Object Detection in Refrigerator and Calorie Estimation
using EfficientDet1 and YOLOv5S Techniques on Mobile Devices

by

Rohit Agarwal
B.Tech., Uttar Pradesh Technical University, India, 2015

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER IN SCIENCE

in the Department of Computer Science

Object Detection in Refrigerator and Calorie Estimation
using EfficientDet1 and YOLOv5S Techniques on Mobile Devices

by

Rohit Agarwal
B.Tech., Uttar Pradesh Technical University, India, 2015

Supervisory Committee

Dr. Hausi A. Müller, Supervisor
(Department of Computer Science)

Dr. Alex Thomo, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Hausi A. Müller, Supervisor
(Department of Computer Science)

Dr. Alex Thomo, Departmental Member
(Department of Computer Science)

ABSTRACT

In this research, we developed a tool to automate object detection and calorie estimation of food items in home refrigerators compatible with both windows and mobile devices. This tool detects objects using the state-of-the-art one-stage methods EfficientDet1 and Yolov5s (You look only once). The tool's performance is assessed by comparing speed, accuracy, time, and mean average precision (mAP) to learn the advantages in real-time applications. Open-source frameworks such as TensorFlow or PyTorch are used to train the object detection models. Using these models, the object detection of items is identified and classified. A Python algorithm is developed to count the number of items and their calorie estimations. The algorithm automates the process by generating the inventory of items in the refrigerator and sending it to users at their designated e-mail addresses. If any of the desired food items are not present or fall below their threshold values, then the items are flagged.

The experimental results indicate that object detection models (EfficientDet1 and Yolov5s) are suitable to run on smaller models for real-time applications. The associated web-based graphical user interface (GUI) (compatible with mobile devices as well) displays the list of items in a detected image, the calorie estimation of each item, and the list of missing items needed for further action. Though our GUI Python code is tested on the local computer, it can be repurposed on any platform such as mobile or embedded devices. The advantage of automatic detection of refrigerator items through the mobile app is helpful in making life easier and healthier, especially for diet-conscious people.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

Dr. Hausi A. Müller, my supervisor, for his supervision, enthusiasm, motivation, and encouragement throughout my Master's program. I want to thank him for providing me with the opportunity to work with him. I am really grateful to him for his continuous guidance, support, and feedback during this research.

Prof. Alex Thomo, for being my committee member and mentor. I am really grateful to him for guiding me and providing feedback throughout this research work.

Kirti, Sunil and all Rigi and Pita group members, for their valuable support, advice, and feedback on my research and academics over the entire course of this beautiful journey.

My wife, family members and friends, for encouraging and inspiring me all through my journey. It would be impossible to realize this dream without their immense support and love.

Chapter 1

Introduction

1.1 Problem Definition and Motivation

Nowadays people are very busy in their lives. They may even forget what they have in their refrigerator and results in food wastage due to spoilage. People who forget food in their refrigerator either do not have regular diet habits or have to go shopping again when they come back to their home because they forget what they have in their refrigerator. This situation is not harmful but it increases carbon emissions. Our climate is changing dramatically every year, and for this reason, we should not waste our fuel energy. On the other hand, we must have a regular diet to keep healthy. The goal of this research project is to develop a fast and efficient object detection technology that can run on microcomputers such as Raspberry Pi. This involves an algorithm to count the number of items in the refrigerator and estimate their calories. Finally, an inventory of items in the refrigerator is generated and sent to the user. If any of the desired food items are not present or fall below the threshold values, then the users are alerted by e-mail as well as through a Graphical User Interface (GUI) which will list the names of items that need to be replenished. The advantage of this automation and object detection is an easier and healthier life, especially for diet-conscious people.

Although Internet of Things (IoT) technologies are common for home devices, newly manufactured refrigerators are unable to assess refrigerator stock inventory situations or do not support diet-conscious people. Hence, we have developed a built-in option for object detection and calorie estimation for future refrigerators so that food wastage may be reduced and the choice of healthy diet options can be promoted. People will

be fitter knowing their food calories and reduce their carbon emissions.
Thus, this thesis presents a new approach to reducing food waste in refrigerators and estimating food calories for people who care about their diet. At the same time, the approach will provide new search and development ideas to automate object detection in the home refrigerator.

1.2 Related state-of-the-art detection techniques

There is significant research interest in facilitating and automating tasks in home refrigerators by big companies such as LG and Samsung to develop smart refrigerators. However, their results of object identification are still limited both in terms of accuracy and speed as they can not install big computers into refrigerators. Instead, it is possible to install microcomputers such as Raspberry Pi or Arduino or to run an app to provide good and dedicated results for the user. Raspberry Pi has an inbuilt SD card slot and has a memory of 1 GB RAM. It can connect to a computer and utilize input devices such as a keyboard and mouse. Python is one of the most widely used languages to develop software for Raspberry Pi. The use of such software or an app in the refrigerator may make people's lives easier and healthier.
Most of the object detection techniques used in the home refrigerator are based on convolution neural networks (CNN) and are installed using large computers. Recently, Pachon et al. 2018 [1] used the region-based convolution neural network (R-CNN) [2] technique in the development of product detection system in a home refrigerator based on pattern recognition.
Agarwal 2018 [3] in her thesis work used a single shot detector (SSD) and faster region-based CNN (R-CNN) techniques and did show that SSD is faster than faster R-CNN for detecting object identification in a home refrigerator. Also, she suggested that there is a complex training process that makes Faster R-CNN [4] slower than SSD [5]. Despite the fact that these approaches have overcome the constraints of data limitation and modeling in object detection, they are not capable of detecting objects in real-time applications.
So far, YOLO (You Only Look Once) a real-time object detection technique has not been studied for home refrigerators yet. When you want to detect in real-time and in one go, the YOLO algorithm has gained prominence to achieve greater performance than SSD or Faster R-CNN [6].
YOLO uses CNN to recognize the object in real-time and takes a single forward

propagation through a neural network, and hence, it is called YOLO (You look only once). It also performs well when the object size is small. Further, the algorithm has exceptional learning abilities allowing it to learn object representations that facilitate object detection. YOLO uses the following detection technique:

- Residual blocks: The image is first divided into several grids and objects appear within grid cells and will be detected by each grid cell.
- Bounding box regression: A bounding box is an outline that draws attention to a certain object in a picture.
- Intersection Over Union (IOU): IOU is used by YOLO to create an outbox that properly surrounds the objects or items.

The key difference between the YOLO and SSD architectures is that the YOLO architecture utilizes two fully connected layers, whereas the SSD network uses convolutional layers of varying sizes.

There are several variations of the YOLO algorithm. Firstly, Redmon et al. 2016 proposed a new YOLO algorithm to detect objects which achieved double mean Average Precision (mAP) in real-time detectors [6]. Then, Redmon and Farhadi 2017 proposed the improved algorithm YOLO V2 [7]. This was further optimized by Wei et al. 2017 to increase the average accuracy rate of the detection network [8]. Redmond and Farhadi, 2018 provided the third version YOLO V3 which is as accurate as SSD but three times faster (YOLOv3[1] source code)[9]. This uses the features learned by a deep convolution neural network for detecting objects in real-time. Zhao and Shuaiyang 2020 [10] presented an improved version of Yolov3 using the k-means cluster method and found that it has better performance than the original Yolov3 [9] in terms of recall, mean average precision and F1-score. Yolov3 is widely used because of improved speed, high accuracy, and excellent learning capabilities.

Recently, Tan et al. 2020, the Google Brain team, published their state-of-the-art EfficientDet [0-7] models using Tensor Flow Lite (TFlite) software for object detection for the use of mobiles or embedded devices [11]. These EfficientDet models outperform the similar-sized models in comparison to popular object detection models like Yolov3 [7], Mask R-CNN [12] and RetinaNet [13] as shown in Figure 1.1. This figure shows that the performance of EfficientDet (D0-D4) models speed (Flops) and accuracy

[1] `https://pjreddie.com/darknet/yolo/`

(COCO AP) is better than previous detectors including Yolov3 [11]. The source code of EfficientDet models is available to use by these authors on github.[2]

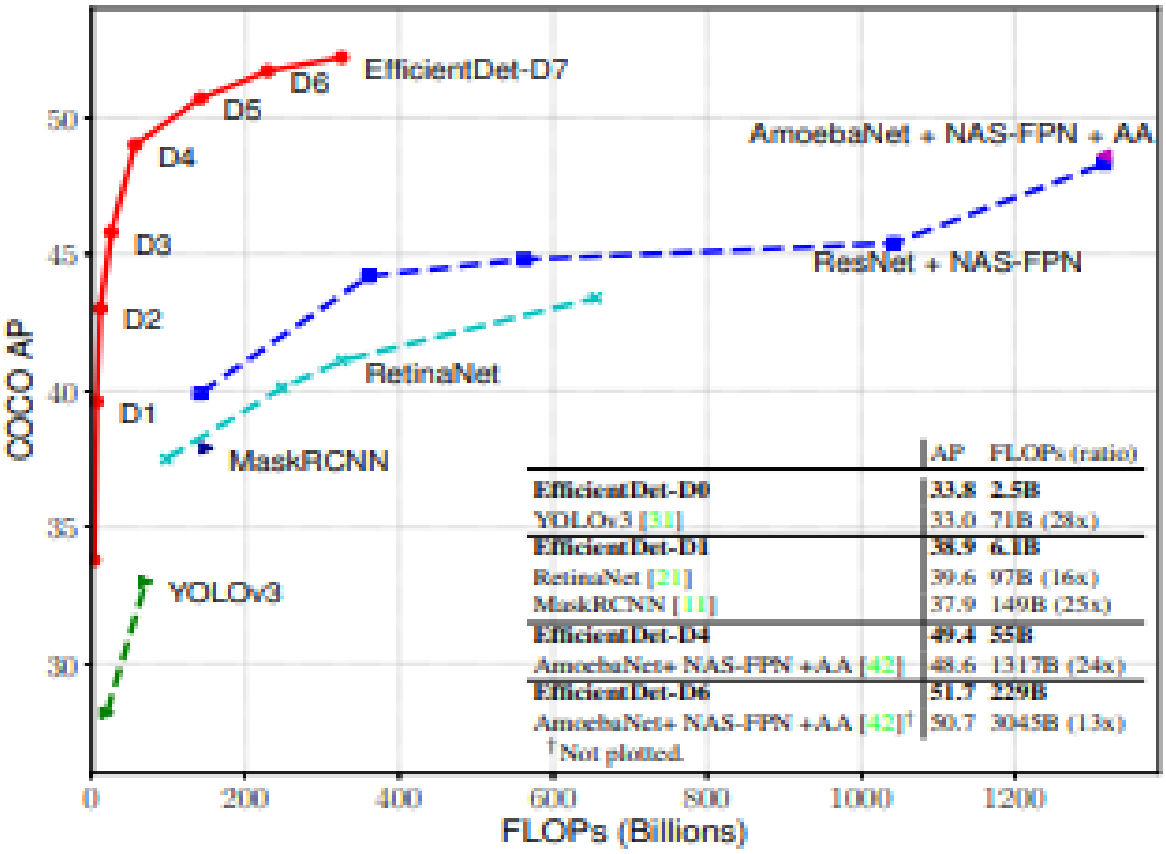


Figure 1.1: Model FLOPs vs. COCO accuracy – All numbers are for single-model single-scale. Our EfficientDet [0-7] achieves new state-of-the-art 52.2% COCO AP with much fewer parameters and FLOPs than previous detectors.

Further, Tan et al. 2020 showed that EfficientDet-Lite[0-4] are a family of mobile/IoT-friendly object detection models derived from the EfficientDet architecture which can be used for mobile or embedded devices. Table 1.1 compares the performance of EfficientDet-Lite models. In Table 1.1, size is the integer quantized model, latency is measured on Raspberry Pi4 using four threads on the CPU and Average Precision is the mAP (mean Average Precision) on the COCO 2017 validation dataset.

In this comparison, authors used EfficientDet-Lite [0-4] to train their models and present their results. We find that EfficientDet1 architecture is highly suitable for

[2] `https://github.com/google/automl/tree/master/efficientdet`

Table 1.1: Comparison of Latency and Average Precision of the EfficientDet-Lite [0-4] models using the COCO 2017 dataset

Architecture	Size(MB)	Latency(ms)	Average Precision
Efficient-Lite0	4.4	1.46	25.69%
Efficient-Lite1	5.8	259	30.55%
Efficient-Lite2	7.2	396	33.97 %
Efficient-Lite3	11.4	716	37.70 %
Efficient-Lite4	19.9	1886	41.96 %

small-size models to run on mobile devices (like Raspberry Pi) as a trade-off between speed (Latency) and accuracy. Hence we will be using EfficientDet1 for our research project in this thesis.

Soon after the discovery of EfficientDet models surpassing the importance of Yolov3, Glen Jocher (May 18, 2020) introduced the state-of-the-art Yolov4 and Yolov5 models whose performance turned out to be even much better than EfficientDet[0-4], as shown below in Figure 1.2 [14]. The open-source code is readily available by Glen Jocher for use at Ultralytics[3] [15].

As seen from Figure 1.2, yolov5 has various types of models with software designed to generate small-size models (Yolov5s), medium-size models (Yolov5m), large-size models (Yolov5l) and extra large size model (Yolov5x). Their performance in terms of speed (GPU speed ms/image) and accuracy (COCO AP value) is impressive and better than EfficientDet models (D0-D4).

Table 1.2: The performance parameters of different sizes of Yolov5 models

Model	APval	APtest	AP50	SpeedGPU	FPSGPU	params	FLOPS
YOLOV5s	36.6	36.6	55.8	2.1ms	476	7.5M	13.2B
YOLOV5m	43.4	43.4	62.4	3.0ms	333	21.8M	39.4B
YOLOV5l	46.6	46.7	65.4	3.9 ms	256	47.8M	88.1B
YOLOV5x	48.4	48.4	66.9	6.1 ms	164	89.0M	166.4B
YOLOV3-SPP	45.6	45.5	65.2	4.5 ms	222	63.0M	118.0B

Table 1.2 shows the performance of actual numbers for these Yolov5 models [14].

[3] `https://docs.ultralytics.com/yolov5`

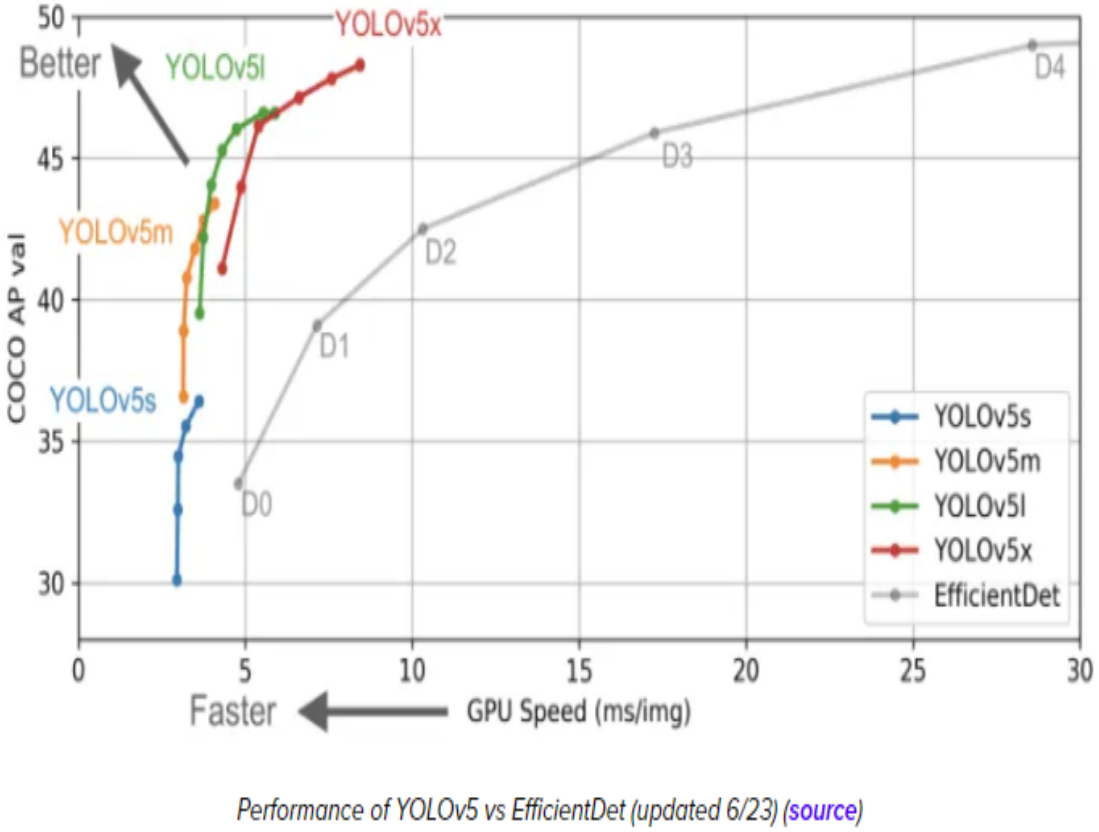


Figure 1.2: Model FLOPs vs. COCO accuracy AP for EfficientDet [0-4] and Yolov5 models

It will be interesting to see whether the average precision at 50% IOU, even for smaller model YoloV5s with fewer parameters (7.5M) has a good accuracy of 55.8% (AP_50) and very good FLOPS (13.2B) to make it suitable for real-time applications. Hence, we selected the Yolov5s to run on the smaller models of mobile devices for our research investigation.

In view of the above recent prominent discoveries in the field of object detection for small models to run on mobile devices, we selected two state-of-the-art object detection techniques (EfficientDet1 and Yolov5s) for our research investigation and will be discussing more in detail.

1.3 Open-source frameworks for object detection and proposed approach

This project applies different software-based techniques such as image processing, sending emails, calorie estimation, and a graphical user interface (GUI). All these functions have been developed using Python. For image processing, we need datasets to train object detection models and many existing datasets available on the internet. Thus, we have downloaded the selected item's photos from the internet and if they are unlabeled, we have annotated them manually. After preparing the dataset, we will use open-source frameworks such as Google's Tensorflow Lite (TFlite) Object detection API developed by the Google team for EfficientDet models [16, 17, 18, 19], and Yolov5s which uses the PyTorch framework created by Facebook team [20]. Both frameworks are very powerful and have good documentation. These frameworks make it possible to create our application-specific object detectors.

To train these models, we have used Google Colaboratory[4] since they allow anybody to write and execute arbitrary Python code through the browser. It is well suited to machine learning, and data analysis and offers free GPU cycles for this task. Also, we developed Python code to count the items of each class and implemented calorie estimation for each food item by pre-defining the approximate calories (taken from the internet) of each item in the code. For sending email, Python has various libraries but we have employed the Pyqt libraries. Training is the most critical part of this project. We trained the Yolov5s model with PyTorch software and the EfficientDet1 model with TFlite. We focused on finding the fastest and most efficient trained model that works effectively on Raspberry Pi. The results of both the models are compared in terms of speed and accuracy. Our proposed workflow is shown in Figure 1.3.

Our comparison results in terms of speed and mean average Precision (mAP) between the two models will help us to learn if they are suitable for real-time applications or not. Furthermore, new research models will illuminate our way to find the best models for our case.

(i) Speed-accuracy Trade-off of different object detection models will help us to determine which model is best for our application.

(ii) Our methodology and results could potentially help other researchers to design a custom object detector and further enhance the precision of their data sets.

[4] `https://colab.research.google.com/`

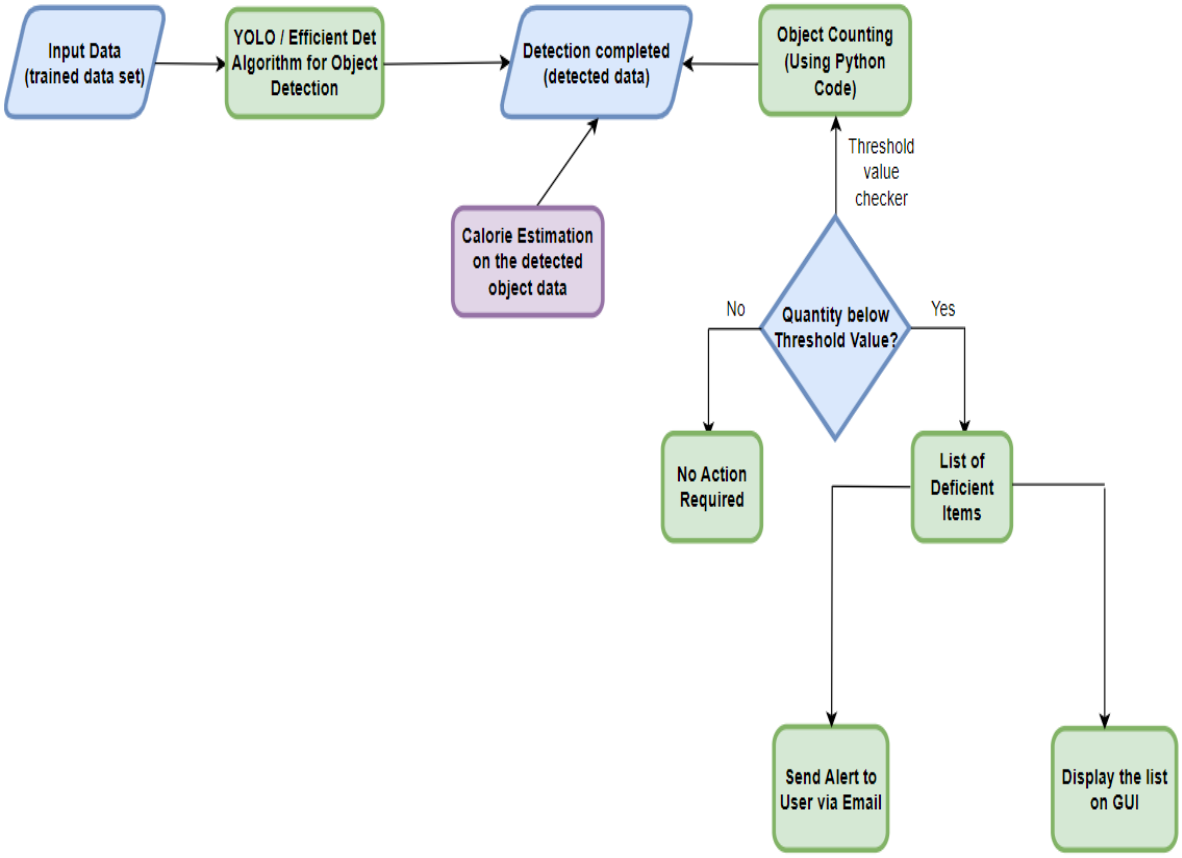


Figure 1.3: Work-flow of our research project

As depicted in Figure 1.3, this research project is not limited to only object detection but also calorie estimation, and counting the number of items. We will develop an alert system and have used a graphical user interface (GUI) to provide the inventory of refrigerator items at regular intervals to the user. The process will be automated by incorporating that if any of the desired food items are not detected or fall below their thresh hold values, then the user is alerted by e-mail and through the GUI by listing the names of the items that are missing.

1.4 Research objectives

The objectives of the research presented in this thesis are as follows:

1. Further confirm, discover, and discuss practices that yield improved performance in terms of accuracy and speed in the domain of object-detecting convolution neural networks
2. Implement an object detection model suitable in terms of size and speed to run on mobile devices such as Raspberry Pi and detect home items in real-time
3. Show the good performance of object detection models in a complex environment with a large number of classes (around 25 or more) and taking the ready-made large image dataset available from the internet
4. Develop an automatic system of object detection of refrigerator food items to avoid wastage with an alert system and GUI for the user
5. Investigate how the small-size object detection model performs in terms of precision and accuracy for mobile devices in real-time applications
6. Investigate how the training of object detection model parameters for small models behaves on mobile devices.
7. Investigate whether our object detection methods are successful for small models and which techniques are more suitable for real-time applications.

1.5 Thesis Outline

Chapter 1 presents the motivation, state-of-art detection techniques for mobile devices, proposed research model, and a summary of contributions.

Chapter 2 incorporates the key concepts of our literature review of the state-of-the-art object detection methods. It also presents relevant related work in this area of research.

Chapter 3 presents the fast and efficient EfficientDet1 and Yolov5s object detection techniques that work on smaller computers such as Raspberry Pi.

Chapter 4 presents the large image datasets, training of models (EfficientDet and Yolov5s), and their evaluation metrics parameters used throughout this research. It also computes the box, object, and class loss functions during training and validation of the image dataset and finds the best-computed model for detecting the test image.

Chapter 5 evaluates and analyzes the performance of EfficientDet and Yolov5s on the test dataset. This chapter also includes comparing two techniques and highlights the important differences between the two.

Chapter 6 summarizes our research conclusions and offers insights for future research on mobile devices for automation.

Chapter 2

Background and Related work

2.1 Basic Concepts

Most of the previous research studies primarily focused on understanding and modeling a particular type of constraint, such as technological, contractual, resource, spatial, and information constraints. However, they did not use the detection information for making a decision.

The present research is more focused on integrating object detection with decision-making. Object detection mainly consists of three parts (i) recognizing, (ii) classifying and (iii) localizing the objects and drawing bounding boxes around them. Most of the successful object detection networks make use of neural network-based image classifiers in conjunction with object detection techniques. The neural networks used are Artificial Neural Networks (ANN), Deep Learning Networks (DNN), and Convolution Neural Networks (CNN).

2.1.1 Artificial neural network (ANN)

Artificial neural networks (ANN) are one of the main tools used in machine learning. They are brain-inspired systems designed to mimic how humans learn, as the word "neural" in their name suggests. Neural networks consist of input and output layers, as well as (in most cases) two or more hidden layers consisting of units that transform the input into something that the output layer can use. The first hidden layer learns the simple and basic features and then passes them to the second layer. The second layer can learn more complex features than the first one. Thus, having more hidden layers can learn more complicated features and are useful to solve sophisticated

problems like image classification, object detection, etc.
An ANN system with one input layer and output layer along with two hidden layers is shown in Figure 2.1 [21]. While neural networks are around since 1940, they did not become popular until the idea of "backpropagation" came into the last several decades. Now they are a major part of artificial intelligence (AI).

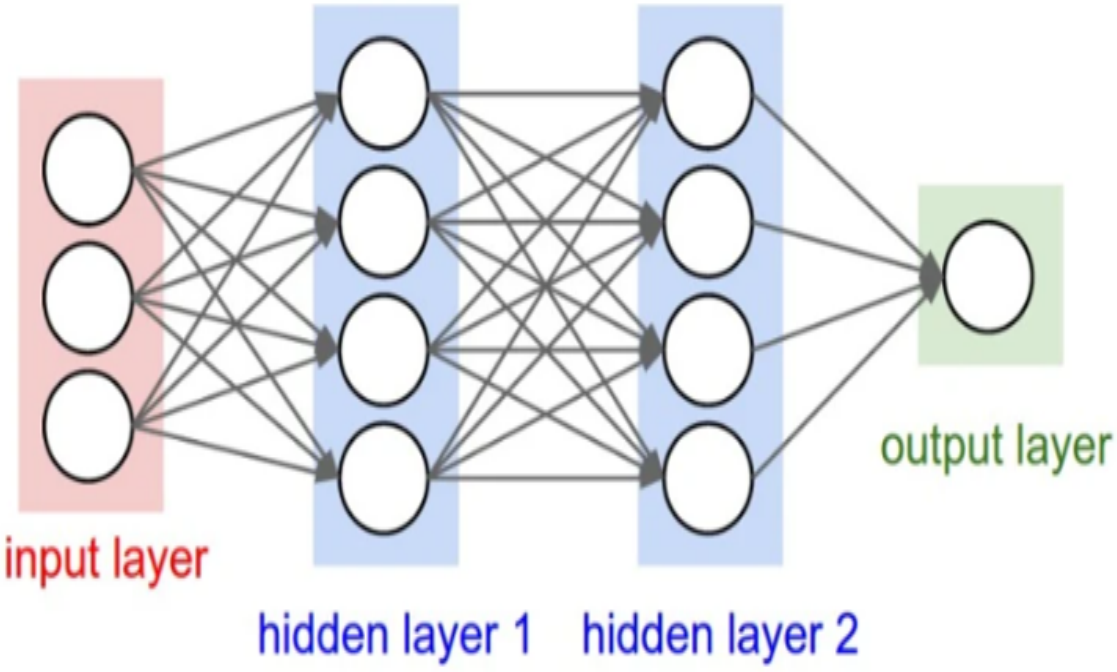


Figure 2.1: An architecture of artificial neural network (ANN)

2.1.2 Deep learning neural network (DNN)

Another important development has been the introduction of deep learning neural networks (DNN). DNN uses different layers of a multi-layer network and extracts different features until it can recognize what it is looking for. In other words, the multi-layered ANN is known as a DNN. Having more layers, more complex structures can be trained through the deeper layers [22].
There are different types of learning such as supervised or unsupervised learning or

reinforcement learning, in which the network learns for itself by trying to maximize its score.

(i) How do the neural networks actually learn?

Neural networks learn the same way as we learn from our experiences in our lives. Neural networks require more data to learn. The larger the data to learn, the more accurate will be the learning, and the more accurate will be the outcomes. Once the input data is sent to the first hidden layer, it might analyze the brightness of its pixels. Based on lines of identical pixels, the following layer may then determine if there were any edges in the image. Following this, a further layer might pick up on textures and shapes, and so forth. Once this is done, then back-propagation can be used to correct any mistakes if any.

When researchers or computer scientists want to train a neural network, they will typically divide their data into three sets (training, validation, and testing datasets). The dataset assists the network in determining the various weights between its nodes during training. In the validation stage, data is used to fine-tune it. Finally, the test data is used to see if it can successfully turn the input into the desired output.

(ii) Do neural networks have any limitations?

The time it takes to train networks, which might demand a significant amount of computing power for more sophisticated tasks, is one of the greater technological challenges. However, the major problem is that neural networks are "black boxes," where the user inputs data and receives results. They can fine-tune the answers, but they do not have access to the exact decision-making process.

2.1.3 Convolutional neural network (CNN)

There are various types of neural networks depending on specific use cases and levels of complexity such as feedforward neural networks (FNN) and recurrent neural networks (RNN). However, Convolutional Neural Network (CNN) is a type of ANN used in image recognition and processing that is optimized to process pixel data. CNNs are the fundamental and basic building blocks for the computer vision task of image

segmentation. The basic architecture of CNN is shown in Figure 2.2 [23].

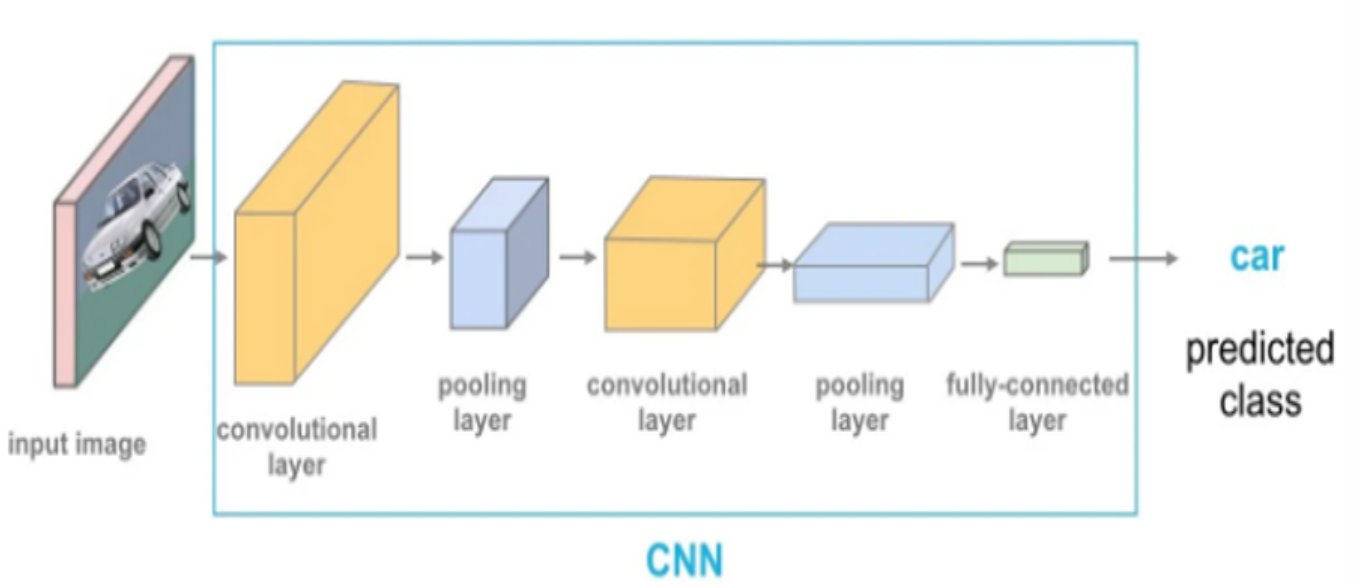


Figure 2.2: Showing the basic three types layers used in the CNN architecture

As seen from the Figure, there are three types of main layers in CNN architecture but one does sometimes use RelU and Softmax layers (not shown) depending on their specific use.

- **Convolution layer**: It helps to abstract the valuable features from an input image as a feature map with the use of filters and kernels.
- **Pooling layer**: It helps to downsample feature maps by summarizing the presence of features in patches of the feature map. It uses various filters to identify the different patches of the image like edges, corners, body, and eyes. There are various types of pooling such as max-pooling, sum pooling, and average pooling. But max-pooling is more popular to downsize the feature map. The next process is flattening which converts all the resultant two-dimensional arrays into a single continuous linear vector.
- **Fully connected layer:** The flattened matrix is fed as input to the Fully connected layers to classify the input image into classes. The fully connected layer uses an activation function such as softmax, sigmoid, or support vector

machine (SVM). The term fully connected comes from the fact that every neuron in one layer is connected to every neuron in other neighboring layers to generate the final model [24].

- **RelU / Softmax function:** Sometimes one does use RelU in the hidden layer to avoid vanishing gradient problems and for better computation performance, whereas the Softmax function is always used as a final layer of CNN classifier. The advantage of this Softmax layer is to identify the classes by assigning the decimal probabilities to each class in multi-classes problems and the target class is always selected with the highest probability. The decimal probabilities range from 0 to 1. For example, if the decimal probability is 0.9 it means 90% is the probability that the correct class will be assigned and 10% is the probability the incorrect class will be assigned. However, both correct and incorrect probabilities should add up to 1 (100%) always. RelU and Softmax layers are not shown in Figure 2.2.

(i)How to Train a Convolutional neural network (CNN)?

Training a CNN model requires a significant number of annotated images. Before starting the training process, the labeled dataset of images is divided into three parts, training dataset, validation dataset, and test dataset. During the training process, the network adjusts the filter weights by an algorithm known as back-propagation. Back-propagation consists of mainly four steps, the forward pass, calculating the loss function, the backward pass, and the weight update. These four steps constitute one iteration of learning. To minimize the loss functions, we perform a fixed number of iterations till it reaches a stage of minimum loss. Then the trained model is validated over the validation data set. If successful, then the model is ready to test any image for object detection.

2.2 Object detection techniques (Traditional)

Object detection is the task of detecting instances of objects of a certain class within an image. The state-of-the-art methods can be categorized into two main types:

(i) Two stage-methods:

Two-stage methods consist of (i) a regional proposal network and (ii) classification and localization steps. These methods prioritize detection accuracy, and example models include Region-based CNN (R-CNN), Fast R-CNN, Faster R-CNN, Mask R-CNN, and Region-based Fully Convolution Network (R-FCN).

(ii) One-stage methods:

In one-stage methods, both object classification and bounding-box regression are done directly without using pre-generated region proposals. One-stage methods prioritize inference speed, and example models include Single stage multi-box detector (SSD), RetinaNet, You look only one (Yolo), and others. Two-stage detectors usually reach better accuracy but are slower than one-stage detectors.

2.2.1 Backbone networks

Backbone networks are CNN classification architectures used by object detection models. There is a trade-off between the depth and width of networks. A shallow but wide feed-forward network may present any function, but the layers might be massive and prone to overfitting the data. Therefore by building deeper architectures the layers can be significantly smaller and less prone to overfitting. Though there exists a problem of vanishing gradient in deep architectures (deep as in many layers). In view of this, we will discuss the three popular backbone networks used in object detection models.

(i) Deep residual network Resnet (ResNet-101)

A deep residual network uses residual learning a "skip connection method" to solve the degrading of the gradient when using a high amount of layers. ResNet-101 is a 101-layer residual network and is inspired by the VGG architecture [25]. Most of the convolutional layers have 3x3 filters; they follow two rules: (i) for the same feature map size, the layers have the same number of filters; and (ii) if the feature map size is halved, the number of filters is doubled so as to preserve the time complexity per layer. Downsampling is accomplished by using convolutional layers that have a stride

of 2. At last, the network has global average pooling layers and a 1000-way fully-connected layer with softmax. This is so far just a plain network, such as VGG, but by applying shortcut connections this is evolved to a residual network [26].

(ii) Deep network inception V2 (Inception V2)

A combination of the inception network and the residual "skip connection" approach. An inception network performs convolution on input with 3 different sizes of filters, typically 1x1,3x3 and 5x5. To speed up the convolution process of the 3x3 and 5x5 filters, n x 1 and 1 x n filters are stacked. This speeds up the convolution process by about 370% compared to the original 5x5 filtering. These convolutions are performed on the same level, and the 3x3 combined with a previous 1x1 is used for dimension reduction. Instead of applying kernels (filters) and creating a deeper architecture inception goes wider. This is a sub-solution of solving the vanishing gradient problem [27].

(iii) Yolov3 specific Darknet 53

Darknet 53 consists of 53 convolutional layers. The network uses successive 3x3 and 1x1 convolutional layers. It uses residual skip connections like ResNet mentioned in (i) [9].

2.2.2 Localization

In object detection, localization is the task of finding the object's position relative to the input image. A bounding box marks the object's position, and in the localization process, several anchor boxes may be used to help detect objects of different ratios. These concepts are explained in this section.

(i) Bounding boxes

A bounding box is an area defining the position, width, and height of an object where it may exist in the image.

(ii) Anchor boxes

Anchor boxes are much similar to bounding boxes, a definition of an area defining the width and height of an object's position. First, an anchor box serves as the initial guess of the bounding box. Multiple anchor boxes of different ratios are used to cover different areas, and therefore more likely to find the correct ratio for the object.

(iii) Non-max suppression

Non-max suppression is used in object detection to identify an object only once. A bounding box often contains more than one object (it contains another object when it's center y,x position is within the bounding box). First, all low-probability detections are removed by using a pre-set threshold. Then the maximum probability within the bounding box is chosen and the intersection over union (IoU value) for the other objects is calculated and for some prefixed IoU threshold, each object is set to be suppressed.

2.2.3 Inputs, outputs, and dimensions

Inputs and dimensions are essential aspects when feeding images to the model. Some models require a small fixed-size resolution (e.g., 300x300), which sometimes leads to a loss of information. At the same time, others make use of larger images (e.g., 600x1200) and keep the aspect ratio while re-scaling. This is an essential step in the pre-processing, to re-scale and crop images without distorting the information.

2.2.4 Region-based CNN models and its variants (Two-Stage detectors)

These are a family of machine learning models specially designed for object detection. The primary goal of any Region-based CNN model is to detect objects in any input image. Like in Figure 2.3, there are three objects (i) Sleepy Monk, (ii) upset Monk, and (iii) happy Monk [28]. Object detection aims to successfully locate and classify all three Monks in an image defining boundaries around them.

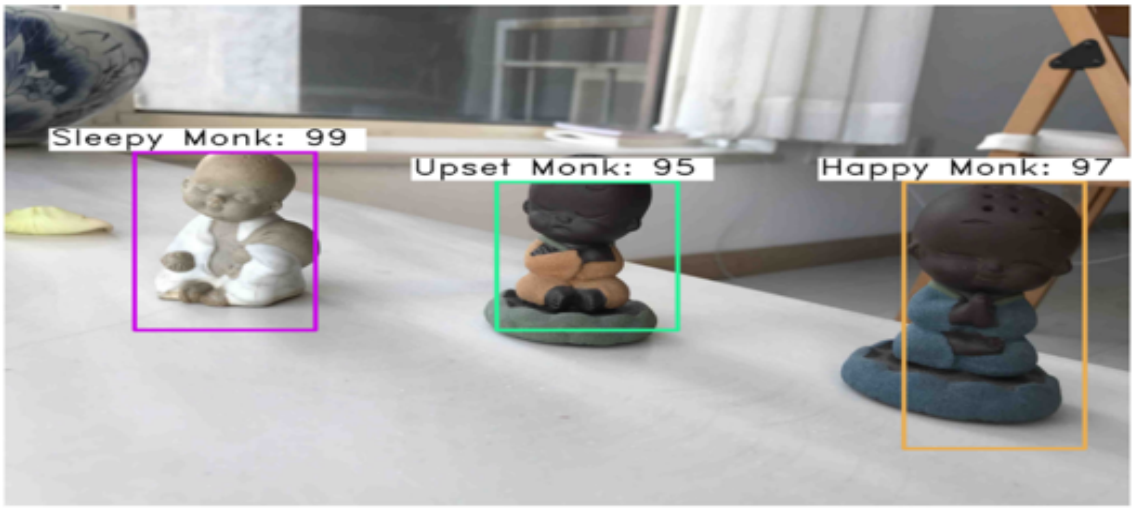


Figure 2.3: Showing the three different types of Monks in a Single image

Since CNNs are the widely used models to complete the tasks of image processing, there are some popular algorithms used in object detection and they are based on RCNN, Fast RCNN, Faster RCNN, Region-based fully convolutional network (RFCN), and Mask RCNN.

(i) R-CNN: A R-CNN is simply an extension of a CNN with a focus on object detection, while "normal" CNNs are usually used for image classification. The model consists of two separate tasks that are localization and classification. The key concept behind the R-CNN is region proposals used to localize objects within an image along with the image classification and its system overview is shown in Figure 2.4 [29].

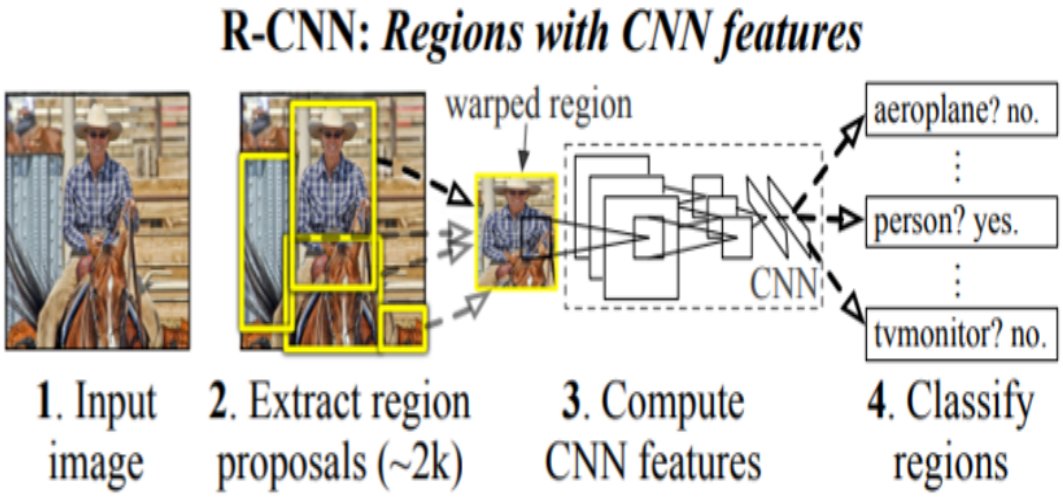


Figure 2.4: Object detection system overview. The system (1) takes an input image, (2) extracts around 2000 bottom-up region proposals, (3) computes features for each proposal using a CNN, and then (4) classifies each region using class-specific linear SVMs. R-CNN achieves a mean average precision (mAP) of 53.7% on PASCAL VOC 2010.

Figure 2.4 shows the procedures of an R-CNN while detecting an object using it. Using the R-CNN within an image, the regions of interest are extracted using the region extraction algorithm. The number of regions can be extended to 2000. For each region of interest, the model manages the size to be fitted for CNN, where CNN computes the features of the region. Support vector Machine (SVM) classifiers classify the objects present in the region. The following tasks are performed by R-CNN:

Selective Search algorithms:

These algorithms are a basic phenomenon of object localization. Using sliding filters of different sizes on the image to extract the object from the image can be one approach that we call an exhaustive search approach. As the number of filters or windows will increase, the computation effort will increase in an exhaustive search approach.

After localization in object detection, there are three processes from which an extracted object will go.

- Warping
- Extracting features with a CNN
- Classification

Warping:

After the selection of the region, the image with regions goes through a CNN where the CNN model extracts the objects from the region. Since the size of the image should be fixed according to the capacity of CNN, one requires some time or most of the time to reshape the image. In basic R-CNN, one wraps the region into 227 x 227 x 3 size images.

Extract objects with a CNN :

A wrapped input for CNN will be processed to extract the object of size 4096 dimensions.

Classification:

The basic R-CNN consists of an SVM classifier to segregate different objects into their class. The whole process architecture of R-CNN can be represented as shown in Figure 2.5 [28, 29].

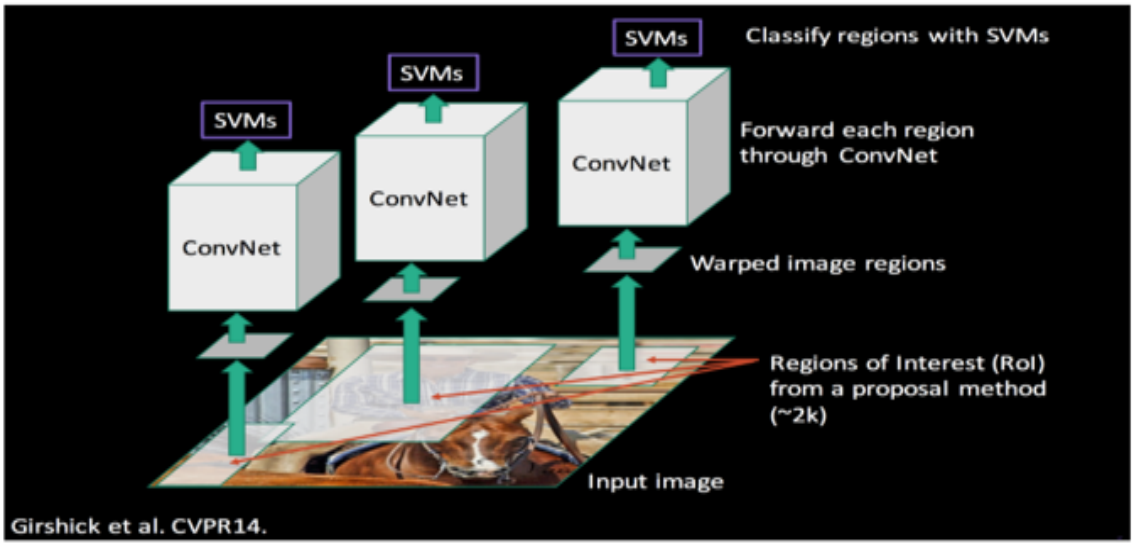


Figure 2.5: The architecture of R-CNN

At the end of the model, the boundary box regressor works for defining objects in the image by covering the image with the rectangle.

(ii) Fast R-CNN:

Due to the 2000 regions that must be calculated to finish the procedure and the CNN that each area goes through in order to extract the feature, basic R-CNN is quite sluggish during training and testing.

SPPNet (Spatial Pyramid Pooling Network) serves as a model for Fast R-CNN. SPP-Net turns the entire image on the feature map at once, as seen in Figure 2.6 below, rather than working on the 2000 regions to make them into feature maps [28, 30].

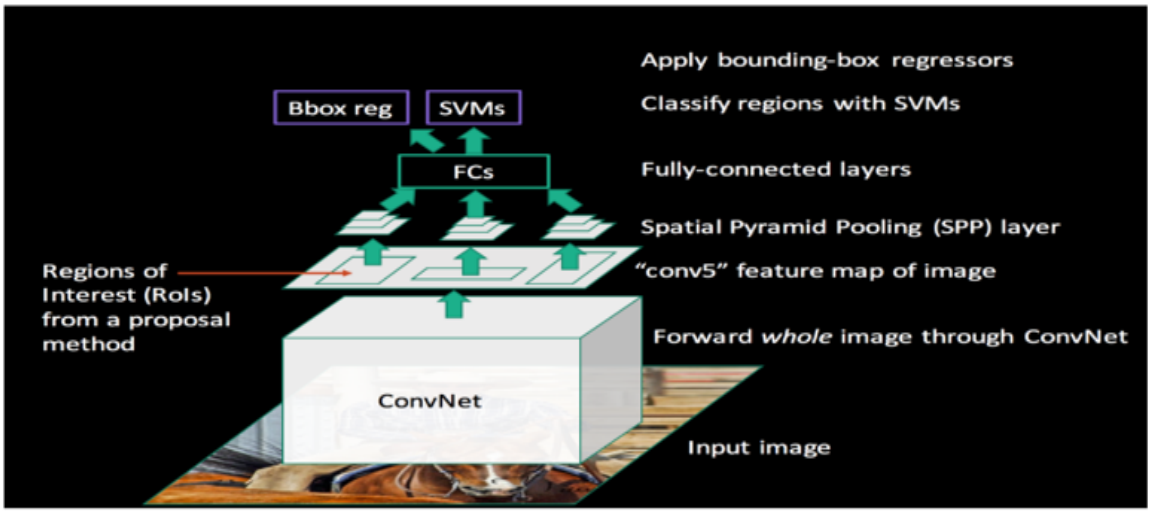


Figure 2.6: Shows Architecture of SPPNet

SPPNet uses a maximum pooling layer to extract the most highlighted color from a pixel matrix which causes the wrapping of the ROI below the image to represent the whole image as a feature map. Then this map is passed to a fully connected network and uses an SVM for classification and a linear regressor for the bounding box.

In Fast-R-CNN instead of performing maximum pooling, we perform ROI pooling for utilizing a single feature map for all the regions. This warps ROIs into one single layer; the ROI (Region of Interest) pooling layer uses max pooling to convert the features as shown in Figure 2.7 [28, 31]. It is shown in this Figure that instead of generating layers in a pyramid shape, it only generates one layer (ROI pooling). Since max pooling is still working and is present here, Fast R-CNN is considered to be an upgrade of the SPPNet.

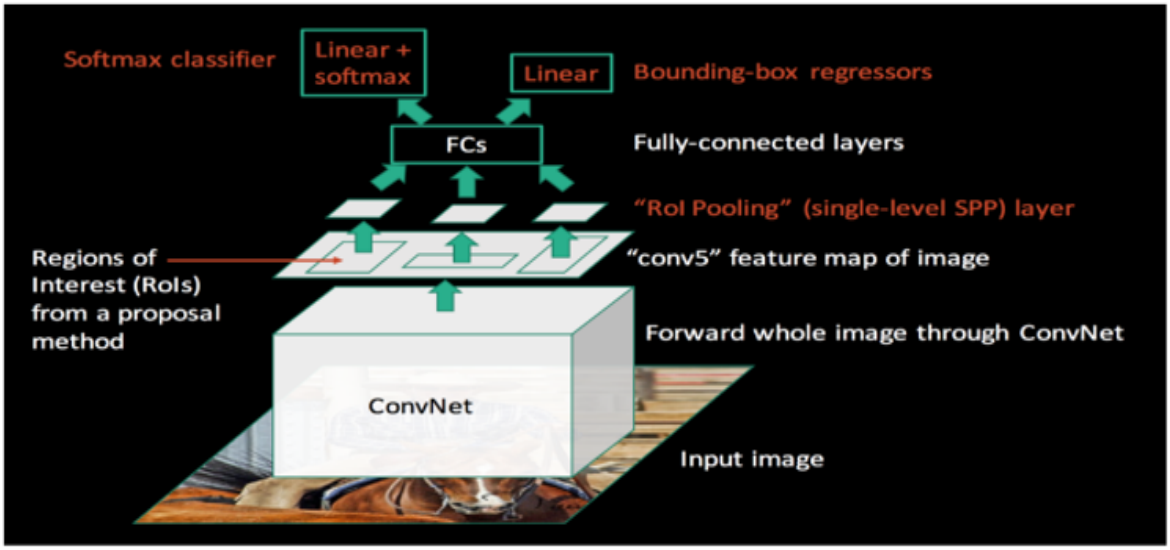


Figure 2.7: Showing Architecture of Fast R-CNN

Finally, a fully connected network for classification using softmax and linear regression is shown in Figure 2.8 [31]. The bounding box is further refined with linear regression. Fast R-CNN is faster than SPPNet.

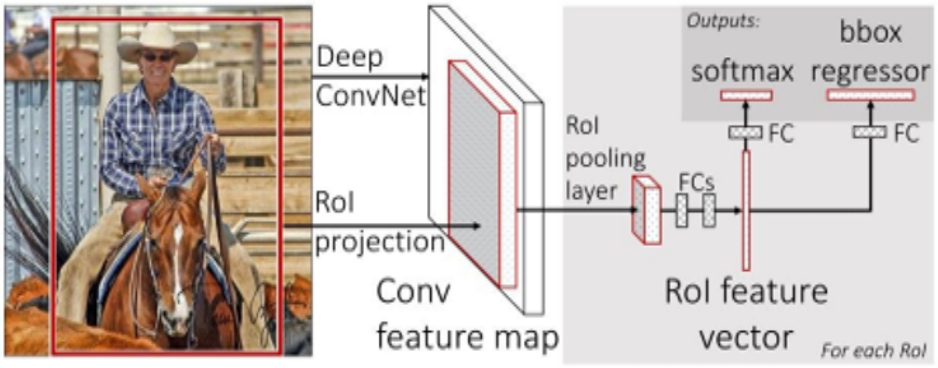


Figure 2.8: Fast R-CNN architecture. An input image and multiple regions of interest (ROIs) are input into a fully convolutional network. Each RoI is pooled into a fixed-size feature map and then mapped to a feature vector by fully connected layers (FCs). The network has two output vectors per RoI: softmax probabilities and per-class bounding-box regression offsets. The architecture is trained end-to-end with a multi-task loss

(iii) Faster R-CNN

For the region proposals in both SPPNet and Fast R-CNN, there are no methods for choosing ROIs. Faster R-CNN uses a region proposal network (RPN) to create the sets of regions which makes the process faster than Fast R-CNN. The only distinction between Fast R-CNN and Faster R-CNN is this. Figure 2.9 [28, 4] illustrates how faster R-CNN has an additional CNN for acquiring the regional proposition, which we refer to as an RPN.

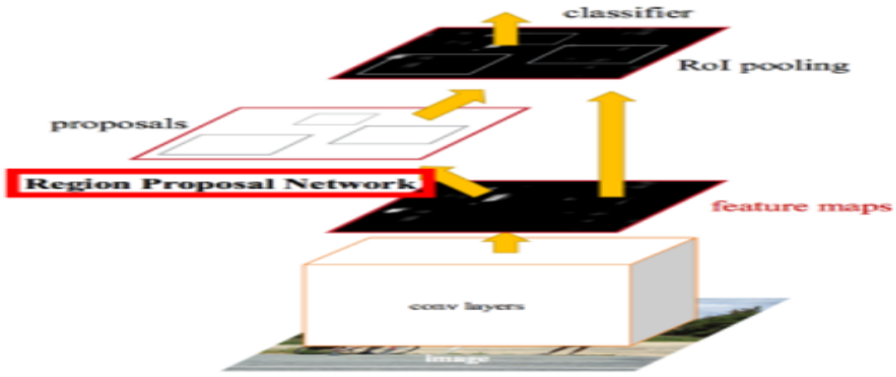


Figure 2.9: Showing Faster R-CNN uses an extra CNN layer for Region Proposal Network

An RPN is a fully convolutional network that simultaneously predicts object bounds and objectness scores at each position. The RPN is fully trained to produce superior region proposals, which Fast R-CNN uses for detection.

The proposal network in the training region uses the feature map as input and produces region proposals. These proposals go to the ROI pooling layer for further procedure and are sent to the classifier (Figure 2.9). The network can accurately and quickly predict the locations of different objects.

Faster R-CNN makes use of a region proposal network (RPN) to extract regions of interest (RoI) as shown in Figure 2.10 [32]. It uses a CNN-based region proposal method where the region proposal network takes the output from the CNN backbone and slides 3x3 filters over the feature maps to make region proposals using a convolutional network. Softmax with cross-entropy is used as the classification loss function for the region proposal network in Faster R-CNN, and a bounding box regressor func-

tion is used with smooth L1 loss. The total loss of the RPN is classification loss and localization loss combined also called multi-task loss. [31]

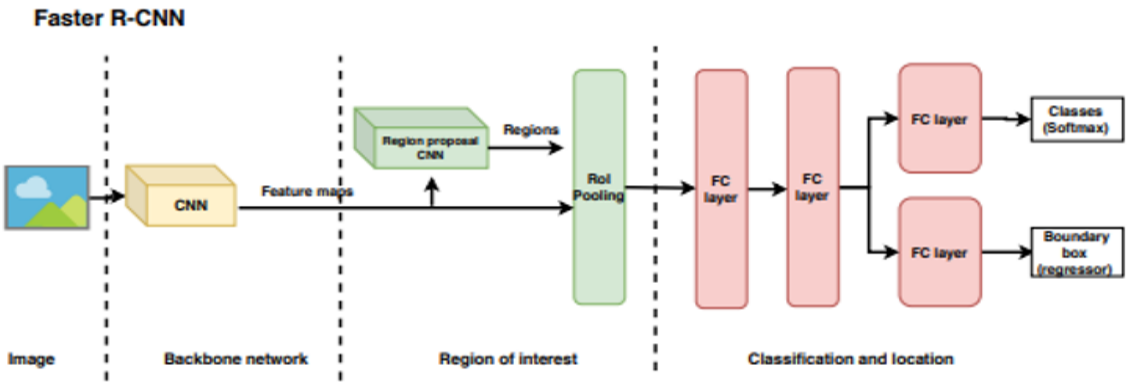


Figure 2.10: Showing the Region Proposal Network (RPN) to extract Regions of Interest (RoI) for Faster R-CNN Architecture

The comparison of important features of R-CNN, Fast R-CNN, and Faster R-CNN is presented in Table 2.1 [28].

Table 2.1: Showing the comparison of different features of R-CNN, Fast R-CNN, and Faster R-CNN

	R-CNN	Fast R-CNN	Faster R-CNN
region proposals method	Selective search	Selective search	Region proposal network
Prediction timing	$40 - 50$ sec	2 seconds	0.2 seconds
computation	High computation time	High computation time	Low computation time
The mAP on Pascal VOC 2007 test dataset(%)	58.5	G6.9 (when trained with VOC 2007 only) 70.0 (when trained with VOC 2007 and 2012 both)	69.9 (when trained with VOC 2007 only)

It is clearly seen how the above models in Table 2.1 differ from each other and how

pooling methods and region proposal methods can make changes in object detection prediction time. It can also make the process faster and even improve the average precision (mAp) from R-CNN to Faster R-CNNN.

(iv) RFCN (Region-based fully convolutional network)

Region-based Fully Convolutional Network (RFCN), is a region proposal detector as shown in Figure 2.11 [15] It is very similar to Faster RCNN but requires less effort to investigate each region of interest (RoI). It makes use of a region proposal network, like RCNN (RPN). The primary difference is that it employs a position-sensitive ROI pool, which generates position-sensitive score maps by computing individual regions of the object on each of nine feature maps ($K \times K = 3 \times 3$). Each region (location) has a prediction for how likely it is to be the real region. The class score is then calculated by averaging the results from all 9 regions. To determine the class probability, the resulting score map is subjected to Softmax. Boundary box regression is used to determine the object's location.

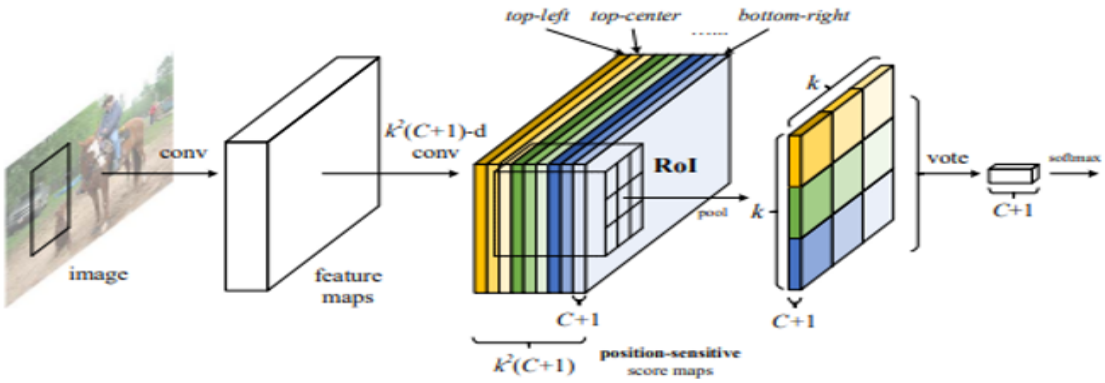


Figure 2.11: Key idea of R-FCN for object detection. In this illustration, there are $K \times K = 3 \times 3$ position-sensitive score maps generated by a fully convolutional network. For each of the $k \times k$ bins in an RoI, pooling is only performed on one of the k 2 maps (marked by different colors).

The identical feature map seen in Figure 2.11 is subjected to a second convolutional filter. These generate a $4-x-k-x-k$ map (4 positions, where k is the filter's size), which is then applied to the position-based RoI pool to determine a k-by-k boundary box array. The average of the array is calculated to determine the final border box.

RFCN is fully convolutional and maximizes the shared computation hence RFCN is much faster than Faster RCNN and achieves comparable performance [33].

(v) Mask R-CNN

Mask R-CNN is developed on top of Faster R-CNN, a region-based CNN. It is state-of-the-art in terms of image segmentation and instant segmentation. The technique of dividing a digital image into several parts is called image segmentation (sets of pixels, also known as image objects). This segmentation is used to locate objects and boundaries (lines and curves etc.). Two types of image segmentation fall under Mask R-CNN.

(i) Semantic Segmentation :

This classifies each pixel into a fixed set of categories without differentiating object instances. It is also known as background segmentation because it separates the subjects of the images from the background (See Figure 2.12) [34, 35].

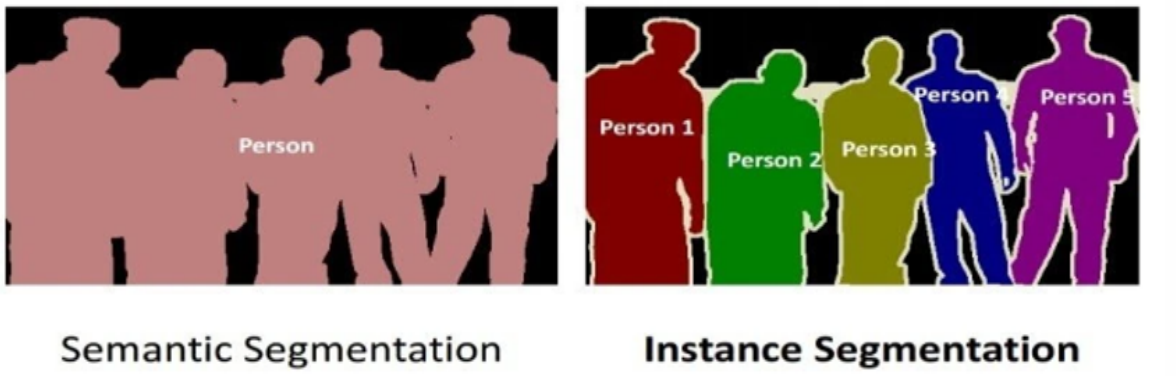


Figure 2.12: Difference between Semantic Segmentation and Instance Segmentation

(ii) Instance Segmentation :

This is also known as Instance Recognition as it deals with the correct detection of all objects in an image while also precisely segmenting each instance. It is therefore the combination of object detection, object localization, and object classification. As

seen in Figure 2.12, all objects are persons but this segmentation process shows each person as a single entity [34, 36].

Using this instance segmentation, the Mask R-CNN framework is exhibited in Figure 2.13 [37]. The essential component of Fast/Faster R-CNN that is lacking from Mask R-CNN is the pixel-to-pixel alignment. A class label and a bounding-box offset are the two outputs that Mask R-CNN, an extension of Faster R-CNN, provides for each object. The object mask is then output by a third branch that Mask R-CNN adds. The extraction of a considerably more precise spatial arrangement of an object is necessary for this additional mask output, which differs from the class and box outputs. It operates by simultaneously adding to Faster R-current CNN's branch for bounding box recognition a branch for predicting an object mask (Region of Interest).

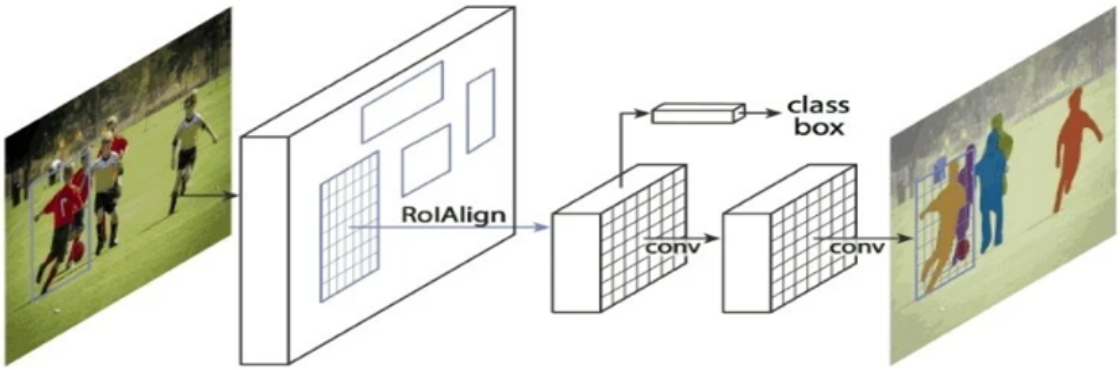


Figure 2.13: The Mask R-CNN framework for instance segmentation

Mask R-CNN is simple to train and outperforms all existing, single-model entries on every task. The method is very efficient and adds only a small overhead to Faster R-CNN. It is easy to generalize to other tasks.

2.2.5 Single stage detectors

In spite of significant progress made in region-based detectors (two-stage detectors), they are computationally expensive for both mobile and wearable devices which have limited storage and computational capability. Instead of trying to optimize the individual components of a complex region-based pipeline, researchers have begun to

develop unified detection strategies like single-stage detectors.

In single-stage detectors, both object classification and bounding-box regression are done directly in one shot without using pre-generated region proposals and thus saving a lot of computing time. Hence these methods are much faster than two-stage detectors for both mobile and non-mobile applications and good for real-time applications.

The advantages of single-stage detectors are as follows:

(i) Unified pipelines refer broadly to architectures that directly predict class probabilities and bounding box offsets from full images with a single feed-forward CNN network in a monolithic setting that does not involve region proposal generation or post-classification.

(ii) The approach is simple and elegant because it completely eliminates region proposal generation and subsequent pixel or feature resampling stages, encapsulating all computation in a single network.

(iii) Since the whole detection pipeline is a single network, it can be optimized end-to-end directly on detection performance.

Single-shot Multi-box Detector (SSD) and improved SSD

SSD methodology:

- Liu et al. in 2016 [5] proposed a Single Shot Multi-Box Detector (SSD) using convolution features layers to the end of the truncated base network (VGG-16).
- The architecture of SSD is shown in Figure 2.14 [5], where the layers decrease in size progressively and allow predictions of detections at multiple scales.
- An input image passes through all the convolution layers to generate multiple convolutional feature maps of different sizes.
- A convolution filter of size say, 3x3 is slid over each feature map to create anchor boxes at different scales and aspect ratios. By using filters of different sizes both large and small objects may be detected.

- To reduce duplicates covering the same position, non-max suppression, and Hard Negative Mining (HNM) techniques are used to provide the final detection.
- During training time, all the predicted boxes are matched with the ground truth based on IoU > 0.5 which are considered positives and others are negatives.
- Multi-Box means that both Ground truth and predicted box are in this box.
- The total loss of SSD is calculated as (classification loss+α localization loss)/(Positive bounding boxes) where α is the weight for localization loss. [5]

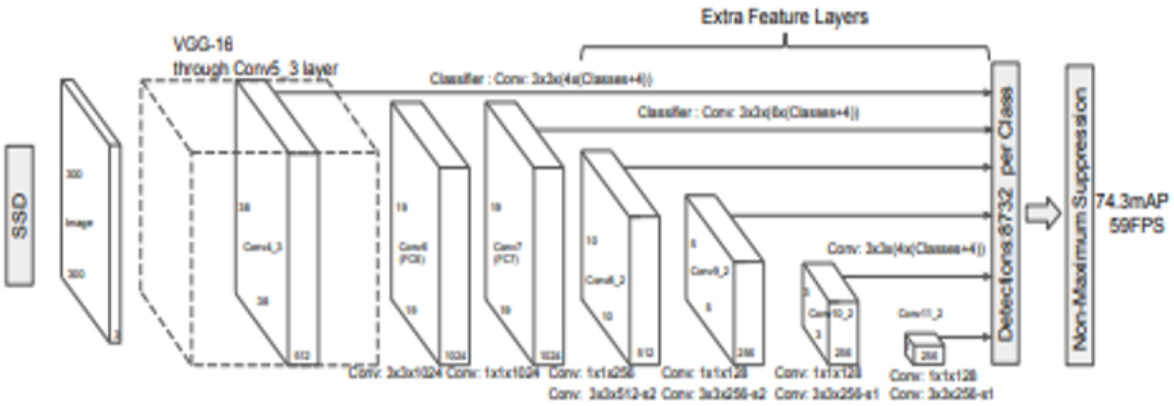


Figure 2.14: The Architecture of Single-Shot Multi-Box Detector with both classes/bounding boxes together

- SSD predicts both bounding boxes and class probabilities in one shot which makes it faster than all region-based algorithms discussed above and its accuracy is comparable to Faster R-CNN [4].

Improved SSD methods:

(i) Single shot multi-box detector (SSMBD) SSD algorithm is not appropriate to detect tiny objects. To overcome this problem, Kumar et al. 2020 proposed an improved Single shot multi-box detector (SSMBD) algorithm for real-time, where they increased the classification accuracy of detecting objects while keeping the speed constant. The authors proposed improved model architecture as shown in Figure 2.15 [38].

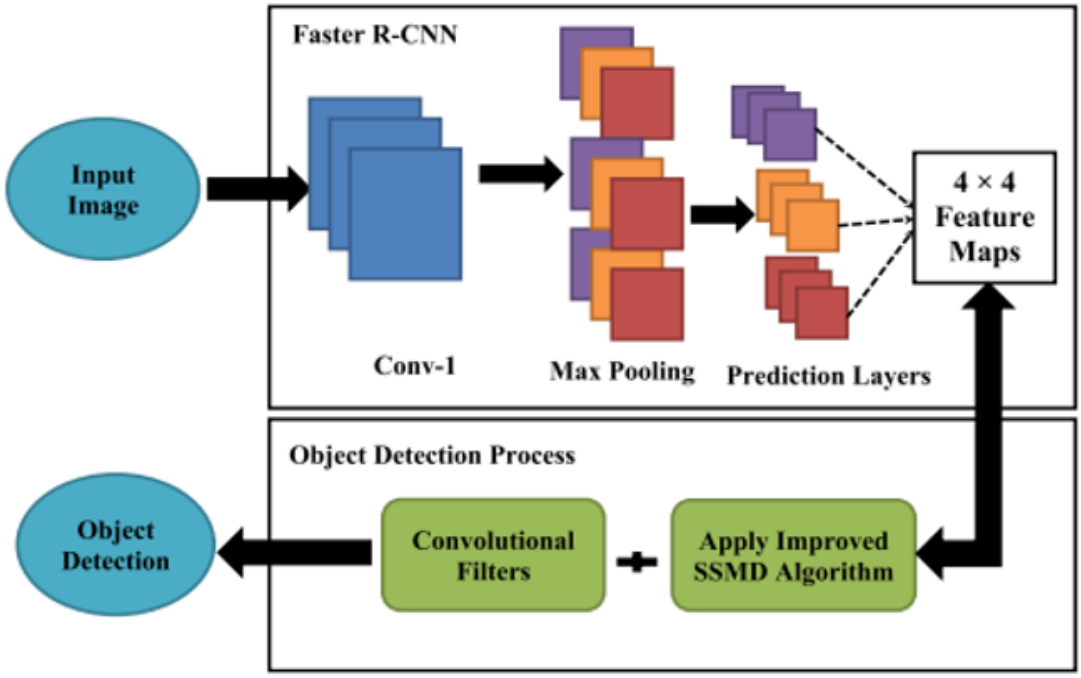


Figure 2.15: The Architecture of improved SSMB model proposed by Kumar et al. 2020

The model uses depth-wise separable convolution and spatial separable convolutions in their convolutional layers, thus using a combination of multilayer of CNN. It has main two phases : (i) feature maps extraction of spatial dimensions by using resolution multiplier (upper diagram) (ii) object detection process performed by applying the small convolution filters and by using the best aspect ratio values (lower diagram). The multilayer CNN uses a larger number of default boxes and results in more accurate detection of objects including small tiny objects as compared to the standard SSD [5].

(ii) DenseNet and feature fusion Single shot multi-box detector (DF-SSD) To overcome the lack of feature complimentary between the feature layers of SSD and the weak detection ability of SSD for small objects, Zhai et al. 2020 proposed an improved SSD algorithm based on Dense Convolutional Network (DenseNet) and feature fusion, which is called DF-SSD [39]. They used the DenseNet-S-32-1 network to enhance the feature extraction ability of the model, whereas the original SSD work utilized the backbone network of VGG-16. The authors proposed model architecture

as shown in Figure 2.16 [39].

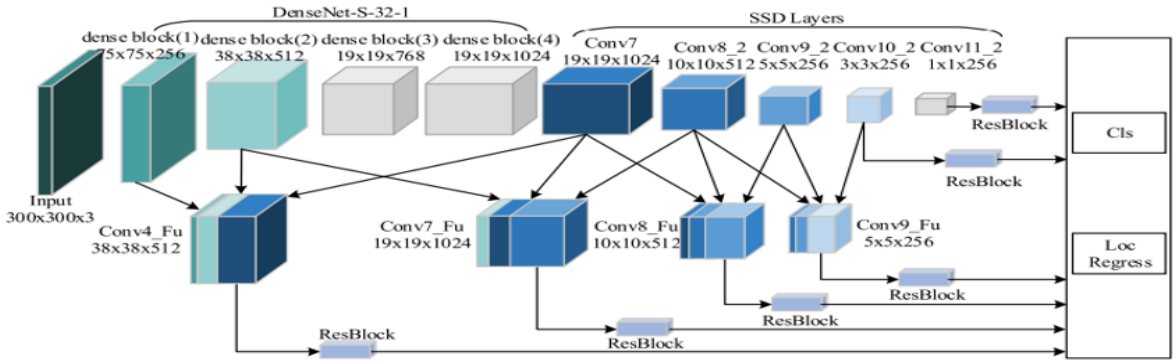


Figure 2.16: The architecture of DF-SSD with DenseNet-S-32-1 network as proposed by Zhai et al. 2020

As evident from the above figure, ConvX_Fu blocks are the feature fusion modules and the purple modules are the residual prediction blocks (ResBlock) proposed by the authors for the object detection on multi-scale feature maps to further improve the model performance. They have shown improved performance by 3.1% over standard SSD [5]. DF-SSD uses only $\frac{1}{2}$ parameters to SSD and injects more semantic information to introduce the advanced detection effects on small objects and other related specific objects.

2.2.6 Yolo (You look only once) algorithms: Yolo, Yolov2, and Yolov3

Another important development in Single stage detectors is the YOLO which is an abbreviation for the term 'You Only Look Once'. Its algorithm can detect and recognize various objects in a picture. This employs CNN layers to detect objects in real-time. As the name suggests, the algorithm requires only a single forward propagation through a neural network to detect objects. This works using the following three techniques:

- Residual blocks
- Bounding box regression
- Intersection Over Union (IOU)

Residual blocks

First, the image is divided into various grids. Each equal grid has a dimension of $S \times S$. Every grid cell will detect objects that appear within them. For example, if an object center appears within a certain grid cell, then this cell will be responsible for detecting it.

Bounding box regression

A bounding box is an outline that highlights an object in an image. Every bounding box in the image consists of the following attributes:

- Width (bw)
- Height (bh)
- Class (for example, person, apple, car, traffic light, etc.). This is represented by the letter c.
- Bounding box center (bx,by)

As an example, the bounding box and its attributes are shown in Figure 2.17 [40]. The bounding box has been represented by a yellow outline.

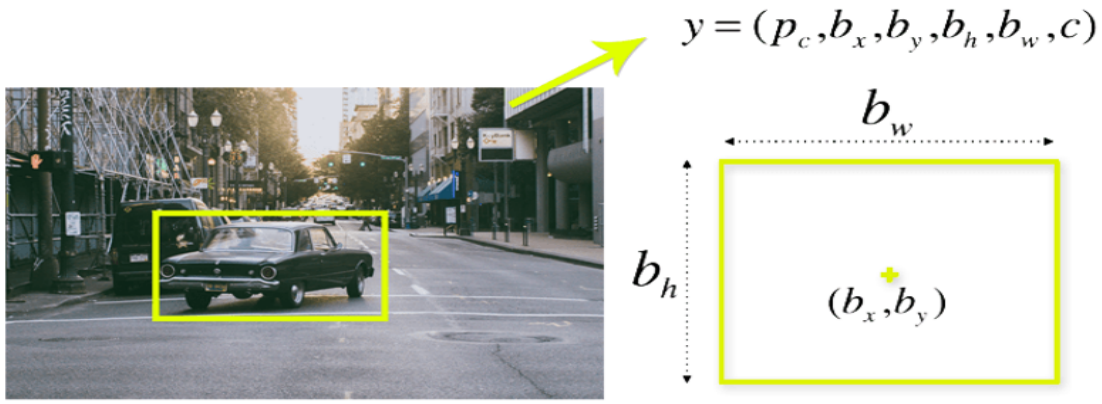


Figure 2.17: Representation of the bounding box in the Yolo algorithm

YOLO uses a single bounding box regression to predict the height, width, center, and class of objects. In the image, Pc represents the probability of an object appearing in the bounding box. To forecast the height, width, center, and class of objects, YOLO uses a single bounding box regression.

Intersection over union (IOU)

The concept of intersection over union (IOU) illustrates how boxes overlap in object detection. IOU is used by YOLO to create an output box that properly surrounds the items. The bounding boxes and their confidence scores are predicted by each grid cell. If the anticipated and real bounding boxes are identical, the IOU is 1. This approach removes bounding boxes that are not the same size as the actual box.

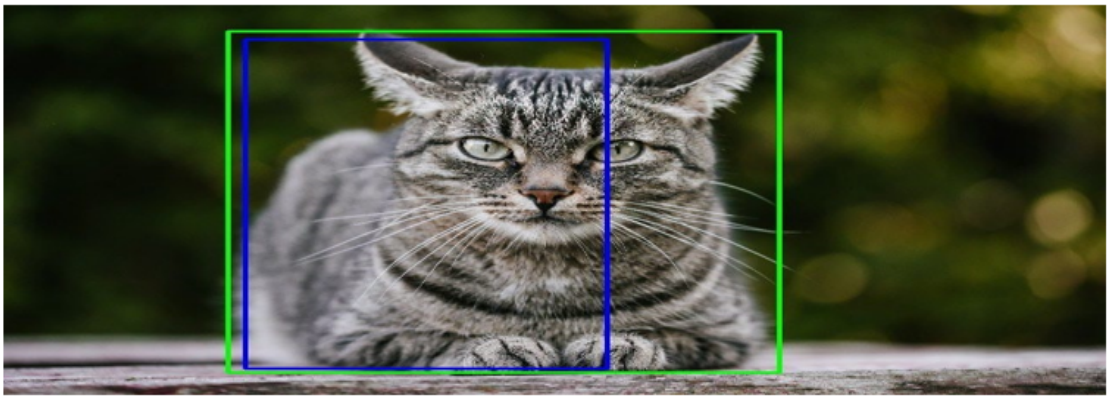


Figure 2.18: The concept of intersection over union (IOU) of real box (green) and predicted box (blue)

In the image of Figure 2.18 [40], there are two bounding boxes, one in green and the other one in blue. The blue box is the predicted box while the green box is the real box. YOLO ensures that the two bounding boxes are nearly or equal.

Combination of the three techniques

The following image in Figure 2.19 [40] shows how the three techniques (residual grid blocks, bounding box regression, and IOU) are applied to produce the final detection results in the YOLO algorithm.

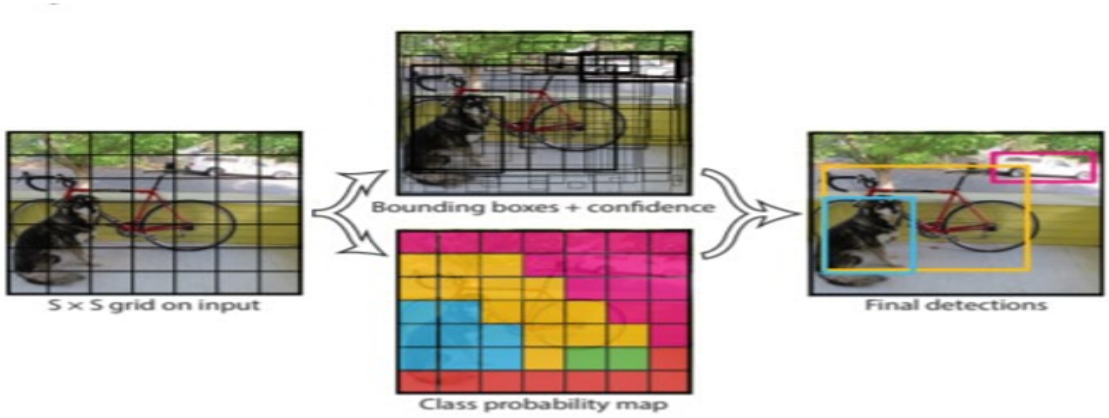


Figure 2.19: The representation of all three techniques grouped to detect the final objects

First, the image is divided into grid cells. Each grid cell forecasts B bounding boxes and provides their confidence scores. The cells predict the class probabilities to establish the class of each object.

For example, in Figure 2.19, we can notice at least three classes of objects: a car, a dog, and a bicycle. All the predictions are made simultaneously using a single convolutional neural network. This Yolo technique provides improved detection results compared to other object detection techniques such as Fast R-CNN [31], Faster R-CNN [4], and SSD [5]. The details of the development of various versions of the YOLO algorithm (YOLO, YOLOv2, and Yolov3) are given below.

Firstly, Redmon et al. 2016 proposed a new YOLO algorithm for object detection, which achieved double mean average precision (mAP) in real-time detectors [6]. It uses softmax to the class scores and takes the class with the maximum score to be the class of the object contained in the bounding box.

Then, Redmon and Farhadi 2017 [7] proposed the improved algorithm YOLO9000 which was better and faster than YOLO. However, it was further optimized by Wei et al. 2017 [8] to increase the average accuracy rate of the detection network and called YOLOV2.

Redmond and Farhadi, 2018 [9] provided the third version of YOLO V3, which is as

accurate as SSD [5] but three times faster (YOLOv3 source code:[1]).

This new Yolo V3 uses the features learned by a deep convolution neural network to detect objects in real-time. This uses a variant of Darknet, which initially had a 53-layer network trained on Imagenet. For the detection task, 53 more layers are stacked onto it, giving us a 106-layer fully convolutional underlying architecture for YOLO v3 as shown in Figure 2.20 [41].

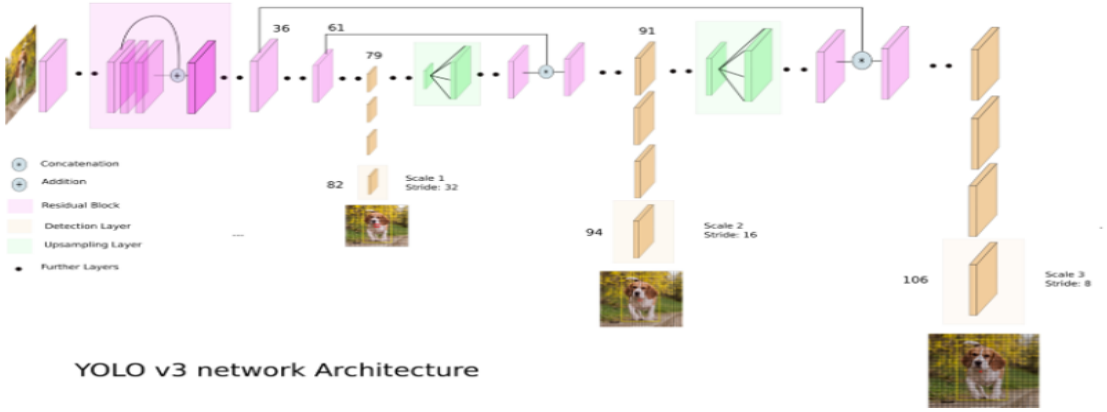


Figure 2.20: Detailed three features of yolov3 model with Darknet backbone network

YOLO V3 predicts boxes at three different scales, also called multi-scale prediction. For each scale, three anchor boxes are used, so YOLO V3 uses nine anchor boxes. These anchor boxes are best approximated using k-means clustering on the data set to output approximated anchors.

The first detection on scale one is made by the 82nd layer and is used to detect small objects. The second detection is made by the 94th layer and is used to detect medium objects. Finally, detections on scale three are made by the 106th layer and are used to detect large objects [41].

The objectness score for each bounding box using logistic regression is one of the bounding boxes prior overlaps a growth truth object by more than any other bounding box prior. If the bounding box prior is not the best but does overlap a ground truth object by more than some threshold (value .5), we accept the prediction.

[1] `https://pjreddie.com/darknet/yolo/`

YOLO V3 uses independent logistic classifiers and binary cross-entropy loss for classifications. For localization loss, it uses the sum of squared error, and for object scores, it uses logistic regression. [41] [9]. The advantages of YOLO V3 algorithm are :

- **Speed:** The algorithm improves detection speed by predicting objects in real-time.
- **High accuracy:** It is a predictive technique that provides accurate results with minimal background errors.
- **Learning capabilities:** The algorithm has excellent learning capabilities that enable it to learn the representations of objects and apply them in object detection.

2.3 Open-source frameworks for object detection

The important part of model detectors is their training through open-source frameworks. Various types of frameworks are available for its use [20]. However, important and popular ones are outlined here.

2.3.1 TensorFlow

Google Team developed the TensorFlow software, an open-source deep learning framework. This software library defines, trains, and deploys machine learning models. A nice tour of TensorFlow is written by Goldsborough, 2016, in his research paper [17]. He has also compared it with other frameworks such as Thea, Torch, and Caffe.

TensorFlow supports Python and R languages and uses dataflow graphs to process data which helps see data flow through neural networks. It provides TensorBoard for data visualization as well.

2.3.2 PyTorch

PyTorch software is developed by the Facebook AI Research lab (FAIR) team, and its source code is given on GitHub[2]. It is based on Torch, another deep-learning framework based on Lua. It supports Python and CUDA, along with C/C++ libraries for processing. Other advantages are:

(I) Provides flexibility and speed due to its hybrid frontend

(II) Enables "torch distributed" back end

(III) Deep integration with Python libraries

2.3.3 Microsoft CNTK

The Microsoft research team develops this CNTK software framework. It builds a neural network as a series of computational steps via a direct graph and supports interfaces such as Python and C++. The source code is given in GitHub[3].

CNTK is designed for speed and efficiency but has less community support. It can be used for CNN and RNN-type neural networks to handle images and speech recognition.

2.3.4 Keras

This is a high-level deep learning API developed by Google for implementing neural networks. It is written in Python as a frontend and runs on top of other different back ends (such as TensorFlow and CNTK), with ease to switch from one to another. The source code is available in GitHub[4]

Keras has been embedded in TensorFlow and has inbuilt models to run faster for all types of neural networks. The research community for its use is also vast and highly developed with lots of documentation and help. This is widely used by big companies like Netflix, Yelp, Uber, and square and runs smoothly on both CPUs and GPUs.

[2] `https://github.com/pytorch/pytorch`
[3] `https://github.com/microsoft/CNTK`
[4] `https://github.com/keras-team/keras-io/blob/master/guides/sequential_model.py`

2.4 Related work

There are various types of real-time applications, such as autonomous driving, video surveillance, face identification, medical object detection, and identification of food items in Home Refrigerators. However, training object detection models on large-scale datasets remain computationally expensive and time-consuming.

Chen et al. 2019 present an efficient and distributed open-source object detection framework called SimpleDet, which enables training state-of-the-art detection models on consumer-grade hardware at a large scale [42]. This covers a wide range of models, including both high-performance and high-speed ones. SimpleDet is well-optimized for both low-precision training and distributed training and achieves 70% higher throughput for the Mask R-CNN detector compared to other existing frameworks. It was concluded that this framework would help users design and help new detection systems more efficiently.

Tan et al. 2021 used SSD and Yolov3 detectors for real-time pill sample identification and their associated performance in hospital pharmacy [43]. The values of mean average precision (mAP) and Frame per second (FPS) were computed in both cases. Although the mAP of Yolov3 was slightly lower than SSD, it has a significant advantage in terms of detection speed. Yolov3 performed better when tasked with hard pill sample detections and was overall found to be more suitable for deployment in a hospital environment.

Viraktamath and Neelopant 2021 [44] studied the comparison of Yolo and SSD techniques to recognize and locate cases in computerized images and recordings for semantic artifacts of a particular class such as persons, structures, or vehicles in automated objects and observations. They concluded that despite a low mAP, Yolo has an appropriate mAP to be used in real-time applications and has a much faster speed than SSD. They suggested that Yolo is much easier and simpler to use in real-time than other classifier algorithms.
Bathija and Sharma 2019 [45] used Yolo and SORT to track the visual object (vehicle or pedestrian) detection in consecutive video frames and connect them over time. Yolo detects the objects, and the SORT algorithm does tracking.

Pachon et al. 2018 [46] studied "home refrigerator product system through a region-based CNN network." They used pattern recognition by the Region-based Convolutional Neural Network (R-CNN). Five types of products (classes) were used (butter, juice, milk, sauce, and soda) as objects of interest. A Graphical User Interface (GUI) alerted the user if any product was undetected.

Kirti Agarwal 2018 [3] studied "object detection in Refrigerators using Tensorflow" using region-based detectors (Faster R-CNN, R-FCN) and single shot detectors (SSD). Her mean Average Precision (mAP) performance was highest for faster R-CNN, followed by R-FCN, and lowest for SSD. However, SSD results were more immediate than region-based detectors, including faster R-CNN. Her results were limited to a small number of classes (only 8).

Though useful for home object detection in refrigerators, both studies of Pachon et al. [46] and Kirti Agarwal 2018 [3] were limited to a small number of classes and applied to only larger models. They can not run on smaller devices/computers like Raspberry Pi, and hence automation of object detection is not possible.

Summary of Region-based, SSD and Yolov3 detectors

One of the factors for the tremendous successes in generic object detection is due to development of better detection frameworks, both region-based (R-CNN [29], Fast R-CNN [31], Faster R-CNN [4], Mask R-CNN [37]) and one-state detectors (YOLOv3 [9], SSD [5]).

The development of these frameworks has been discussed in detail in a recent survey, "Deep Learning for Generic Object Detection: A Survey," presented by Liu et al. 2018 [47]. The goal of this survey is to provide recent comprehensive achievements in the field of deep learning techniques. The pictorial presentation of these leading frameworks of two-stage and single-stage detectors is shown in Figure 2.21 [47].

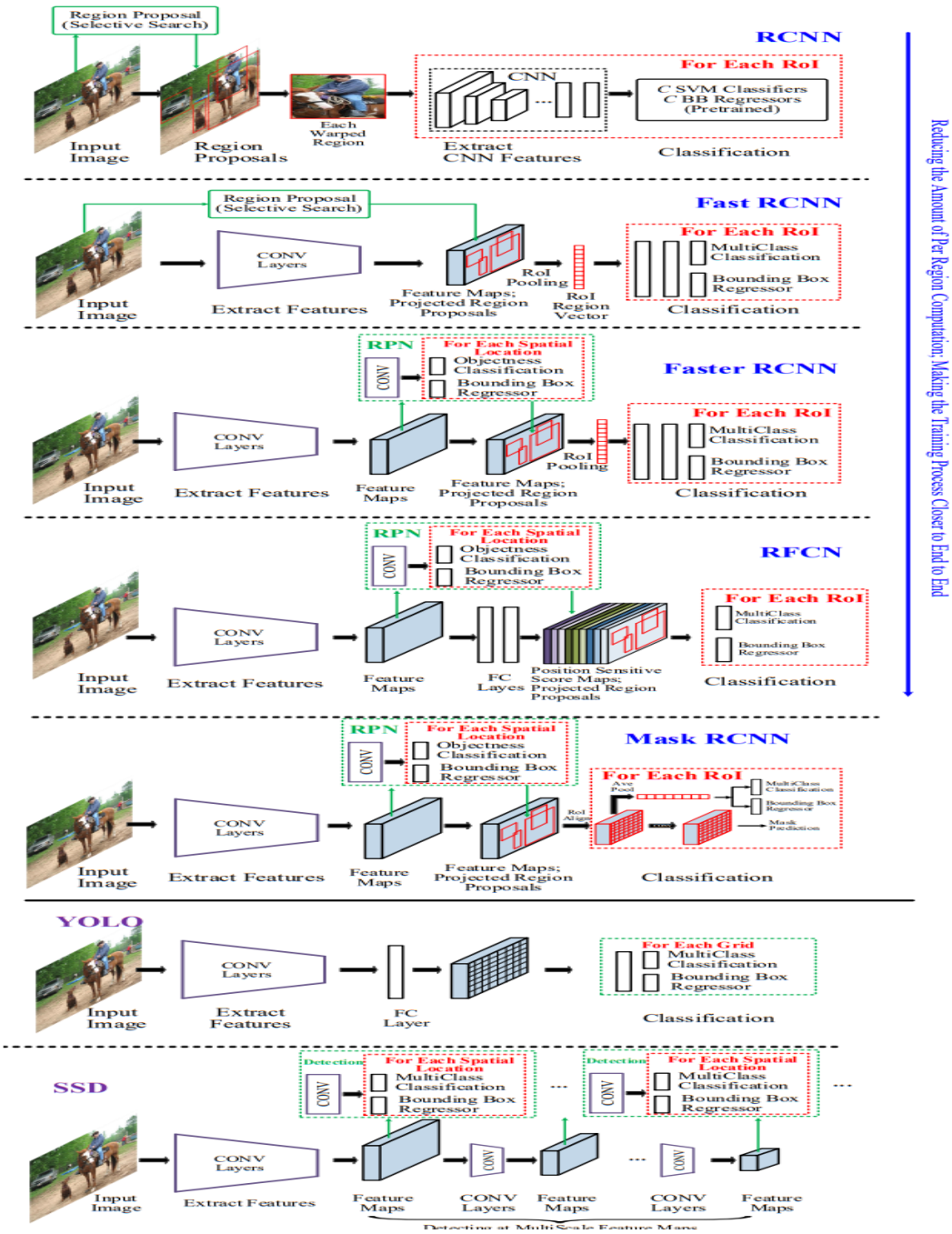


Figure 2.21: Summary of leading frameworks of both two-stage and single-stage detectors

Region-based detectors have the highest accuracy but are too computationally intensive for embedded or real-time systems.

One-stage detectors have the potential to be faster and simpler but have not yet reached the accuracy of region-based detectors as suggested by Viraktamath et al. 2021 [44].

Even though SSD [5] and Yolov3 [9] are faster. However, they are still limitations for the larger model applications, and they are not very well suited for small-scale models to run on mobile or wearable devices; hence automation of object detection in some of the applications is not possible using these detectors.

More recent state-of-the-art detectors such as EfficientDet [11], and Yolov5s [14] are found to be more mobile user-friendly. These detectors can run small in size, give better accuracy, and are good to run on small computers such as raspberry Pi. They are also suitable for automation in real-time applications and are utilized in our research project and discussed in detail in the following chapters.

Chapter 3

Fast and efficient object detection techniques for mobile devices

The recent advent of new modern single-stage state-of-the-art detectors such as EfficientDet [11], and Yolov5 [15, 14] are not only useful for real-time applications for the larger models but also found to be very useful for smaller models, user friendly for mobile and embedded devices. Such detectors can help in automation with good speed and accuracy of object detection in real-time applications.

In the present chapter, we will discuss the architectures of these two modern state-of-the-art models and their corresponding open-source framework used for training and validating these models for object detection.

3.1 EfficientDet1 model architecture

Recently, Tan et al. 2020 [11], the Google Brain team, published their state-of-the-art EfficientDet models using Tensor Flow Lite (TFlite) software for object detection in mobiles or embedded devices. The open-source code is available for use (Source code:[1]. The characteristic features of these models are :

1. **EfficientDet** models largely follows the one-stage detectors paradigm employing ImageNet-pretrained **EfficientNet** as the backbone.

2. One-stage detection models skip the region proposal stage present in two-stage

[1] `https://github.com/google/automl/tree/master/efficientdet`

object detection models and run detection directly over a dense sampling of locations.

They require only a single pass through the neural network and predict all the bounding boxes in one go. These models usually have faster inference (possibly at the cost of performance) and are much more suitable for mobile devices.

3. CNNs are classifier-based systems that can process input images as structured data arrays and identify patterns between them.

4. EfficientNet is a CNN architecture and scaling method that uniformly scales all dimensions of depth/width/resolution using a compound coefficient. The details of its architecture are discussed by Tan and Le 2020 and are shown in 3.1 [48].

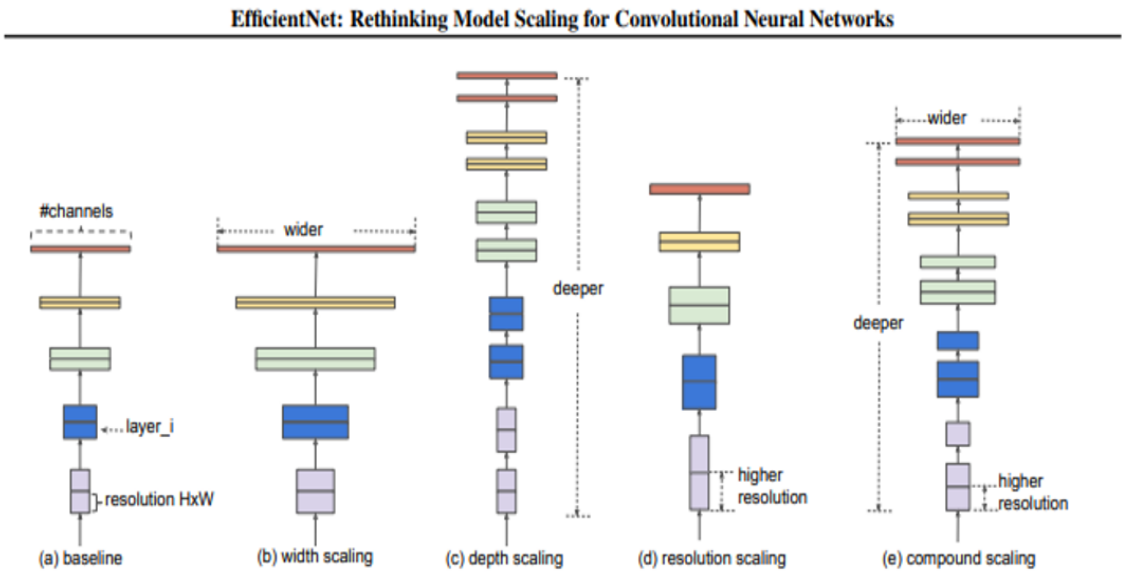


Figure 3.1: Model Scaling. (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution. (e) is our proposed compound scaling method that uniformly scales all three dimensions with a fixed ratio.

As shown in Figure, the authors conducted a thorough analysis of model baseline scaling and showed that performance could be improved by carefully balancing network depth, width, and resolution. Based on this fact, they suggest a new scaling technique that employs a straightforward but incredibly potent compound coefficient

to scale all depth, breadth, and resolution parameters equally. They use scaling up MobileNets and ResNet to show the effectiveness of this strategy, and its performance results are presented in Figure 3.2 [48].

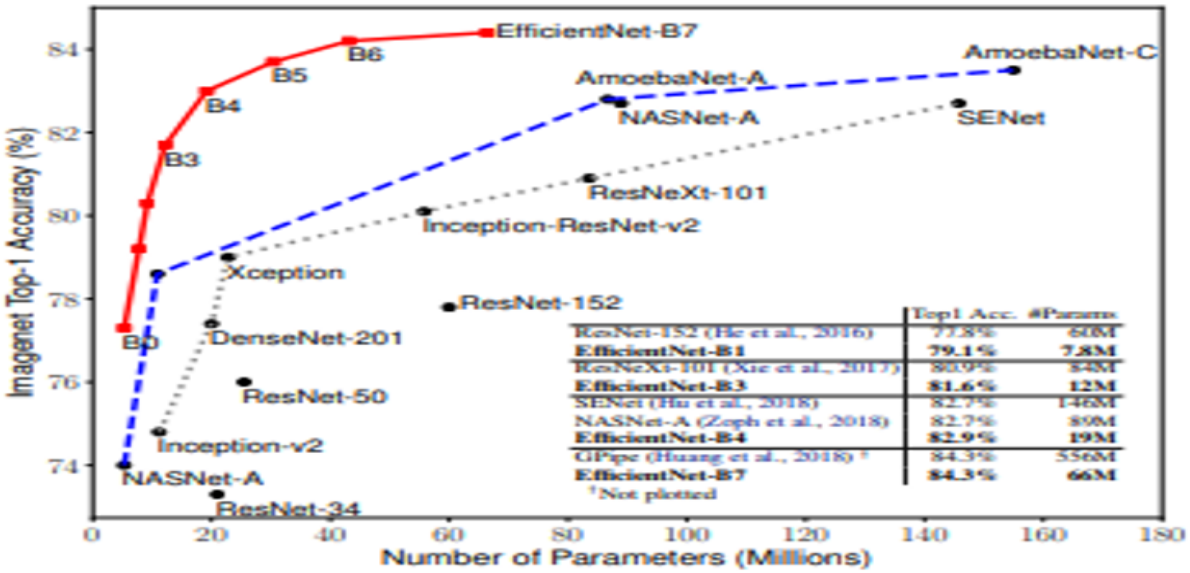


Figure 3.2: Model Size vs. ImageNet Accuracy. All numbers are for single-crop, single-model. Their EfficientNets significantly outperform other CNNs. In particular, EfficientNet-B7 achieves new state-of-the-art 84.3% top-1 accuracy.

It is visible that the accuracy of all EfficientNet [B0-B7] detectors is much better than that of Convolutional Neural Networks (CNNs).

5. The **EfficientDet** models architecture utilizes several key optimization and backbone network (**EfficientNet**) tweaks, such as the use of a **BiFPN**, and a compound scaling method that uniformly scales the resolution, depth, and width for all backbones, feature networks, and box/class prediction networks at the same time. These models' architecture is presented in Figure 3.3 [11].

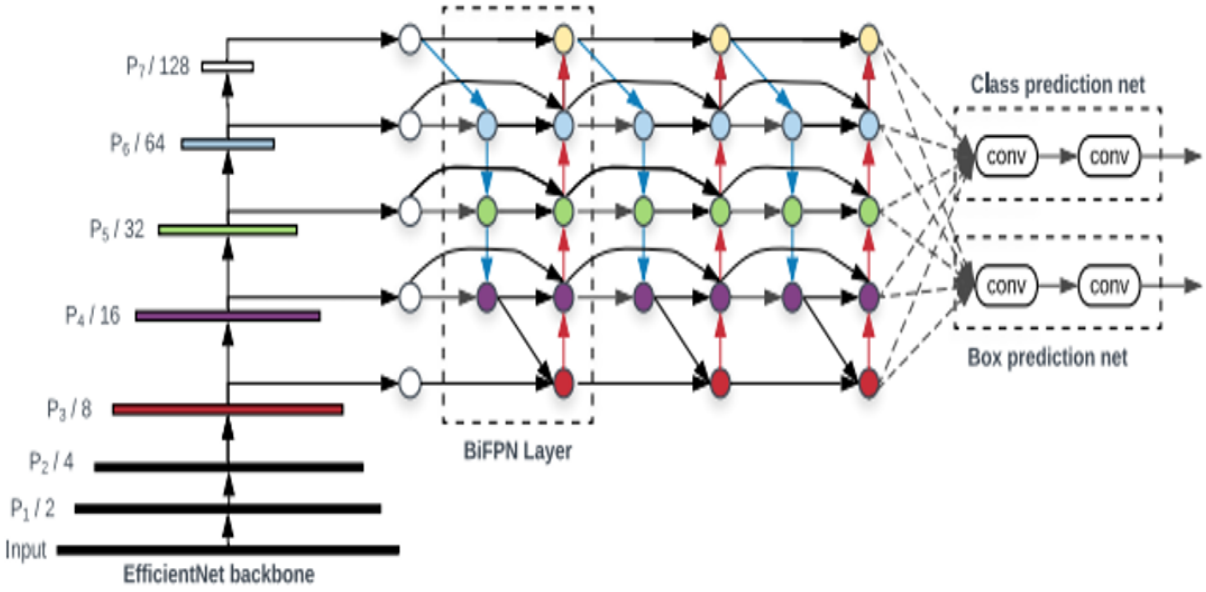


Figure 3.3: EfficientDet architecture It employs EfficientNet as the backbone network, BiFPN as the feature network, and a shared class/box prediction network.

Different EfficientDet models [0-7] were designed using different input sizes, backbone network, BiFPN number of channels and layers, and class/box net layers as shown in Table 3.1 [11].

Table 3.1: Showing the Different Models with Different Input Sizes, Backbone Network, Number of Channels and Layers, Box/Class Layers

	Input size R_{input}	Backbone Network	BiFPN #channels W_{bifpn}	BiFPN #layers D_{bifpn}	Box/class #layers D_{class}
D0 ($\phi = 0$)	512	B0	64	3	3
D1 ($\phi = 1$)	640	B1	88	4	3
D2 ($\phi = 2$)	768	B2	112	5	3
D3 ($\phi = 3$)	896	B3	160	6	4
D4 ($\phi = 4$)	1024	B4	224	7	4
D5 ($\phi = 5$)	1280	B5	288	7	4
D6 ($\phi = 6$)	1280	B6	384	8	5
D7 ($\phi = 7$)	1536	B6	384	8	5

6. A **BiFPN**, or Weighted Bi-directional Feature Pyramid Network (FPN), is a type of feature pyramid network that allows easy and fast multi-scale feature

fusion. The advantage of using BiFPN over other Feature networks (FPN [49], PANet [50], NAS-FPN [51]) is illustrated in Figure 3.4 [11].

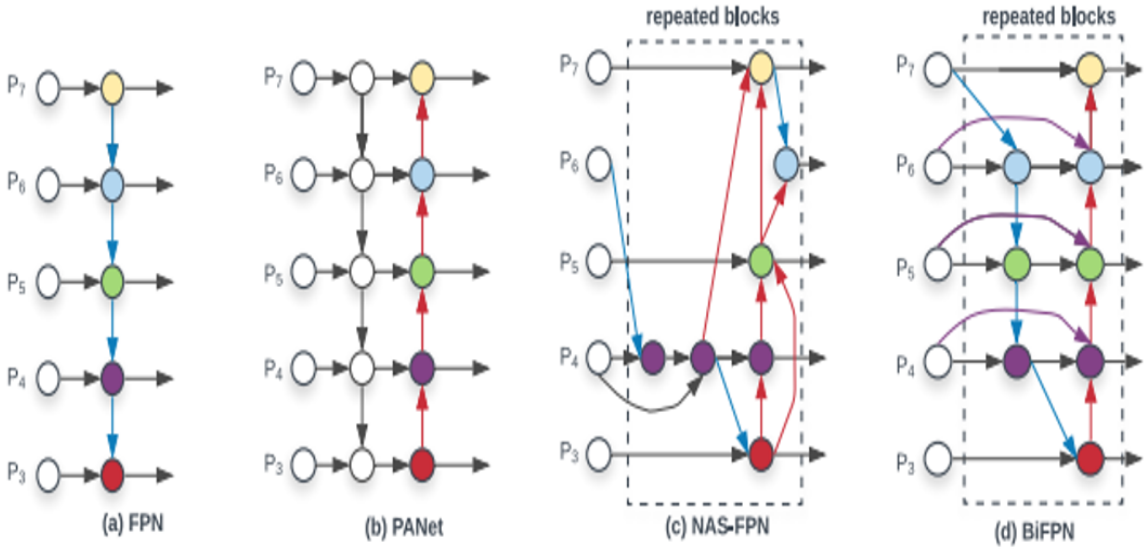


Figure 3.4: Feature network design – (a) FPN introduces a top-down pathway to fuse multi-scale features from level 3 to 7 (P3 - P7); (b) PANet adds bottom-up pathway on top of FPN; (c) NAS-FPN use neural architecture search to find an irregular feature network topology and then repeatedly apply the same block; (d) is our BiFPN with better accuracy and efficiency trade-offs.

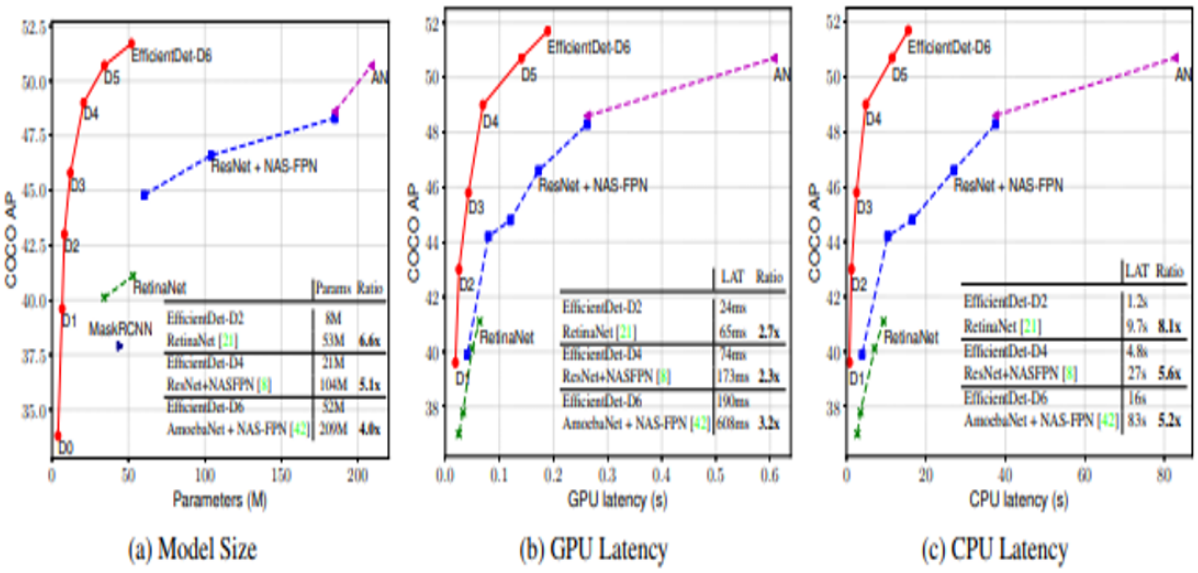
7. **A Feature pyramid network (FPN)** is a feature extractor that takes a single-scale image of arbitrary size as input and outputs proportionally sized feature maps at multiple levels in a fully convolutional fashion. This process is independent of the backbone convolutional architectures [49].

8. **Path aggregation network (PANet)** allows the feature fusion to flow backward and forward from smaller to larger resolution [50].

9. **Neural architecture search- feature pyramid architecture (NAS-FPN)** use NAS to find an irregular feature network topology and then repeatedly apply the same block to fuse features across scales [51].

10. **EfficientDet** paper uses to edit the structure of NAS-FPN to settle on the BiFPN blocks on top of each beginning of the channel that is learnable.

11. **EfficientDet for object detection**. This is evaluated using 118K training images on COCO (Common Objects in Context) 2017 detection datasets [52]. With a momentum of 0.9 and weight decay of 4e-5, each model is trained using the SGD (Stochastic gradient descent) optimizer. In the first training period, the learning rate is linearly increased from 0 to 0.16 and then annealed down using the cosine decay method. The comparison of model size, GPU delay, and single-thread CPU latency is shown in Figure 3.5 (a,b,c) respectively [11].



(a) Model Size (b) GPU Latency (c) CPU Latency

Figure 3.5: Model size (a) and inference latency comparison (b and c) on COCO dataset – Latency is measured with the same batch size on the same machine equipped with a Titan V GPU and Xeon CPU

As seen from the Figure, the model sizes for all EfficientDet detectors are much smaller than other previous detectors, Yolov3, Faster R-CNN, and MobileNet. Also, these models are up to 3.2x quicker on GPU and 8.1x faster on CPU than earlier detectors, indicating they are also effective on real-world hardware [11].

12. The authors also suggest that EfficientDet models are quite promising for semantic segmentation [11].

13. Chen et al. 2021 suggested that model-dependent data augmentation is better than the data augmentation in the fine-tuning stage of object detectors to obtain robust and accurate object detection via Adversarial learning [53].

14. Jacob Solawetz 2020 studied comparing "Yolov3 vs. **EfficientDet** models" training period time for 100 epochs using the COCO dataset. They find that EfficienDet trains slightly faster (18.3 minutes) than **yolov3** (24.5 minutes) with their implementations [54]. This is promising because the software libraries around these models will continue to improve and this initial training time lead will drop over time as well.

15. The advantages of EfficientDet models are (i) a lesser amount of data required to generalize a new domain, (ii) the need to carefully set up the model design and tune several hyperparameters, (iii) a reduced amount of training time, (iv) smaller memory size required to store the model weights and (v) quicker predictions time for object detection in a given image. (v) model sizes being smaller, they are very suitable for mobile applications as well.

3.1.1 Open-source framework TensorFlow Lite (TFlite)

The EfficientDet models can be trained using open-source frameworks such as TensorFlow or TensorFlow Lite (TFlite) object detection API developed by the Google team. Much documentation about TensorFlow is available online for its use, and a nice description of "A tour of TensorFlow" is given by Peter Goldsborough 2016 [17]. The description of the TensorFlow Lite(TFlite) guide and its API reference is also provided by the Google team developers [19]. EfficientDet model with custom data can be trained using roboflow software by following their instructions [55].

The Key features of the TFLite model are :

(i) It optimized the five important constraints such as latency (no round trip to a server), privacy (keeps personal data on the device), connectivity (no internet is needed), size (reduced model and binary size) and power consumption (efficient detection and no network connections).

(ii) Support on multiple platforms (Android, IOS devices, embedded Linux, and microcontrollers).

(iii) Support in various languages (Python, Java, and C++)

(iv) Model optimization and hardware acceleration to obtain high performance

(v) End to End detection of multiple objects with bounding boxes

TFLite model can be generated using the following methods:

(i) Select a pre-existing TFLite model with and without metadata

(ii) Create a custom model with metadata

(iii) Use the TFLite Converter to convert the TensorFlow (TF) model into TFLite model by applying optimizations such as quantization to reduce the model size, and latency with minimal or no loss in accuracy

(iv) The process of training a TensorFlow Lite model with a customized dataset is made simpler by the TensorFlow Lite Model Maker package. The usage of transfer learning decreases the amount of training data needed and training time.

(v) The model maker library currently supports the various machine learning (ML) tasks listed in this link[2].

(vi) If the tasks are not supported, then use TensorFlow to retrain an existing TensorFlow model using transfer learning (following instructions like images, text, or voice) or train a new TensorFlow model from scratch before converting it to a TensorFlow Lite model.

Once the TFLite model is ready, it can run inference (or detection) to make predictions on any input image or a set of images from the custom dataset.

3.2 Yolov4 and Yolov5 model architecture

Soon after the discovery of state-of-the-art EfficietDet detectors by Tan et al. 2020 [11], the improved version of yolov3, called Yolov4 by Bochkovskiy et al. on April 23, 2020 [56] and Yolov5 by the Glen Jocher in May 18, 2020 [15, 14] were introduced. These models' performances were even better than the EfficientDet [0-4] models. They surpass all the benchmarks of speed and accuracy for both larger and smaller models, which are particularly useful for mobile or embedded devices (see Figure 1.2 in chapter-1). It is also noticed that all the variants of Yolov5 training are much

[2] `https://www.tensorflow.org/lite/models/modify/model_maker`

faster than EfficientDet detectors. This is a remarkable development in the history of the Yolo series of detectors.

The main difference between Yolov4 over Yolov3 is the use of CSP networks as a backbone which shows a considerable increase in the accuracy of Yolov4. Though Yolov5 performs similarly to Yolov4, it is effortless to use, train and deploy. This breakthrough in Yolov5 has made the Yolov5 more popular for object detection in real-time applications, and its performance improvements are due to the PyTorch training procedure. The source code of yolov5 is freely available for its use in GitHub[3]).

Yolov4 model architecture :

Yolov4 model architecture consists of three components developed by Bochkovskiy et al. 2020 [56] and also explained by Jacob Solawetz 2020 [57].

1. Backbone
2. Neck
3. Head

Backbone : Typically, an object detector's backbone network receives pre-training on ImageNet categorization. Pre-training refers to the network's weights being modified for the new task of object recognition even when they have already been trained to recognize important features in an image.

For the YOLOv4 object detector, the authors considered the following backbones.

1. CSPResNext50 [58]
2. CSPDarknet53 [58]
3. EfficientNet-B (0-7) [48]

The first two backbones depend on DenseNet as shown in Figure 3.6 and studied by Huang et al. 2018 [59]. As shown in Figure 3.6, the authors use a 5-layer dense block that connects each layer and its subsequent layer in a feed-forward fashion, rather than traditional convolutional networks with L layers having a connection between

[3]`https://github.com/ultralytics/yolov5`

each layer and its subsequent layer. In other words, all previous layers' feature maps are utilized as inputs for each layer, and that layer's feature maps are used as inputs for all subsequent layers.

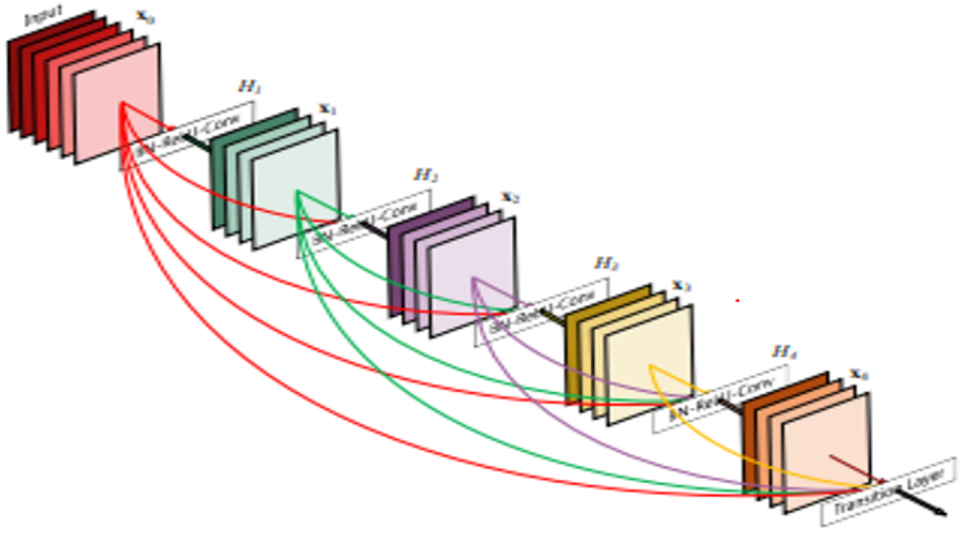


Figure 3.6: A 5-layer dense block that connects each layer and its subsequent layer in a feed-forward fashion as used by Huang et al. 2018

DenseNet was created to connect layers in convolutional neural networks with the following goals in mind: to improve feature propagation, encourage the network to reuse features, and decrease the number of network parameters; to address the vanishing gradient problem (it is challenging to backdrop loss signals through a very deep network). The denseNet architecture is illustrated in Figure 3.7(a) [58].

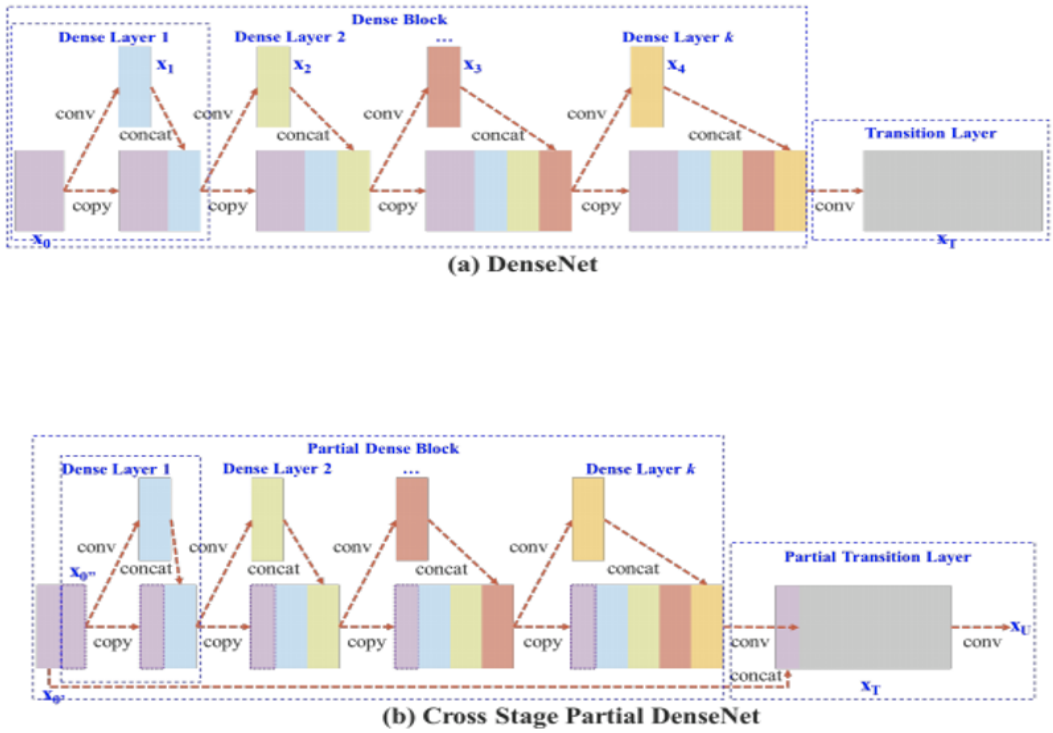


Figure 3.7: Examples of (a) DenseNet and (b) the Cross Stage Partial DenseNet (CSPDenseNet) proposed by the authors

To overcome the problem of vanishing gradient problem, Wang et al. 2019 [58] introduce the Cross Stage Partial Network (CSPNet) as shown in Figure 3.7 (b). CSPNet separates the feature map of the base layer into two parts, one part will go through a dense block and a transition layer; the other part is then combined with the transmitted feature map to the next stage. The advantage of this in two backbones CSPResNext50 and CSPDarknet53 is (i) strengthening the learning ability of a CNN, (ii) removing computational bottlenecks, and (iii) reducing the memory costs to have smaller models and achieve better accuracy.

The Google Brain team created EfficientNet to study the scaling up of the CNN network (ConvNet) by varying the input size, width scaling, depth scaling, and scaling of all these, as shown in Figure 3.1 [48]. The team found that there is an optimal point for all these scalings to outperform the other networks of comparable size on image classification. However, Yolov4 authors [56], based on their intuition and experimental results, selected the CSPDarknet53 as the final backbone network for their study.

Neck : The next step is to mix and combine the features in the convolution networks. For this, Yolov4 authors tried the various options for the neck, such as (i) FPN [49], (ii) PAN [50], (iii) NAS-FPN [51], (iv) BiFPN [11], (v) ASFF [60] and (vi) SFAM [61].

Usually, a neck comprises several bottom-up paths and several top-down paths among layers and connects only a few layers at the end of the Convolutional network. As shown in Figure 3.4 [11], the structure of various P_i layers represents a feature layer in the CSPDarknet53 backbone.

EfficientDet uses the BiFPN layers [11]. However, the authors of Yolov4 authors selected the feature layers of SPP [30], and PAN [49]. Then they modify the SAM [30] from spatial-wise attention to point-wise attention [Figure 3.8] and replace shortcut connection of PAN [50] to concatenation [Figure 3.9].

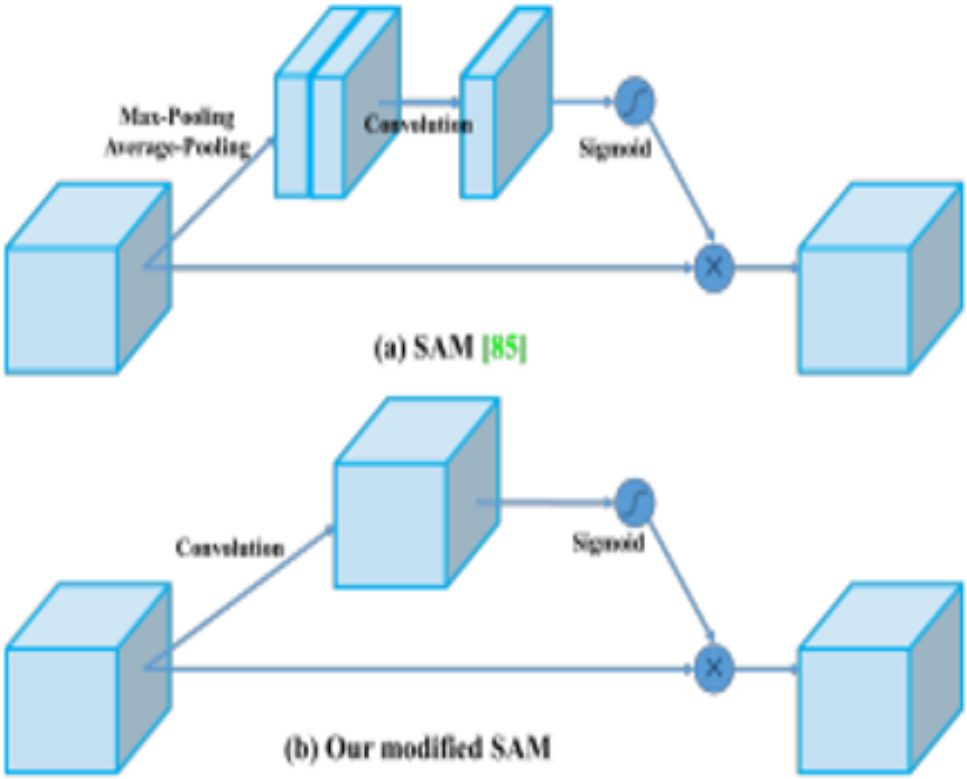


Figure 3.8: (a) original SAM and (b) modified SAM

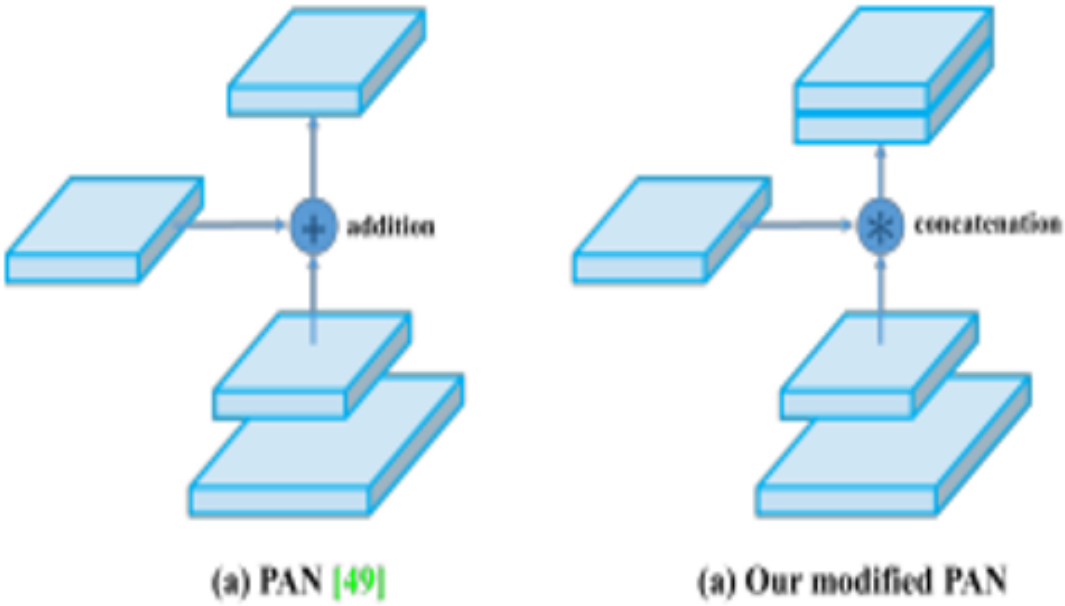


Figure 3.9: (a) original PAN (left) and (a) authors modified PAN (Right)

Head : The authors of Yolov4 employ three degrees of detection granularity and anchor-based detection, using the same head as that of Yolov3 [9] for detection.

Yolov4 also uses (i) a bag of freebies and (ii) a bag of specials.

Bag of Freebies (BoF) :

The "Bag of Freebies" used by [56] YOLOv4 authors boosts network performance without increasing production-related inference (object detection) time. The majority of the freebies in the bag are related to data augmentation. They use data augmentation to increase the size of their training set and expose the model to semantic scenarios that it would not have otherwise encountered. The computer vision community is aware of many of these techniques; YOLOv4 is only confirming their efficacy. However, their new contribution is **mosaic data augmentation**, which tiles four images together, teaching their model to detect smaller items and pay less attention to surroundings that are not immediately around the object. Another important new contribution to data augmentation is **self-Adversarial Training (SAT)**. SAT operates in 2 forward-backward stages seeks. In the 1st stage, the neural network alters the original image instead of the network weights. By changing the original

image in this way, the neural network engages in an adversarial attack against itself, creating the illusion that the desired object is not present in the image. In the second stage, the neural network is trained to recognize an object on this altered image in a conventional manner.

The YOLOv4 authors give an ablation study (i.e., removing features from your model one by one to see how much each one individually contributes to the performance) to justify the data augmentations that they have used to support the usage of data augmentations.

Bag of Specials (BoS):

The so-called "Bag of Specials" techniques used by YOLOv4 only slightly lengthen inference time but greatly improve performance, making them worthwhile.

The authors test out different activation functions. Features are changed as they move through the network using the activation functions. It can be challenging to convince the network to drive feature creations toward their optimal point when using conventional activation functions like **ReLU**. Therefore, research has been conducted to develop features that very slightly enhance this process **Mish** is an activation function designed to push signals to the left and right.

Predicted boundary boxes are separated out by the authors using **DIoU NMS**. It would be helpful to quickly select the optimal bounding box because the network might predict several bounding boxes for a single object.

Cross mini-Batch Normalization **(CmBN)**, which can be executed on any GPU, is the method the authors utilize for batch normalization. Multiple GPUs working together are required for many batch normalization approaches.

DropBlock regularisation is used in YOLOv4. Sections of the image are hidden from the first layer in DropBlock. This is a method for making the network discover features it might not have otherwise. As an example, one might picture a dog whose head is hidden behind a bush. The network ought to be able to recognize the dog's torso in addition to its head.

Finally, Yolov4 uses BoF and BoS [56] :

(i) BoF for Backbone: CutMix and Mosaic data augmentation, DropBlock regularization, Class label smoothing

(ii) BoS for Backbone: Mish activation, Cross-stage partial connections (CSP), Multi-input weighted residual connections (MiWRC)

(iii) BoF for detector: CIoU-loss, CmBN, DropBlock regularization, Mosaic data augmentation, Self-Adversarial Training, Eliminate grid sensitivity, Using multiple anchors for a single ground truth, Cosine annealing scheduler [62], Optimal hyper-parameters, Random training shapes

(iv) BoS for detector: Mish activation, SPP-block, SAM-block, PAN path-aggregation block, DIoU-NMS. In contrast to CIoU loss, which simultaneously takes into account the overlapping area, the distance between center points, and the aspect ratio, DIoU loss also considers the distance of an object's center [63]. CIoU can achieve better convergence speed and accuracy on the BBox regression problem.

The main contribution of Yolov4 authors are :

(i) It is a robust and effective object detection model, lightweight and easy to use

(ii) It uses cutting-edge Bag-of-Freebies and Bag-of-Specials items detection techniques.

(iii) Possible to train on your custom datasets, easy to run on GPU or Google Colab.

(iv) It can be trained on a darknet framework or PyTorch.

The implementation of using the darknet framework in Yolov4 is shown in Figure 3.10. The new features are Weighted-Residual-Connections (WRC), CSP, CmBN, SAT, Mish activation, Mosaic data augmentation, DropBlock regularization, and CIoU loss.

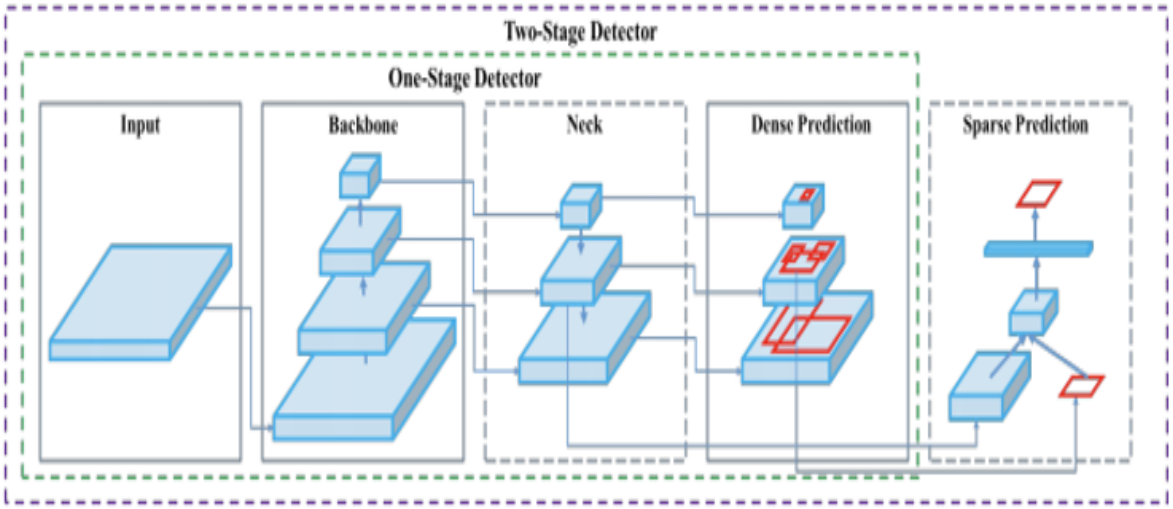


Figure 3.10: The Yolov4 Architecture

Using the MS COCO dataset [64], the experimental results using the new Yolov4 architecture show excellent performance compared to other state-of-the-art object detectors, including the EfficientDet detectors [11]. The average precision (AP) vs. frame per second (FPS) for all detectors is shown in Figure 3.11.

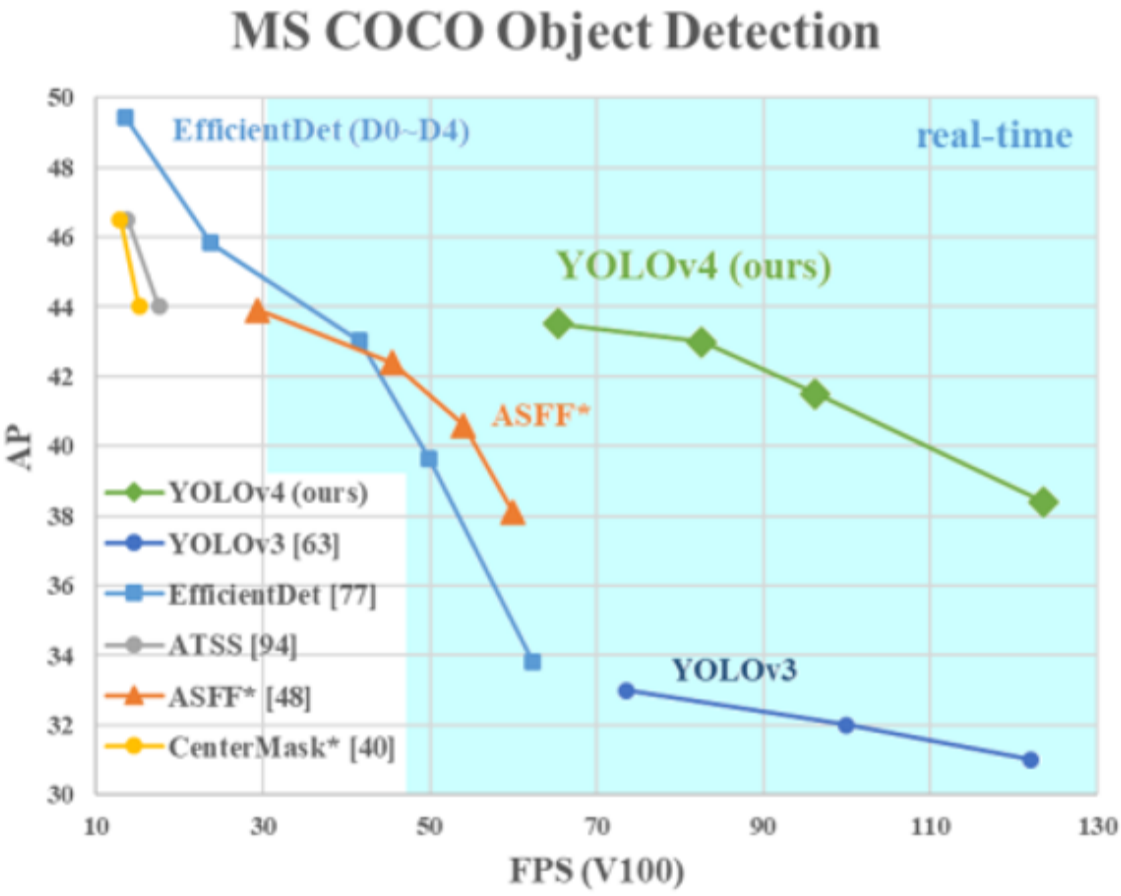


Figure 3.11: Comparison of the proposed YOLOv4 and other state-of-the-art object detectors.

It is evident from the Figure that Yolov4 runs two times faster than EfficientDet with comparable performance. It outperforms YOLOV3 performance for AP, and FPS is about 10% and 12%, respectively, as suggested by Yolov4 authors [56]. Compared to earlier object detection algorithms, which only had either excellent performance or fast inference speed, YOLOv4 has both astonishingly good performance and very high FPS. This is an outstanding achievement.

Yolov5 model Architecture :

On March 18, the Google team opened the source of their implementation of EfficientDet, and then Yolov4 was released on April 23, 2020.

Soon after the discovery of Yolov4, Glen Jocher, on May 18, 2020, introduced Yolov5 using the Pytorch framework [15, 14]. Though he did not write his official research paper for it like Yolov4 [56], he did officially release the open-source code for use on June 29, 2020, in Github[4] [15]. It is very fast and has good development in real-time object detection applications.

Model structure:

YOLOv5 derives most of its performance improvement from PyTorch training procedures, while the model architecture remains close to Yolov4 described. However, Glen Jocher [15] introduced many differences and improvements.

The Yolov5 network consists of three main pieces.

1. Backbone: CSP-Darknet53
2. Neck: SPPF, CSP-PAN
3. Head Yolov3

Training Strategies :

The procedure for the training process is as important as any factor to the end performance of an object detection system. In Yolov5, two important training procedures are as follows:

1. **Data Augmentation :** It transforms the base training data to expose the model to a wider range of semantic variation than the training set in isolation. The data loader uses three kinds of augmentations (i) Scaling, (ii) Color space adjustments, and (iii) Mosaic augmentation. The most novel is the Mosiac data augmentation which combines images into four tiles of random ratio. It is native to Yolov3 PyTorch and now in Yolov4 and Yolov5 PyTorch.
2. **Loss Calculation :** It calculates a total loss function from the GIoU [65], objectness, and class loss functions. GIoU stands for generalized intersection over Union. These functions can be carefully constructed to maximize the objective of mean average precision (mAP).

[4] `https://github.com/ultralytics/yolov5`

3. **Auto learning bounding box anchors:** The YOLOv5 network predicts bounding boxes as deviations from a set of anchor box dimensions in order to produce box predictions. If we attempt to discover an object like exceptionally tall and skinny giraffes or very wide and flat manta rays, the most extreme discrepancy in anchor boxes may occur. However, when we enter the custom data in YOLOv5, all YOLO anchor boxes are automatically learned. In Yolov3, the learning of anchor boxes uses K-means and genetic learning algorithm.

4. **Yolov5 Labeling Format :** Yolov5 uses PyTorch TXT annotation text format, which is similar to Yolov4 Darknet text format, but it contains a YAML file containing model configuration and class values for model data training.

5. **Yolov5 Labelling Tool :** There are various annotation tools available for this purpose. However, Roboflow [66] provides a Yolov5 labeling tool, and it partners with Ultralytics, the creator of Yolov5. It can work with datasets, labeling, and the active learning section of the Yolov5 Github repo.

Advantages of Yolov5 algorithm :

(i) Provides a new and very efficient PyTorch training and development framework that improves the state-of-the-art for object detectors.

(ii) Easy to install the torch and some lightweight python libraries

(iii) Inferences (detection) can be obtained through individual images, batch images, video feeds, and webcam ports.

(iv) Intuitive data file system easy to navigate during developing the algorithm

(v) Easy to run on mobile devices as Yolov5 PyTorch weights can be transported to ONXX, CoreML, and IOS platforms.

Evaluation metrics results on COCO dataset :

The metrics result on the COCO dataset [64], using the Yolov5 repository code, are computed and presented in Figure 1.2 and Table 1.1 in Chapter-1. These are published by Glen Jocher 2020 [15], and Jacob Solawetz 2020 [14]. It is evident from Figure 1.2 that the Yolov5 model's performance is much better than the EfficientDet models both in speed (ms/image) and in accuracy (mAP). Reverting the speed in ms

will give you the FPS (Frame per second), which is around 200-300 FPS for Yolov5 models.

Another important point in Figure 1.2 is that Yolov5 has 4 types of models such as (i) small (s), (ii) medium (m),(iii) large (l), and (iv) extra large (x). All four models' speed is higher, and their accuracy is progressively increasing with the increase in the size of the model from small (s) to extra large (x). Even with the smaller size of the model (Yolov5s), both speed and accuracy are better than EfficicientDet models.

Table 1.1 also shows the performance of actual numbers for these Yolov5 models [14]. It is evident that the average precision at 50% IOU, even for smaller model YoloV5s with fewer parameters (7.5M), has a very good accuracy of 55.8% (AP_50) and very good FLOPS (13.2B) to make it suitable for mobile devices and for real-time applications.

Summary of Yolov5 is as follows [67]:

(i) This is implemented in PyTorch rather than the Darknet framework as used in Yolov4. Though Darknet is an incredibly flexible research framework, but not good in production. Yolov5 being in PyTorch is easy to support, and good for development and production, as it is easily available for the broader research community. This can be deployed on mobile devices more simpler as the model can be compiled to ONNX and CoreML with ease.

(ii) Yolov5 is very fast (about 7 ms per image in Figure 1.2 in Chapter -1) giving rise to about 140 frames per second (FPS). On the contrary, Yolov4 gets only 50 FPS having converted to the same Ultralytics PyTorch library.

(iii) Another very important point is that the Model weight file size for Yolov5 is very small (about 27 megabytes) as compared to the weights file for Yolov4 (244 megabytes with Darknet structure), which is almost 90% smaller than Yolov4 and hence making it ideal for running on mobile or embedded devices.

(iv) Yolov5 accuracy is found to be acurate by the Roboflow authors [66]. The authors achieve about 0.895 mean average precision (mAP) in their tests on the "blood cell count and detection (BCCD)" dataset. Even though EfficientDet

and YOLOv4 performed similarly, it is rare to witness such broad performance gains in Yolov5 without sacrificing accuracy.

(v) Yolov5s has good speed and accuracy for object detection in real-time applications for mobile devices and is useful for automation.

3.2.1 Open-source framework PyTorch in Yolov5

Ultralytics/yolov5 has provided the PyTorch hub by Glenn Jocher [15]. The website link is[5].

This hub is useful for training the custom dataset and for the object inference settings. It contains the requirements.txt file having the Python >=3.7.0 environment, including PyTorch>=1.7. The preferred Yolov5s model can be loaded from the PyTorch hub as a "model". After training the model with the aid of custom images dataset and validation, the model weights can be used for the object inference using any desired image.

The model can be run on a single GPU through Google colab or on multiple GPUs in parallel with threaded inference. Training attributes such as image size, batch size, number of epochs, etc. can be set. Also, inference attributes such as confidence threshold, IoU threshold, classes (number of classes), max_det (maximum number of detection per image), etc can be set to the desired values. Both training and inference results are displayed on the desktop and saved in files under the runs/train folder for training and runs/detect folder for inference. Training losses and performance metrics are saved to Tensorboard and also to a log file called yolov5s_results. Yolov5s is the lightest and fastest Yolov5 model.

In the next following chapters, the implementation of these two stat-of-art detection techniques (EfficientDet1 and Yolov5s) is highlighted and their results are discussed for the application in "Home object items detection in refrigerators".

[5] `https://github.com/ultralytics/yolov5/issues/36`

Chapter 4

Implementation of EfficientDet1 and Yolov5s techniques for mobile devices

The first and most important step in implementing object detection techniques is preparing and processing an existing or custom image dataset. Then the question arises of what type of food items (called classes) will be selected and how to obtain the images of these classes for the home refrigerators. Class selection and data collection are important to increase the accuracy of object search by training the model. Varying sizes of items covering from smaller to larger items and selecting larger number classes give credibility to the object detection techniques employed.

In the present chapter, we will discuss the collection of image datasets and its preprocessing of data annotation required for the model training using open-source software such as Labellmg and Roboflow.

4.1 Data Collection and pre-processing

So far, the previous studies of Pachon et al. 2018 [1] and Kirti Agarwal 2018 [3], useful for home object detection in refrigerators, were limited to a small number of classes only. Pachon et al. 2018 utilized five classes (butter, juice, milk, sauce, and soda), whereas Agarwal used eight classes (Apple, Bell pepper, Cauliflower, Lemon, Orange, Pear, Tomato, and Turnip.)

The images for these classes can be either obtained from a camera installed in a refrigerator or for experimental purposes it can be obtained from the internet on popular websites for object detection.

In our research experiment, we selected 27 classes. A variety of items are taken into consideration such as 3 fruits (apple, orange, lemon), 4 vegetables (potato, garlic, onion, tomato), 5 drinks (chocolate_drink, coke, grape_juice, sprite, orange_juice), 8 miscellaneous (butter, eggs, sausages, cereal, noodles, paprika, potato_chips, pringles) and 7 other home object items (Cloth_opl, crackers, basket, tray, help_me-carry_opl, scrubby, spong_opl). The objective of having a large number of refrigerator items plus other home items mixed together is to create a complex image dataset and identify them through object detection techniques. Secondly, the images of all items are collected from the following different public dataset links (Kaggle.com and ieee-dataport.org) and we did not have the option to isolate them. The following image dataset links are used to make a complete image dataset for this purpose. The data format for the images is .jpg files.

Dataset-1 link:[1]

Dataset-2 link:[2]

Dataset-3 link :[3]

In order to create a large image dataset, we have to use the last data link which provides mixed items and we wanted to see if we can detect the desired object from the mixed items or not. Our image dataset consists of a total of 725 images. Out of the total images, about 70% (about 507) are kept for training the data, 20% (about 145) for validating the model, and 10% (about 73) for testing. These total images can be annotated by putting the bounding boxes and labeling each image using popular software like either labeling or Roboflow as shown in Figure 4.1.

[1] https://www.kaggle.com/datasets/kritikseth/fruit-and-vegetable-image-recognition

[2] https://www.kaggle.com/datasets/trolukovich/food11-image-dataset

[3] https://ieee-dataport.org/open-access/annotated-image-dataset-household-objects-robofeihome-team

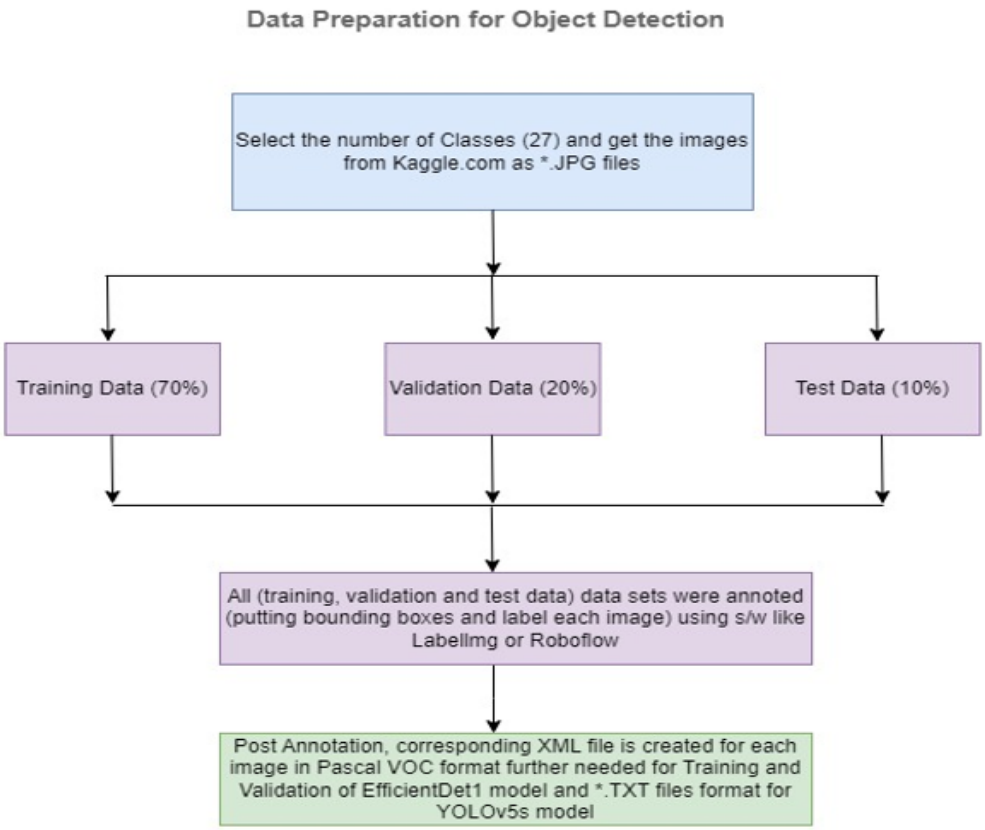


Figure 4.1: Management of image dataset and labellmg or Roboflow software

4.1.1 Data annotation tools

There are various tools available for labeling and annotating images. However, the popular ones are Labellmg and Roboflow whose details are given below.

4.1.2 Labellmg software for annotating image dataset

LabelImg is a popular barebones graphical image annotation tool that is written in Python and uses Qt for its graphical interface. The installation is relatively simple and is generally done through a command prompt/terminal. The details of the instructions for installing and labeling images for object detection with Labellmg are given by Phoebe Parker, 2022 [68]. The graphical interface looks as shown in Figure 4.2 [69]. The key buttons on the left side are self-explanatory, we can open an image, label it, go to the next image, and so on.

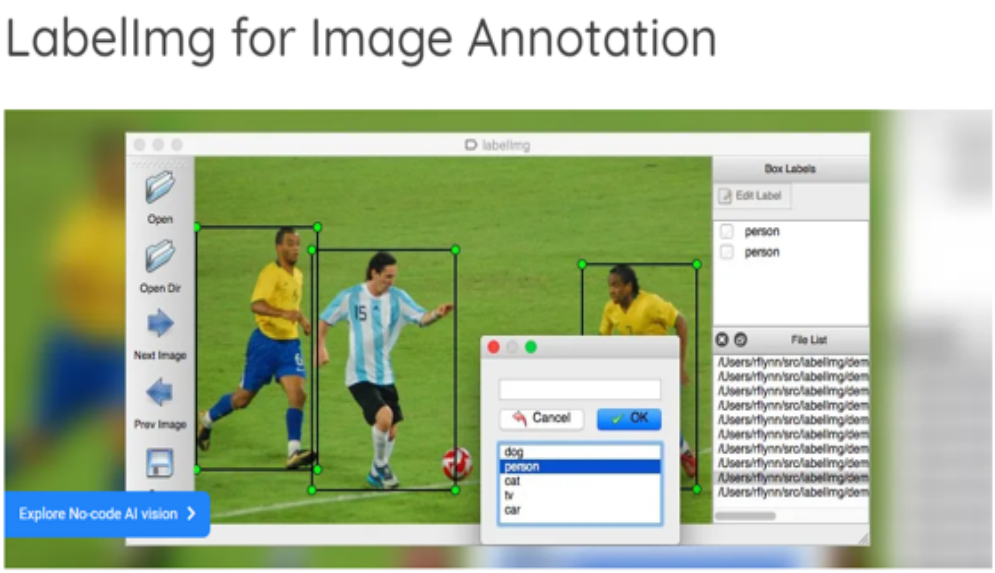


Figure 4.2: The graphical interface of Labellmg for image annotation

This tool is great for datasets under 10,000 images, as it requires a lot of manual interaction and is created to help annotate datasets for object detection models. This software annotates image by image (putting bounding boxes and labeling each item in the image) which makes it a time-consuming tool for object detection. After annotating an image, it can be converted either to XML files in the PASCAL VOC file format required for EfficientDet1 or the text files format required for Yolov5vs models for further training, validating, and testing the object detection models.

4.1.3 Roboflow software for annotating the image dataset

This is more advanced software to label and annotate large image datasets. It provides an official Python package that interfaces with the Roboflow API. The details of this software usage and its advantages are provided by Brad Dwyer [70]. It is capable of managing, pre-processing, augmenting, and versioning datasets for the object detection model datasets. Key features of Roboflow are importing and exporting image datasets into any supported formats. For annotating each image, it draws a box around the object you want to detect and selects the class label for each box you drew. Once the annotation of all the images is done, they can be converted to desired format such as XML files in PASCAL VOC required for the EfficientDet1 model and the text files required for the Yolov5s model.

In fact, the advantage of Roboflow software is that one can convert annotated datasets in many formats like Pascal VOC, Yolo, COCO, and JSON and annotated datasets can be used for further training, validation, and testing in Roboflow or can be exported for further processing to other platforms.

4.1.4 Data processing with Roboflow

Though both Labellmg and Roboflow software is suitable for the size of our image dataset (705), we preferred to use Roboflow for image annotation (putting the bounding boxes and labeling each image) for object detection. The reason for selecting Roboflow is that it is faster than Labellmg and saves time in the annotation process and is easy to import and export datasets as well. The Roboflow graphical interface used for annotation is shown in Figure 4.3.

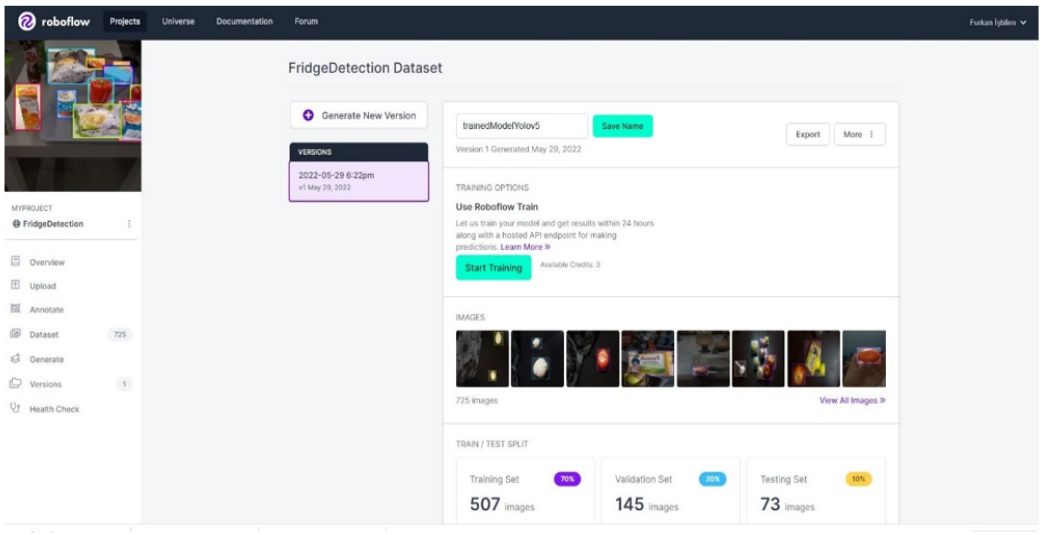


Figure 4.3: The graphical interface of Roboflow for image dataset annotation

As seen from our Roboflow diagram, the whole image dataset is divided into three parts (i) training the dataset, 507 images (20%), (ii) validating the dataset, 145 images (20%), and (iii) testing, 73 images (10%) as required for better performance of object detection techniques.

Then the same image dataset has been used for both EfficientDet1 and yolov5s models for training and validation on mobile devices for home refrigerator object detection. Note that the distribution of instances (occurrences) of all classes in the image dataset is shown in Figure 4.4. Most classes in general have more than 100 instances and are useful for generating efficient detection models.

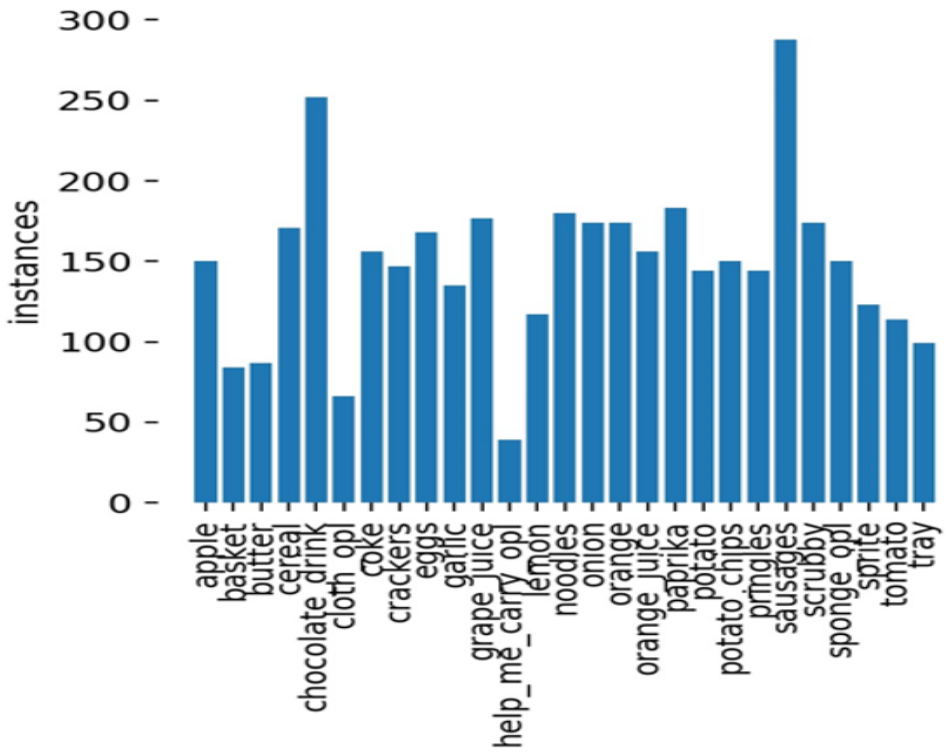


Figure 4.4: The distribution of instances of each class in the image dataset

4.2 Model Training and validation procedure for smaller models

Once the annotated image dataset is ready, then the next big step is how to train and validate the EfficientDet1 and Yolov5s smaller models on mobile devices such as Raspberry Pi. This point is discussed in the following section using open-source code frameworks.

4.2.1 EfficientDet1 using Tensorflow Lite (TFLite)

Recently, Tan et al. 2020 (Google Brain Team), not only published the state-of-the-art EfficientDet [0-4] models using the Tensor Flow Lite (TFLite) software for object detection for the use of mobiles devices, but they did provide the open-source code to use the software in GitHub for the use of research community[4] [11].

This source code is written in Python and is utilized by us for the EfficientDet1 [d1] model with our pre-processed custom annotated image dataset, to train and validate the object detection model. This uses the image dataset in XML files for each image in PASCAL VOC format and can run in Google colab notebook. This colab provides a Python programming environment equipped with GPU computing resources and is free to use it. This environment is found to be consistent with the results of EfficientDet1 authors published on the COCO dataset [11].

In Google colab, the Python source code is divided into two parts : (i) Training and validating the model code called "**Model_Maker_Object_Detection_for_tflite_model.ipynb**" and (ii) the inference (or detection) from test images called "detection-fromimages.ipynb.

Both source codes are developed and tested in Google colab. This code is found to be running successfully and the final codes are kept in the Source code area under Appendix A. The implementation details of both codes are given below.

(A) Model_Maker_Object_Detection_For_tflite_model.ipynb code :

The training and validation of a custom object detection model with a TFLite model maker consist of the following steps. It tells how to run a model on a Raspberry Pi. Transfer learning is used by the Model Maker package to streamline the process of training a TFLite model with a custom dataset. Retraining a TFlite with a custom dataset will require less training data and take less time.

1. **Install the packages:**
   Install the required packages, including the Model Maker package from GitHub and the pycocotools library needed for model evaluation

[4] `https://github.com/google/automl/tree/master/efficientdet`

2. **Load the pre-processed image dataset :**

   (a) Images in train_data (507 images in the dataset) is used to train the custom object detection model

   (b) Images in val_data (145 images in the dataset) are used to check if the model can generalize well to new images which have never been seen before.

3. **Define the name of classes :**
   Define all the 27 classes selected in the image dataset and mentioned above in the data collection section. Classes are eggs, lemon, tomato, garlic, butter, onion, potato, orange, cereal, apple, paprika, sprite, coke, noodles, chocolate_drink, potato_chips, scrubby, sponge_opl, crackers, pringles, sausages, grape_juice, orange_juice, cloth_opl, basket, help_me_carry_opl, tray.

4. **Select a model architecture :**
   Select the desired model architecture in code which is efficientdet_lite1 (EfficientDet1)

5. **Train the TensorFlow Lite (TFLite) model with training data:** Train the TensorFlow model with the training data (70% of the total dataset) using the arguments as

   (a) epochs = 50, batch_size = 4 and train_whole_model=True.

   (b) Having epochs = 50 means that it will go through the training dataset 50 times.

   (c) Batch_size = 4 means that it will take 118 steps to go through the 475 images in the training dataset.

   (d) Train_whole_model=True means that it will fine-tune the whole model instead of just training the head layer to improve accuracy. The trade-off is that it may take longer to train the model.

   (e) One can change the number of epochs and train the model again to improve validation accuracy during training and stop when the validation loss (val_loss) stops decreasing to avoid overfitting.

6. **Evaluate the model with validation data:**
   Evaluate the model with the validation data (20 % data of total images = 145

images) to learn how the model performs against new data it has never seen before.

Here the default batch size is 64, it took 4 steps to go through the total 145 images in the validation dataset. The evaluation metrics in the model are the same as the COCO dataset which is each precision value of all 27 classes, average precision (AP) at threshold (0, .5, .75), and average recall (AR) values.

7. **Export as a TFLite model:** Export as TF Lite model after training the validation of the Tensor Flow model. This will generate TF Lite format and export this quantized model in a folder you prefer.

   The default post-training quantization technique is full integer quantization. This allows the TensorFlow Lite model to be smaller, run faster on Raspberry Pi CPU, and also be compatible with the Google Coral EdgeTPU.

8. **Evaluate the TFLite model:**
   Finally, the evaluation of the TFlite model (quantized model) is done using the validation dataset again and computing the metrics parameters again to make sure accuracy and precision have not gone down much as a result of Quantization.

   Quantization helps shrink the model size by 4 times at the expense of some accuracy drop. The original TensorFlow model uses per-class non-max suppression (NMS) for post-processing, while the TF Lite model uses global NMS that's much faster but less accurate. Hence it is necessary to re-evaluate the exported TFLite model and compare its accuracy with the original TensorFlow model.

   The final evaluation model is saved on the local computer with your desired file name such as **raspi1.tflite**.

   Now we are ready to test any image from the test dataset (10% of total images = 65) or any other image from the Refrigerator or internet using this final quantized model. The results are discussed in the next chapter using this model.

(B) detectionfromimages.ipynb code :

Once the quantized model (raspi1.tflite) is ready, the weights of this model can be utilized to infer (or detect) any image from the test image dataset or any other image to see the performance of the object detection model. This Python source code can be run on Google colab and the steps are as follows:

1. Install the necessary packages

   Install tflite-model-maker, tflite-support, opencv-Python-headless packages

2. Load any input image for inference

   Test any image from the test dataset

3. Load the TFLite quantized model

   This model file is raspi1.tflite. Use the tf.lite.Interpreter to infer the image

4. Inference of the image

   The post-process output of the TFLite model of any image consists of
   Boxes: Bounding boxes of detected objects
   Classes: class index of the detected objects
   Scores: Confidence scores of detected objects
   Count: Number of detected objects
   Image_width: Width of input image
   Image_height: Height of the input image.

   The detected image is stored in tflite/detected folder.

EfficientDet1 model learning training time and loss functions

The effect of changing the number of epochs on model learning time and validation loss functions is discussed in this section.

Time taken during model training and validation:

The above TFLite model is run for four different epochs values (50, 100, 200 and 300). It is noticed that the time taken for training and validation of model is in the number of hours rather than in minutes. Further, the model training time increases linearly with the increase in the number of epochs (especially towards higher iterations) as shown in Figure 4.5.

Figure 4.5: model training and validation time (hrs) vs number of epochs

It appears that the training time in hours is rather large for the efficientDet1 models. This is partly attributed due to the large number of classes used in training the model and partly due to the model's libraries (TF and TFLite models) used in training the smaller model size to run on mobile devices.

Computation of loss function:

Loss functions measure how far an estimated value is from its true value.
When training deep learning models, in general, there are three types of loss functions (i) Regression loss functions, (ii) Classification, and (iii) Multi-class

functions [71, 72].

Loss_bbox: a loss that measures how "tight" the predicted bounding boxes are to the ground truth object (usually a regression loss, L1, smoothL1, L2, Regression L2, etc.). L1 loss is the mean absolute error (MAE) and L2 loss is the mean squared error.

loss_cls: a loss that measures the correctness of the classification of each predicted bounding box: each box may contain an object class, or a "background". This loss is usually called cross-entropy loss.

The multi-class entropy loss is usually called a multi-class classification loss functions. The types of computation of loss functions are not fixed, but they can change depending on the task at hand and the detection technique used for the goal to be met.

EfficientDet1 computes the various parameters such as (Det_loss, Class_loss, Box_loss, Reg l2_loss, loss, learning rate, and gradient norm) during training and validation of object detection models. Det_loss is the determinant loss and the loss function is the sum of all loss functions.

We have computed these parameters for four different values of epochs (50, 100, 200, and 300) to find the best efficient model. The validation values of these functions are distinguished from the training functions by prefixing the word as Val_ (i.e. Val_det_loss, Val_Class_loss, Val_Box_loss, Val_Reg l2_loss, Val_loss).

The comparison of these loss functions for four epochs during the training period and validation period with increasing the number of epochs is shown in Figure 4.6. The loss functions are shown for epochs 50 in (a), epochs 100 in (b), epochs 200 in (c), and epochs 300 in (d) of this figure.

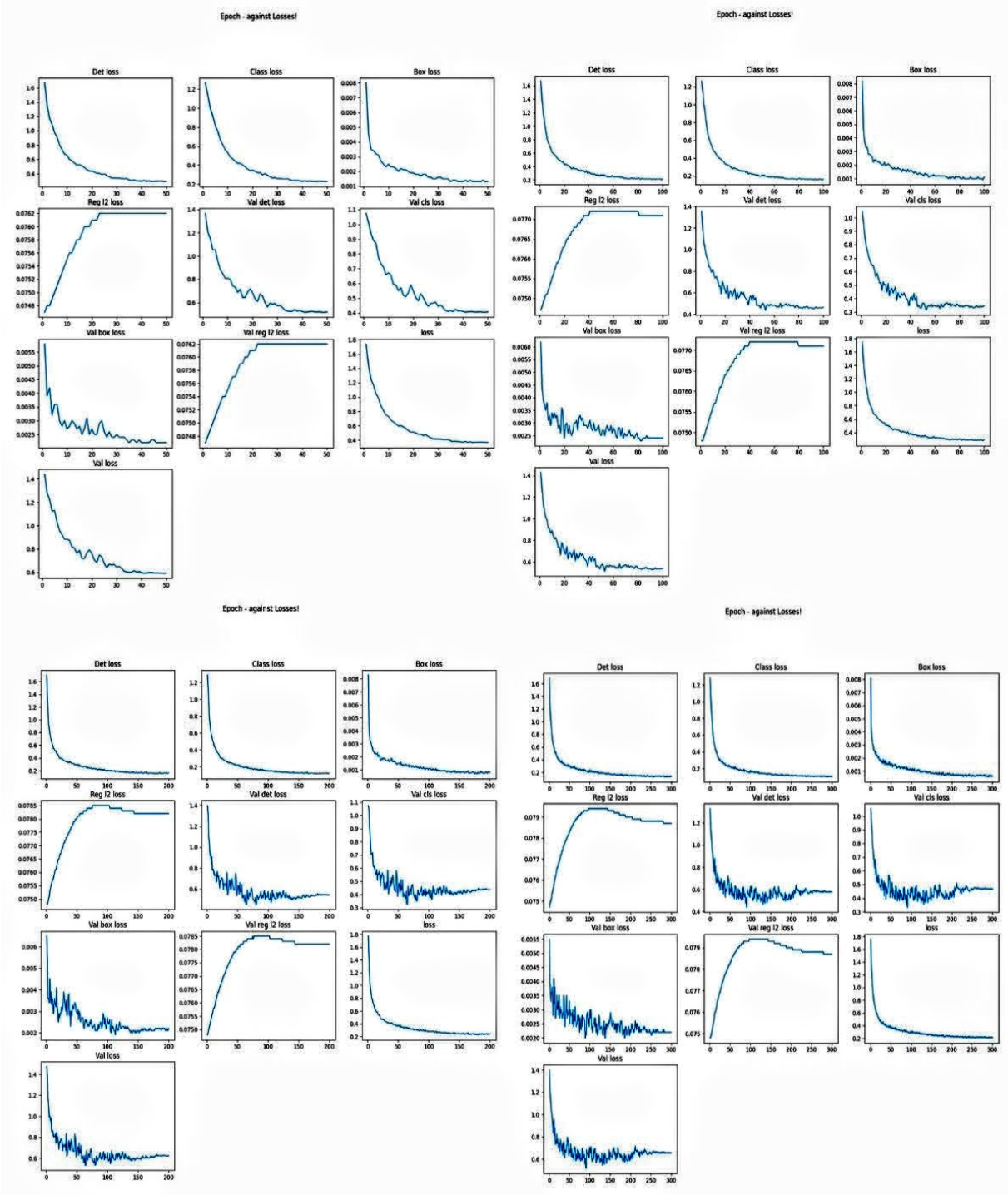


Figure 4.6: EfficientDet1 model training and validation loss functions are shown for epochs 50 in (a) left top, for epochs 100 in (b) right top, for epochs 200 in (c) left down, and for epochs 300 in (d) right down.

It is evident from Figure that all the det, class, box, and total loss functions sharply

fall at the beginning of the first 10 to 20 iterations and then slowly these functions keep improving with the increase of the number of iterations. Similarly, the performance of total loss and val_loss functions attains very good value at the end of iterations during both training and validation of the model. If the number of epochs is increased from 50 to 100, the model validation loss (Val_loss) functions first show the improvement in decreasing the loss values. However, increasing the number of epochs from 100 to 200 or 300 values, the Val_loss functions stop decreasing and start increasing in loss values indicating that the local minima of the loss function are around 100 epochs. Thus the best EfficientDet1 model (raspi1.tflite) is selected with 100 epochs to avoid overfitting at higher iterations of 200 or 300.

Learning rate and gradient norm :

The learning rate is a tuning hyperparameter in an optimization algorithm that determines the step size at each iteration while moving towards minimizing a loss function [73]. It gives you control over how big or small step is needed so that the model can learn faster. Choosing the bigger step can result in unstable training and small steps may result in a failure to train. Hence, the trade-off between the bigger and smaller steps is useful to reach faster towards the minimum of loss function without creating too many oscillations in the gradient norm.

Gradient norm scaling involves changing the derivatives of the loss function when the L2 vector norm (sum of the squared values) of the gradient vector exceeds a threshold value. During gradient descent, the magnitude of the parameter update is scaled using the learning rate. The value for the learning rate can impact two things: (i) The speed at which the algorithm learns, and (ii) whether or not the loss function is minimized. The traditional value of learning rate is either 0.1 or 0.01 which generally ranges from 0 to 1.

The efficientDet1 is trained using the Stochastic gradient descent (SGD) optimizer. The default learning rate is selected to be 0.01, and no momentum is used by default. While training the EfficienDet1 model, two important parameters, learning rate, and gradient norm functions are also computed and presented for four epochs (50, 100, 200, and 300) in Figure 4.7.

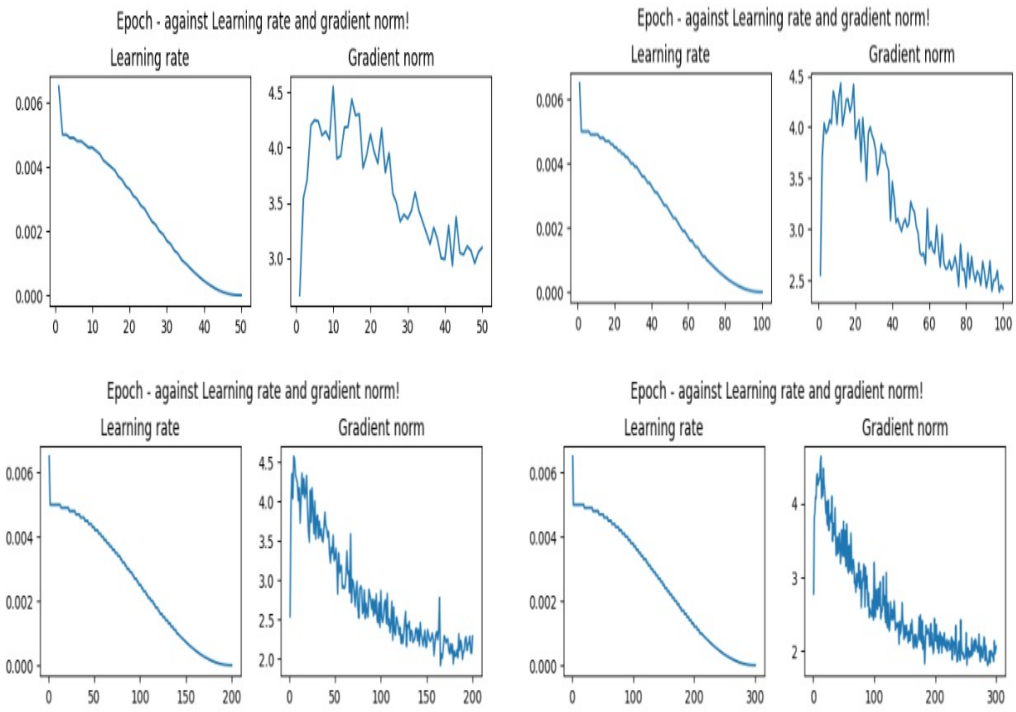


Figure 4.7: Learning rate and gradient norm computed during EfficientDet1 model training are shown for epochs 50 in (a) left top, for epochs 100 in (b) right top, for epochs 200 in (c) left down, and for epochs 300 in (d) right down.

It is suggested that the learning rate is very fast at the beginning of the first few training epochs and then slowly improves with the increase in the number of epochs. The gradient norm values jump in the beginning for a few epochs, and then the gradient norm keeps decreasing with minor fluctuations with increasing the number of epoch values in all four cases. Though the learning rate and gradient norm function for all four cases is good during model training, the fluctuations in gradient norm are smaller in 100 epochs (case b).

In view of learning rate and gradient norm behavior, and together with loss functions in Figure 4.6, the best EfficientDet1 model (raspi1.tflite) is selected for the case of 100 epochs to test the object detection results in chapter 5.

4.2.2 Yolov5s using PyTorch

As mentioned in Chapters 1 and 3, Glen Jocher 2022 found the importance of Yolov5 models (small, medium, larger) over the EfficientDet [0-4] smaller models [14, 15].

Since we want to run a smaller size model on Raspberry Pi, we will be using Yolov5s, the smaller version of the Yolov5 model, for the present research work.

Though the author did not publish any research paper, his first official version of Yolov5 source code is released for its use by Ultralytics and is available on GitHub[5][14].

This source code is based on the Pytorch framework and requires the installation of the torch and some lightweight Python libraries. The models can learn training incredibly fast, which lowers the cost of experimenting. Further, the model PyTorch weights can be translated from ONXX weights to CoreMl to IOS for use on any mobile device such as Raspberry Pi. The inference (or detection) input can either be a single image, a batch of images, video feeds, or camera ports.

The code can run on Google colab notebook. This colab provides a Python programming environment equipped with GPU computing resources and is free to use it. For training and validating the Yolov5s models, the code uses the images in text format (rather than XML files for each image in PASCAL VOC format used for EfficientDet1) of any input of image which is *.txt. Roboflow software online tool was used to convert these XML files into text files for Yolov5s.
For Yolov5s model training and validating the same image dataset is used for the EfficientDet1 model as well to make the comparison of the two models' results more meaningful. This image dataset consists of three distinct datasets for

(i) training (70 %),

(ii) validation (20 %),

(iii) test (10 %).

The instructions for "How to train Yolov5 models on the custom dataset" are given on Roboflow by Jacob Solawetz and Joseph Nelson on Jun 10, 2020 [74] and also on the Wiki page of Ultralytics website under the topic of "training custom data"[6] [75].

[5] https://github.com/ultralytics/yolov5
[6] https://github.com/ultralytics/yolov5/wiki/Train-Custom-Data/

The source code for the Yolov5s model consists of a single Python script called as "Training_YOLOV5_.ipynb". This code is developed and tested in Google colab. It is found to be running successfully and the final code is kept in the Source code area under Appendix A.

The first part of this code describes how to infer the objects in any given input image or video images using the pre-determined model weights (if already available). If not available, then use the second part of the code to train and validate the model using the custom image dataset, and obtain the desired model weights for the use of inference in the first part. The implementation details of this code are given below.

(A) Training Custom data (Second part of code)

To train the Custom data in the second part of the code, the steps are as follows [74, 75].

- Install YOLOv5 repository and the dependencies
  First, clone the YOLOv5 repository and install the necessary dependencies. By doing this, we will prepare our programming environment for the execution of commands for object detection training, validation, and inference.

- Download Custom YOLOv5 Object Detection Data
  We will download our custom image dataset with labels into Roboflow. Use the "YOLOv5 PyTorch" export format to convert image files in txt format (*.txt). Note that the Ultralytics implementation calls for a YAML file defining where your training and test data are.

- Define the name of all 27 classes used in the image dataset (similar in Efficient-Det1).
  Classes are eggs, lemon, tomato, garlic, butter, onion, potato, orange, cereal, apple, paprika, sprite, coke, noodles, chocolate_drink, potato_chips, scrubby, sponge_opl, crackers, pringles, sausages, grape_juice, orange_juice, cloth_opl, basket, help_me_carry_opl, tray.

- Define Model Architecture and Configuration

Since we want to use model Architecture Yolov5s, the code uses this model and will use the yolov5s.yaml config script.

Roboflow writes this yaml config script that defines the parameters for our model like the number of classes, anchors, and each layer.

(One need not edit these cells, but you may if needed).

- Train a custom dataset using YOLOv5s Detector Once the image dataset is loaded, classes and config files are defined, then the model is trained by passing the following number of arguments.

  Img : define input image size (416)
  Batch : determine the batch size (16)
  epochs: define the number of training epochs (50)
  data: set the path to our yaml file
  cfg: specify our model configuration file (models/custom_yolv5s.yaml)
  weights: specify a custom path to weights
  name: results name folder name (yolov5s_results)
  nosave: only save the final checkpoint
  cache: cache images for faster training

After defining these parameters, Python script train.py is used to train the yolov5s model and the command looks as
Python train.py –img 416 –batch 16 –epochs 50 –data dataset.location/data.yaml –cfg ./models/custom_yolov5s.yaml –weights ” –name yolov5s_results –cache

The main important arguments used are batch size 16 and epochs 50 to get the model trained and validate for further testing of any image.

- Evaluate Custom YOLOv5s model performance
  Training losses and performance metrics are saved to Tensorboard and also to a log file defined above with the –name flag when we train. In our case, we

named this yolov5s_results. (If given no name, it defaults to results.txt.) The results file can be plotted as a png file after training completes.

- Visualize YOLOv5 training data
  In the code, there is a provision to view train*.jpg images to see training images, labels, and augmentation effects (if needed when training starts).

- Run YOLOv5 Inference on test images
  Once the training and validation are complete, the trained weights are ready to run inference on any image or video images from the test image dataset. These weights are saved in folder **runs/train/yolov5s_results/weights**

- Export Saved YOLOv5 Weights for Future Inference
  Now that we have trained and validated our custom model, we can save and export the weights to any desired folder for inference on our device elsewhere.

Inference of images (First part of code)

If Yolov5s model weights are ready, then run inference from the test/images folder and follow the steps:

1. Install the necessary Yolov5s software dependencies.

2. Upload any input image either as any image (*.jpg) or input video name (was.mp4).

3. Run inference with Yolov5s model trained weights in Google Colab using Python script detect.py. The weights are kept in this folder runs/train/yolov5s_results/weights, and the file name is best.pt

4. Only the first four cells of code are required to run any image or video images from the test dataset. The output results are saved in the runs/detect folder and the file name will be exp* with a number increment after each next run.

5. The speed and accuracy of Yolov5s detection are found to be better than EfficientDet1 detector.

Yolov5s model learning training time and loss functions

The effect of changing the number of epochs on model learning rate and validation loss functions is discussed in this section.

Time taken during model training and validation:

By training the model with different epochs, one can find the best model without underfitting or overfitting. The best model will give improved accuracy of the model. To do so, the Yolov5s model is run for four different epoch values (50, 100, 200 and 300) and the total time taken vs the number of epochs is plotted in Figure 4.8. It is noticed that the total time for training and validation of the model increases linearly (approx..) with the increase in the number of epochs and it is much small (in minutes rather than in hours) as compared to the EfficientDet1 model time (see Figure 4.5).

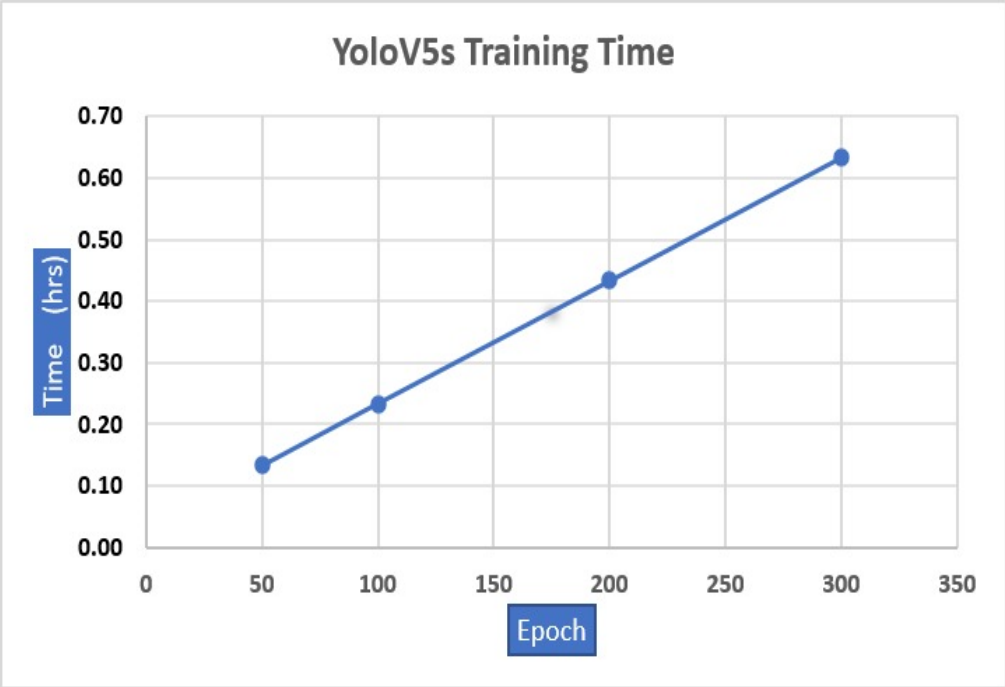


Figure 4.8: Yolov5s model training and validation time (minutes) vs number of epochs

Computation of loss function :

The loss functions are indicators of how well an algorithm predicts an object. In the case of the Yolov5s model, three types of loss functions are calculated (i) box_loss, (ii) obj_loss, and (iii) cls_loss functions [76]. The total loss is the sum of all these three loss functions.

box_loss: a loss that measures how "tight" the predicted bounding boxes are to the

ground truth object. It uses L2 loss which is the sum of the mean squared error calculated based on the predicted box location (x,y,h,w).

The box loss measures how accurately the algorithm can pinpoint an object's center and how completely the anticipated bounding box encloses an object.

Obj_loss: If a bounding box prior is not assigned to a ground truth object, then it incurs no loss for coordinate or class predictions, only for the objectness. Mean squared error calculated for the objectness-Confidence score.

The probability that an object exists in a suggested zone of interest is basically measured by objectness. If the objectivity is high, an item is probably present in the image window.

Cls_loss: a loss that measures the correctness of the classification of each predicted bounding box: each box may contain an object class, or a "background". It uses the binary cross-entropy loss for the object class predictions.

How successfully the algorithm can determine the proper class of a given object is shown by the classification loss.

Yolov5s algorithm computes these three loss functions (box_loss, Obj_loss, and train_loss). We have competed for four epochs (50, 100, 200, and 300) to find the best model. The loss functions during validation are distinguished from training loss functions by using the word Val (i.e. Val/box_loss, Val/obj_loss, and Val/cls_loss).

The comparison of these loss functions for four epochs during training and validation periods, with increasing the number of epoch values are shown in Figure 4.9 (a) for 50 epochs, (b) for 100 epochs, (c) for 200 epochs and (d) for 300 epochs.

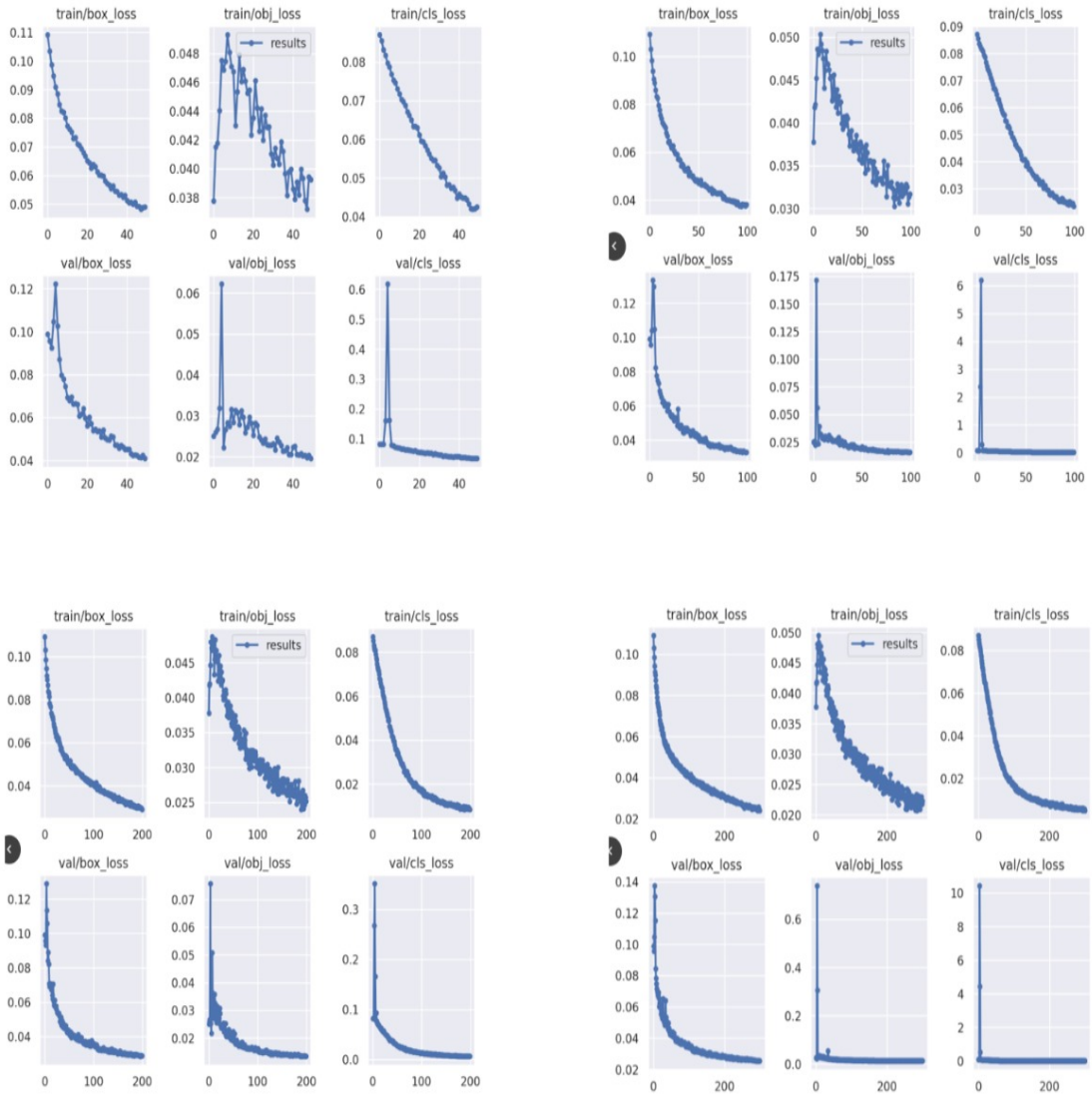


Figure 4.9: Yolov5s Model training and validation loss functions are shown for epochs 50 in (a) left top, for epochs 100 in (b) right top, for epochs 200 in (c) left down, and for epochs 300 in (d) right down.

It is seen from Figure 4.9 that all three loss functions (box_loss, Obj_loss, and Cls_loss) are sharply falling at the beginning of first 10 to 20 iterations, and then slowly these functions keep decreasing with increase in the number of epochs to show continuous improvement in the model. The performance of validation loss functions showed a rapid decline and attains an improvement till the end of each iteration.

With an increase in the number of epochs from 50 to 300 values, the Val_loss functions keep decreasing and attain a very good plateau and lowest at 300 epoch where there is minimal change in loss functions. Hence, the model for 300 epochs is considered the best optimal model for this experiment.

4.3 Evaluation Metrics

There are two very important parameters (precision and recall) of Evaluation metrics which are known as performance indicators for data retrieval from a collection or sample data[77]. In addition to this, F1-score, average precision (AP), mean average precision (mAP), average recall (AR), mean average recall (mAR), intersection over union (IOU), precision-recall curves, confusion matrix are used to compute various other parameters to evaluate the performance of the model detections.

In object detection, there are four possible cases (i) object detected to a correct class is called a Ture Positive (TP), (ii) object detected belongs to another class, then it means a False positive (FP), (iii) if the object should have been detected but not detected is known as False negative (FN) and (iv) object detected but should not be detected means True Negative (TN). These all cases are shown in Figure 4.10 [77]. However, it is important to note that TN result does not apply in real object detection context as there are an infinite number of detections (a large number of bounding boxes) that should not be detected within an image and hence not considered in the computation of precision and recall [78].

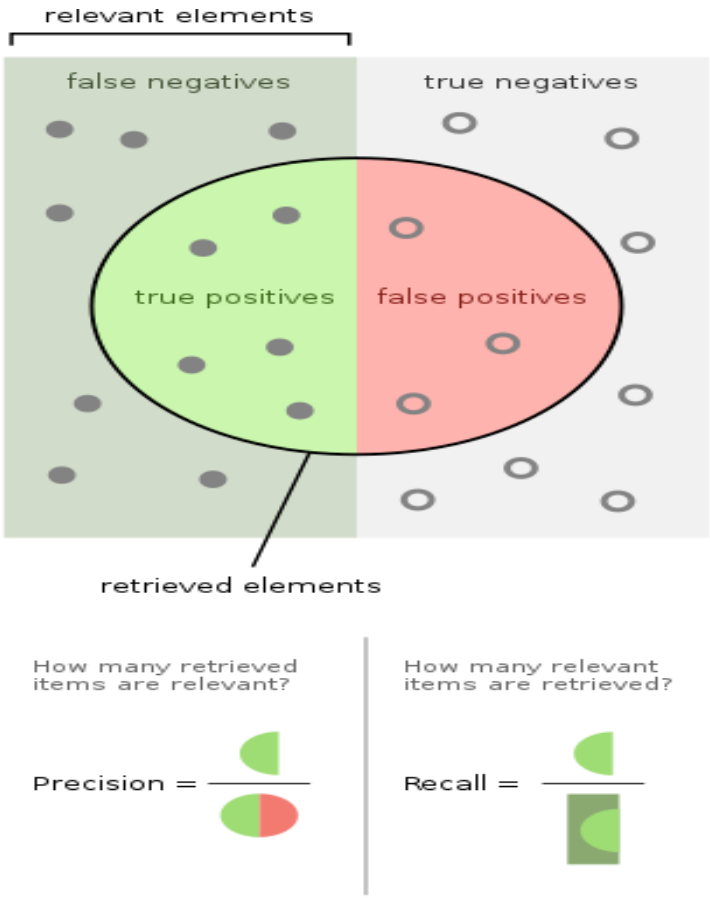


Figure 4.10: Showing the representation of Precision and Recall

4.3.1 Precision and Recall

The proportion of relevant cases among the retrieved instances is known as precision (also known as positive predictive value), whereas the proportion of relevant instances that were retrieved is known as recall (also known as sensitivity). Thus, relevance serves as the foundation for both precision and recall [78].

Precision	= TP / (TP+FP)	= TP/ (All elements declared as positive)
	= TP / ALL Detections	
Recall	= TP / (TP+FN)	= TP/ (All positive elements)
	= TP / All Ground Truths	

In the above Figure 4.10, TP is 5, FN is 7 and FP is 3 (TN not considered). So the Precision and Recall will be given as

Precision	= 5/(5+3)	= 5/8	= .625
Recall	= 5/(5+7)	= 5/12	= .416

4.3.2 F1 score

F1 is the harmonic mean of precision and recall (not the arithmetic mean)
F1 = 2* Precision * Recall / (Precision + Recall)

F1 = TP / [TP + 0.5 (FP+FN)]

In the above Figure 4.9,

F1	= 5/[5+.5(3+7)]
	= 5/[5+5]
	= 5/10 = .5

F1-score has a value between zero and 1; the higher the value, the higher the accuracy of detecting an object.

4.3.3 Intersection Over Union (IoU)

The important question is how to define the correct detection or incorrect detection (TP, FP, FN) to evaluate the performance of the detector. In view of this, the concept of Intersection over Union (IoU) is very necessary to understand. In object detection, IoU measures the degree of overlapping area between the bounding box of ground truth (GT) and predicted truth (PT) divided by the area of union between them as shown in Figure 4.11 [78].

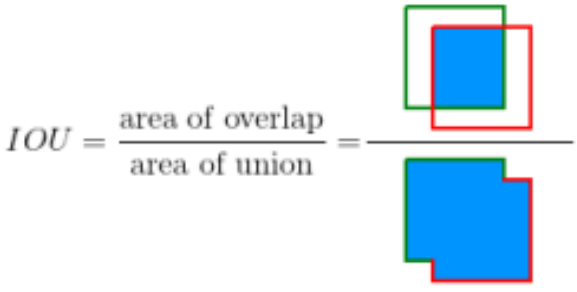


Figure 4.11: Intersection over Union (IoU)

IoU ranges between 0 and 1 where 0 shows no overlap and 1 means perfect overlap between GT and PT.

One can categorize a detection as correct or incorrect by comparing the IOU with a specified threshold value (say t=.5). The detection is deemed to be accurate if IOU is greater than the threshold value and if it is less then the detection is incorrect. For example, see below Figure 4.12 with different cases of overlap and computed values of IoU [79].

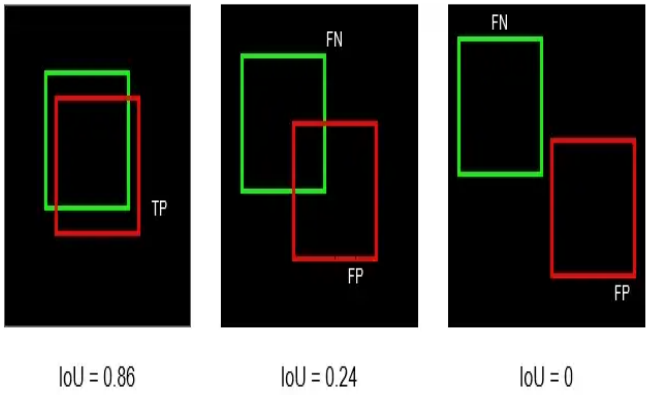


Figure 4.12: Identification of TP, FP, and FN through IoU thresholding

If the threshold value is greater than .5, then the only first case is TP. If the threshold value is less than .24, then both the first and second cases become TP. Hence, the decision to make detection as TP or FP and ground-truth as FN completely depend on the choice of IoU threshold used. The precision-recall curves for each class will depend on the threshold value.

4.3.4 Average precision (AP)

The average precision for any specific class is calculated as the area under the Precision-Recall curve defined as

$$AP = \int_0^1 P(R)dR$$

The precision $\times$ recall curve can be seen as a trade-off between precision and recall for various confidence values associated with the bounding boxes generated by a detector.

If the confidence of a detector is such that its FP is low, the precision will be high (close to 1.0). However, in this case, many positives may be missed, yielding a high FN, and thus a low recall. The recall will rise if one accepts more positives, but the FP may rise as well, lowering the precision. A competent object detector should, however, locate every ground truth object (FN = 0; high recall) and recognize only pertinent objects (FP = 0; high accuracy). Hence, a high area under the curve (AUC) tends to indicate both high precision and high recall. Unfortunately, in practical cases, this curve is often not monotonic rather it is zigzag-like and hence poses a challenge to compute the area under the curve. To overcome this problem, basically, there are two approaches (i) N-point interpolation and all point interpolation and then AP is calculated [80].

To do this unambiguously, Timothy C. Arlen 2018 [81] suggested that one can calculate the AP value as the mean precision at N equally spaced recall values (for example, 11 points at .1 interval between 0 to 1.0) divided by the total number of recall values. The AP values will be different for each P-R curve corresponding to each threshold value. AP values are generally taken as the accuracy of the detection model. In view of this, AP values or the accuracy of the model are invariably used in my thesis in accordance with scientific research papers.

In the COCO dataset, AP is calculated for IoU = 0.5 to .05:0.95, AP (IoU=.50), AP (IoU=.75). AP was also calculated area-wise object size: AP (small), AP (medium), and AP (large) [82].

4.3.5 Mean Average Precision (mAP)

The mean average Precision is calculated by taking the average of AP over all classes. As shown in the following equation, AP_i is the AP value of the i-th class and C is the total number of classes [80].

$$mAP = \frac{1}{c} \sum_{i=1}^{C} AP_i$$

Since the AP value is different for each threshold value, mAP will be different too. If desired, the mAP can be taken as the average of all classes and overall threshold values.

4.3.6 Average Recall (AR)

Average Recall is not calculated in the same manner as done in the case of average precision [AP].

Hosang et al. 2015 [83] introduced a novel metric, the average recall (AR) which rewards both high recall and good localization. This correlates surprisingly well with detection performance. In their method, two kinds of plots are generated (i) for the fixed proposal (i.e. say 100 proposals, 1000 proposals, or 10,000 proposals per image) and then (ii) for fixed IoU values ranging in between from .5 to 1).

In the first kind of plots, for a fixed number of proposals (i.e. say 100 detection proposals per image), the recall values are plotted as a function of IoU overlap threshold values ranging from .5 to 1. Similar plots are done for 1000 proposals per image and 10,000 proposals per image also.

In the second kind of plot, for a fixed IoU threshold value (say .5), the recall values are varied as a function of the number of proposals per image. Similar curves are generated for other IoU threshold values ranging between .5 to 1.0 (say .8 and 1).

Now the novel metric, the average recall (AR) is calculated by taking the average of all threshold recall values (IoU from .5 to 1) against the number of proposals. In other words, the AR metric evaluates a wide range of IoU thresholds. The least acceptable IoU according to most metrics is 0.5, which can be read as a rough localization of an object. An IoU of 1 corresponds to the exact location of the identified object. By taking averaging the recall values in the range [.5 to 1], the model is evaluated on the assumption that object location is extremely accurate [80].

Note that the study of AR in the COCO dataset reports that it is not calculated exactly the same way as done here. Instead, AR is the average of maximum obtained recall across several IoU thresholds. COCO computes AR (max=1) given 1 detection per image, AR (max=10) with 10 detections per image, and AR (max=100) with 100 detections per image. They did calculate AR area-wise for the object size: AR (small) for small objects, AR (medium) for medium objects, and AR (large) for large objects [82].

4.3.7 Mean Average Recall (mAR)

Similar to mAP in section 4.1.5, the mean average Recall is calculated by taking the average of AR over all classes. AP_i is the AR value of the i-th class and C is the total

number of classes [80].

$$mAR = \frac{1}{c}\sum_{i=1}^{C} AR_i$$

4.3.8 Matrix parameters (EficientDet1 models)

Different detection models evaluate different types of matrix parameters. Though EfficientDet1 computes many parameters for the model performance but the important ones are AP and mAP.

In view of this, AP values for each class are calculated and are made available for all four models (50, 100, 200, and 300 epochs). For these cases, the variation of AP with classes is shown in Figure 4.13. The mAP value for 50 epochs is .51, 100 epochs is .58, 200 epochs are .55 and 300 epochs are .58. The mean average value is shown as a solid orange line in each diagram.

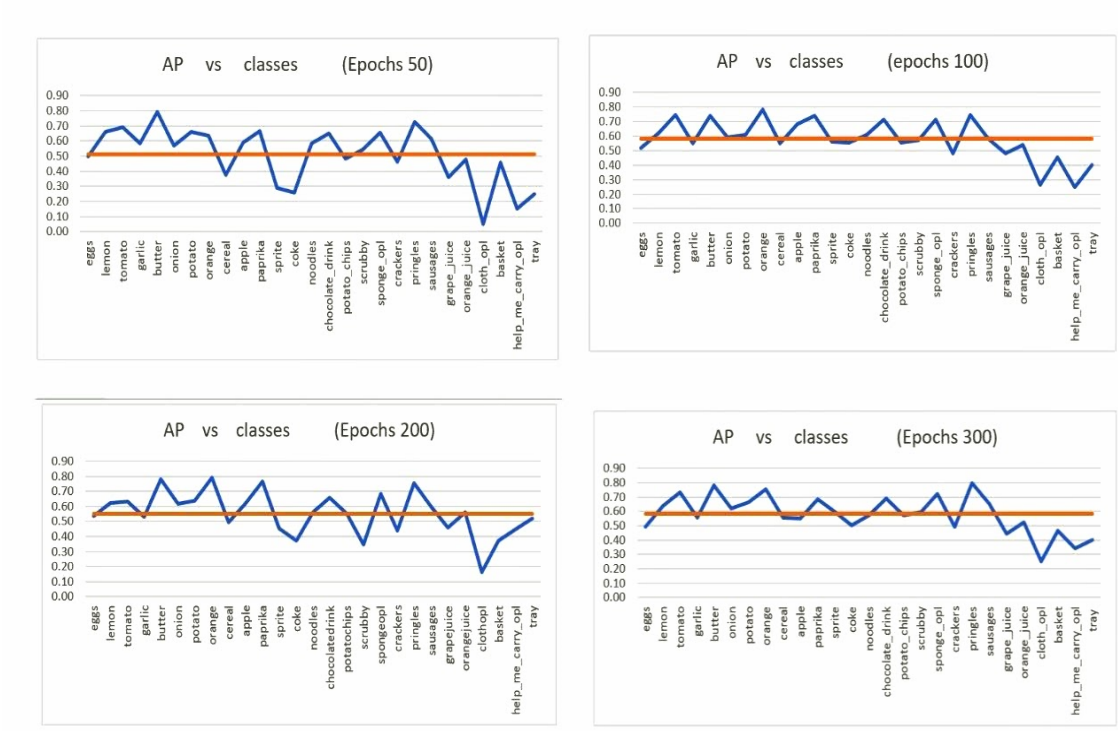


Figure 4.13: EfficientDet1 model matrix parameters of AP values for each class for the epochs of 50 (a) left top, 100 (b) right top, 200 (c) left down, and 300 (d) right down.

It is seen that AP values for each class vary from class to class and show some improvement as the number of epochs increases from 50 epochs to 100 epochs. However, the variation of AP values shows very little improvement as the number of epochs is increased from 100 to 200 or 300 epochs. The mAP value is good (.58) for 100 epochs as compared to other epochs models and the loss functions also show a minimum of around 100 epochs as seen in Figure 4.6. Hence the model weights of 100 epochs are used for inferences (detection) of images from the test image dataset in the following next Chapter 5.

4.3.9 Matrix parameters (Yolov5s models)

Yolov5s algorithm computes very important parameters such as average precision (AP), average recall (AR), mAP@.5, and mAP@.5:.95 during training and validation of 4 epochs models (50, 100, 200, and 300 epochs). The variation of these parameters as a the function of each iteration for 4 epochs is presented in Figure 4.14.

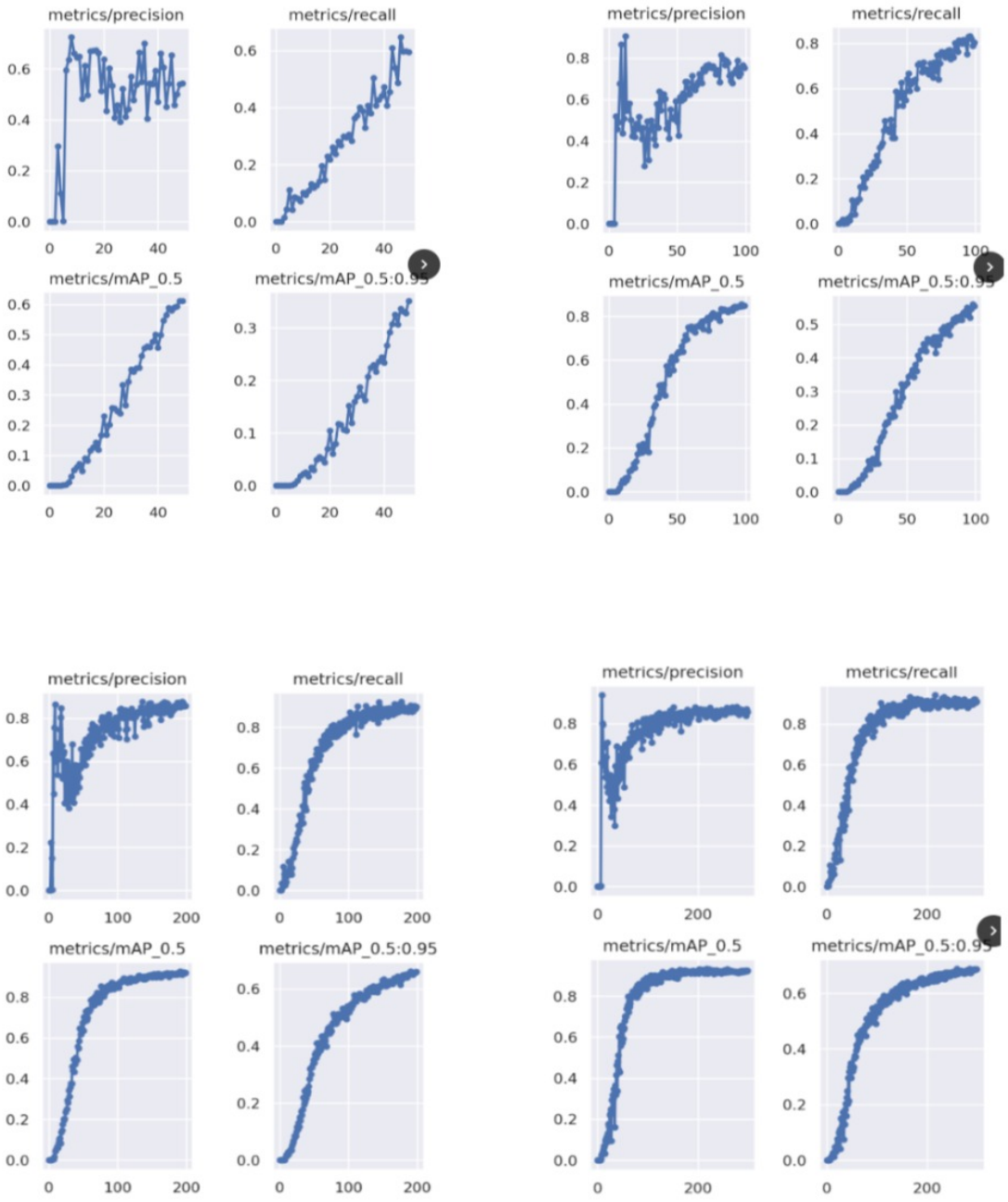


Figure 4.14: Yolov5s matrix parameters of AP, AR, map@.5 and mAP@.5:.95 values for 4 epochs of 50 (a) left top, 100 (b) right top, 200 (c) left down, and 300 (d) right down.

It is evident from Figure 4.14 that performance evaluation parameters show good

progress as the iteration cycle increases for all 4 epochs. The max values of all these 4 parameters keep increasing as the number of epochs increases from 50 to 300 epochs and attains the optimal values for 300 epochs indicating the model with 300 epochs as the best final model.

Yolov5s also computes the Precision-Recall Curve for all classes at mAP@.5. These curves for four epochs 50 (a), 100 (b), 200 (c), and 300 (d) are shown in Figure 4.15.

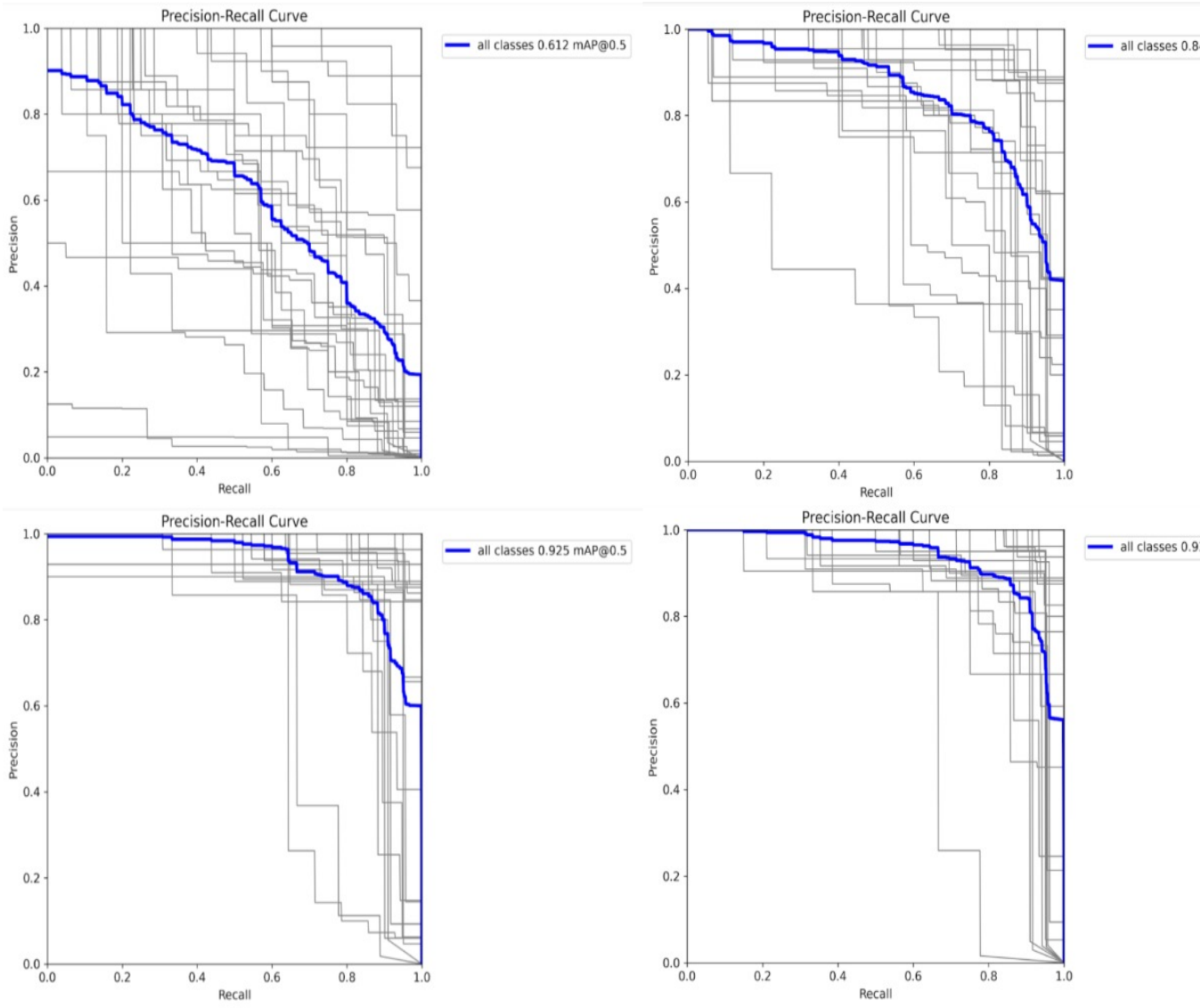


Figure 4.15: Yolov5s matrix evaluation of Precision-Recall curves for four epochs of 50 (a) left top, 100 (b) right top, 200 (c) left down, and 300 (d) right down.

It is seen from Figure 4.14 that average Precision and recall values for all classes (blue

line) are small at .612 (mAP@.5) at 50 epochs, .847 (mAP@.5) for 100 epochs, .925 mAP@.5 for 200 epochs and .931 mAP@.5 for 300 epochs. This clearly indicates the model performance with 300 epochs is best with both high precision and high recall values and is in agreement with the conclusion reached in Figure 4.14. Note that black lines represent the precision-recall curve of all 27 classes used in the analysis and the average of all classes is shown by the blue line.

4.3.10 Confusion matrix (yolov5s models) :

Another matrix evaluation parameter for the performance of an object detector is the Confusion matrix. Yolov5s algorithm calculates this, but not available for the EfficientDet1 model. The advantage of the confusion matrix is explained by Kukil 2022 [84].

The Confusion matrix is a good way of showing the analysis of all individual classes (True) on the X-axis and the predicted classes on the Y-axis. The predicted class shows the confidence score on each square of the predicted class.
The detection of each class can be correct (TP), or incorrect (FN or FP). The correctness depends on the threshold value. For example, say we have an apple and predicted detection is correct for the apple (TP), then on the Confusion matrix, it will mark the apple on Y-axis with a confidence score. The confidence value on apple will show us how good the detected model is. A higher value ranging from .8 to 1.0 indicates good detection. If the detection of an apple is incorrect FN or FP, then either it did not detect (FN) or it mistook some other class (onion) as an apple. In this case, a lower confidence score value and only a few missing classes still approve the detection model to be very successful.
The confusion matrix of yolov5s models is shown for four epochs 50 (a), 100 (b), 200 (c), and 300 (d) in Figure 4.16.

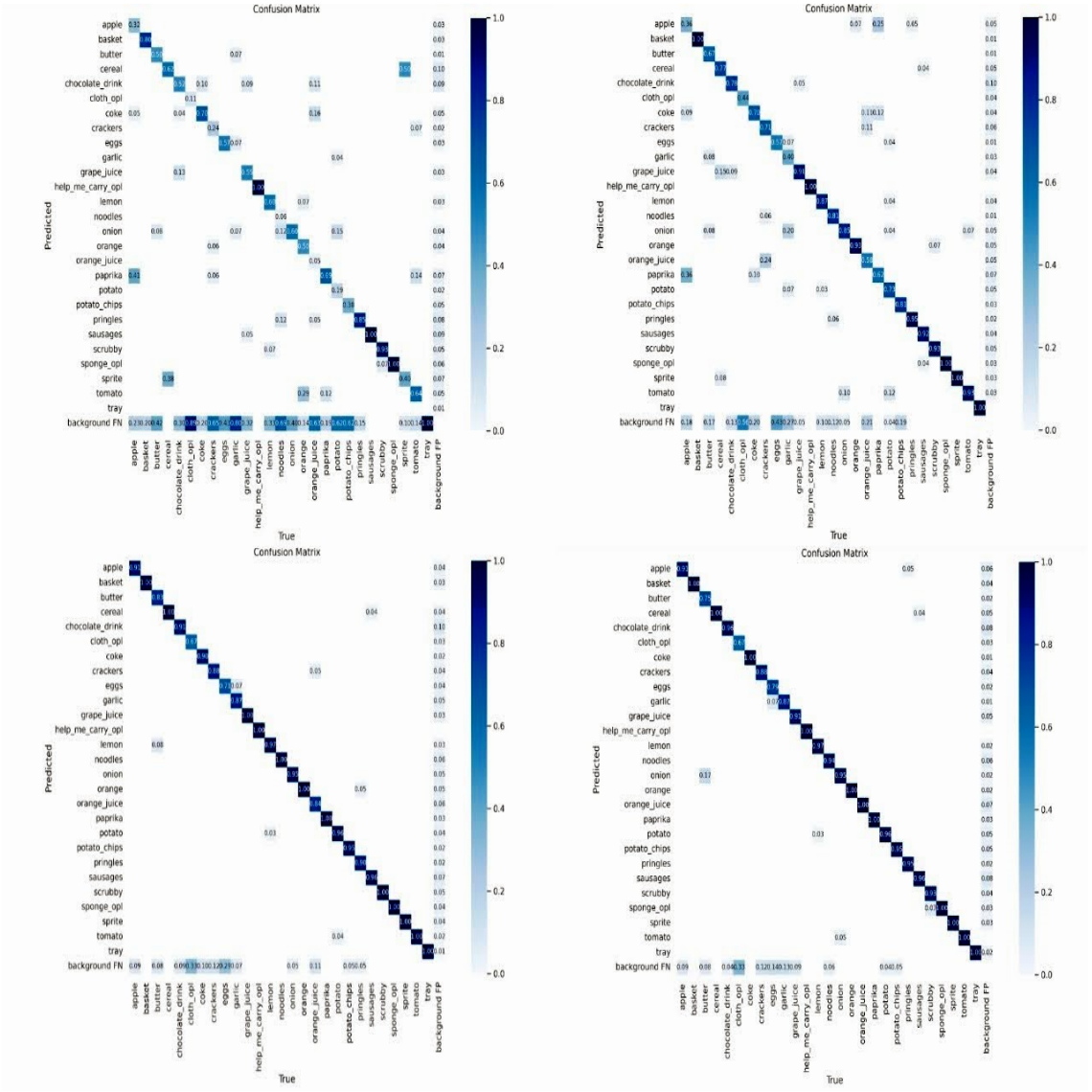


Figure 4.16: Yolov5s Confusion matrix for four epochs of 50 (a) left top, 100 (b) right top, 200 (c) left down, and 300 (d) right down.

It is seen from Figure 4.16 that for a low number of epochs 50, the Confusion matrix shows most of the classes as misclassified with other classes. The reason for the mistake to be with other classes is due to the fact that the model is not yet properly trained and loss functions are not fully minimized. Increasing the model training with a large number of epochs with 100, 200, and 300 epochs, the misclassification with other classes keeps decreasing and it is much less for a model with 200 epochs and least with the 300 epochs indicating that this model is best. The confidence score in each class square (diagonal) keeps increasing with an increase in the number

of epochs from 50 to 300. The prediction of all classes in the 300 epochs model is almost perfect with not much misclassification and a very reliable high confidence score (>90% in most of the classes) in the square of all classes (Figure 4.16 (d)). Hence Yolov5s model weights of 300 epochs are used for inferences of test images in the dataset in next Chapter 5.

4.4 Summary

- This chapter highlights the importance of two state-of-the-art models (EfficienDet1 and yolov5s) and their implementation as smaller models for the use of mobile devices such as Raspberry Pi.
- Open-source code of both models is used and it is run through Google colab successfully. Google Colab provides the correct GPU resources with the correct Python libraries environment required to run these source codes. It is easy, free, and efficient to run on these resources.
- Data annotation (putting bounding boxes and labeling them) tools such as labellmg or Roboflow are used.
- Procedure for training and validating both models is discussed. It is noticed that the Yolov5s model's training time is much faster than that of EfficientDet1 models for the same image dataset and same classes (27 items).
- The summary of the object detection flow chart for smaller models on mobile devices is shown in Figure 4.17.

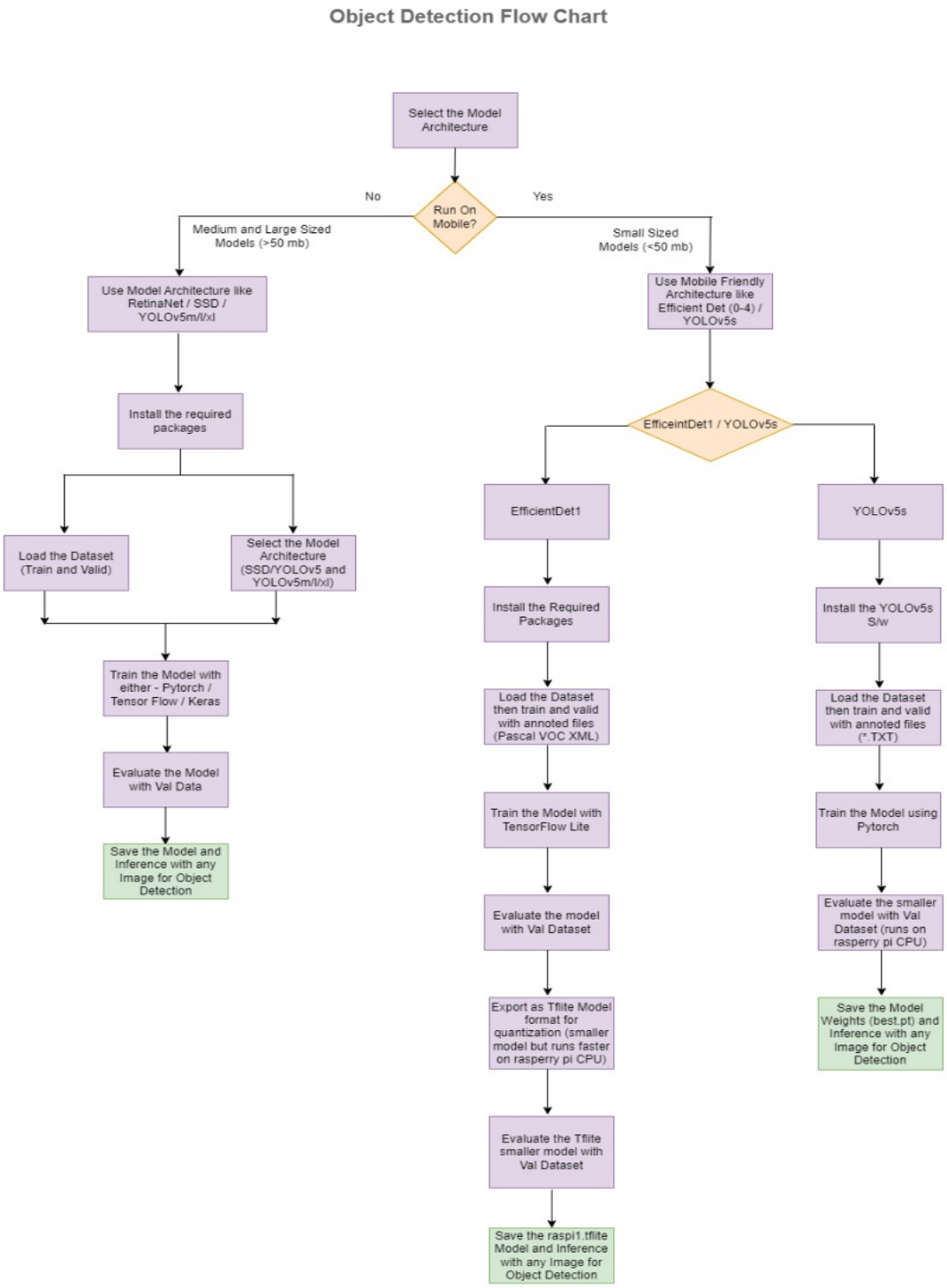


Figure 4.17: Object detection Flow chart for smaller models to run on mobile devices

- EfficientDet1 model with 100 epochs is found to be good, whereas Yolov5s is

best with 300 epochs.

- These two final models are selected for inferences of object detection using the images in the test image dataset. It is discussed in the following chapter 5.
- Smart homes are sure to make a huge impact in the future by using the speed, accuracy, and efficiency of these two modern object detectors, particularly the Yolov5s model.

Chapter 5

Results and Discussion

5.1 Evaluation performance of detection models (EfficientDet1 and Yolov5s) using test images

In the previous chapter, the EfficientDet1 model with 100 epochs is considered the best model with a weights file (called raspi1.tflite), whereas Yolov5s with 300 epochs is selected as the best model with a weights file (called best.pt). We have used these two model weights files for the inferences of test images in the evaluation of matrix parameters and the development of an automatic detection system of graphical user interface (GUI) along with an alert system for the help of the user.

Both detection models run on Google colab employing GPUs on cloud cluster, and the specification of GPUs are Tesla T4, 15110MiB. To evaluate the performance of detection models (EfficientDet1 and Yolov5s), the following four images are selected for detection purposes from the test image dataset.

Case-1: Simple 2 eggs image (two eggs close together)

The purpose of this simple image is to test whether the object detection models can infer these two tiny eggs in real-time refrigerator home items or not. First, using the EfficientDet1 model weights (raspi1.tflite) and the source code of **detection-fromimages.ipynb** defined in chapter-4, the inference predicted truth (middle) of this simple input image as shown in Figure 5.1 together with the ground truth image (upper). The detected image is stored in **tflite/detected** folder. Now, the same

image is subjected to Yolov5s model weights (best.pt) and the inference of image source code **Training_Yolov5_.ipynb (First part of code)** defined in Chapter-4 is used to detect the predicted truth of the Yolov5s model. The output result is saved in **runs/detect** folder. The file name is exp* with a number increment after each run. This output is shown in Figure 5.1 (bottom) below.

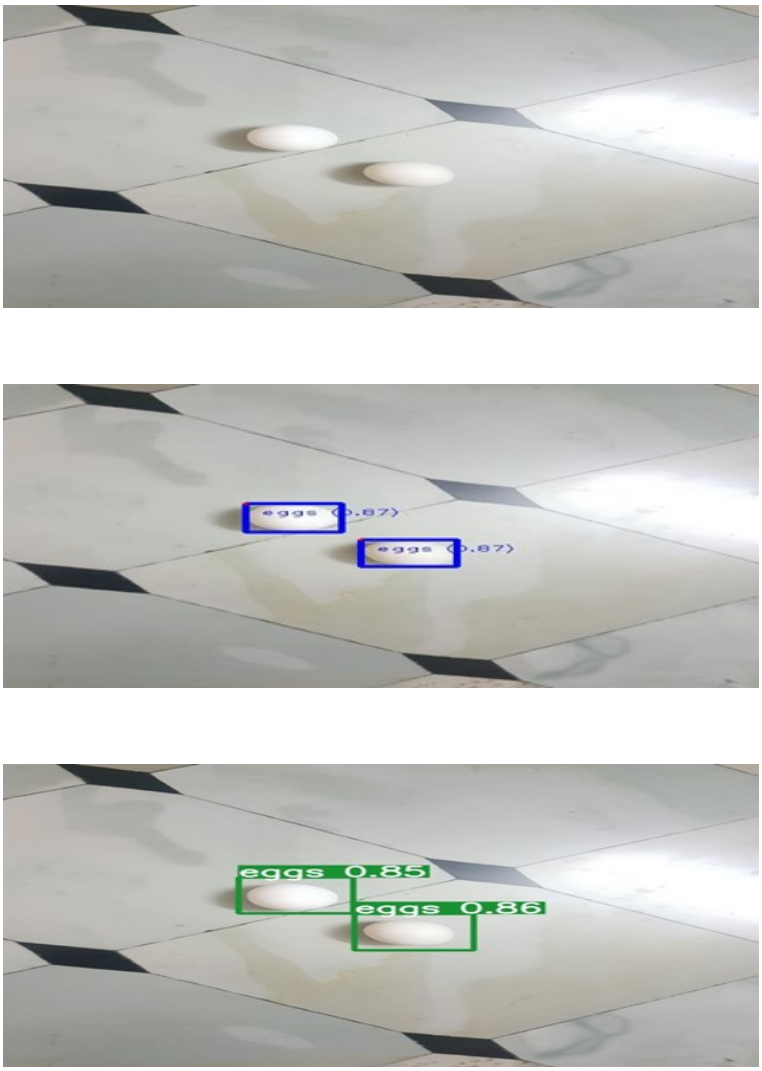


Figure 5.1: Truth (top), predicted EfficientDet1 (middle) and predicted Yolov5s (bottom)

It is evident from Figure 5.1 that both eggs, even though small and close by, are detected by both detector models correctly, and confidence is around 85%. Both detection models are run on Google colab using a GPU processor. The inference time for the EfficientDet1 model is 992 ms, whereas the inference time for Yolov5s is 9.6

ms which is much smaller than that of EfficientDet1 model. It has only one single class detection (2 eggs) application.

Case-2: Apple, orange and other refrigerator items (mixed small, medium, and large items)

The second case is considered with many mixed small, medium, and large items in a single image with 13 classes. The classes are paprika (1 item), chocolate_drink (1 item), orange_juice (1 item), cereal (1 item), orange (1 item), sponge_opl (1 item), sausages (1 item), noodles (1 item), grape_juice (1 item), potato_chips (1 item), apple (1 item), crackers (1) and scrubby (2 items). Most of the items are refrigerator classes, but few of them belong to home objects. This is due to the restrictions of data collection from the internet (Kaggle.com), which we could not avoid. Hence, we must live with a mixed refrigerator and home objects items. However, it is useful to check the usefulness of detection even in the presence of unwanted items.

As in case-1, the detection is performed on both the models to infer the predicted truth of EfficientDet1 and Yolov5s models. These predicted truths are compared and studied against the ground truth. First, the image is subjected to the inference code of EfficientDet1 (detectionfromimages.ipynb) using the model weights (raspi1.tflite), and then the inference code of Yolov5s (Training_Yolov5_.ipynb, First part of code) with model weights (best.pt) are used. The computed results are stored in tflite/detected for EfficientDet1, runs/detect folder for the Yolov5s model. The ground truth (top) of the image, predicted truth of EfficientDet1 (middle) and predicted truth of Yolov5s (bottom) are shown in Figure 5.2.

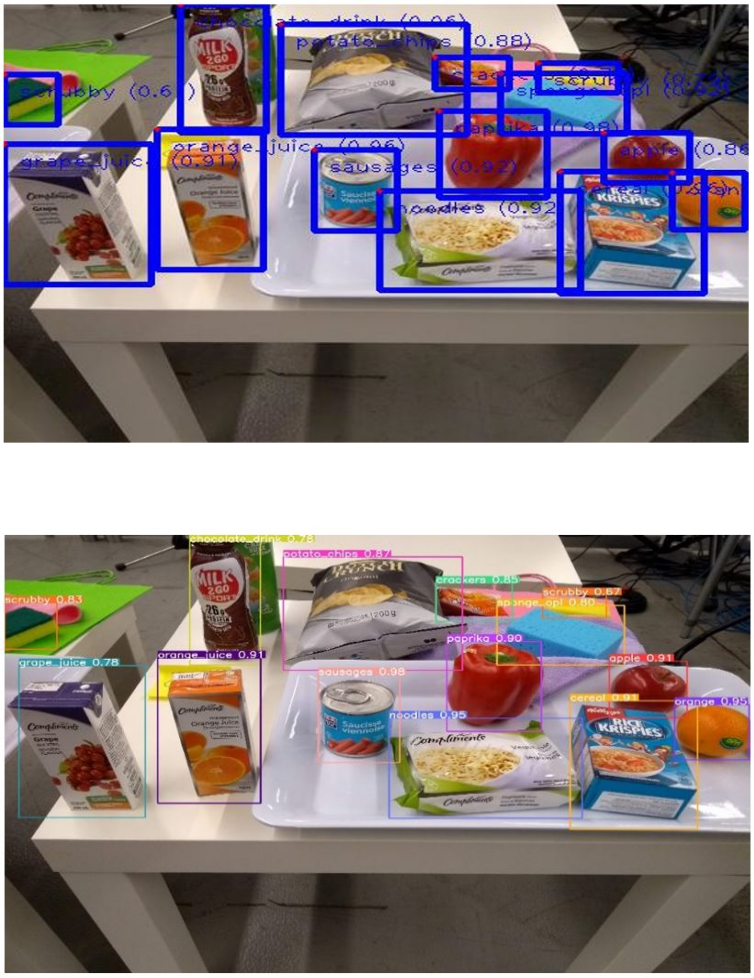


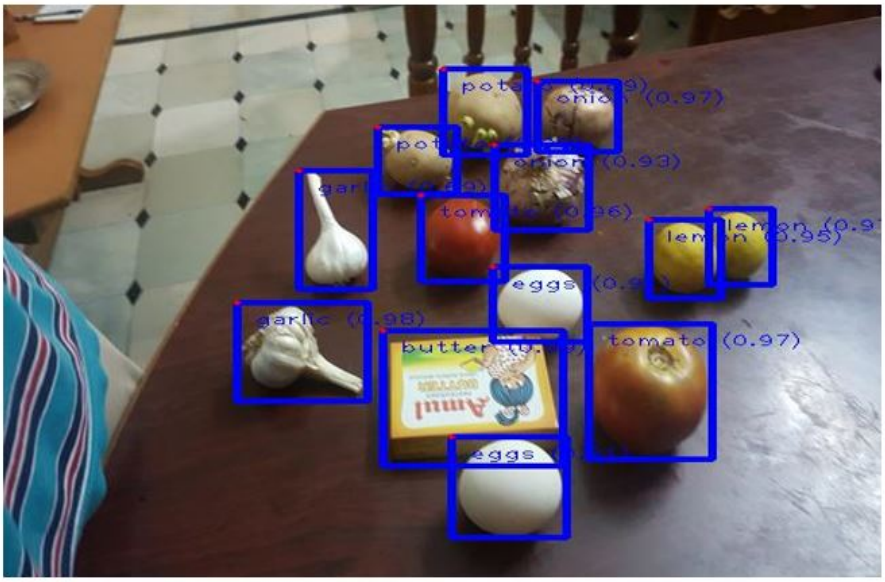
Figure 5.2: Truth (top), predicted EfficientDet1 (middle) and predicted Yolov5s (bottom)

It is interesting to note that all 13 classes with the mixed small, medium, and large items, despite the closeness, are well detected by both models (EfficienDet1 and Yolov5s) successfully. In this case, the inference detection time for EfficientDet1 is 880 ms, and for the Yolov5s it is around 14.4 ms.

Case-3: Tomato, potato, and other mixed items (medium-sized items)

The third case is considered with 7 classes consisting of a total of 13 items (medium-sized). The classes are eggs (2 items), potato (2 items), tomato (2 items), lemon (2 items), garlic (2 items), onion (2 items), and butter (1 item). This third image

is subjected to the inference code of EfficientDet1 **(detectionfromimages.ipynb)** using the model weights (raspi1.tflite) and then the inference code of Yolov5s **(Training_Yolov5_.ipynb, First part of code)** with model weights (best.pt). The ground truth (top) of the image, predicted truth of EfficientDet1 (middle), and predicted truth of Yolov5s (bottom) are shown in Figure 5.3.



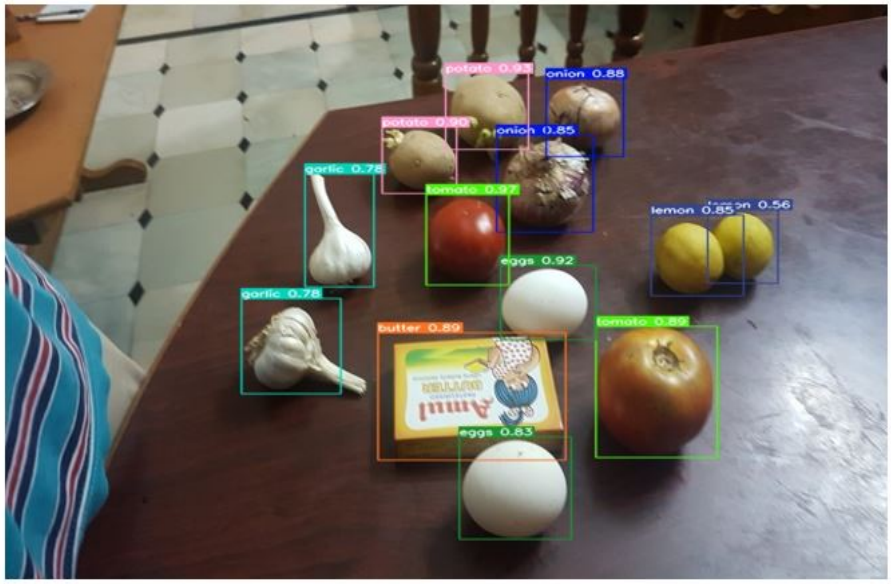


Figure 5.3: Truth (top), predicted EfficientDet1 (middle) and predicted Yolov5s (bottom)

The identification and location of 13 mixed items in the congested environment are predicted correctly by both detection models, giving credence to their usage in real-time applications for refrigerator object identifications. It has 7 classes (1 butter, 2 eggs, 2 garlic, 2 lemons, 2 onions, 2 potatoes, 2 tomatoes), totaling 13 mixed items. The inference time of EfficientDet1 is 929 ms, and for Yolov5s is 9.2 ms.

Case-4: Tomato, potato, and other mixed items (medium-sized items as in Case-3, but mixed at different orientations)

In the fourth case, it was decided to select an image as in Case-3 (same 7 classes), but with different orientation and locations in comparison to the original orientation and locations of all 13 mixed items (1 butter, 2 eggs, 2 garlic, 2 lemons, 2 onions, 2 potatoes, 2 tomatoes).

The inference results of this fourth image using both models are obtained and compared to assess the robustness of these two detection techniques. For this, again, the inference code of EfficientDet1 **(detectionfromimages.ipynb)** with model weights (raspi1.tflite), and the inference code of Yolov5s **(Training_Yolov5_.ipynb, First part of code)** with model weights (best.pt) are utilized to get the predicted truths of

EfficientDet1 and Yolov5s models. The comparison of ground truth (top), predicted truth of EfficienDet1 (middle), and the predicted truth of Yolov5s are shown in Figure 5.4.

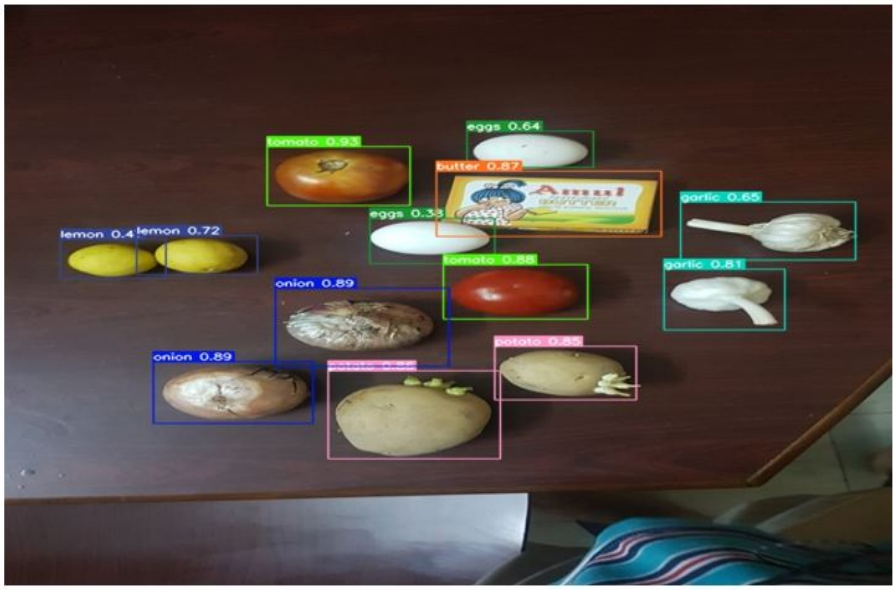


Figure 5.4: Truth (top), predicted EfficientDet1 (middle) and predicted Yolov5s (bottom)

It is interesting to note that even after changing the orientation and location of all 13 object items as used in Case-3, are still resolvable for all 7 classes with respect to their correct locations and class identification. The inference detection time for EfficientDet1 is 937 ms, and for Yolov5s is 9.1 ms.

On the basis of all four images, the average inference time per image for EfficientDet1 by using the GPU processor on the Google Colab platform is about 935 ms [(880 ms + 929 ms + 939 ms + 992 ms)/4.0], whereas for Yolov5s it is about 10.6 ms [(14.4 ms + 9.2 ms + 9.1 ms + 9.6)]. Note that inference detection time is highly dependent on the use of the processor of your computing platform.

5.2 Comparison of AP values for Two Models' Performance

In research studies, the comparison of the model's performance is not limited to detection abilities and the inference speed of detection models (fastness). The average accuracy precision (AP) value of each class is also equally important to assess the performance of detection models.

In view of this, the comparison of average precision (AP) values of each class for EfficientDet1 and Yolov5s models for all the above four cases are shown in Table 5.1.

Table 5.1: Showing speed and Accuracy (AP values) of all four cases for comparison of EfficientDet1 and Yolov5s models

CASE 1	**EfficientDet1**		**Yolov5s**		
Detected Item	**Speed (ms)**	**Accuracy**	**Speed (ms)**	**Accuracy**	**Quantity**
Egg 2	992	0.98828125	9.6	0.9095298	2

CASE 2	**EfficientDet1**		**Yolov5s**		
Detected Item	**Speed (ms)**	**Accuracy**	**Speed (ms)**	**Accuracy**	**Quantity**
Paprika	880	0.98375	14.4	0.8561	1
Chocolate_drink	880	0.96875	14.4	0.9168117	1
Orange_juice	880	0.96875	14.4	0.8692796	1
Cereal	880	0.96484375	14.4	0.9180394	1
Orange	880	0.9296875	14.4	0.8714119	1
Noodles	880	0.9296875	14.4	0.8883986	1
Sponge_opl	880	0.921875	14.4	0.8006564	1
Grape_juice	880	0.9140625	14.4	0.90267634	1
Sausages	880	0.9140625	14.4	0.81413925	1
potato_chips	880	0.8828125	14.4	0.8387507	1
apple	880	0.87109375	14.4	0.9449726	1
crackers	880	0.8125	14.4	0.9054897	1
Scrubby	880	0.79296875	14.4	0.80635256	2

CASE 3	EfficientDet1		Yolov5s		
Detected Item	**Speed (ms)**	**Accuracy**	**Speed (ms)**	**Accuracy**	**Quantity**
Potato	929	0.98828125	9.2	0.8741332	2
Garlic	929	0.984375	9.2	0.90285385	2
Butter	929	0.984375	9.2	0.912405	1
Lemon	929	0.97265625	9.2	0.90141004	2
Tomato	929	0.96875	9.2	0.86735666	2
Onion	929	0.96875	9.2	0.8914253	2
Eggs	929	0.953125	9.2	0.91058534	2

CASE 4	EfficientDet1		Yolov5s		
Detected Item	**Speed (ms)**	**Accuracy**	**Speed (ms)**	**Accuracy**	**Quantity**
Potato	937	0.984375	9.1	0.83320737	2
Garlic	937	0.94921875	9.1	0.88838893	2
Butter	937	0.9453125	9.1	0.9269697	1
Lemon	937	0.97265625	9.1	0.8551765	2
Tomato	937	0.953125	9.1	0.8195916	2
Onion	937	0.953125	9.1	0.8546588	2
Eggs	937	0.984375	9.1	0.9085769	2

It is evident from Table 5.1 that the average precision values (Accuracy) of each class for both models are not very much different from each other. The mean average precision values for most of the classes in both models are about 90% (.90) which gives credence to the reliability of both these models in the object detection performance. The inference detection time is much faster for Yolov5s (approx. 9.1 ms) than that of EfficientDet1 (approx. 937 ms).

5.3 Object detection models using GUI for mobile devices

In this research project, we propose to automate object detection using the instant images of a ready-made camera installed at the refrigerator door. The image can be taken at any time or at a specific interval (or once a day). These images can be transmitted through wi-fi and processed on an object detection app installed on mobile devices. The object detection results can be displayed on his/her mobile device through a Graphic card and can be sent to the user through his e-mail account as an update of objects present in the refrigerator.

In addition to the object detection of items in an image, it is very useful to count the total number of items of each class and also to calculate the total calorie estimation of each item. The results can be communicated to the refrigerator user for the selection of a healthy diet each day and to monitor the inventory of object items present at any time inside the refrigerator. For this, we have developed a graphical user interface (GUI) displaying the objects detected in an image (predicted image objects), displaying the total number of items in each class and the total calories of each class.

Although we have two different models of object detection techniques, we used the common approach for counting the items, calorie estimation, an alert e-mail system, and building GUI. The details of Python code, setting up an alert email system for the user, and testing of GUI are discussed in the following sections.
Though the developed Python source code is tested and run on a laptop or desktop using the windows operating system for simplicity, the same code can be repurposed to run on any other platform like a mobile device or embedded devices.

5.3.1 Counting and calorie estimation of food items

The detection of items in each class is more meaningful by adding counting and calorie estimation features in our app. In the development of Python code for counting and calorie estimation of food items, each item is an individual class for the app. When the app detects the item, it creates a list of detected items in the backend. The feature of counting list items in Python is used to count the item's count. The Python codes of EfficientDet1 and Yolov5s are listed in Figure 5.5 and Figure 5.6 respectively.

```
detectedItems=[]
scores=[]
for i in range(0,len(detections)):
    detectedItem=detections[i][1][0][0]
    score=detections[i][1][0][1]
    detectedItems.append(detectedItem)
    scores.append(score)

scoressDf=pd.DataFrame(scores,columns=["Accurancy"])

res={}
for i in detectedItems:
    res[i] = detectedItems.count(i)

itemQuantity=res.items()
itemQuantities = pd.DataFrame (itemQuantity, columns = ['Product Name', 'Quantity'])
self.fill_quantities(itemQuantities,howMuchTime,scoressDf)
self.calc_cals(itemQuantities)
self.diff_count(itemQuantities)
print(itemQuantities)
```

Figure 5.5: EfficientDet1 (TFLite) detected items counting Python code

```
item=[]
quantity=[]
for c in det[:, -1].unique():
    n = (det[:, -1] == c).sum()  # detections per class
    s += f"{n} {names[int(c)]}{'s' * (n > 1)}, "  # add to string
    itemName=names[int(c)]
    itemQuantity=n.item()
    item.append(itemName)
    quantity.append(itemQuantity)
itemDf=pd.DataFrame(item,columns=["Product Name"])
quantityDf=pd.DataFrame(quantity,columns=["Quantity"])
scores=det[:, 4]
scoressDf=pd.DataFrame(scores,columns=["Accurancy"])
itemQuantities=pd.concat([itemDf,quantityDf],axis=1)
self.fill_quantities(itemQuantities,dt[1],scoressDf)
self.calc_cals(itemQuantities)
self.diff_count(itemQuantities)
```

Figure 5.6: Yolov5s detected Items counting Python code

As seen from Figure 5.5 & Figure 5.6, EfficientDet1 and Yolov5s counting codes are

very similar but with minor differences, since these two models have different detection approaches. We prefer to use the same variable names as we did not want to create discrepancies.

The calorie information is obtained from Google. We researched the item's calorie values on Google.com and used these values for individual items to provide meaningful calorie estimation of nutritional value. Each item's calories are shown in Figure 5.7. These values are just guidelines to start the process. The values can be changed at any time in the code.

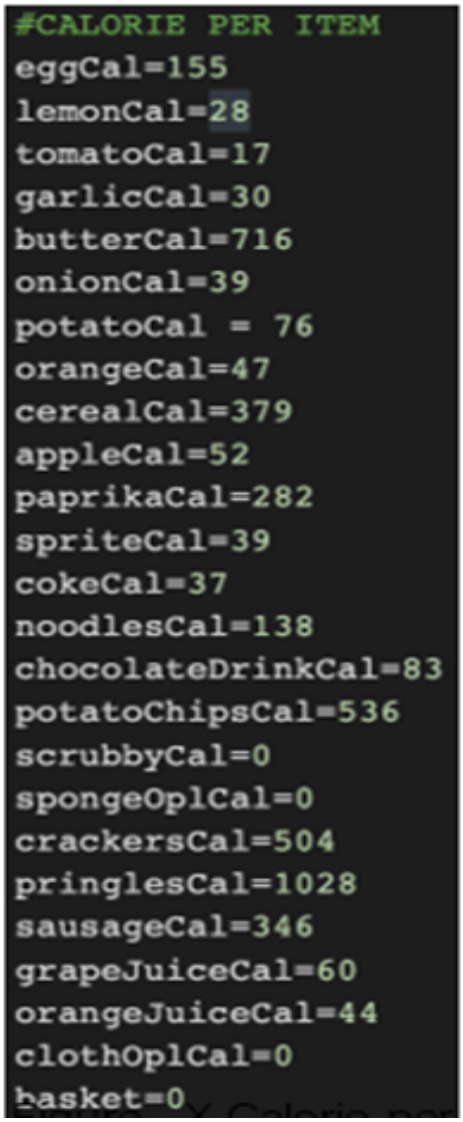


Figure 5.7: Calories per item used in app

Once the counting of items is done, then we have written a very simple math multiplication function to calculate the total estimated calories of each item. The Python code for calorie calculations for a few items is shown in Figure 5.8 (for example only).

```
for i in range(0,len(itemQuantities["Product Name"])):
    item = itemQuantities["Product Name"][i]
    itemQuantity=f'{item}'+"Threshold"
    quantity=0
    if itemQuantity == "eggsThreshold" and eggsThreshold>itemQuantities["Quantity"][i]:
        print("Counted egg number is below dedicated threshold, you should add egg for shopping chart")
        shoppingChart.append(item)
        quantity=eggsThreshold-itemQuantities["Quantity"][i]
        diffirenceItem.append(quantity)
        thresholdDf.append(eggsThreshold)
    elif itemQuantity == "lemonThreshold" and lemonThreshold>itemQuantities["Quantity"][i]:
        print("Counted lemon number is below dedicated threshold")
        shoppingChart.append(item)
        quantity=lemonThreshold-itemQuantities["Quantity"][i]
        diffirenceItem.append(quantity)
        thresholdDf.append(lemonThreshold)
    elif itemQuantity == "tomatoThreshold" and tomatoThreshold>itemQuantities["Quantity"][i]:
        print("Counted tomato number is below dedicated threshold")
        shoppingChart.append(item)
        quantity=tomatoThreshold-itemQuantities["Quantity"][i]
        diffirenceItem.append(quantity)
        thresholdDf.append(tomatoThreshold)
    elif itemQuantity == "garlicThreshold" and garlicThreshold>itemQuantities["Quantity"][i]:
        print("Counted garlic number is below dedicated threshold")
        shoppingChart.append(item)
        quantity=garlicThreshold-itemQuantities["Quantity"][i]
        diffirenceItem.append(quantity)
        thresholdDf.append(garlicThreshold)
```

Figure 5.8: Calorie calculation of few items for EfficientDet1 and Yolov5s

It is to be noted that the same code is used for both models to get consistent results in our app.

5.3.2 Setting up an alert email system for the user

Alert systems are useful for people who are very busy with their heavy lifestyles. These systems are used in various fields such as e-commerce or the business world to remind users of important tasks to make their day-to-day life easier. Alert systems are a very common feature in kitchen machines such as toast machines or ovens. In this thesis, we wanted to implement an alert system for refrigerators that can be very useful for old people and diet-conscious people.

In order to develop an alert system, we needed certain decisions of setting alert conditions in our app. Like we pre-selected the threshold values of each item. If the counted item is below the threshold value that we set, the system alerts the user by sending an e-mail. The pre-selected threshold values of each item are shown in Figure 5.9

```
eggsThreshold=5
lemonThreshold=5
tomatoThreshold=5
garlicThreshold=2
butterThreshold=1
onionThreshold=4
potatoThreshold=5
orangeThreshold=5
cerealThreshold=2
appleThreshold=5
paprikaThreshold=3
spriteThreshold=2
cokeThreshold=2
noodlesThreshold=3
chocolateDrinkThreshold=2
potatoChipsThreshold=2
scrubbyThreshold=3
spongeOplThreshold=3
crackersThreshold=3
pringlesThreshold=1
sausageThreshold=2
grapeJuiceThreshold=1
orangeJuiceThreshold=1
clothOplThreshold=2
basketThreshold=2
```

Figure 5.9: Threshold values of each item for setting an Alert system

It is evident from Figure 5.9 that each item threshold value is adjusted randomly. If the user wants to change these values, they can be changed in the backend code easily. The threshold detection system algorithm is developed using simple math calculations. Once items are counted, each item is compared with a threshold value, calculated quantity, and calories of items. If the value is below the threshold, it is added to the shopping cart list to send an e-mail to the user. This algorithm is shown in Figure 5.10.

```
shoppingChart=[]
diffirenceItem=[]
thresholdDf=[]
for i in range(0,len(itemQuantities["Product Name"])):
    item = itemQuantities["Product Name"][i]
    itemQuantity=f'{item}'+"Threshold"
    quantity=0
    if itemQuantity == "eggsThreshold" and eggsThreshold>itemQuantities["Quantity"][i]:
        print("Counted egg number is below dedicated threshold, you should add egg for shopping chart")
        shoppingChart.append(item)
        quantity=eggsThreshold-itemQuantities["Quantity"][i]
        diffirenceItem.append(quantity)
        thresholdDf.append(eggsThreshold)
    elif itemQuantity == "lemonThreshold" and lemonThreshold>itemQuantities["Quantity"][i]:
        print("Counted lemon number is below dedicated threshold")
        shoppingChart.append(item)
        quantity=lemonThreshold-itemQuantities["Quantity"][i]
        diffirenceItem.append(quantity)
        thresholdDf.append(lemonThreshold)
    elif itemQuantity == "tomatoThreshold" and tomatoThreshold>itemQuantities["Quantity"][i]:
        print("Counted tomato number is below dedicated threshold")
        shoppingChart.append(item)
        quantity=tomatoThreshold-itemQuantities["Quantity"][i]
        diffirenceItem.append(quantity)
        thresholdDf.append(tomatoThreshold)
    elif itemQuantity == "garlicThreshold" and garlicThreshold>itemQuantities["Quantity"][i]:
        print("Counted garlic number is below dedicated threshold")
        shoppingChart.append(item)
        quantity=garlicThreshold-itemQuantities["Quantity"][i]
        diffirenceItem.append(quantity)
        thresholdDf.append(garlicThreshold)
```

Figure 5.10: Algorithm preparing the inventory list made from the difference between item quantity and threshold value

Once the shopping cart list is prepared, the e-mail content is ready to send to the user's e-mail address. For setting up an e-mail system, we have used the free Google "**gmail**" infrastructure. Google LLC allows sending automatic e-mails for free. Only we need to set the configuration for allowance. Google LLC gives the user password for each user. Also, we can add more e-mail users' addresses if we want. In Figure 5.11, the code of sending e-mail as an alert system is shown.

```
from smtplib import SMTP

# Simple Mail Transfer Protocol
# try:
    # Mail Message Configration
    subcjet = "Inventory List Reminder"
    message = "Hi!\n\nThe following items in your refrigerator are running out. Please have a look at them and shop for
    the desired items."+"\n\n\n"+f'{df2}'+"\n\n\n"+"Regards,"+"\n"+"Refrigerator Alert System"+"\n\n"+"**********This is
    an automated email. Please DO NOT reply to this email**********"
    content = "Subject: {0}\n\n{1}".format(subcjet,message)

#     # Hesap Bilgileri
    myMailAdress = "rohitcsc599@gmail.com"
    password = "ovkhyubhztrmrmye"
#"ovkhyubhztrmrmye"

    emailList=["rohitcsc599@gmail.com"]

#     # Kime Gönderilecek Bilgisi
    for email in emailList:
        sendTo=email

        mail = SMTP("smtp.gmail.com", 587)
        mail.ehlo()
        mail.starttls()
        mail.login(myMailAdress,password)
        mail.sendmail(myMailAdress, sendTo, content.encode("utf-8"))
        print("Inventory list was sent to" ,f'{email}' ,  "successfully!")
except Exception as e:
    print("Mail Error!\n {0}".format(e))
```

Figure 5.11: Sending Email System Configuration

5.3.3 Setting up the GUI

Most of the apps have two sides namely the backend and frontend. Until this topic, we talked about the backend functions of the app. But apps need frontend designs for user usage. It is called a graphical user interface (GUI). In our app, we used the Python library (PyQt5) for building GUI. There are two ways to build GUI. First, the developer may write backend codes like HTML or Flutter or draw GUI by using the designer app. In this GUI, we used both of them.

PyQt5 library has its own designer app to build the GUI easily. We preferred to use the PyQt5 designer app for the beginning and then we had to use the backend GUI design for updates. By using PyQt5, we built our app's GUI. In the designer app, you can edit all items and configurations.

Once build GUI by using the designer app, we created GUI's backend code in Visual Studio Code to connect with our app backend codes. The source code of GUI backend codes are attached in the Annex field of this thesis, and our PyQt5 Designer App

view of GUI is shown in Figure 5.12.

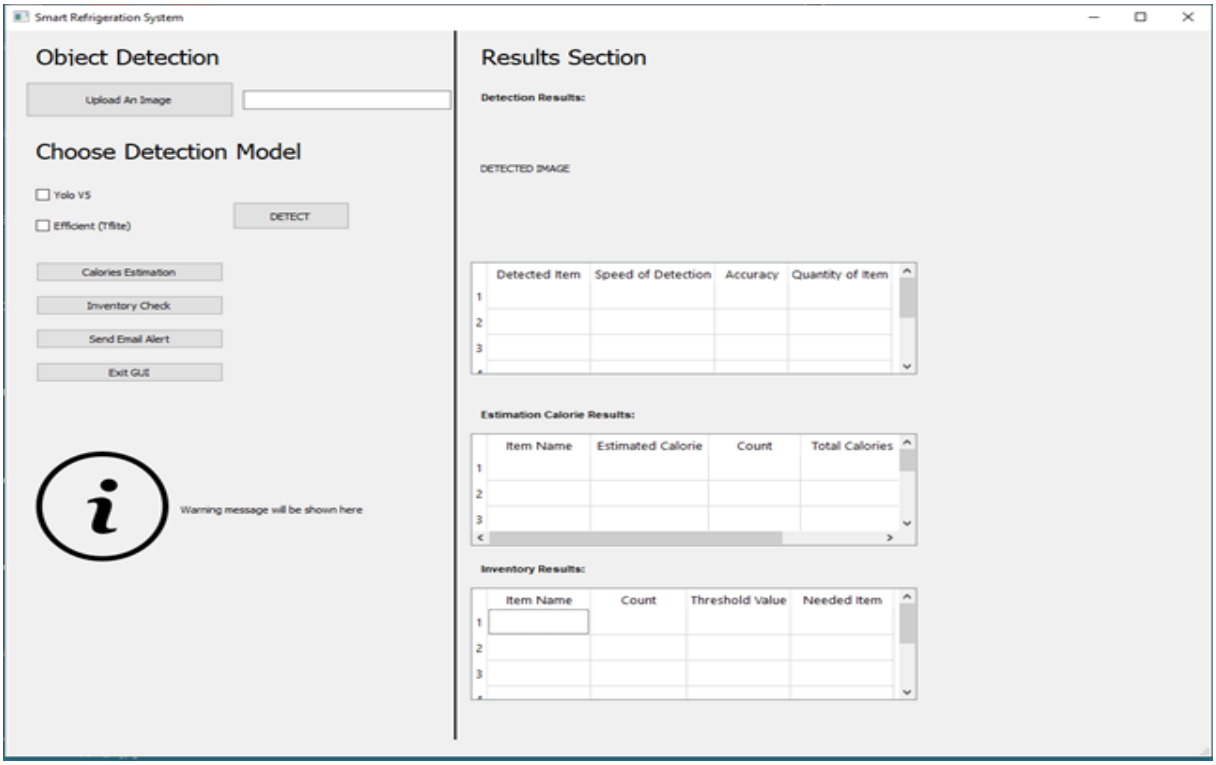


Figure 5.12: PyQt5 Designer App View of GUI

5.3.4 Frontend view of GUI

In Figure 5.12, the frontend view of GUI shows that it is divided into two sections (i) the First section has the Object detection model (left side) and (ii) the Second section has the Results Section (right side).

The left section consists of the names of detection models (Yolov5s or EfficientDet1) and the "Detect" button. By clicking the "Detect" button, one can select the type of model that is desired. Once the detection model is selected, upload the image from the local computer or mobile device and the detection will be performed. Once the detection is over, then the "detected image" is displayed in the right top section. In addition to the detected image, three tables are generated : (1) the top table displays the list of detected items, speed of detection, accuracy, and quantity of each item (class), (2) the middle table lists the item name, estimated calorie, count and total calories and (3) bottom table lists item name, count, the threshold value of each item

(class) and needed item (if any item falls below the threshold value of that class, an alert is generated and e-mail is sent to the refrigerator user). If no detection is required, one can click the "Exit GUI" button towards the left, to exit the GUI.

5.3.5 Python source code for inference of detection model

In addition to GUI backend source codes, the Python code for running the inferences of EfficientDet1 and Yolov5s models (described in Chapter-4) is developed. It is also required to run models in GUI along with the correct model weights. The best model (with 100 epochs) for the EfficientDet1 weight file is raspi1.tflite and is stored in tflite folder. The predicted image is stored in tflite/detected folder.
Yolov5s best model (with 300 epochs) weights are stored in runs/train/yolov5s_results2/ folder and the weights file name is best.pt. The predicted image is stored in the runs/detect folder and the file name is exp* where * is the number incremented after each run.

We have put test images (*.jpg) from the test folder in the home directory where the main python source code file (newYolo.py) resides. This is the file that initiates the GUI and asks to select the input image to load and also to run the detection model either EfficientDet1 or Yolov5s. The main source code file can be run on any mobile device such as Raspberry Pi or laptop or desktop with the proper operating system installed on these devices. The Python source code, along with the GUI backend code mentioned in section 5.3.5 and discussed above, is zipped in Rohit_GUI_code.zip. This zipped file is made available at the end under the Annex field of this thesis.

5.3.6 Installing, running, and testing object detection in GUI

For object detection in GUI, the first step is to download the Rohit_GUI_code.zip file on any device where the source code is to be run. Though the source code can run on any mobile device such as Raspberry Pi, any laptop, or desktop, we preferred to run on our desktop where the operating system is readily available to try our GUI. The specific of our desktop processor is Intel ® Core ™ i5-1035G1 CPU@1.00GHz.

In view of this, we prefer to demonstrate the object detection on the desktop where the windows environment is available and GUI is installed and tested. For installing

the source code on the desktop, we first unzip the Rohit_GUI_code.zip in our home directory. Then change to the Yolov5-Rohit-Final directory.

1. **Unzip Rohit_GUI_code.zip**

   Cd Yolov5-Rohit-Final
   Now before running the Python source code file newYolo.py, it is important to install the required Python libraries and source code. For this, there is a requirements.txt file that contains all the important modules required to run the source code. To run this requirement.txt file, use the pip as **py -m pip install -r requirements.txt**

   It might be possible, it may complain that certain modules are missing. If so, make sure you install these missing modules first on your platform as **py -m pip install tflite_support (missing module name)**

   Similarly, install all the missing modules one by one. Once all the required modules are installed, then the Python source code file (newYolo.py) is ready to run as follows: **py newYolo.py**

   Then it will pop on GUI on the desktop and you are ready to run any image from the test image dataset or any arbitrary image, using the EfficientDet1 or Yolov5s models, on this GUI.

2. **Testing the GUI**

   The aim is to test GUI for detecting the refrigerator items by two different models, counting items, estimating the calories of detected items, and sending an alert system to the user (if needed). All functions are executed through GUI and displayed in GUI. The results are run on a local computer rather than Google colab cloud cluster (GPUs).

For testing we have selected two image cases, discussed in Section 5.1, to evaluate the performance of the models. They are (A) **Case-2**: This image has a total of 13 classes which are paprika (1 item), chocolate_drink (1 item), orange_juice (1 item), cereal (1 item), orange (1 item), sponge_opl (1 item), sausages (1 item), noodles (1 item), grape_juice (1 item), potato_chips (1 item), apple (1 item), crackers (1) and

scrubby (2 items) and (B) **Case-3 :** This image has 7 classes consisting of a total of 13 items (medium-sized). The classes are eggs (2 items), potato (2 items), tomato (2 items), lemon (2 items), garlic (2 items), onion (2 items), and butter (1 item).

The images are uploaded from a local computer via GUI. The uploaded images can be detected using any desired model either EfficientDet1 or Yolov5s. Once the model is selected, then it will upload the image, and detection will perform. Upon completion, the detected (predicted) image will be displayed in the top section under the "Result section" on the right-hand side. The app will fill up the entries of all tables automatically. An e-mail will be sent with the contents of detection items to the designated user (rohitcsc599@gmail.com). We have tested GUI and all functions are found to perform correctly. An example of two test image results is exhibited below.

a. Running Case-2 image in GUI

EfficientDet1 model results:

On a local computer, run the following python source code file

Py newYolo.py

It will pop on GUI. Then select by clicking the desired model (EfficientDet1) and load the desired image from the local computer (in our Case-2 image). Once the image is loaded, detection will be performed. The app fills up the detected image and the entries of all three tables. The output of the detected model in GUI is shown in Figure 5.13.

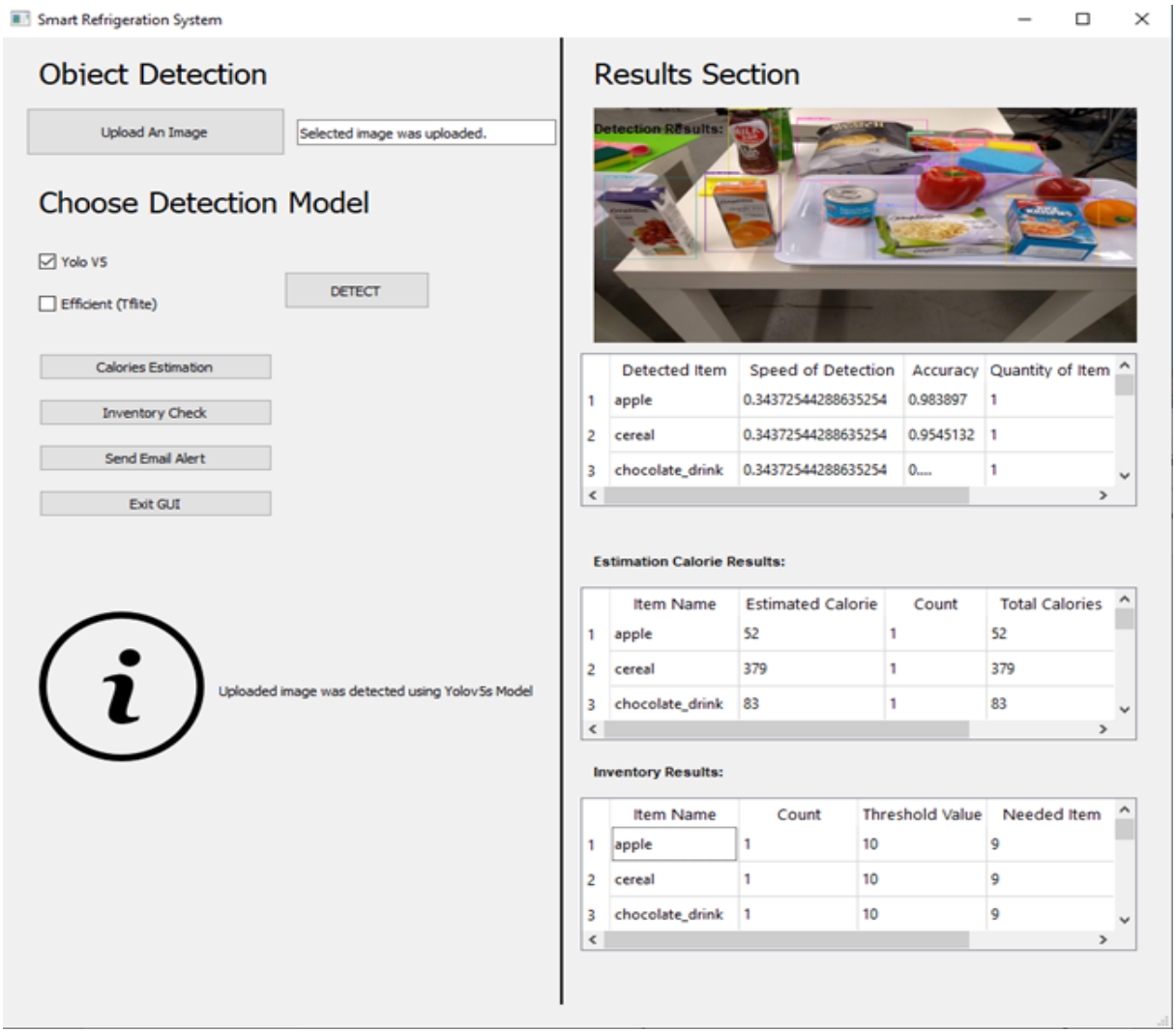


Figure 5.13: EfficientDet1 detected image running on GUI with filled table entries (case-2)

Yolov5s model results:

Now de-select the EfficientDet1 box. Select by clicking another model box (Yolov5s) and load the same image from the local computer (in our Case-2 image). Once the image is loaded, detection is performed. The app fills up the detected image and the entries of all three tables. The output of the detected model for Yolov5s is shown in Figure 5.14.

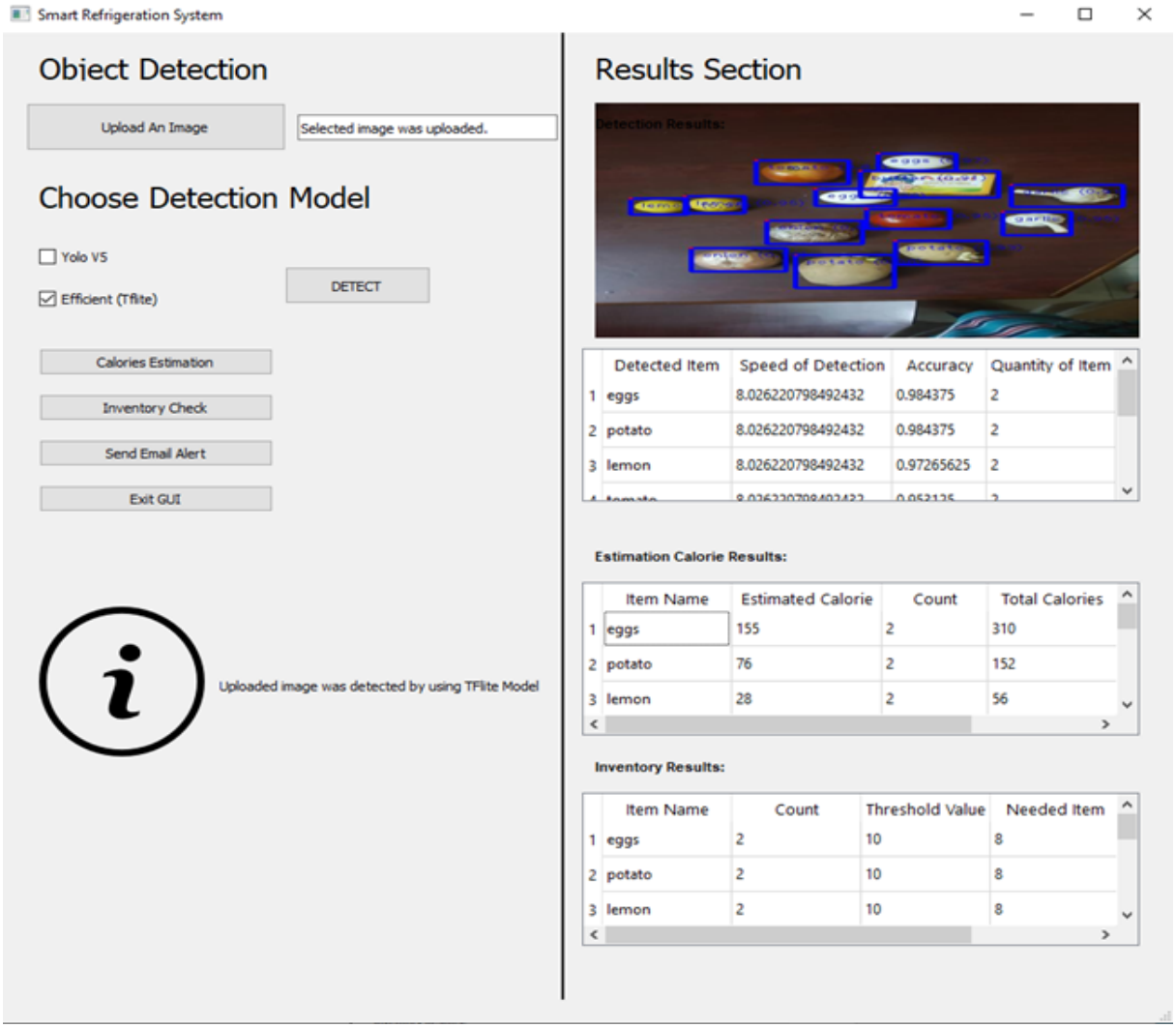


Figure 5.14: Yolov5s detected image running on GUI with filled table entries (case-2)

It is noticed in Figure 5.13, and Figure 5.14 for case-2 that both models detected images, speed, and detection of items, and their number quantity are found to be correct (upper table) in GUI. Even total calorie estimation (middle table) and inventory of items (bottom table) for each class are filled correctly. This indicates that testing of object detection of all GUI functions is very successful.

b. Running Case-3 image in GUI

EfficientDet1 model results :

On a local computer, run the following python source code file

Py newYolo.py

It will pop on GUI. Then select by clicking the desired model (EfficientDet1) and load the desired image from the local computer (in this Case-3 image with 7 classes). Once the image is loaded, detection is performed. The app fills up the detected images and the entries of all three tables. The output of the detected model is shown in Figure 5.15.

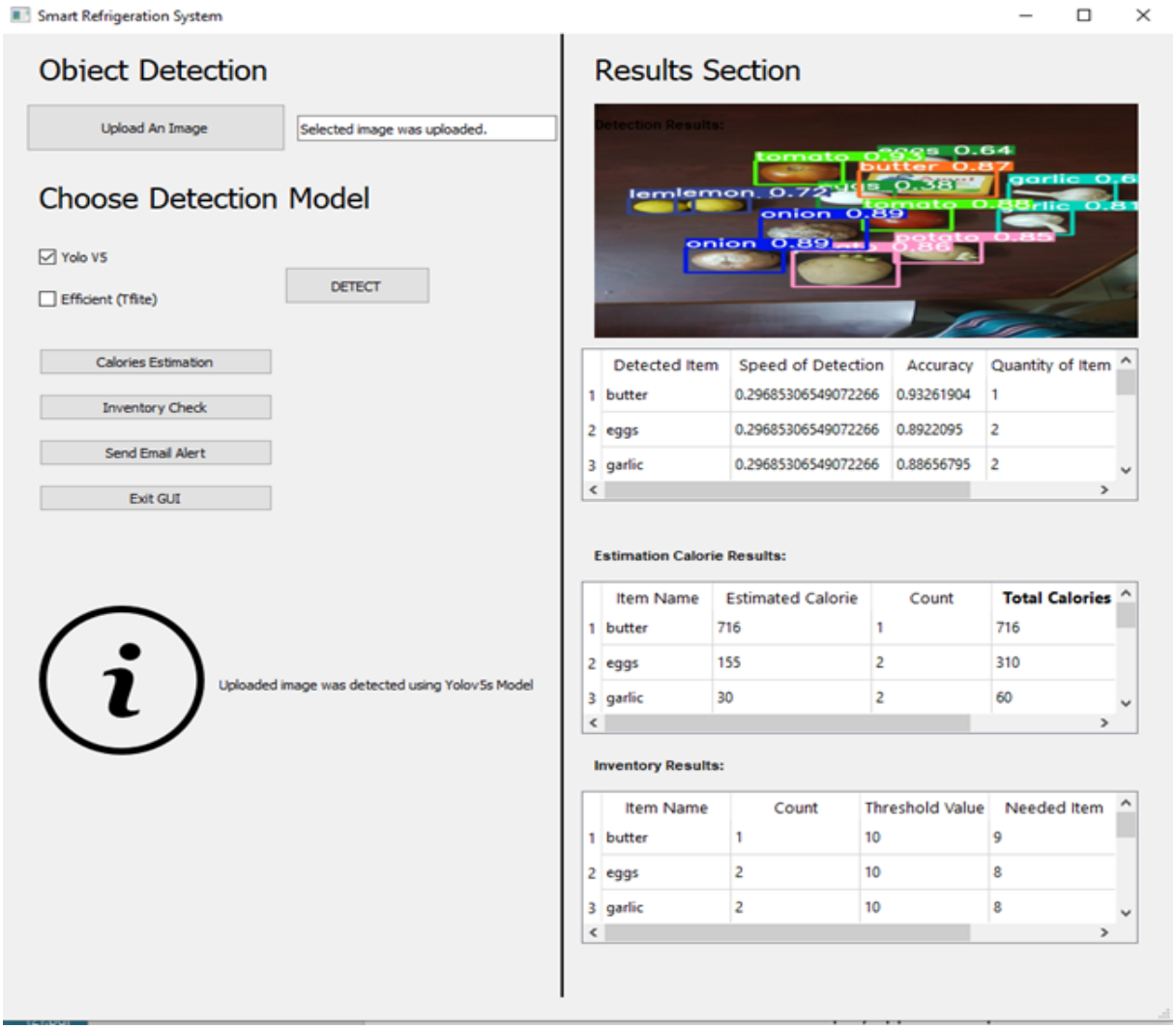


Figure 5.15: EfficientDet1 detected image running on GUI with filled table entries (case-3)

Yolov5s model results :

Now de-select the EfficientDet1 box.
Select by clicking another model box (Yolov5s) and load the same image from the local computer (in our Case-3 image). Once the image is loaded, detection is performed. The app again fills up the detected image and the entries of all three tables. The output of the detected model for Yolov5s is shown in Figure 5.16.

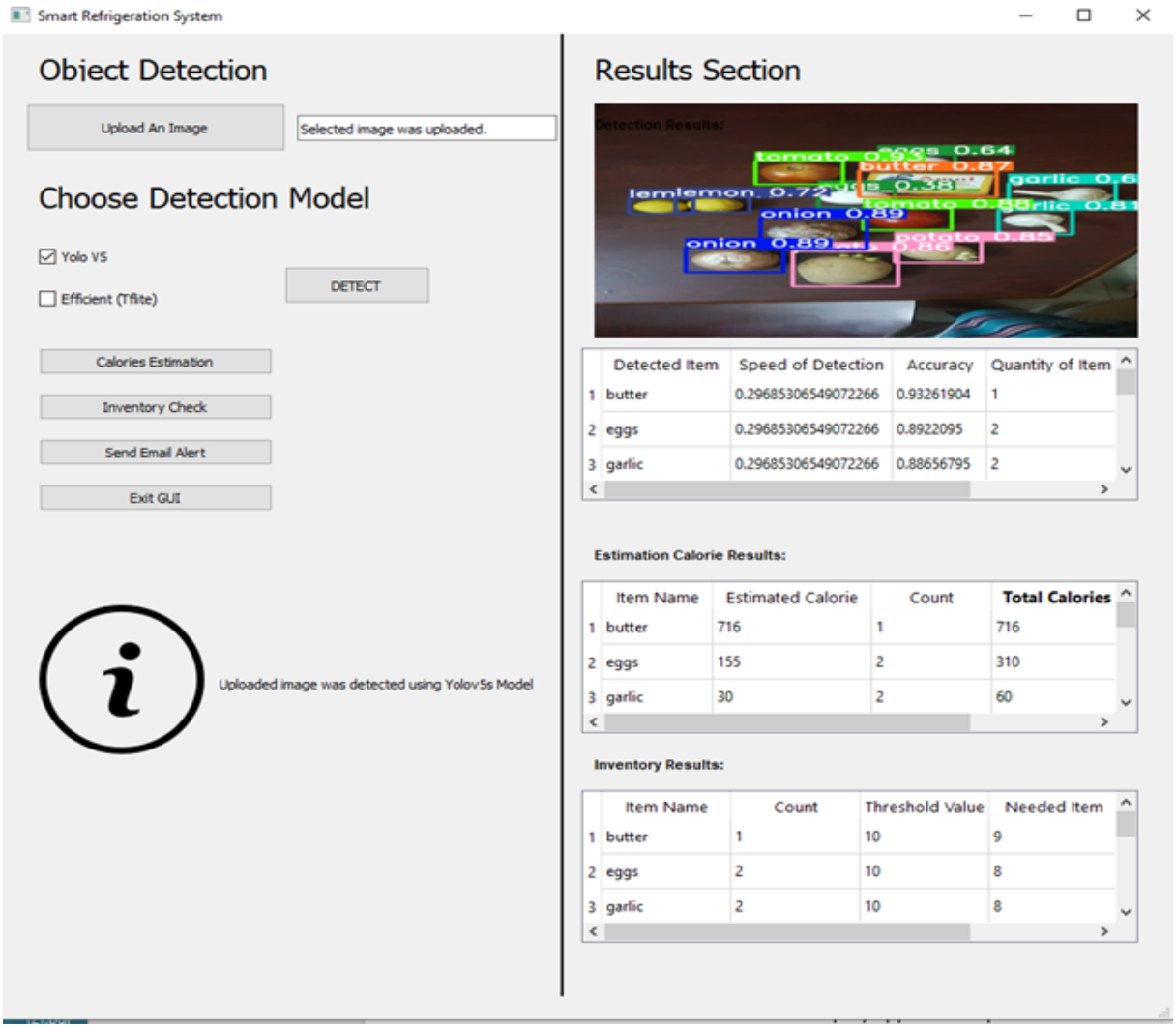


Figure 5.16: Yolov5s detected image running on GUI with filled table entries (case-3)

Similarly, Figure 5.15 and 5.16 for case-3 show how both models correctly display the detected images, speed, detection items, and their quantity in GUI (upper table). Even the estimation of the total number of calories (middle table) and the items

inventory (bottom table) for each class are completed accurately. This shows that all GUI functions of object recognition testing are pretty successful.

5.4 Interpretation of tables results in GUI and an automatic alert system

Though we have shown above only two cases (2 and 3) for object detection results in GUI, we have run all four cases to interpret the table results in each case. Case-1 and Case-4 results are not shown above. It would be interesting to compare the GUI detection results with corresponding results run on Google colab GPU. The local computer Processor is Intel ® Core ™ i5-1035G1 CPU@1.00GHz, whereas the Google Colab GPU processor is Tesla T4, 15110MiB.

1. **Speed detection values (upper table) in GUI**
   Though the accuracy values of all cases are very similar, the detection speed running on Google colab is much smaller than that of the local computer for both models (EfficientDet1 and Yolov5s). Speed comparison results on two platforms are shown in Table 5.2.

Table 5.2: Speed comparison on Google colab (GPU) and local computer (intel Processor)

CASE No.	**Detection Speed - Google Colab GPU (Tesla T4, 15110MiB)**		**Detection Speed - Local Machine (i5-1035G1 CPU@1.00GHz)**	
	EfficientDet1	**Yolov5s**	**EfficientDet1**	**Yolov5s**
CASE 1	992 ms	9.60 ms	6.32 s	0.35 s
CASE 2	880 ms	14.4 ms	5.88 s	0.29 s
CASE 3	929 ms	9.20 ms	5.79 s	0.32 s
CASE 4	937 ms	9.10 ms	5.05 s	0.33 s

It is evident from Table 5.2 that speed detection on Google colab GPU for both models is in ms (average 935 ms for EfficientDet1 and 10.6 ms for Yolov5s). It is very impressive. In the case of the local computer, the detection time is larger for both models (5.76 s for EfficientDet1 and .325 s for Yolov5s). However, Yolov5s is more suitable for running on mobile devices such as Raspberry Pi, laptop, or

local Computers, as their detection time is shorter than 1s, make them suitable for real-time applications. Note that the detection time of EfficientDet1 models is also in secs, though somewhat larger than Yolov5s, making it also suitable to run on mobile devices.

2. **Calorie estimation (middle table) in GUI**
   Keeping track of calorie estimation of items present in the refrigerator is very helpful for diet-conscious people to assess their daily calorie intake. In view of this, the total calories of items in all four cases are calculated and are shown in Table 5.3.

Table 5.3: Total Calories estimation in all four cases

CASE 1			
Item Name	**Estimated Calories**	**Count**	**Total Calories**
Eggs	155	2	310

CASE 2			
Item Name	**Estimated Calories**	**Count**	**Total Calories**
Paprika	282	1	282
Chocolate_drink	83	1	83
Orange_juice	44	1	44
Cereal	379	1	379
Orange	138	1	138
Noodles	100	1	100
Sponge_opl	60	1	60
Grape_juice	346	1	346
Sausages	536	1	536
potato_chips	52	1	52
apple	504	1	504
crackers	100	1	100
Scrubby	0	2	0

CASE 3			
Item Name	**Estimated Calories**	**Count**	**Total Calories**
Potato	76	2	152
Garlic	30	2	60
Butter	716	1	716
Lemon	28	2	56
Tomato	17	2	34
Onion	39	2	78
Eggs	155	2	310

CASE 4			
Item Name	**Estimated Calories**	**Count**	**Total Calories**
Potato	76	2	152
Garlic	30	2	60
Butter	716	1	716
Lemon	28	2	56
Tomato	17	2	34
Onion	39	2	78
Eggs	155	2	310

Table 5.3 shows the total calories calculated for each class whenever image detection is performed. The Calories results are also communicated to the user through his e-mail by an alert system. User can decide how frequently he needs an update of calorie estimation of their refrigerator items daily or at specific intervals.

3. **Items left in the refrigerator and a shopping list of needed items (lower table) in GUI**

   The refrigerator user is very keen to know the status of his refrigerator items present each day or at a specific interval so that he can decide when to go for his next shopping. Given this, the contents of the lower table in GUI are very advantageous to monitor the inventory of refrigerator items and also to know the shopping list of needed items for purchasing or placing an order based on the threshold values defined in the app. Whenever the image is subjected to perform detection, this table is generated and communicated to the refrigerator user through email. A sample of this table for all four cases is shown in Table 5.4.

Table 5.4: Items left in the refrigerator (First column) and a shopping list of needed items (Last column)

CASE 1			
Item Name	**Count**	**Threshold Value**	**Needed Item**
Eggs	2	5	3

CASE 2			
Item Name	**Count**	**Threshold Value**	**Needed Item**
Paprika	1	3	2
Chocolate_drink	1	2	1
Orange_juice	1	1	0
Cereal	1	2	1
Orange	1	5	4
Noodles	1	3	2
Sponge_opl	1	3	2
Grape_juice	1	1	0
Sausages	1	2	1
potato_chips	1	2	1
apple	1	5	4
crackers	1	3	2
Scrubby	2	3	1

CASE 3			
Item Name	**Count**	**Threshold Value**	**Needed Item**
Potato	2	5	3
Garlic	2	2	0
Butter	1	1	0
Lemon	2	5	3
Tomato	2	5	3
Onion	2	4	2
Eggs	2	5	3

CASE 4			
Item Name	**Count**	**Threshold Value**	**Needed Item**
Potato	2	5	3
Garlic	2	2	0
Butter	1	1	0
Lemon	2	5	3
Tomato	2	5	3
Onion	2	4	2
Eggs	2	5	3

The first column in Table 5.4 shows the number of items present in the refrigerator in each case, whereas the last column indicates the number of items running low in the refrigerator, and this constitutes the shopping list of needed items. The last column is calculated by taking the difference between threshold values and the items present for each class.

5.5 Summary

- In this chapter, we have compared the object detection results of two state-of-the-art models (EfficientDet1 and Yolov5s) on home refrigerator items. Both models are found to be very successful in detecting the items correctly.
- Average speed detection on Google colab GPU (Tesla T4, 15110MiB) for Yolov5s is about 10.6 ms, whereas for EfficientDet1 is 935 s which makes both smaller models suitable to run on mobile devices such as Raspberry Pi, laptops, and desktops.
- The average precision values of Yolov5s is around 0.90 (90%) and for Efficient-Det1 is 0.95 (95%).
- Utilizing the effectiveness of these two smaller models, a tool is developed to automate the object detection process on mobile devices.
- An automatic object detection is executed instantly for any given image of refrigerator items, and total calorie estimation, inventory of detected items, and the items running out in the refrigerator are simultaneously displayed in a Graphical user interface (GUI), built based on PyQt5 library.
- Though the detection speed on mobile devices is smaller than Google colab GPUs, still, both models' run times are around 1s which makes these models suitable for real-time applications.
- Monitoring Calories estimation for diet-conscious people and keeping control of inventory of refrigerator items are useful features of our app.

Chapter 6

Conclusion and Future Work

6.1 Summary and Contributions

In this research work, we investigated two object detection techniques EfficientDet1 and Yolov5s for running the smaller models on mobile devices such as Raspberry Pi as well as laptops or desktops. Their model architecture, source codes, and training procedure of object classes for food items in home refrigerators are discussed. The model performances are compared on selected image data sets from the internet. The general statistics of the important parameters for these single-stage detector models are presented in Table 6.1.

Table 6.1: Showing the comparison of important parameters for EfficientDet1 and Yolov5s

Sr. No.	**Parameters**	**Single Stage Detector**	**Single Stage Detector**
1	Model Name	Efficientdet1	Yolov5s
2	Model Training Time	7 hrs (100 epochs)	0.62 hrs (300 epochs
3	Mean Average Precision	0.95 (95%)	0.90 (90%)
4	Inference Time (Google Colab GPU)	935 ms/image	10.6 ms/image
5	Inference Time (Local Computer)	5.70 s/image	0.33 s/image
6	Speed	Fast	Faster

We conclude that the model training time and inference detection time on Google

colab cluster GPUs are much smaller for Yolov5s than the EfficientDet1. However, taking into account the high percentage of precision and the processing times for detecting objects being less than 1s, we conclude that both detection models are highly suitable for automating viable detection systems in refrigerators. We provide a novel and automatic system for recognizing objects of interest, with total calorie estimation and developing an alert system on mobile devices for end users.
An automatic object detection system to run on mobile devices is developed and tested through a graphical User Interface (GUI), designed with the help of the Python library PyQt5. Though this system will work on Raspberry Pi or any mobile device, we have shown its implementation, running, and performance on laptops or desktops. The performance results of all functions on GUI, such as instant object detection in an image, total calorie estimations, inventory of refrigerator items, and the shopping list of items running low in refrigerators, are displayed correctly on GUI and prove to be highly beneficial for end users, in the development of future endeavors in the design of refrigerators.

6.2 Future work

With technological advancements, the research and development of smart homes have reached a new level. Our app is a new approach to making human life easier. As we have seen, using two detection models in GUI makes this app user-friendly.
Since the automatic object detection in GUI in our research project is successful from the sample and training images taken from the internet, it would be interesting to install a ready-made camera at the refrigerator door. This can provide instant images anytime through a wi-fi module on a user's mobile with Raspberry Pi or Android applications.
The advantage of using a built-in camera will be to capture the instant image of refrigerator fruits and vegetables at any time. Also, getting email alerts on the devices where our app is installed. Once our app gets any image, detection of either Yolov5s or EfficientDet1 will be performed automatically. The detected image objects, along with the total calorie estimations, inventory of refrigerator items, and the shopping list of items running low in refrigerators, will be displayed correctly in GUI. In addition to the display, an automatic alert will be generated and sent to the refrigerator user e-mail containing all the table entries and detected objects in the image. This will make smooth communication between the refrigerator and the user, making day-to-

day life easier and may reduce carbon emissions because of the unnecessary re-driving car to the grocery store.

The proposed framework can be added to any already existing refrigerators. The framework uses open-source codes and low-cost hardware design and will make a significant impact in the future.

Further, Yolov5s being a smaller model, is fast enough, accurate, and lightweight to be easily deployed on mobile devices, which will open a new feature of automatic object detection in real-time applications using the built-in camera in refrigerators.

Another advantage of the Yolo series is that it is continuously a part of research and development, and newer advances in this series are continuously being advertised. Already Yolov6, Yolov7, and Yolov8 have emerged in recent months. We hope that this research work will keep going in the near future.

As of January 2023, the latest version of YOLO, which is the state-of-the-art model Yolov8 has been launched, claiming advancements in structure and architectural changes with faster and better accuracy results. It would be interesting to incorporate this newer version of Yolov8 in our app in the near future [85][86].

Bibliography

[1] César Giovany Pachón, Javier Orlando Pinzón, and Robinson Jimenez Moreno. Product detection system for home refrigerators implemented though a region-based convolutional neural network. *International Journal of Applied Engineering Research*, 13(12):10381–10388, 2018.

[2] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[3] Kirti Agarwal. Object detection in refrigerators using tensorflow. *M.Sc Thesis, University of Victoria*, 2018.

[4] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in Neural Information Processing Systems*, 28, 2015.

[5] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European Conference on Computer Vision*, pages 21–37. Springer, 2016.

[6] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016.

[7] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7263–7271, 2017.

[8] Yongming Wei, Jicheng Quan, and Yuuqingyang Hou. Aerial image location of unmanned aerial vehicle based on yolo v2. *Laser & Optoelectronics Progress*, 54(11):111002, 2017.

[9] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[10] Liquan Zhao and Shuaiyang Li. Object detection algorithm based on improved yolov3. *Electronics*, 9(3):537, 2020.

[11] Mingxing Tan, Ruoming Pang, and Quoc V Le. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10781–10790, 2020.

[12] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international Conference on Computer Vision*, pages 2961–2969, 2017.

[13] Tsung-Yi Lin. Y, dollár p, girshick r, he k, hariharan b, belongie s. feature pyramid networks for object detection. In *Proceedings-30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, volume 2017, pages 936–944, 2017.

[14] Jacob Solawetz. Yolov5 new version-improvements and evaluation. *Roboflow.https://blog.roboflow.com/yolov5-improvements-and-evaluation/*, 2020.

[15] Glen Jocher. Yolov5 dcumentation. *Ultralytics, https://docs.ultralytics.com/yolov5*, 2020.

[16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[17] Peter Goldsborough. A tour of tensorflow. *arXiv preprint arXiv:1610.01178*, 2016.

[18] Ladislav Rampasek and Anna Goldenberg. Tensorflow: biology's gateway to deep learning? *Cell systems*, 2(1):12–14, 2016.

[19] Google Team developers. Tensorflow lite api reference. *Google. https://www.tensorflow.org/lite/api$_d$ocs*, 2022.

[20] Simplilearn. Top 10 deep learning frameworks you should know in 2022. *Simplilearn, https://www.simplilearn.com/tutorials/deep-learning-tutorial/deep-learning-frameworkstensorflow*, 2022.

[21] Facundo Bre, Juan M Gimenez, and Víctor D Fachinotti. Prediction of wind pressure coefficients on building surfaces using artificial neural networks. *Energy and Buildings*, 158:1429–1441, 2018.

[22] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[23] Elisha Odemakinde. Everything about mask r-cnn: A beginner's guide. *https://viso.ai/deep-learning/mask-r-cnn/*, 2022.

[24] Stanford-Spring 2022. Cs231n convolutional neural networks for visual recognition. *http://cs231n.github.io/convolutional-networks/*, Spring-2022.

[25] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[27] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI Conference on Artificial Intelligence*, 2017.

[28] Yugesh Verma. R-cnn vs fast r-cnn vs faster r-cnn a comparative guide. *https://analyticsindiamag.com/r-cnn-vs-fast-r-cnn-vs-faster-r-cnn-a-comparative-guide/*, 2022.

[29] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(9):1904–1916, 2015.

[31] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international Conference on Computer Vision*, pages 1440–1448, 2015.

[32] Henrik Eklund. Object detection: Model comparison on automated document content interpretation-a performance evaluation of common object detection models, 2019.

[33] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. *Advances in Neural Information Processing Systems*, 29, 2016.

[34] Elisha Odemakinde. Everything about mask r-cnn A beginner's guide. *https://viso.ai/deep-learning/mask-r-cnn/*, 2022.

[35] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[36] Yi Li, Haozhi Qi, Jifeng Dai, Xiangyang Ji, and Yichen Wei. Fully convolutional instance-aware semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2359–2367, 2017.

[37] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international Conference on Computer Vision*, pages 2961–2969, 2017.

[38] Ashwani Kumar, Zuopeng Justin Zhang, and Hongbo Lyu. Object detection in real time based on improved single shot multi-box detector algorithm. *EURASIP Journal on Wireless Communications and Networking*, 2020(1):1–18, 2020.

[39] Sheping Zhai, Dingrong Shang, Shuhuan Wang, and Susu Dong. Df-ssd: An improved ssd object detection algorithm based on densenet and feature fusion. *IEEE access*, 8:24344–24357, 2020.

[40] G Karimi. Introduction to yolo algorithm for object detection. *S ection. io. https://www. section. io/engineering education/introduction-to-yolo-algorithm-for-objectdetection/(accessed May 7, 2021)*, 2021.

[41] Ayoosh Kathuria. What's new in yolo v3. *Towards Data Science*, 23, 2018.

[42] Yuntao Chen, Chenxia Han, Yanghao Li, Zehao Huang, Yi Jiang, Naiyan Wang, and Zhaoxiang Zhang. Simpledet: A simple and versatile distributed framework for object detection and instance recognition. *J. Mach. Learn. Res.*, 20(156):1–8, 2019.

[43] Lu Tan, Tianran Huangfu, Liyao Wu, and Wenying Chen. Comparison of retinanet, ssd, and yolo v3 for real-time pill identification. *BMC Medical Informatics and Decision Making*, 21(1):1–11, 2021.

[44] DS Viraktamath, Pratiksha Navalgi, and Ambika Neelopant. Comparison of yolov3 and ssd algorithms. *Int. J. Eng. Res. Technol*, 10:1156–1160, 2021.

[45] Akansha Bathija and Grishma Sharma. Visual object detection and tracking using yolo and sort. *International Journal of Engineering Research Technology*, 8(11), 2019.

[46] César Giovany Pachón, Javier Orlando Pinzón, and Robinson Jimenez Moreno. Product detection system for home refrigerators implemented though a region-based convolutional neural network. *International Journal of Applied Engineering Research*, 13(12):10381–10388, 2018.

[47] Li Liu, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, Xinwang Liu, and Matti Pietikäinen. Deep learning for generic object detection: A survey. *International Journal of Computer Vision*, 128(2):261–318, 2020.

[48] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[49] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2117–2125, 2017.

[50] Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, and Jiaya Jia. Path aggregation network for instance segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8759–8768, 2018.

[51] G Chiasi, Tsung-Yi Lin, and N Le QV. Learning scalable feature pyramid architecture for object detection. In *Proceedings of the IEEE Computer Vision and Pattern Recognition*, pages 7029–7038.

[52] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision*, pages 740–755. Springer, 2014.

[53] Xiangning Chen, Cihang Xie, Mingxing Tan, Li Zhang, Cho-Jui Hsieh, and Boqing Gong. Robust and accurate object detection via adversarial learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16622–16631, 2021.

[54] Jacob Solawetz. Yolov3 versus efficientdet for state-of-the-art object detection. *https://blog.roboflow.com/yolov3-versus-efficientdet-for-state-of-the-art-object-detection/*, 2020.

[55] Jacob Solawetz and Joseph Nelson. Training efficientdet object detection model with a custom datase. *https://blog.roboflow.com/training-efficientdet-object-detection-model-with-a-custom-dataset/*, 2020.

[56] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[57] Jacob Solawetz. What is yolov4? a detailed breakdown. *https://blog.roboflow.com/a-thorough-breakdown-of-yolov4/*, 2020.

[58] Chien-Yao Wang, Hong-Yuan Mark Liao, Yueh-Hua Wu, Ping-Yang Chen, Jun-Wei Hsieh, and I-Hau Yeh. Cspnet: A new backbone that can enhance learning capability of cnn. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 390–391, 2020.

[59] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017.

[60] Songtao Liu, Di Huang, and Yunhong Wang. Learning spatial fusion for single-shot object detection. *arXiv preprint arXiv:1911.09516*, 2019.

[61] Qijie Zhao, Tao Sheng, Yongtao Wang, Zhi Tang, Ying Chen, Ling Cai, and Haibin Ling. M2det: A single-shot object detector based on multi-level feature pyramid network. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9259–9266, 2019.

[62] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[63] Zhaohui Zheng, Ping Wang, Wei Liu, Jinze Li, Rongguang Ye, and Dongwei Ren. Distance-iou loss: Faster and better learning for bounding box regression. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 12993–13000, 2020.

[64] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European Conference on Computer Vision*, pages 740–755. Springer, 2014.

[65] Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union: A metric and a loss for bounding box regression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 658–666, 2019.

[66] Brad Dwyer. Roboflow and ultralytics partner to streamline yolov5 mlops,. *https://blog.roboflow.com/yolov5-partnership/*, 2021.

[67] Joseph Nelson and Jacob Solawetz. Yolov5 is here : State-of-art object detection at 140 fps. *https://blog.roboflow.com/yolov5-is-here/*, 2020.

[68] Phoebe Parker. Labelling images for object detection with labellmg. *https://www.altisconsulting.com/insights/labelling-images-for-object-detection-with-labelimg/*, 2022.

[69] Gaudenz Boesch. Labellmg for image annotation. *https://viso.ai/computer-vision/labelimg-for-image-annotation/*, 2022.

[70] Brad Dwyer. Getting started with roboflow. *https://blog.roboflow.com/getting-started-with-roboflow/*, 2021.

[71] Prashanth Saravanan. Understanding loss functions in machine leaning. *https://www.section.io/engineering-education/understanding-loss-functions-in-machine-learning/*, 2021.

[72] Khyati Mahendru. A detailed guide to 7 loss functions for machine learning algorithms with python code, 2019.

[73] Jason Brownlee. Understand the impact of learning rate on neural network performance. *Machine Learning Mastery*, pages 1–27, 2019.

[74] Jacob Solawetz and Joseph Nelson. How to train yolov5 on a custom dataset. */urlhttps://blog.roboflow.com/how-to-train-yolov5-on-a-custom-dataset/*, 2020.

[75] Glenn Jocher. Train custom data. *https://github.com/ultralytics/yolov5/wiki/Train-Custom-Data/*, 2022.

[76] Yolov5s architecture summary. *https://docs.ultralytics.com/tutorials/architecture-summary/44*.

[77] Precision and recall. *https://en.wikipedia.org/wiki/Precision*$_a$*nd*$_r$*ecall*.

[78] Rafael Padilla, Sergio L Netto, and Eduardo AB Da Silva. A survey on performance metrics for object-detection algorithms. In *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*, pages 237–242. IEEE, 2020.

[79] Kiprono Elijah Koech. Object detection metrices with worked example. *https://towardsdatascience.com/on-object-detection-metrics-with-worked-example-216f173ed31e*, 2020.

[80] Rafael Padilla, Wesley L Passos, Thadeu LB Dias, Sergio L Netto, and Eduardo AB Da Silva. A comparative analysis of object detection metrics with a companion open-source toolkit. *Electronics*, 10(3):279, 2021.

[81] Timothy C Arlen. Understanding the map evaluation metric for object detection. *Source:¡ https://medium. com/@ timothycarlen/understanding-themap-evaluation-metric-for-object-detection-a07fe6962cf3*, 2018.

[82] Common objects in context. *https://cocodataset.org/detection-eval/*.

[83] Jan Hosang, Rodrigo Benenson, Piotr Dollár, and Bernt Schiele. What makes for effective detection proposals? *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(4):814–830, 2015.

[84] Kukil. Mean average precision (map) in object detection. *https://learnopencv.com/mean-average-precision-map-object-detection-model-evaluation-metric/*, 2022.

[85] Glenn Jocher and Ayush Chaurasia. Ultralytics yolov8, the state-of-the-art yolo model. *https://github.com/ultralytics/ultralytics/*, 2023.

[86] Chinmay Bhalerao. Yolo v8! the real state-of-the-art? my experience experiment related to yolo v8. *https://medium.com/mlearning-ai/yolo-v8-the-real-state-of-the-art-eda6c86a1b90/*, 2023.

Appendix A

Object Detection Models and GUI Source Code HTTP Links

- **EfficientDet1: HTTP Link for downloading source code and data**
- **Yolov5s: HTTP Link for downloading source code and data**
- **Graphical User Interface (GUI): HTTP Link for downloading source code folder (Yolov5-Rohit-Final)**
  or
  HTTP Link for downloading source code of Zip file (Rohit_GUI_Code.zip)