarios.

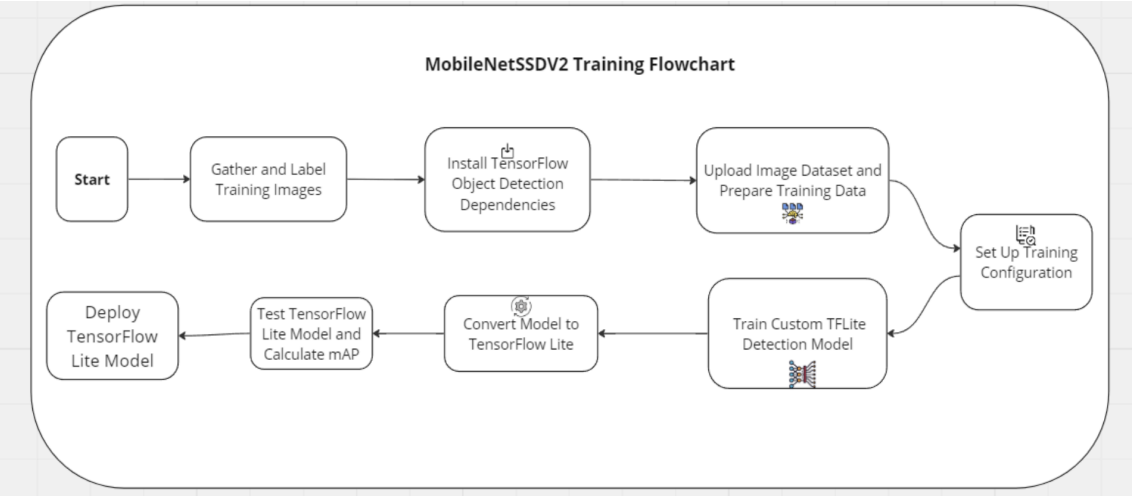

Fig 3.6: MobileNet model flowchart

CHAPTER 4
IMPLEMENTATION

4.1 Implementing The Dataset

In our dataset, we have organized the data into 10 classes, with each class having its own separate folder. Each folder contains an average of 230 images. This structured arrangement is a practical way to manage and access the data for implementation and analysis. We will be using Google Colab for designing the MobileNet model

4.2 Install TensorFlow Object Detection Dependencies

First, we'll install the TensorFlow Object Detection API in this Google Colab instance. This requires cloning the TensorFlow models repository and running a couple installation commands. The latest version of TensorFlow this Colab has been verified to work with is TF v2.8.0.
Then we will test our installation by running model_builder_tf2_test.py to make sure everything is working as expected by running the following command:
!python
/content/models/research/object_detection/builders/model_builder_tf2_test.py

4.3 Upload Image Dataset and Prepare Training Data

In this section, we'll upload our data and prepare it for training with TensorFlow. We'll upload our images, split them into train, validation, and test folders, and then run scripts for creating TFRecords from our data.
First, on your local PC, zip all your training images and XML files into a single folder called "images.zip". The files should be directly inside the zip folder, or in a nested folder as shown below in figure.

Finally, we need to create a labelmap for the detector and convert the images into a data file format called TFRecords, which are used by TensorFlow for training. We'll use Python scripts to automatically convert the data into TFRecord format. Before running them, we need to define a labelmap for our classes.
The code section below will create a "labelmap.txt" file that contains a list of classes as discussed in Fig 4.1 and Fig 4.2:

```
images.zip
-- images
  -- img1.jpg
  -- img1.xml
  -- img2.jpg
  -- img2.xml
  ...
```

Fig 4.1: images.zip

```
### This creates a a "labelmap.txt" file with a list of classes the object detection
#  model will detect.
%%bash
cat <<EOF >> /content/labelmap.txt
door
cabinetDoor
refrigeratorDoor
window
chair
table
cabinet
couch
openedDoor
pole
EOF

# Download data conversion scripts
! wget https://raw.githubusercontent.com/EdjeElectronics/TensorFlow-Lite-Object-Detectior
! wget https://raw.githubusercontent.com/EdjeElectronics/TensorFlow-Lite-Object-Detectior

# Create CSV data files and TFRecord files
!python3 create_csv.py
!python3 create_tfrecord.py --csv_input=images/train/train_labels.csv --labelmap=labelmap
!python3 create_tfrecord.py --csv_input=images/validation/validation_labels.csv --labelma

train_record_fname = '/content/train.tfrecord'
val_record_fname = '/content/val.tfrecord'
label_map_pbtxt_fname = '/content/labelmap.pbtxt'
```

Fig 4.2: Create Labelmap and TFRecords

4.4 Set Up Training Configuration

In this section, we'll set up the model and training configuration. We will specify which pretrained TensorFlow model we want to use from the TensorFlow 2 Object Detection Model Zoo. Each model also comes with a configuration file that points to file locations, sets training parameters (such as learning rate and total number of training steps), and more. We'll modify the configuration file for our custom training job.

Set the "chosen_model" variable to match the name of the model that we are training with. It's currently set to use the popular "ssd-mobilenet-v2" model as given in Fig 4.3

```
# Change the chosen_model variable to deploy different models available in the TF2 object detection zoo
chosen_model = 'ssd-mobilenet-v2-fpnlite-320'

MODELS_CONFIG = {
    'ssd-mobilenet-v2': {
        'model_name': 'ssd_mobilenet_v2_320x320_coco17_tpu-8',
        'base_pipeline_file': 'ssd_mobilenet_v2_320x320_coco17_tpu-8.config',
        'pretrained_checkpoint': 'ssd_mobilenet_v2_320x320_coco17_tpu-8.tar.gz',
    },
    'efficientdet-d0': {
        'model_name': 'efficientdet_d0_coco17_tpu-32',
        'base_pipeline_file': 'ssd_efficientdet_d0_512x512_coco17_tpu-8.config',
        'pretrained_checkpoint': 'efficientdet_d0_coco17_tpu-32.tar.gz',
    },
    'ssd-mobilenet-v2-fpnlite-320': {
        'model_name': 'ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8',
        'base_pipeline_file': 'ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8.config',
        'pretrained_checkpoint': 'ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8.tar.gz',
    },
    # The centernet model isn't working as of 9/10/22
    #'centernet-mobilenet-v2': {
    #    'model_name': 'centernet_mobilenetv2fpn_512x512_coco17_od',
    #    'base_pipeline_file': 'pipeline.config',
    #    'pretrained_checkpoint': 'centernet_mobilenetv2fpn_512x512_coco17_od.tar.gz',
    #}
}

model_name = MODELS_CONFIG[chosen_model]['model_name']
pretrained_checkpoint = MODELS_CONFIG[chosen_model]['pretrained_checkpoint']
base_pipeline_file = MODELS_CONFIG[chosen_model]['base_pipeline_file']
```

Fig: 4.3 Setting the chosen model

Now that we have downloaded our model and config file, we need to modify the configuration file with some high-level training parameters. The following variables are used to control training steps: (Refer Fig: 4.4)

- num_steps: The total amount of steps to use for training the model. A good number to start with is 40,000 steps. You can use more steps if you

notice the loss metrics are still decreasing by the time training finishes. The more steps, the longer training will take. Training can also be stopped early if loss flattens out before reaching the specified number of steps.

- batch_size: The number of images to use per training step. A larger batch size allows a model to be trained in fewer steps, but the size is limited by the GPU memory available for training. With the GPUs used in Colab instances, 16 is a good number for SSD models and 4 is good for EfficientDet models.

```
# Set training parameters for the model
num_steps = 40000

if chosen_model == 'efficientdet-d0':
  batch_size = 4
else:
  batch_size = 16
```

```
[ ] # Set file locations and get number of classes for config file
pipeline_fname = '/content/models/mymodel/' + base_pipeline_file
fine_tune_checkpoint = '/content/models/mymodel/' + model_name + '/check

def get_num_classes(pbtxt_fname):
    from object_detection.utils import label_map_util
    label_map = label_map_util.load_labelmap(pbtxt_fname)
    categories = label_map_util.convert_label_map_to_categories(
        label_map, max_num_classes=90, use_display_name=True)
    category_index = label_map_util.create_category_index(categories)
    return len(category_index.keys())
num_classes = get_num_classes(label_map_pbtxt_fname)
print('Total classes:', num_classes)
```

Fig: 4.4 : Setting training parameters

4.5 Train Custom TFLite Detection Model

We are ready to train our object detection model! Before we start training, we have to load up a TensorBoard session to monitor training progress. It won't

show anything yet, because we haven't started training. Once training starts, come back and click the refresh button to see the model's overall loss.
Model training is performed using the "model_main_tf2.py" script from the TF Object Detection API. Training will take anywhere from 2-3 hours, depending on the model, batch size, and number of training steps. We have already defined all the parameters and arguments used by model_main_tf2.py in previous sections, here is a snippet :

```
# Run training!
!python /content/models/research/object_detection/model_main_tf2.py \
    --pipeline_config_path={pipeline_file} \
    --model_dir={model_dir} \
    --alsologtostderr \
    --num_train_steps={num_steps} \
    --sample_1_of_n_eval_examples=1
```

4.6 Convert Model to TensorFlow Lite

Alright! Our model is all trained up and ready to be used for detecting objects. First, we need to export the model graph (a file that contains information about the architecture and weights) to a TensorFlow Lite-compatible format. We'll do this using the export_tflite_graph_tf2.py script as discussed in Fig 4.6.
Next, we'll take the exported graph and use the TFLiteConverter module to convert it to .tflite FlatBuffer format.

```
# Make a directory to store the trained TFLite model
!mkdir /content/custom_model_lite
output_directory = '/content/custom_model_lite'

# Path to training directory (the conversion script automatically chooses the highest checkpoint file)
last_model_path = '/content/training'

!python /content/models/research/object_detection/export_tflite_graph_tf2.py \
    --trained_checkpoint_dir {last_model_path} \
    --output_directory {output_directory} \
    --pipeline_config_path {pipeline_file}


# Convert exported graph file into TFLite model file
import tensorflow as tf

converter = tf.lite.TFLiteConverter.from_saved_model('/content/custom_model_lite/saved_model')
tflite_model = converter.convert()

with open('/content/custom_model_lite/detect.tflite', 'wb') as f:
  f.write(tflite_model)
```

Fig 4.6 : Convert Model to TensorFlow Lite

4.7 Test TensorFlow Lite Model and Calculate mAP

We have trained our custom model and converted it to TFLite format. But how well does it actually perform at detecting objects in images? This is where the images we set aside in the test folder come in. The model never saw any test images during training, so its performance on these images should be representative of how it will perform on new images from the field. Here is a little snippet of the code:

```
# Import packages
import os
import cv2
import numpy as np
import sys
import glob
import random
import importlib.util
from tensorflow.lite.python.interpreter import Interpreter

import matplotlib
import matplotlib.pyplot as plt

%matplotlib inline

### Define function for inferencing with TFLite model and displaying results

def tflite_detect_images(modelpath, imgpath, lblpath, min_conf=0.5, num_test_images=10,

  # Grab filenames of all images in test folder
  images = glob.glob(imgpath + '/*.jpg') + glob.glob(imgpath + '/*.JPG') + glob.glob(img

  # Load the label map into memory
  with open(lblpath, 'r') as f:
      labels = [line.strip() for line in f.readlines()]

  # Load the Tensorflow Lite model into memory
  interpreter = Interpreter(model_path=modelpath)
  interpreter.allocate_tensors()
```

Fig 4.7: Test TensorFlow Lite Model and Calculate mAP

4.8 Deploy TensorFlow Lite Model

Now that our custom model has been trained and converted to TFLite format, it's ready to be downloaded and deployed in an application! This section shows how to download the model and provides links to instructions for deploying it on the our PC, or other edge devices as shown in Fig 4.8

```
[ ]  # Move labelmap and pipeline config files into TFLite model folder and zip it up
     !cp /content/labelmap.txt /content/custom_model_lite
     !cp /content/labelmap.pbtxt /content/custom_model_lite
     !cp /content/models/mymodel/pipeline_file.config /content/custom_model_lite

     %cd /content
     !zip -r custom_model_lite.zip custom_model_lite
```

```
     from google.colab import files

     files.download('/content/custom_model_lite.zip')
```

Fig 4.8: Download TFLite model

4.9 Appendix: Common Errors

Here are solutions to common errors that can occur while stepping through this notebook.

1. Training suddenly stops with ^C output:

If training randomly stops without any error messages except a ^C, that means the virtual machine has run out of memory. To resolve the issue, try reducing the batch_size variable in Step 4 to a lower value like batch_size = 4. The value must be a power of 2. (e.g. 2, 4, 8 ...)

Source:

https://stackoverflow.com/questions/75901898/why-my-model-training-automatically-stopped-during-training

CHAPTER 5
RESULTS

5.1 Class Description

Given below is Table 5.1 showing different classes we took for training the proposed model.

Table 5.1 : Representing Class labels and corresponding Class names

Class Number	Class Name	Number of Images
1	Backpack	696
2	Book	256
3	Cup	390
4	Key	497
5	Laptop	685
6	Mouse	331
7	Phone	461
8	Remote	574
9	Wallet	561
10	Bottle	388

Accuracy: The proportion of correctly classified instances in the total number of instances. This is a simple and commonly used metric for classification tasks. Table 5.2 shows all mAP calculated for different models on the same dataset.

$$Accuracy: \frac{Number\ of\ Correct\ Predictions}{Total\ Number\ of\ Predictions}$$

5.2 Performance comparison with existing models on same dataset

Now let's analyze the performance of other models and calculate their mAP and see how our model performs than the rest as discussed in table 5.2

Table 5.2 : Feature based model metrics

Model	**mAP**	**Training Time**	**Epoch**
CNN	92	35 min	100
YOLO v5 s	82.4	32 min	150
YOLO v5 m	42	52 min	299 (249)
YOLO v5 l	85	44 min	150
YOLO v5 l	88	129 min	299
MobileNet (proposed method)	54.74	180 min	300

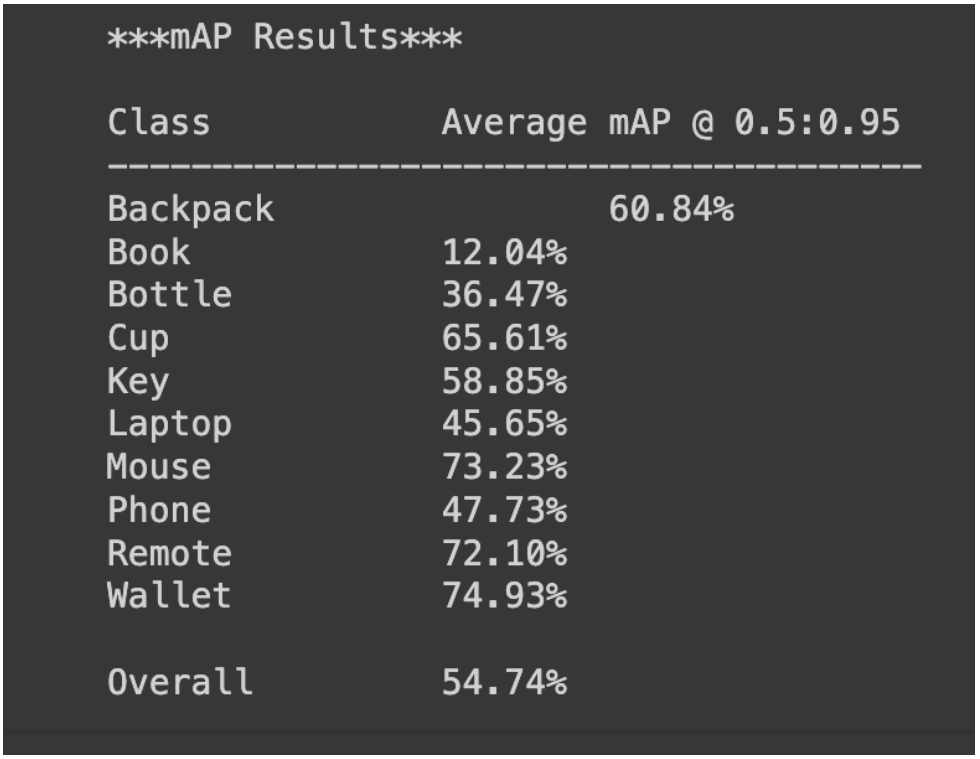

Fig 5.1 mAP results of proposed MobileNet model

5.3 Static Image classification using MobileNet:

Here we provide random static images as input to the model which classifies the object and presents it within a bounding box. Some input images give are described in Fig 5.2 , 5.3, 5.4 and 5.5

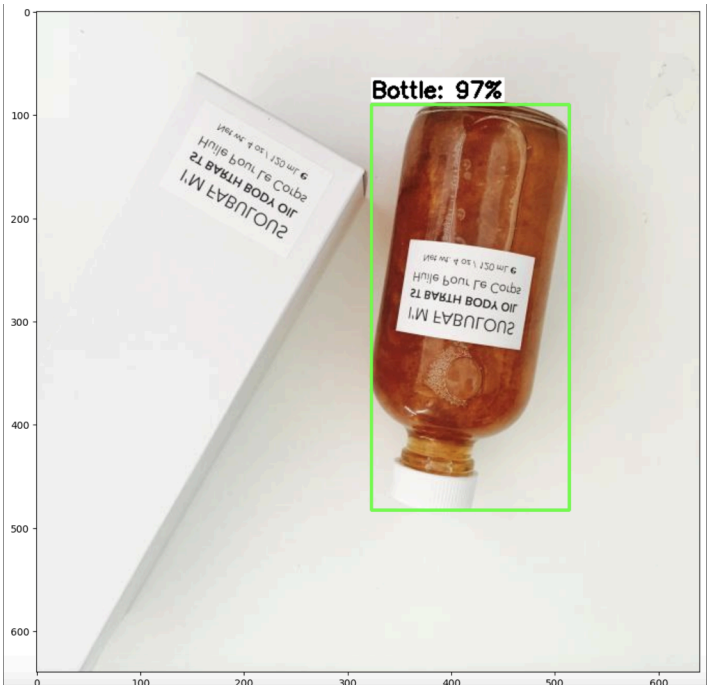

Fig 5.2 Bottle classification

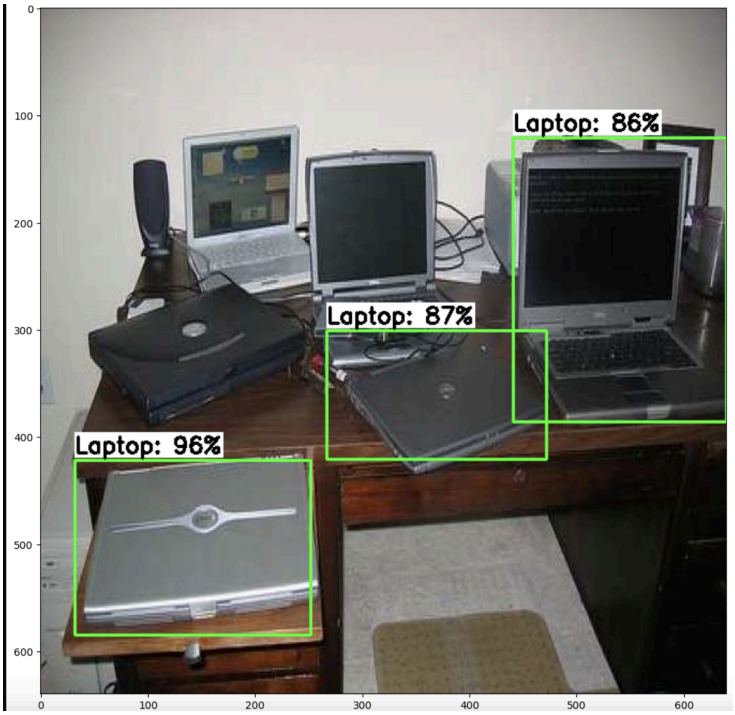

Fig 5.3 Laptop classification

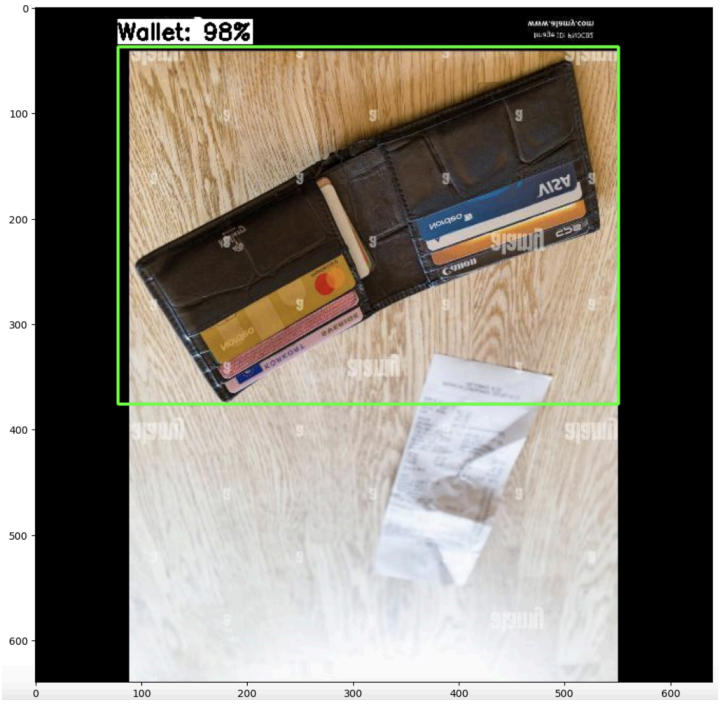

Fig 5.4 Wallet classification

5.4 Live Inferencing using Webcam:

Here we provide live images as input to the model from the webcam, which classifies the object and presents it within a bounding box. Some input images give are described in Fig 5.6 , 5.7, 5.8. We set up a TensorFlow Lite Runtime environment on a macOS device and used Anaconda to create a Python environment to install the TFLite Runtime in.

Fig 5.5 Live Cup classification

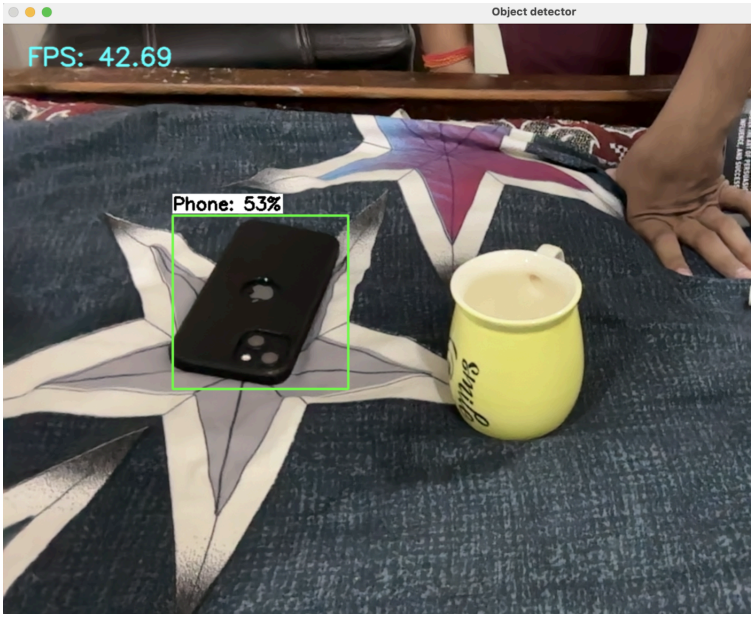

Fig 5.6 Live Phone classification

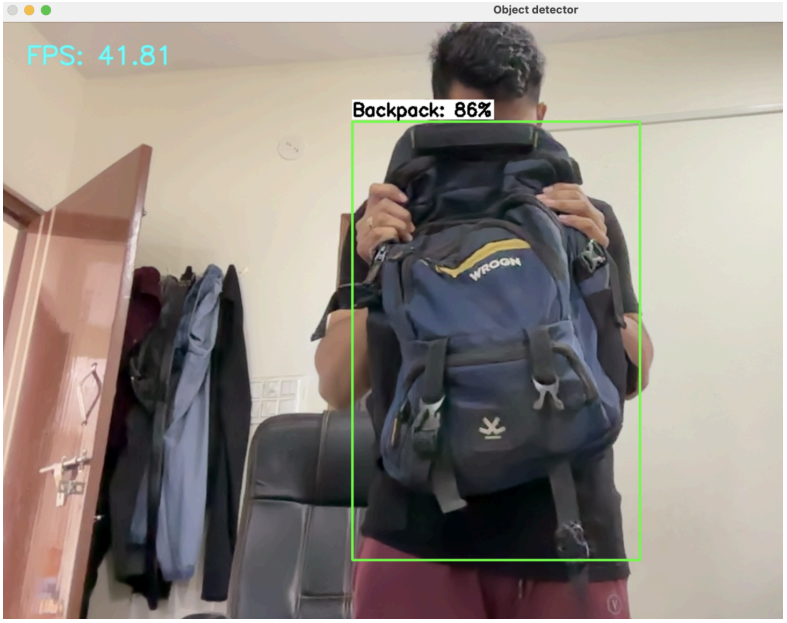

Fig 5.7 Live Backpack classification

CHAPTER 6
CONCLUSION AND FUTURE SCOPE

CONCLUSION:

In conclusion, this project presented an innovative approach to object detection utilizing lightweight deep learning models, specifically MobileNet. By leveraging the efficiency and compactness of MobileNet, we achieved real-time object detection capabilities for indoor and outdoor objects with high accuracy.

Our evaluation on benchmark datasets and real-world scenarios demonstrated the effectiveness of the proposed methodology. The MobileNet-based object detection model outperformed traditional deep learning models in terms of computational efficiency while maintaining competitive detection accuracy.

Moreover, the deployment of the model on the Jetson Xavier platform further enhanced its performance, enabling efficient processing and compact deployment in edge computing environments. This highlights the practical utility of lightweight models like MobileNet for real-time object detection tasks.

Additionally, the project adhered to ethical principles, ensuring that the object detection system does not promote harm, discrimination, or unethical practices. It serves as a responsible and safe solution for object detection in various applications.

Overall, the utilization of lightweight models for object detection offers promising prospects for enhancing efficiency, accuracy, and real-time performance in a wide range of indoor and outdoor object detection scenarios.

Future Scope:

1. **Enhanced Model Optimization:** Continued research and development efforts can focus on further optimizing the lightweight models, such as MobileNet, for improved efficiency and accuracy in object detection tasks. This may involve exploring advanced optimization techniques and model architectures tailored specifically for edge computing environments.

2. **Integration of Advanced Features:** The project can explore the integration of advanced features, such as attention mechanisms or context-awareness, into the lightweight object detection models to enhance their capabilities in understanding complex scenes and objects.

3. **Multi-Object Detection**: Extend the project to support the detection of multiple objects simultaneously in real-time scenarios. This could involve adapting the lightweight models to handle overlapping objects and diverse object categories efficiently.

4. **Domain-specific Applications**: Investigate the application of lightweight object detection models in specific domains such as healthcare, retail, or agriculture. Tailoring the models to address domain-specific challenges and requirements can unlock new opportunities for practical deployment.

5. **Hardware Acceleration:** Explore hardware acceleration techniques to further optimize the performance of lightweight models on edge devices like Jetson

Xavier. Leveraging specialized hardware accelerators such as GPUs or TPUs can significantly enhance processing speed and efficiency.

6. **Integration with IoT Ecosystem**: Explore integration opportunities with IoT ecosystems to enable seamless communication and interaction between lightweight object detection models and other IoT devices and systems for broader application scenarios.

REFERENCES

1] Anisimov, D., & Khanova, T. (2017, August). Towards lightweight convolutional neural networks for object detection. In 2017 14th IEEE international conference on advanced video and signal based surveillance (AVSS) (pp. 1-8). IEEE.

2] Chiu, Y. C., Tsai, C. Y., Ruan, M. D., Shen, G. Y., & Lee, T. T. (2020, August). Mobilenet-SSDv2: An improved object detection model for embedded systems. In 2020 International conference on system science and engineering (ICSSE) (pp. 1-5). IEEE.

3] Li, Y., Dua, A., & Ren, F. (2020, June). Light-weight RetinaNet for object detection on edge devices. In 2020 IEEE 6th World Forum on Internet of Things (WF-IoT) (pp. 1-6). IEEE.

4] Tang, Q., Li, J., Shi, Z., & Hu, Y. (2020, May). Lightdet: A lightweight and accurate object detection network. In ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 2243-2247). IEEE.

5] Galvez, R. L., Bandala, A. A., Dadios, E. P., Vicerra, R. R. P., & Maningo, J. M. Z. (2018, October). Object detection using convolutional neural networks. In TENCON 2018-2018 IEEE Region 10 Conference (pp. 2023-2027). IEEE.