# A Mobile Application for Dish Detection in Food Platters by Deploying an Object Detection Computational Protocol


by
**Nitesh Narwade**


**Under the supervision of**
**Professor. Ganesh Bagler**

**Master of Technology, Computational Biology**

**Center for Computational Biology Indraprastha Institute of Information Technology - Delhi**


**June, 2023**

# Certificate

This is to certify that the thesis titled *"A Mobile Application for Dish Detection in Food Platters by Deploying an Object Detection Computational Protocol"* being submitted by **Nitesh Narwade** to the Indraprastha Institute of Information Technology Delhi, for the award of the Master of Technology is an original research work carried out by him under my supervision. In my opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree/diploma.

June 2023

Professor. Ganesh Bagler

Department of Computational Biology
Indraprastha Institute of Information Technology Delhi
New Delhi 110 020

# Acknowledgements

This entire thesis was possible due to help from a multitude of people. The biggest contributor is my supervisor, Dr. Ganesh Bagler, without whose help, guidance, and patience this thesis would have been completed. It's through his advice that this work was able to reach this level. Throughout this work members of the Reggen lab were always there to lend a hand and help me in exploring uncharted areas of knowledge and I would also like to extend my heartfelt gratitude to my teachers at IIIT Delhi, my seniors, and my batchmates. I would also like to thank the Department of Biotechnology, the Government of India, for the student fellowship and support to the MTech (Computational Biology) program. Last but not the least, I express my gratitude to my family members who were always there as a pillar of support.

# Abstract

The training and deployment of YOLOv4 and YOLOv5 models in an Android application play a vital role in achieving real-time object detection on mobile devices. This abstract overviews the critical steps in training and integrating these models into an Android application. The training process for YOLOv4 and YOLOv5 begins with collecting and annotating a labelled dataset, where bounding box coordinates and object classes are assigned to the images. Deep learning techniques are then applied to train the models, optimizing their parameters through iterative processes to improve object detection accuracy. PyTorch and TensorFlow are commonly employed for training YOLOv4 and YOLOv5 models offer comprehensive support for model architecture design, data preprocessing, and optimization algorithms. Once trained, the models must be deployed in an Android application. This involves converting the trained models into a format compatible with mobile devices like TensorFlow Lite. TensorFlow Lite enables the efficient execution of deep learning models on Android smartphones and other resource-constrained devices. The loaded YOLOv4 or YOLOv5 model is utilized in the Android application for real-time object detection. The application captures frames from the device's camera, feeds them into the model, and receives predictions through bounding boxes and class labels. These predictions are then superimposed on the camera feed, providing the user a real-time object detection experience. To optimize the performance of the models in the Android application, techniques like model quantization and hardware acceleration can be employed. Model quantization reduces memory and computation requirements, leading to faster inference on mobile devices. Hardware acceleration, such as utilizing the GPU, further enhances the speed and efficiency of the object detection process. In conclusion, training YOLOv4 and YOLOv5 models involve dataset collection, annotation, and deep learning model training, followed by deployment in Android applications using frameworks like TensorFlow Lite. Integrating these models into an Android application enables real-time object detection, enhancing the functionality and usability of various applications, including augmented reality and image recognition.

# Contents


**1 INTRODUCTION AND MOTIVATION** **7**

**2 LITERATURE SURVEY** **9**

2.1 Object Detection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2 Object Detection With OpenCV . . . . . . . . . . . . . . . . . . . . . 10
2.3 YOLO And Similar Models . . . . . . . . . . . . . . . . . . . . . . . 10
2.3.1 YOLO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.3.2 RCNN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.3.3 SSD . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.4 Deep Learning (Classification) . . . . . . . . . . . . . . . . . . . . . . 13
2.4.1 CNN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.4.2 RNN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

**3 DATA COLLECTION AND DATA PREPARATION FOR DIFFERENT YOLO MODELS** **15**

3.1 Food Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.1.1 Dataset Prepration . . . . . . . . . . . . . . . . . . . . . . . . 16

**4 IMAGE DETECTION FOR THE INDIAN DISHES** **18**

4.1 YOLO Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.2 Training YOLOV4 Model . . . . . . . . . . . . . . . . . . . . . . . . 21
4.3 Training YOLOv5 Model . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.3.1 YOLOV5 Nano Model . . . . . . . . . . . . . . . . . . . . . . 22
4.3.2 YOLOv5 Small Model . . . . . . . . . . . . . . . . . . . . . . 23
4.3.3 YOLOv5 Medium Model . . . . . . . . . . . . . . . . . . . . . 23
4.3.4 YOLOv5 Large Model . . . . . . . . . . . . . . . . . . . . . . 24
4.3.5 YOLOv5 Extra Large Model . . . . . . . . . . . . . . . . . . . 25
4.4 Performance Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . 25

**5 MOBILE APPLICATION DEVELOPMENT** **29**
5.1 WhatDish Application Introduction . . . . . . . . . . . . . . . . . . . . . . 29
5.2 Application UI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.2.1 Camera Activity . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.2.2 User Activity . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
5.3 Deployment Of YOLO Model In An Android Application . . . . . . . . . 37

**6 CONCLUSION** **39**

# List of Figures

4.1 YOLO Architecture. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
4.2 Performance results of YOLO models . . . . . . . . . . . . . . . . . . . . 26
4.3 Graphical Representation of Performance results for YOLOv5 models . 28

5.1 WhatDish Android application with no progress in progress bar . . . . 31
5.2 WhatDish Android application with progress in progress bar . . . . . . 32
5.3 WhatDish Android application with detected food information stored in table . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
5.4 WhatDish Android application user Activity . . . . . . . . . . . . . . . . 36

# Chapter 1

# INTRODUCTION AND MOTIVATION

Object detection has emerged as a crucial task in computer vision, with numerous applications across various domains. In recent years, the YOLO (You Only Look Once) models, specifically YOLOv4 and YOLOv5, have gained significant attention for their outstanding performance in real-time object detection tasks. Integrating these advanced models into an Android application focused on food detection and recognition offers excellent potential for enhancing user experiences, promoting healthy eating habits, and aiding dietary monitoring. This thesis aims to develop an object detection Android application that utilizes the YOLOv4 and YOLOv5 models to accurately detect and classify food classes in real time. The application aims to assist users in making informed dietary choices, providing nutritional information based on the identified food items. By leveraging the power of YOLOv4 and YOLOv5, the Android application can directly achieve fast and accurate food detection on the device. An Android application eliminates the need for internet connectivity or server-side processing, enabling users to receive instant feedback and information about the food items they encounter. One of the primary advantages of integrating YOLOv4 and YOLOv5 into the Android application is their capability to handle complex scenes, occlusions, and varying object scales. These models have shown remarkable performance in object detection tasks, including challenging scenarios, making them well-suited for food detection applications that involve diverse food classes and different object sizes. The Android application's focus on food detection and recognition has several practical implications. It can empower users to make healthier dietary choices by providing real-time information about various food items' nutritional content. Additionally, the application can offer personalized calorie goal setting. Moreover, integrating YOLOv4 and YOLOv5 into the Android applica-

tion presents research and exploration opportunities. This thesis aims to evaluate the performance of these models, specifically in the context of food detection and recognition on mobile platforms. It opens avenues for investigating optimizations, fine-tuning techniques, and architectural modifications to enhance the models' accuracy and efficiency for food-related tasks. In conclusion, integrating YOLOv4 and YOLOv5 models into an Android food detection and recognition application holds immense potential. By leveraging the strengths of these models, the application can provide real-time, accurate, and context-aware information about various food classes. The thesis seeks to contribute to advancing mobile vision applications in food detection and support users in making healthier dietary choices.

# Chapter 2

# LITERATURE SURVEY

## 2.1 Object Detection

Object detection involves identifying and localising objects within digital images or video frames. Its significance extends to various domains, such as autonomous driving, surveillance systems, robotics, image and video analysis, and augmented reality. Object detection aims to accurately and efficiently identify multiple objects within an image or video. These objects can range from everyday items like cars, pedestrians, and animals to domain-specific objects in medical imaging or industrial inspection areas. Object detection surpasses traditional image classification, which assigns a single label to an entire image. Instead, it aims to identify and locate individual objects by drawing bounding boxes around them. Additionally, object detection often involves assigning a label or class to each detected object, providing further contextual information about its identity. The progress in object detection algorithms has been substantial in recent years, primarily driven by annotated datasets. Deep learning models, particularly convolutional neural networks (CNNs), have revolutionised object detection by directly learning hierarchical representations of visual features from raw image data. Several widely adopted object detection frameworks have emerged, including Faster R-CNN, YOLO (You Only Look Once), and SSD (Single Shot MultiBox Detector). These frameworks utilise techniques like region proposal methods, anchor boxes, and feature pyramid networks to achieve accurate and efficient object detection. Object detection finds practical applications in autonomous driving by identifying and tracking vehicles, pedestrians, and traffic signs to ensure the safety and efficiency of self-driving cars. Surveillance systems enable real-time detection of suspicious activities or individuals in video streams. It also plays a crucial role in retail analytics by counting and tracking products on store shelves or analysing customer behaviour. Continued advancements in

computer vision are leading to increasingly robust, accurate, and faster object detection algorithms, creating new possibilities across a wide range of applications. Researchers and developers continuously explore innovative architectures, optimisation techniques, and training strategies to enhance the performance and efficiency of object detection algorithms.[1]

## 2.2 Object Detection With OpenCV

OpenCV stands for open-source computer vision. It is the library introduced by Intel for image and video analysis. OpenCV library has more than 2500 optimised algorithms. This algorithm is used all over the world. This OpenCV library is used primarily by graduate students and professors for research in the computer vision area. The OpenCV library functions are mainly made in C and C++. After a few years, they also introduced all these libraries in Python. In the YOLO model, python libraries have been used. OpenCV is used in Object Tracking, feature detection, image filtering, image transformation, face detection, and face recognition. In the YOLO model, OpenCV is used for image detection. With the help of OpenCV, we can load the image from the source file. We can convert the image into a grayscale format and create bonding boxes to track or detect the object. OpenCV is used to reduce the noise from the image before giving it to the model. OpenCV used a CUDA-based GPU interface which NVIDIA introduced. To use multiple-thread parallel programming is essential. Users can do parallel programming using CUDA-based GPU in their computer vision model. Processing thousands of images and doing quick work is impossible with single CPU processing. Hence CUDA-based GPU comes into the picture, which divides processing work on different threads and does quick computation. [2]

## 2.3 YOLO And Similar Models

### 2.3.1 YOLO

Introduces YOLO (You Only Look Once), a groundbreaking framework renowned for its speed and accuracy in object detection. YOLO revolutionized the field by proposing a unified approach directly predicting bounding boxes and class probabilities in a single convolutional neural network (CNN) pass. The core concept of YOLO is to treat object detection as a regression problem, where the CNN simultaneously predicts the coordinates of bounding boxes and the probabilities of object classes within an image. Unlike previous methods that relied on region proposal techniques, YOLO eliminates

the need for time-consuming region proposal stages by directly predicting object locations and categories. Images are divided into the grid by YOLO, with each grid cell responsible for predicting a fixed number of bounding boxes and their associated class probabilities. Each bounding box prediction consists of four coordinates (x, y, width, height) and includes a confidence score representing the likelihood of containing an object. YOLO employs multiple convolutional layers with different spatial resolutions to enable predictions at different scales. Lower-resolution feature maps capture global context and larger objects, while higher-resolution feature maps focus on finer details and minor things. This multiscale approach allows YOLO to detect objects of various sizes while maintaining a balance between localization accuracy and computational efficiency. During training, YOLO utilizes a loss function that combines localization loss (related to the accuracy of bounding box coordinates) and classification loss (related to the accuracy of predicted object classes). This loss function encourages precise localization and penalizes incorrect predictions, enabling the network to learn robust representations for object detection. The experimental results demonstrate that YOLO achieves impressive real-time object detection accuracy. It surpasses previous methods in terms of both speed and precision on widely-used datasets like PASCAL VOC and COCO. Despite its success, YOLO does have some limitations. It may struggle with detecting small objects or objects with significant aspect ratio variations. Additionally, it may produce false positives and need help in accurately localizing densely packed objects due to the fixed grid structure. Nevertheless, the YOLO framework represents a significant milestone in real-time object detection. Its speed and accuracy have made it popular for various applications, including video surveillance, autonomous driving, and robotics. YOLO has also inspired subsequent variants such as YOLOv2, YOLOv3, and YOLOv4, further enhancing its strengths and addressing its limitations.[3]

### 2.3.2 RCNN

The R-CNN (Region-based Convolutional Neural Network) was introduced in 2014 by Girshick et al. as a groundbreaking object detection framework. It combined deep learning with region proposal techniques, revolutionizing the field of object detection. The R-CNN approach involves several key steps. First, a selective search algorithm generates region proposals within an image. These proposals group pixels based on image features to capture potential object locations. Next, each region proposal is individually extracted and transformed into a fixed size to match the requirements of a pre-trained CNN, typically based on architectures like AlexNet or VGGNet. The CNN extracts feature for each region. The extracted features are then used for region classification and localization. Linear Support Vector Machines (SVMs) are employed to

classify the content of each region proposal into different object categories. Additionally, bounding box regression is performed to refine object localization. During training, a large dataset with labelled object instances is used. The CNN is pre-trained on a large-scale image classification task (e.g., ImageNet) and then fine-tuned specifically for object detection using the R-CNN framework. SVMs and bounding box regressors are trained to classify and localize objects within the proposed regions. At its introduction, the R-CNN framework exhibited significant performance improvements compared to previous object detection methods. It achieved state-of-the-art results on benchmark datasets like PASCAL VOC and MS COCO, demonstrating the effectiveness of deep learning in object detection. However, R-CNN had certain limitations. Processing each region proposal individually made the method computationally expensive and time-consuming, rendering it unsuitable for real-time applications. Additionally, the multi-stage R-CNN pipeline introduced complexity in training and inference. Despite these limitations, R-CNN laid the foundation for subsequent advancements in object detection. It inspired the development of faster and more efficient variants such as Fast R-CNN, Faster R-CNN, and Mask R-CNN, addressing the original R-CNN approach's computational and efficiency issues. Overall, R-CNN pioneered the integration of deep learning and region proposals for object detection, leaving a significant impact on the field and paving the way for further advancements in subsequent years. [4]

### 2.3.3 SSD

The main idea behind SSD is to directly estimate object categories and adjustments to bounding boxes in a single forward pass of a convolutional neural network (CNN). Instead of relying on region proposal methods like previous techniques, SSD eliminates the need for a separate stage to propose regions, resulting in faster detection. The SSD framework divides the input image into default bounding boxes called anchor boxes with different scales and aspect ratios. These anchor boxes are processed through multiple feature maps of varying resolutions in the CNN, enabling the detection of objects of various sizes. Each anchor box is associated with predictions for object classes and adjustments to bounding boxes, allowing for simultaneous classification and localization. To handle objects of different scales, SSD utilizes feature maps from multiple layers of the CNN with different spatial resolutions. This approach enables the detection of objects of various sizes and enhances the overall detection accuracy[5]. Including using convolutional layers to predict class scores and bounding box offsets, implementing default anchor boxes with diverse scales and aspect ratios, and employing a multi-scale feature fusion strategy. Experimental results demonstrate that SSD achieves state-of-the-art object detection on standard datasets such as PASCAL VOC and MS COCO.

It strikes a good balance between accuracy and speed, making it suitable for real-time applications. [6]

## 2.4 Deep Learning (Classification)

Deep learning classification involves various architectures and techniques for categorizing data. Below are commonly used types:

### 2.4.1 CNN

A Convolutional Neural Network (CNN) is an advanced machine learning algorithm widely used for processing images and videos. It takes inspiration from the human visual system's structure and function, particularly the visual cortex's receptive fields. CNNs have different components, including convolutional, pooling, and fully connected layers. The convolutional layers employ filters (also called kernels) that slide over the input image, performing element-wise multiplication and summation to generate feature maps.

These feature maps capture various patterns and characteristics found in the input image. Pooling layers downscale the feature maps, reducing their spatial dimensions while preserving the most critical information. Common pooling operations involve selecting the maximum or average value within each pooling region, known as max pooling and average pooling. Fully connected layers are typically positioned at the network's end and generate predictions based on the learned features. They establish connections between every neuron in the preceding layer and every neuron in the subsequent layer.

This connectivity enables the network to comprehend intricate relationships among components and produces output predictions. Training CNNs involves utilizing large labelled datasets and employing backpropagation. During backpropagation, the network adjusts the weights and biases of its neurons to minimize the disparity between predicted outputs and the actual labels. This training process allows the network to learn distinctive features and generalize effectively to unseen data. CNNs have succeeded significantly in various computer vision tasks, such as image segmentation and generation. They have also been extended and applied to other domains, including computer vision and image recognition, by incorporating additional layers and adapting the architecture to the specific problem.[7]

### 2.4.2 RNN

RNN, or Recurrent Neural Network, is an artificial neural network commonly used in natural language processing (NLP) and sequential data analysis, unlike traditional feedforward neural networks, which process inputs in a single forward pass. The critical characteristic of RNNs is their ability to process sequential data by maintaining an internal memory state. This memory state enables the network to retain and utilize information from previous steps or time points in the sequence while processing subsequent inputs. This makes RNNs well-suited for tasks that involve sequential or temporal dependencies, such as language modelling, speech recognition, machine translation, sentiment analysis, and time series forecasting.

The structure of an RNN includes recurrent connections, which allow the network to share and propagate information across different time steps. At each time step, the RNN combines an input vector with the previous hidden state to produce an output and update the current hidden state. The secret state acts as the memory of the network, carrying information from one step to the next.

However, standard RNNs can suffer from the "vanishing gradient" or "exploding gradient" problem, where gradients either diminish exponentially or grow uncontrollably during backpropagation through time. Variations of RNNs have been developed to address this issue, such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU). LSTM and GRU are RNNs that incorporate specialized memory cells and gating mechanisms, allowing them to capture long-term dependencies more effectively.

These models have become popular in NLP tasks because they can handle vanishing gradients and retain important contextual information over longer sequences. RNNs, particularly LSTM and GRU variants, have significantly advanced the field of NLP and sequential data analysis. They have proven effective in various applications, including language generation, sentiment analysis, machine translation, speech recognition, and more. Researchers and practitioners continue to explore and refine RNN architectures and techniques to improve their performance and address the challenges associated with sequential data processing.[8]

# Chapter 3

# DATA COLLECTION AND DATA PREPARATION FOR DIFFERENT YOLO MODELS

The team of students collected a dataset of approximately 61,000 images from various sources, such as social media platforms like Instagram. These images were used to train the YOLOv4 and YOLOv5 models for object detection. The dataset consisted of 61 food classes, each with a minimum of 1,000 images.

To annotate the images, the team used an AI tool that assisted in the image annotation procedure. With the help of this tool, bounding boxes were placed around the specific areas of the images where the actual food item of the corresponding class was present. This annotation process helps the model learn and recognize the objects of interest by associating the bounding boxes with their respective class labels.

It is worth mentioning that the annotation process was focused on annotating only the relevant parts of the images containing the actual food items. This approach helps provide precise training data for the model, reducing the need for annotating the entire image and improving the efficiency of the annotation process.

## 3.1 Food Classes

The data has trained for 61 food classes. In 20 food classes, Indian bread (chappati, roti), Rasgulla, Biryani, Uttapam, Paneer, Poha, Khichadi, Omelette, Plain Rice, Dal Makhani, Rajma, Poori, Chole, Dal, Sambhar, Papad, Gulab Jamun, Idli, Vada, and Dosa. On extending these classes to 40, Jalebi, Samosa, Pao bhaji, Dhokla, Barfi, Fishcurry, Momos, Kheer, Kachori, Vadapav, Rasmalai, Kalachana, Chaat, Saag, Du-

maloo, Thupka, Khandvi, Kabab, Thepla, Rasam also added. Again to extend it to 61 classes, Appam, Gatte, Kadhi Pakora, Ghewar, Aloo mattar, Prawns, Sandwich, Dahipuri, Haleem, Mutton, Aloo Gobi, Egg Bhurji, Lemon Rice, Bhendi Masala, Matar Mushroom, Gajarka Halwa, Motichoor Ladoo, Ragi Roti, Chicken Tikka, Tandoori Chicken, and Lauki. The first 20 classes (1 to 20 ) are trained on YOLOV4 and YOLOV5 models. The classes from 1 to 40 trained on the YOLOV5 model only. And again, one set of YOLOV5 models trained for classes from 1 to 61.

### 3.1.1 Dataset Prepration

The dataset used 61 classes of food. Each category contains a minimum of 1000 images and corresponding annotation files. These annotation files provide detailed information about the objects present in the photos, including the object's location, width, height, and the specific class it belongs to. In multiple object cases, each object is correctly annotated with its dimensions. To create this dataset, a team of 9 students was involved. They collected images from various social media platforms, such as Instagram, and utilised AI websites to assist in annotation. This dataset aimed to encompass a comprehensive range of Indian dishes, ensuring that various food items were represented. It's important to note that the dataset needs to be prepared according to the model's specific requirements to utilise the YOLO model effectively. This includes appropriately annotating the images with bounding boxes and class labels, ensuring consistency and accuracy throughout the dataset.

#### YOLOV4 Dataset Prepration

To train the YOLOv4 model, the dataset is divided into training and validation sets using a 90:10 split. This means that out To manage the dataset, a Python script is developed to store the results by saving the file paths of each image into text files. Specifically, two text files are created: one for the training dataset and another for the validation dataset. Storing these files in the "darknet/data" directory is essential to ensure proper organisation. During the training process, the YOLOv4 model produces various outputs and results. These are stored in the result directory, which includes the trained weights saved in the ".pt" format. Additionally, the result directory contains visualisations such as the map value graph, F1 curve, P curve, PR curve, and R curve. These visualisations provide valuable insights into the performance of the trained model. By analysing these results, one can evaluate the effectiveness of the YOLOv4 model in object detection tasks and make informed decisions regarding its performance and potential areas for further improvement.

### YOLOV5 Dataset Prepration

In preparation for training the YOLOv5 model, two distinct folders were created: one for the training data and another for the validation data. Within the training data folder, two sub-folders were established. One sub-folder was designated for storing the images, while the other was dedicated to keeping their corresponding annotation files.

The same structure was replicated in the validation data folder. A 90:10 ratio was employed to split the data between the training and validation folders to ensure an appropriate data division. This means that 90% of the data was allocated to the training folder, while the remaining 10% was assigned to the validation folder. With the data organized this way, the YOLOv5 model is now ready to be executed. To ensure the model utilizes the correct dataset, it is essential to modify the training and validation data paths in the "coco.yaml" file, replacing them with the respective paths to the current dataset. The YOLOv5 model can be effectively trained using the designated training and validation data, enabling accurate object detection and recognition by completing these steps.

# Chapter 4

# IMAGE DETECTION FOR THE INDIAN DISHES

## 4.1 YOLO Architecture

### Introduction

YOLO (You Only Look Once) is an advanced object detection model renowned for its real-time capabilities. YOLOv4, introduced in April 2020, represents the fourth iteration of the YOLO algorithm. This model has demonstrated exceptional performance on the COCO dataset, encompassing 80 diverse object categories. YOLOv4 prioritizes inference speed as a one-stage detector, distinguishing it from the other prevalent method known as two-stage detectors. The model predicts classes and bounding boxes for the image in one-stage sensors without selecting a specific Region of Interest (ROI). This characteristic enables faster processing than two-stage detectors like FCOS, RetinaNet, and SSD. The original implementation of YOLO was developed using the Darknet framework. Darknet is a robust open-source framework for constructing neural networks using C++ and CUDA. It serves as the underlying framework for YOLO, providing a solid foundation for network building and training. The architecture of YOLO divides the object detection task into regression and classification tasks. The regression component predicts the classes and bounding boxes for the entire image in a single pass, allowing for the accurate localization of objects. Subsequently, the classification component determines the specific type or category of the detected object.[9]

In summary, YOLO is a state-of-the-art object detection model, with YOLOv4 being its latest version. It excels in real-time performance, achieving remarkable accuracy on the COCO dataset. YOLO follows a one-stage detection approach, enabling faster

inference by predicting classes and bounding boxes for the entire image without ROI selection. The model was initially implemented in the Darknet framework, providing a reliable network development platform. The YOLO architecture splits the detection task into regression and classification, facilitating precise object localization and identification.[10]

### YOLO Algorithm

The YOLO architecture comprises several components: the input layer, backbone, neck, detection neck, and detection head. The input layer is responsible for receiving the training images and processing them in parallel by the GPU in batches. The backbone and neck layers perform feature extraction and aggregation. The object detector consists of the detection neck and head layers responsible for identifying objects in the input image The final stage of the architecture is the head layer, which is responsible for making predictions and detecting objects. The head layer plays a crucial role in the detection process, as it is accountable for the localization and classification of things in the image.[9]

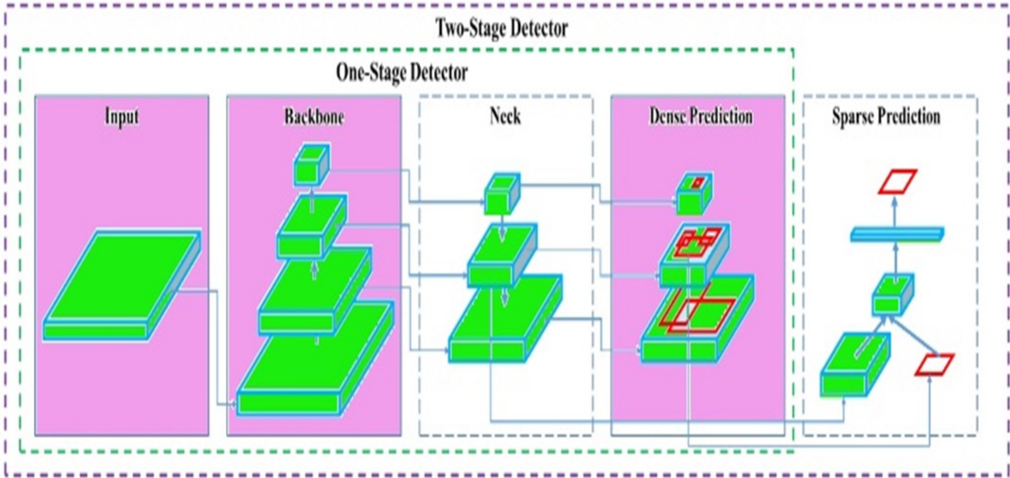


Figure 4.1: YOLO Architecture.

Due to its one-stage detection approach, YOLO performs object localization and object recognition simultaneously, called Dense Detection. On the other hand, a two-

stage detector performs these tasks separately and then combines the results, known as Sparse Detection.

### Backbone Network

backbone network of their object detection model. Initially, they considered CSPResNext50, CSPDarknet53, and EfficientNet-B3, but after extensive testing, they ultimately chose CSPDarknet53. This network is designed based on the DenseNet model and employs the Dense connectivity pattern, where previous inputs are concatenated with the current information before proceeding to the dense layers. CSPDarknet53 comprises a Convolutional Base Layer and a Cross Stage Partial (CSP) Block. The CSP strategy splits the feature map in the base layer into two parts and merges them using Cross-stage hierarchy, allowing for more gradient to flow through the layers and solving the "Vanishing Gradient." The CSP block divides the input into two halves, one sent through the dense block, while the other is routed directly to the next step without any processing. This strategy preserves fine-grained features, stimulates the network to reuse components, and reduces the number of network parameters. The final convolutional block in the backbone network is dense, as using more densely linked convolutional layers may decrease the detection speed.

### Neck

The neck serves as the location for feature aggregation within a system. It gathers feature maps from different stages of the backbone and merges them to ready them for the subsequent phase. Typically, the neck incorporates multiple bottom-up and top-down paths to facilitate this process of combination and integration.

### SPP-Additional Block

A supplementary SPP module (Spatial Pyramid Pooling) is inserted between the CSPDarkNet53 backbone and the feature aggregator network (PANet) to expand the receptive field. This addition allows for extracting essential contextual features while having minimal impact on network speed. The SPP module is connected to the final layers of the densely connected convolutional layers of CSPDarkNet. The term "receptive field" refers to the image area influenced by a single kernel or filter at a given moment. When stacking more convolutional layers, the receptive field grows linearly. However, incorporating dilated convolutions increases the receptive field exponentially, introducing non-linearity to the model.

## 4.2 Training YOLOV4 Model

YOLOv4, an object detection algorithm, is the fourth iteration of the You Only Look Once (YOLO) series. It builds upon the success of YOLOv3 and YOLOv2, introducing several advancements to enhance the accuracy and performance of object detection. One of the critical improvements in YOLOv4 is its upgraded backbone network, which is responsible for extracting features from input images. It employs a more powerful backbone architecture, such as CSPDarknet53, enabling it to capture intricate and abstract object details. YOLOv4 incorporates various optimization techniques to boost performance.

These techniques include advanced data augmentation, mosaic data augmentation (combining multiple images into one), and methods like DropBlock regularization to address overfitting. YOLOv4 introduces PANet (Path Aggregation Network) to enhance detection accuracy further. PANet combines features from different scales to improve the model's ability to detect objects of varying sizes. This improves detection performance for small and large things within an image.

Another notable addition in YOLOv4 is the Mish activation function, which enhances the model's non-linear capabilities and improves its representation learning. The CSPDarknet architecture is introduced in YOLOv4, which facilitates better information flow within the network and reduces computational complexity. This leads to faster inference times while maintaining accuracy. In summary, YOLOv4 strives to achieve state-of-the-art performance in object detection by focusing on accuracy, speed, and robustness. It has been widely adopted in various computer vision applications, including autonomous vehicles, surveillance systems, and object recognition tasks.

The YOLOv4 Model was trained on a dataset containing 20 food classes, including IndianBread, Biryani, Rasgulla, Uttapam, Paneer, Poha, Khichadi, Omellete, Plain Rice, Dal Makhni, Rajma, Poori, Chole, Dal, Sambhar, Papad, Gulab Jamun, Idli, Vada, and Dosa. The training process involved several steps, starting with preparing a labelled dataset consisting of food images. Bounding boxes were annotated around the objects of interest in the photos, and the dataset was split into training and validation sets. The chosen framework for training was Darknet, and the YOLOv4 configuration file (yolov4.cfg) was customized to suit the food dataset's requirements. The number of classes was adjusted to 20, and other parameters were fine-tuned based on the dataset's characteristics. Pre-trained weights for the Darknet-53 backbone network were utilized to initialise the model. These weights provided a good starting point and facilitated effective learning from the food dataset.

The model processed batches of training images during training and iterated over the dataset multiple times. It updated its parameters to minimize the detection loss

by comparing predicted bounding boxes and class probabilities with ground truth annotations. This enabled the model to learn to detect and classify the 20 food classes accurately. The periodic evaluation was performed on the validation set to assess the model's performance. mAP, precision, and recall were calculated. The achieved mAP value of 0.79632 reflects the model's overall accuracy in object detection across all classes. Precision, with a value of 0.75712, represents the model's ability to correctly identify true positives among the predicted positive detections. Recall, at a value of 0.76731, demonstrates the model's effectiveness in detecting the true positives among all ground truth instances.

## 4.3 Training YOLOv5 Model

### 4.3.1 YOLOV5 Nano Model

The YOLOv5 Nano model is the most miniature version of the YOLOv5 architecture, designed specifically for devices with limited computational power, like edge devices, embedded systems, and mobile devices. It aims to balance model size, accuracy, and inference speed. Although compact, the YOLOv5 Nano model can still perform real-time object detection and classification. It utilizes a lightweight architecture that enables faster inference times, albeit with a slight reduction in accuracy compared to larger YOLOv5 models. Training the YOLOv5 Nano model involves using a deep convolutional neural network (CNN) and applying the YOLO approach to object detection. Divides the input image into the grid cell and find the probability for each grid cell. While the Nano model may achieve a different level of accuracy than larger YOLOv5 models, it provides a practical solution for applications that require real-time object detection on devices with limited computational resources. Its reduced complexity and size make it well-suited for scenarios with a trade-off between accuracy and resource consumption. It's important to note that the YOLOv5 Nano model can detect and classify a wide range of objects in various real-world scenarios, making it a valuable option for lightweight object detection tasks.[11]

YOLOv5 Nano Model has a straightforward neural network for the convolution part. In this network architecture, some neurons act as input nodes while others serve as output nodes. There are no hidden layers in this straightforward neural network design. The training process for 100 epochs with 61 food classes took approximately 2 to 2.5 hours. Due to its simplicity, this model exhibits a high Frames Per Second (FPS) rate, ranging from 7 to 10 frames per second. The YOLOv5 Nano Model is particularly well-suited for object detection in Android applications because of its high FPS rate. It can quickly detect objects, making it suitable for real-time applications. However, the

high FPS rate can lead to instability in the system when observing objects. The model's detection speed is fast, but it lacks control. During testing, significant instability was observed in this model. The Nano model exhibits lower accuracy than higher versions of the YOLOv5 model.[12] It can be seen from Figure 4.3 and Figure 4.4 that the Mean Average Precision (mAP) value obtained after training this model on 61 food classes was 0.69441.

### 4.3.2 YOLOv5 Small Model

The YOLOv5 Small model is a variant of the YOLOv5 architecture designed to balance model size and computational requirements while maintaining reasonable accuracy in object detection tasks. Compared to larger versions like Medium, Large, and Extra Large, the YOLOv5 Small model has a more minor architecture with fewer parameters. This compact design allows for faster inference times and makes it well-suited for resource-constrained devices or applications that require real-time object detection. Like other YOLOv5 models, the Small variant follows the YOLO (You Only Look Once) approach for object detection. While the YOLOv5 Small model may sacrifice a small amount of accuracy compared to larger models, it still performs well in detecting and classifying a wide range of objects in various real-world scenarios. It is commonly used in mobile device applications such as surveillance systems, robotics, and object recognition tasks.

The YOLOv5 Small Model's reduced complexity and size make it suitable for scenarios with a trade-off between model accuracy and computational resources. It provides a practical solution for object detection tasks requiring a lightweight model without significantly compromising performance. It's important to note that the version of the YOLOv5 Small model can vary depending on factors such as the specific dataset, training configuration, and application requirements. Fine-tuning and customization techniques can also be applied to adapt the model to particular use cases and optimize its performance for different object detection scenarios.[13] It can be seen from Figure 4.3 and Figure 4.4 that the Mean Average Precision (mAP) value obtained after training this model on 61 food classes was 0.78453.

### 4.3.3 YOLOv5 Medium Model

The YOLOv5 Medium Model is a variation of the YOLOv5 architecture that finds a balance between accuracy and computational demands. It aims to achieve higher precision in object detection while maintaining reasonably fast inference speeds. Compared to smaller models like Nano and Small, the YOLOv5 Medium model offers improved

detection performance. It accomplishes this by employing a more complex architecture and a more significant number of parameters, enabling it to capture finer object details in images. Like other YOLOv5 models, the Medium variant follows the YOLO (You Only Look Once) approach for object detection. Divides the input image in the grid and predicts the probability of each grid. This methodology ensures efficient and real-time object detection and classification.

The YOLOv5 Medium model is well-suited for applications prioritising higher accuracy without compromising real-time processing. It is commonly utilised in surveillance systems, robotics, and autonomous vehicles, where accurate object detection is crucial in decision-making.[14] While the Medium model provides enhanced accuracy compared to smaller YOLOv5 versions, it may require more computational resources during inference. Nevertheless, it maintains a favourable trade-off between accuracy and inference speed, making it a popular choice for various computer vision tasks. It's worth noting that the performance of the YOLOv5 Medium model can vary depending on the specific dataset, training setup, and application requirements. Fine-tuning and customisation can be applied to adapt the model to specific use cases, improving its performance for particular object detection tasks. YOLOV5 medium has a little bit large number of neurons in its convolution layers. This medium model is a little bit complex. It has an input layer, hidden layer and output layer. This YOLOV5 medium model has complex neural networks compared to the nano model, so it takes more time than the nano model to train 61 data classes. YOLOV5 nano model took 2 to 2.5 hours to prepare 61 data types on 100 epochs. Compared to YOLOV5, this YOLOV5 medium model takes 4 to 5 hours to train 61 data classes on 100 epochs. It can be seen from Figure 4.3 and Figure 4.4 that the Mean Average Precision (mAP) value obtained after training this model on 61 food classes was 0.82784.

### 4.3.4 YOLOv5 Large Model

The YOLOv5 Large model is a version of the YOLOv5 architecture that prioritises high accuracy in object detection. Its primary goal is to detect and classify objects with improved precision while maintaining reasonable inference speeds. Compared to minor YOLOv5 variants like Nano, Small, and Medium, the Large model offers enhanced detection performance. This is achieved by employing a more complex and profound architecture, allowing it to capture intricate object details in images more effectively. The YOLOv5 Large model is particularly suitable for applications that require accurate object detection, even if it results in slightly slower inference times. It is commonly employed in advanced robotics, self-driving vehicles, and high-security surveillance systems. Although the Large model offers superior accuracy to more mi-

nor YOLOv5 variants, it may demand more computational resources during inference due to its increased complexity. However, the trade-off between accuracy and inference speed is generally favourable, making it a popular choice for various computer vision tasks prioritising precision. It's crucial to note that the performance of the YOLOv5 Large model can vary depending on factors such as the specific dataset, training configuration, and application requirements. Fine-tuning and customisation can also be applied to adapt the model to particular use cases and optimise its performance for different object detection scenarios.[15] It can be seen from Figure 4.3 and Figure 4.4 that the Mean Average Precision (mAP) value obtained after training this model on 61 food classes was 0.84642.

### 4.3.5 YOLOv5 Extra Large Model

The YOLOv5 Extra Large model is the most substantial and powerful version of the YOLOv5 architecture. Although it requires higher computational resources, it aims to provide exceptional accuracy and detection performance in object recognition tasks. Compared to more minor YOLOv5 variants like Nano, Small, Medium, and Large, the Extra Large model takes detection capabilities to the next level. It achieves this by employing a more intricate and profound architecture, enabling it to capture intricate object details precisely.

The YOLOv5 Extra Large Model is particularly suitable for applications that prioritize uncompromising accuracy and outstanding detection performance. It is commonly utilized in cutting-edge research, advanced computer vision systems, and demanding tasks such as autonomous driving. Due to its significantly larger size and complexity, the Extra Large model demands substantial computational resources during inference. However, it provides an excellent trade-off between accuracy and inference speed for applications requiring the highest precision level. It's important to note that the performance of the YOLOv5 Extra Large model can vary depending on factors such as the specific dataset, training configuration, and application requirements. Fine-tuning and customization techniques can be applied to adapt the model to particular use cases,[16] further optimising its performance for various object recognition scenarios. It can be seen from Figure 4.3 and Figure 4.4 that the Mean Average Precision (mAP) value obtained after training this model on 61 food classes was 0.85125.

## 4.4 Performance Evaluation

From Figures 4.2 and 4.3, it can be seen that the performance of the wise YOLOv5 Extra Large model is very efficient. The mAP Value of the YOLOv5 Extra Large model

is 0.85125 ( the average precision of object detection across different object classes and levels of confidence thresholds is called Map Value). The performance evaluation shows that the YOLOv5 models generally outperform YOLOv4 regarding mAP, precision, and recall. Among the YOLOv5 models, the "YOLOv5 Large" model achieves the highest mAP value of 0.85125, precision of 0.7966, and recall of 0.81973. It is essential to consider both performance and computational resources for Android deployment. The YOLOv5 models, especially minor variants like YOLOv5 Nano and YOLOv5 Small, have balanced performance and efficiency scores, making them suitable for deployment on Android devices with limited computational resources. Therefore, based on the given information, the YOLOv5 Small model may be a preferable choice for Android deployment, as it gives very decent accuracy (high mAP, precision, and recall) and resource efficiency, making it well-suited for real-time object detection on Android devices. After

| Model | No. of classes | No. of Epochs | Map Value | Precision | Recall |
|---|---|---|---|---|---|
| YOLOv4 | 20 | 100 | 0.79632 | 0.75712 | 0.76731 |
| YOLOv5 Nano | 61 | 100 | 0.69441 | 0.65617 | 0.67433 |
| YOLOv5 Small | 61 | 100 | 0.78453 | 0.73552 | 0.75076 |
| YOLOv5 Medium | 61 | 100 | 0.82784 | 0.77185 | 0.79658 |
| YOLOv5 Large | 61 | 100 | 0.84642 | 0.79341 | 0.81171 |
| YOLOv5 Extra Large | 61 | 100 | 0.85125 | 0.7966 | 0.81973 |

Figure 4.2: Performance results of YOLO models

deploying all five YOLOv5 models in an Android application and conducting thorough testing, it has been observed that the YOLOv5s model stands out in terms of performance, efficiency, and FPS (Frames Per Second) rate. The YOLOv5s model, which

represents the "Small" variant of YOLOv5, has proven to be highly efficient regarding resource utilization and computational requirements. This efficiency is crucial when deploying computer vision models on mobile devices with limited processing power, such as Android smartphones. In terms of performance, the YOLOv5s model has exhibited remarkable accuracy and speed. The model demonstrates superior object detection capabilities, accurately identifying and localizing objects in real time. It achieves this while maintaining a high FPS rate, ensuring a smooth and responsive user experience within the Android application. The efficiency of the YOLOv5s model can be attributed to several factors. First, the YOLOv5 architecture itself is designed to strike a balance between accuracy and computational efficiency. The model leverages advanced techniques such as a lightweight backbone network, efficient feature fusion mechanisms, and optimized anchor boxes, contributing to its overall efficiency.

Furthermore, the YOLOv5s model benefits from extensive optimization efforts during the training and deployment. Techniques like model quantization, which reduces the model's memory footprint, and hardware acceleration, such as utilizing the GPU for inference, enhance the model's efficiency without compromising performance. The high FPS rate achieved by the YOLOv5s model is crucial for real-time applications, as it allows for smooth and responsive object detection. The high FPS rate is precious in augmented reality, live video processing, and interactive applications where the system needs to detect objects rapidly and provide timely feedback. Overall, the deployment and testing of the YOLOv5s model in the Android application have demonstrated its exceptional performance and efficiency. Its ability to achieve high accuracy while maintaining a high FPS rate makes it a compelling choice for real-time object detection on resource-constrained mobile devices like Android smartphones. Hence, the YOLOv5s model has been deployed as the final model in this project's Android application.

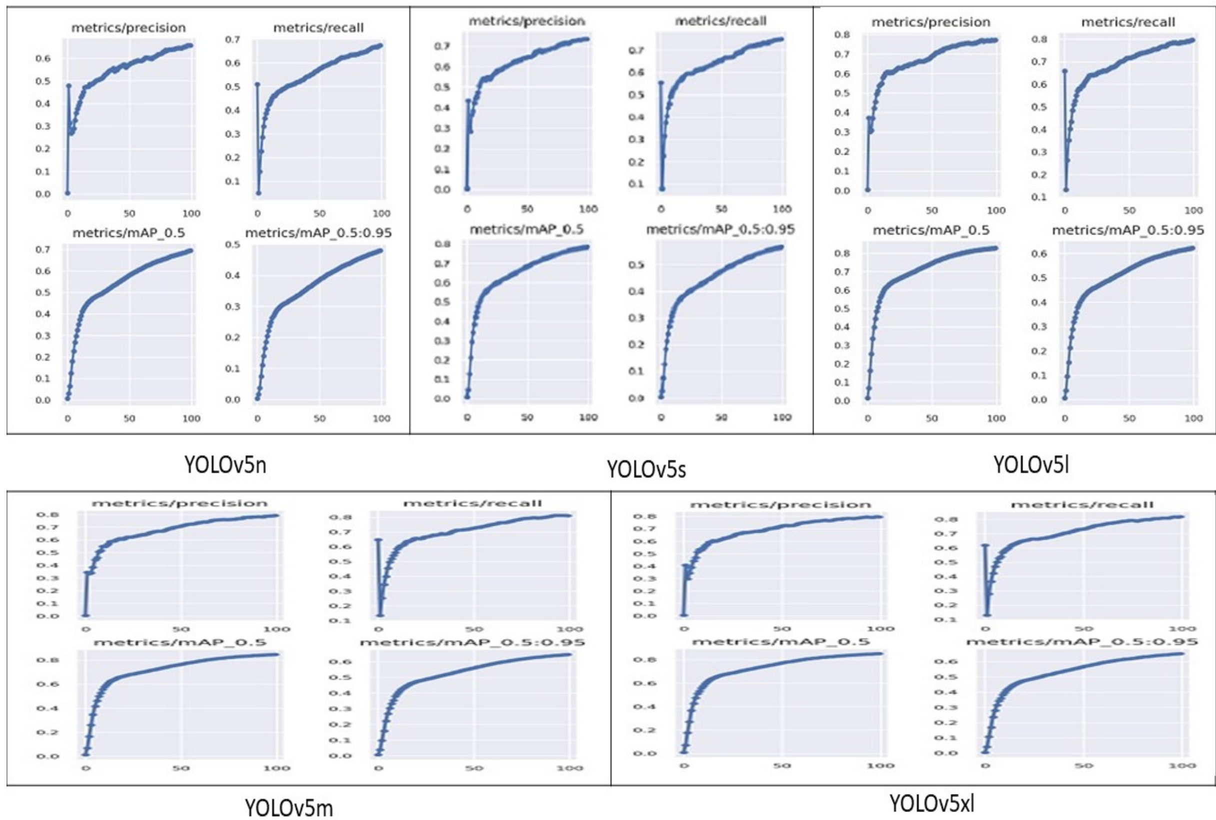


Figure 4.3: Graphical Representation of Performance results for YOLOv5 models

# Chapter 5

# MOBILE APPLICATION DEVELOPMENT

## 5.1 WhatDish Application Introduction

"WhatDish" is an Android application specifically designed for food detection, capable of identifying a wide range of food items and categorizing them into up to 60 different classes. The application offers users valuable insights into food items' estimated calorie, fat, protein, and carbohydrate content. To access the application's features, users must register by clicking on a floating button. During registration, users must provide essential information such as weight, height, age, sex, and activity level. This information is crucial for the application to determine the maximum number of daily calories the user can consume. Once the registration is complete, users are directed to the "MainActivity" screen.

In the "MainActivity" screen, users can utilize the application's food detection capabilities by simply pointing their cameras at different food items. Users must click on a dedicated capture button within the application to capture a food item. Once a food item is captured, the application's sophisticated algorithms analyze the image to identify the type and quantity of the food item. Based on this analysis, the application calculates and presents the captured food item's corresponding calorie, fat, protein, and carbohydrate content. Users can conveniently view these results in a table format by scrolling up from the bottom of the application screen. This table provides a comprehensive overview of all the detected food items and their nutritional information. Moreover, the application generates a ".txt" file in the background, which stores the total calories consumed by the user.[17]

One of the application's key features is a progress bar indicator visually represent-

ing the user's calorie consumption. This progress bar is displayed on the application's "CameraActivity" screen. When the user closes and reopens the application, the progress bar accurately reflects the total calories consumed by the user by reading the stored information from the ".txt" file.

The user interface (UI) of the "CameraActivity" screen consists of a dedicated frame for the Camera-2 API, which enables real-time camera preview and food detection. Additionally, the screen includes a prominent white round button, which serves as the primary action button for capturing food items. Clicking on this button triggers a captivating rotational animation that reveals two additional buttons. One of these buttons allows users to update their personal information by navigating to the "Takinginputfromuser" activity. This activity lets users modify their weight, height, age, sex, or activity level. The other button displays the user's activity history over the past 30 days, providing insights into their progress.[18]

The application incorporates a drop-down table component to assist users in obtaining quick information about the detected food items. This table displays vital details about the identified food items' calories, fat, protein, and carbohydrates. Users can refer to this table to gather accurate nutritional information about the captured food items.

The "WhatDish" Android application offers a comprehensive food detection and nutritional tracking experience. With its user-friendly interface and advanced algorithms, users can effortlessly capture food items, view nutritional details, track calorie consumption, and make informed dietary choices.

## 5.2 Application UI

### 5.2.1 Camera Activity

Figure 5.1 illustrates the "CameraActivity" as the main activity of the "WhatDish" Android application. Upon opening the application, the "CameraActivity" is the first screen that appears. This activity consists of a frame dedicated to the Camera-2 API, a white round button, and a floating button. When the user clicks the floating button, two additional buttons emerge through a rotational animation. One button allows the user to navigate to the "Takinginputfromuser" activity, while the other button displays the user's activity graph for the past 30 days. Within the "CameraActivity," an additional component presents information about the detected food's calories, fat, protein, and carbohydrates. This component takes the form of a drop-down table, providing a comprehensive overview of the nutritional content of the captured food items. Users can view the number of calories consumed from the detected food through

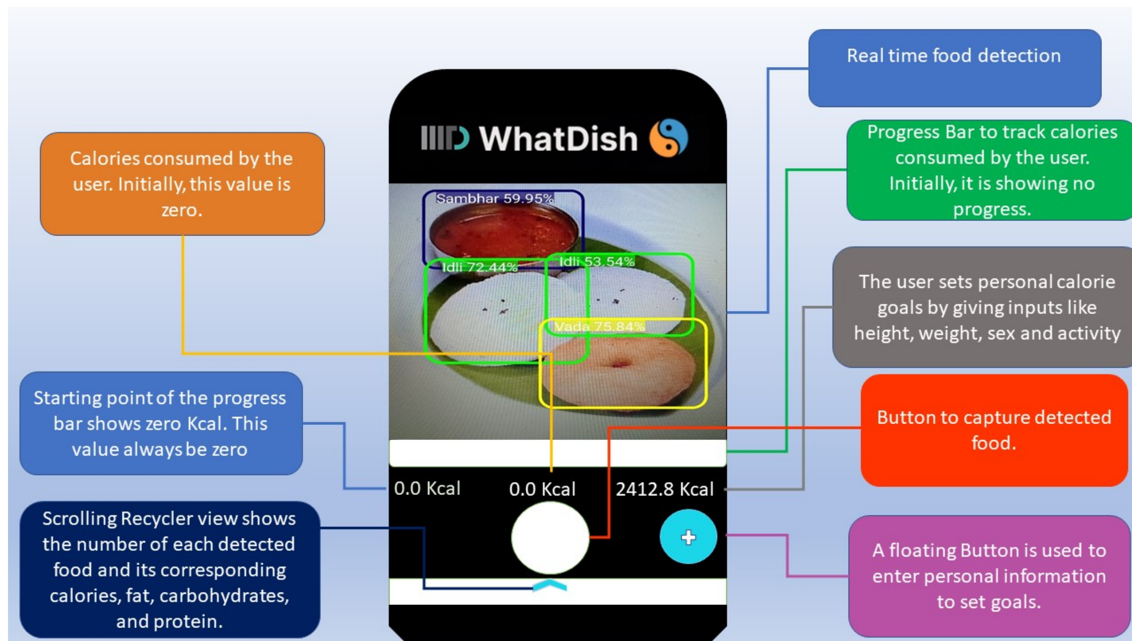


Figure 5.1: WhatDish Android application with no progress in progress bar

this drop-down table.

The following is the user interface (UI) utilized in this activity:

### Frame For Object Detection

The Android application's screen is predominantly dedicated to object detection, accounting for approximately 90% of the available space. This screen detects food objects and stores information about them, including their quantities, in an array. To handle this functionality, a separate fragment is used, which is utilized within the "CameraActivity" of the application.

To create the object detection frame, a custom view is employed, which encompasses four key components: "AutoFitTextureView," "OverLayView," "RecognitionScoreView," and "ResultView." The "AutoFitTextureView" incorporates a TextureView that serves as the display for the camera preview content. When the user launches the "WhatDish" application, this TextureView is presented within the "CameraActivity." As the user initiates the food detection process, the "AutoFitTextureView" captures the detected object's content, displayed on the mainframe screen.

The "OverLayView" component plays a crucial role by capturing the content of the "AutoFitTextureView" on a canvas and presenting it on the TextureView. It acts as

the mechanism for displaying the detection overlay on the camera preview.

The "RecognitionScoreView" determines the accuracy percentage of a particular food detection. It provides valuable information about the reliability of the detected object, allowing users to assess the confidence level of the detection.

After detecting the food objects, the "ResultView" component displays the final results on the mainframe screen. This class includes an array that stores comprehensive information about the detected food items. These details are essential for showcasing various nutritional aspects such as calories, fat, protein, and carbohydrates associated with each detected food item.

All these views mentioned above are integrated and managed within a fragment named "CameraConnectionFragment," which ensures smooth coordination and execution of the object detection process within the overall application flow.

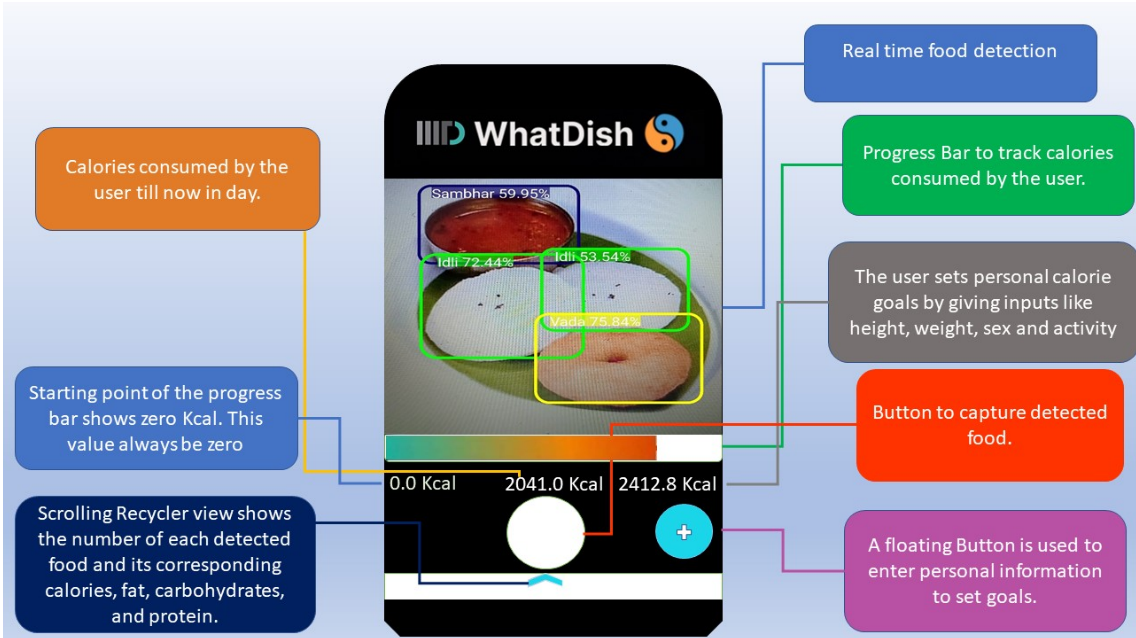


Figure 5.2: WhatDish Android application with progress in progress bar

### Round White Button

The application's primary function is the round white button, allowing users to detect and capture food using the Android camera. When the user has a stable camera view of the food, they must click this round white button. Upon clicking, all the relevant data about the detected food is stored within the Android application.

Users can access the stored data by scrolling up from the bottom of the application frame screen, where they will find a table displaying information about each captured food. The implementation of the round white button is handled on a separate thread. Since the main thread is already occupied with the food detection process using Camera2API, running another method concurrently would be problematic. Android applications typically complete one process before starting the next. The round white button is implemented on a secondary thread to ensure smooth operation and avoid application crashes.

Now users can detect and capture food simultaneously using the round white button while viewing the information about the captured food. The round white button's user interface is created using XML files. Android Studio provides XML file capabilities for developers to design visually appealing buttons and other elements. The round white button consists of three XML files:

**1.** The main XML file.

**2.** An XML file for drawing the round white disk.

**3.** Another XML file for animation.

The round white button appears white when the user is not capturing an image. However, when the user captures an image, the button's colour changes to light green. While the user holds the capture button, the control remains green, indicating an image is being captured. Once the button is unclicked, it reverts to white. This colour animation feature informs the user about the image capture process and provides visual feedback.

#### Float Button

The Float Button is a user interface element in an Android application that provides additional functionality when clicked. When users click the Float Button, it triggers the appearance of more buttons, each with its specific functionality. The primary purpose of the Float Button is to allow users to update their personal information within the application.

Upon clicking the Float Button, a circular animation is displayed, and two additional buttons become visible. One button is dedicated to accessing the user profile, while the second button provides access to the monthly user activity graph. The application optimises screen space by utilizing the Float Button, as most of the screen is dedicated to the Camera frame for food detection.

The Float Button is positioned within the application at the bottom right of the Camera frame. It is easily identifiable by its light green colour and features a white "+" symbol at its centre. Upon clicking the Float Button, the "+" symbol transforms into an "X" symbol, and the circular animation reveals the two additional buttons.

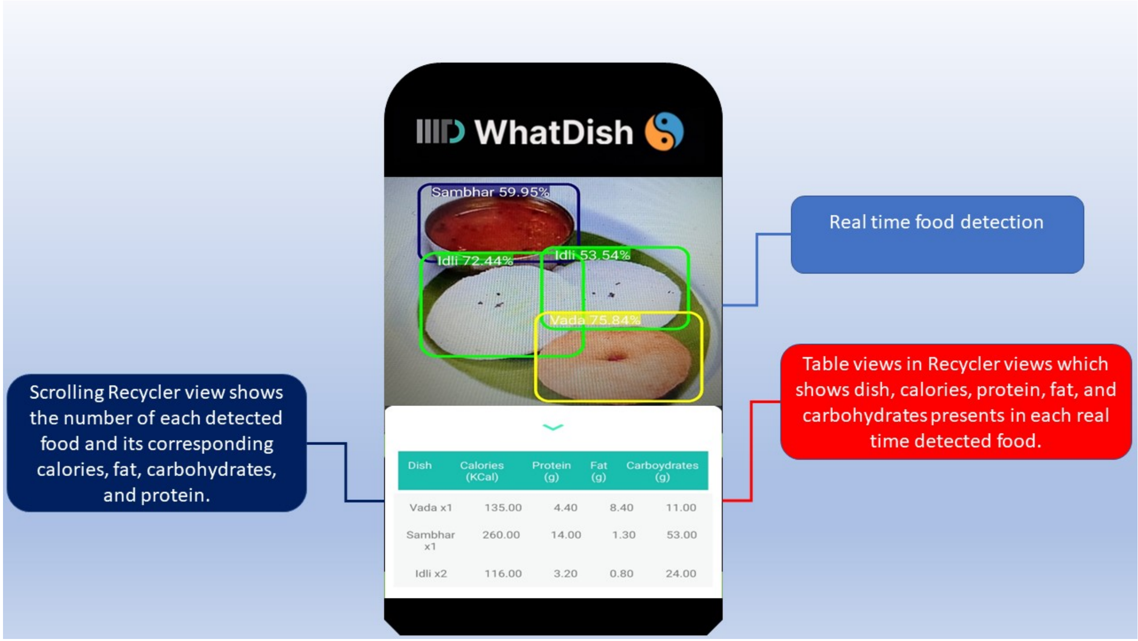


Figure 5.3: WhatDish Android application with detected food information stored in table

### Progress Bar For Per Day Calories

The progress bar used in the application is a cylindrical shape bar that serves as a tracker. Figure 5.1 depicts the progress bar without any progress, while Figure 5.2 illustrates the progress bar with an indication of progress. The cylindrical shape bar tracks calorie consumption, where the maximum calorie intake allowed per day for the user is set as the endpoint of the progress bar. The progress bar reacts and adjusts accordingly based on this maximum calorie limit.

When the user captures a photo, and the model calculates the number of calories in the specific food item, the progress bar indicator increases by a percentage corresponding to the maximum allowed calorie consumption. This allows the user to visually monitor whether their daily calorie intake falls within a safe range. The indicator colour on the progress bar changes as calorie consumption increases. Initially, the colour is

green, indicating 40% of the maximum set calorie. As the user approaches 80% of the set calorie, the indicator colour changes to orange. Finally, the indicator colour turns red when the user reaches 100% of the maximum set calorie.

Below the progress bar, three values are displayed. The first value, located at the far left of the progress bar, is always 0 and represents the starting point of the progress bar. The second value, positioned in the middle of the progress bar, indicates the number of calories consumed by the user thus far. The third value at the far right of the progress bar represents the maximum daily calorie intake recommended for optimal health.

The application effectively handles the functions, and when the user captures a photo, the calorie value of the detected food is calculated and stored in an output ".txt" file on the user's mobile device. This allows the application to keep track of the total calories consumed by the user. If the user closes the application and opens it again, the "CameraActivity" is launched, and the total calories consumed by the user are recalculated from 0. However, the previous total calorie value for the current day is already stored in the user's mobile device as a ".txt" file. When the application is reopened, the "CameraActivity" reads the ".txt" file and considers the previously stored value of the total calories consumed, displaying the actual calorie count in the progress bar.

As the user continues to detect and capture food items, the calories calculated from each capture are added to the day's previous calorie value. Consequently, the progress bar indicator moves forward to reflect the increased calorie consumption.

### 5.2.2 User Activity

As the name implies, the "User Activity" is implemented in the application's code. This activity is designed to gather input from the user, including weight, height, gender, and regular exercise details. If the user leaves any of these fields unfilled and attempts to proceed by pressing the "Done" button, a toast message will prompt the user to enter valid data in the required fields.

After filling out all the necessary fields and pressing the "Done" button, the person's Basal Metabolic Rate (BMR) is calculated based on gender. Additionally, depending on the user's level of regular physical activity, the BMR is multiplied by a specific factor, resulting in the number of calories required for that individual per day. These required calories are stored in an output source file on the user's mobile device, allowing the application to access this value even if the user terminates the activity.

The fields and buttons created in this activity are as follows:

Figure 5.4: WhatDish Android application user Activity

### Edit Text for required fields

The "EditText" field is used to obtain input from the user in string format. In Android Studio, this field is represented by the "EditText" class. In the mentioned activity, I have included two "EditText" fields named "Enter the height" and "Enter the weight". These fields are designed to capture the user's height and weight as input.

Using the "EditText" class, developers can retrieve the user's input as a string variable. However, since the user provides the height and weight in numeric format, string values have been converted into the double type. These corrected values are then used to calculate the Basal Metabolic Rate (BMR).

### Spinner drop-down list

Android provides the flexibility to allow users to select a single item from a list of options. In the application, a spinner drop-down list displays and sets the units for the entered quantity, such as height, weight, sex, and activity. This drop-down list presents users with choices for selecting the desired unit of measurement.

For instance, users can choose to specify weight in either "kg" or "pound", height in either "cm" or "inch", and select their gender as either "male" or "female". The

spinner drop-down list facilitates the selection of the appropriate unit or category from the available options.

**Done button**

Upon completing each field in this activity, the user must click the 'done' button. This action triggers the calculation of the maximum required calories per day based on the input parameters provided by the user. The user's Basal Metabolic Rate (BMR) is calculated, and depending on the selected activity level, the BMR is multiplied by a specific coefficient to determine the maximum daily calorie intake.

Once the user presses the 'done' button, this maximum calorie value is stored in a text file. Even if the user closes the application and opens it again, the application retrieves and utilizes the stored calorie value as the maximum daily calorie intake for the user. Consequently, there is no need to input these activity details daily, as the stored value is a reference for the user's maximum calorie requirement.

## 5.3 Deployment Of YOLO Model In An Android Application

After training the dataset using the "YOLOv5" model, the trained weights are saved in the ".pbt" (PyTorch) format. However, to deploy these weights in an Android application, they need to be converted to the ".tflite" (TensorFlow Lite) design, as Android, only supports the ".tflite" format for model deployment. This conversion process allows for compatibility between the trained weights and the Android app.

Once the weights have been successfully converted to the ".tflite" format, integrating the trained weights into the Android application can begin. To connect either the YOLOv4 or YOLOv5 model with an Android app, the following steps are typically involved:

1. **Preparing the YOLO Model:** Obtain the YOLOv4 or YOLOv5 model trained weights on the 61 food classes. These models are typically implemented in frameworks like Darknet for YOLOv4 and PyTorch for YOLOv5. Model formate is converted, which is compatible with the mobile device. Weights are converted from .pbt format to .tflite format. ONNX tool is used to convert the model to TensorFlow format.

2. **Set Up the Android Development Environment:** Install Android Studio, the official IDE for Android app development, on a computer. Click on the new Android project and name it "WhatDish".

3. **Add Dependencies:** Add the necessary dependencies in the Gradle file for integrating deep learning frameworks. For YOLOv4, TensorFlow dependencies have been added, while for YOLOv5, PyTorch dependencies have been added. Sync the project to download the dependencies.

4. **Load the YOLO Model in Android App:** Copy the converted YOLO model file to the Android project's assets folder. In the Java code in the Android app, load the YOLO model using the corresponding deep learning framework's API. This typically involves reading the model file from the assets folder and initializing the model.

5. **Implement Camera Functionality:** Set up camera permissions in the AndroidManifest.xml file to allow the app to use the device's camera. Implement the camera functionality using the Android Camera-2 API to capture frames from the camera in real-time. This has been done in an Android application's "Cameraactivity" file.

6. **Perform Inference:** Preprocess the camera frames by resizing and normalizing them to match the input requirements of the YOLO model. Pass the preprocessed frames through the YOLO model for object detection. The class labels, bounding box coordinates, and confidence scores have been extracted for object detection. This has been done in the "BoundingBoxactivity" file.

7. **Visualize and Display Results:** Overlay the bounding boxes and class labels on the camera frames to visualize the detected objects. This has been done in an Android application's "Overlay" file.

# Chapter 6

# CONCLUSION

In conclusion, this thesis successfully implemented and evaluated an object detection Android application based on the YOLOv4 and YOLOv5 training models for detecting various food classes and objects. Integrating these state-of-the-art models into an Android application has demonstrated their effectiveness in real-time object detection and recognition tasks, focused explicitly on food detection. The application achieved high accuracy in detecting and localising various food items by training the YOLOv4 and YOLOv5 models on a diverse dataset containing different food classes. The models proved robust and efficient, handling challenges such as occlusions, varying scales, and complex backgrounds. The Android application showcased the potential of utilising these models on mobile devices, enabling real-time object detection directly on the device without relying on server-side processing. This capability provided users instant feedback and valuable information about food items, enhancing their dining experiences, dietary monitoring, and nutritional analysis.

Furthermore, the application demonstrated the practicality of leveraging the device's camera and sensor data to provide contextual information about food objects. It enabled users to receive details about nutritional values in detecting food. The evaluation of the YOLOv4 and YOLOv5 models on the Android application showcased their superior performance in terms of accuracy and speed compared to previous versions. The models successfully addressed the limitations of earlier models by incorporating advancements in architecture, feature extraction, and optimisation techniques. This thesis also highlighted the potential for further research and improvement in food detection on mobile platforms. Future work can focus on expanding the dataset to include a wider variety of food classes and exploring techniques to improve the models' performance in challenging scenarios, such as partial occlusions or low-light conditions. In conclusion, the object detection Android application based on the YOLOv4 and

YOLOv5 models has proven a valuable tool for food detection and recognition. The successful integration of these models into the application demonstrates their potential to enhance user experiences, support dietary choices, and contribute to advancing mobile vision applications in food detection.

# Bibliography


[1] "Object detection in computer vision: A comprehensive review," *Journal of Pattern Recognition Research*, vol. 6, pp. 80–97, 2011.

[2] "Opencv: Open source computer vision library," *Dr. Dobb's Journal of Software Tools*, vol. 25, pp. 120–125, 2000.

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2016.

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

[5] M. Phadtare, V. Choudhari, R. Pedram, and S. Vartak, "Comparison between yolo and ssd mobile net for object detection in a surveillance drone," *Int. J. Sci. Res. Eng. Manag*, vol. 5, pp. 1–5, 2021.

[6] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," *Lecture Notes in Computer Science*, p. 21–37, 2016. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46448-0_2

[7] Q. e. a. Wang, "Deep learning approach to peripheral leukocyte recognition," 2019. [Online]. Available: doi:10.1371/journal.pone.0218808

[8] S. A. Choudhary R, "Potential use of hydroxychloroquine, ivermectin and azithromycin drugs in fighting covid-19," 2020. [Online]. Available: doi:10.1016/j.nmni.2020.100684

[9] T. Diwan, G. Anirudh, and J. V. Tembhurne, "Object detection using yolo: Challenges, architectural successors, datasets and applications," *Multimedia Tools and Applications*, vol. 82, no. 6, pp. 9243–9275, 2023.

[10] D. Pandey, P. Parmar, G. Toshniwal, M. Goel, V. Agrawal, S. Dhiman, L. Gupta, and G. Bagler, "Object detection in indian food platters using transfer learning with yolov4," 05 2022.

[11] S. Li, Y. Li, Y. Li, M. Li, and X. Xu, "Yolo-firi: Improved yolov5 for infrared image object detection," *IEEE access*, vol. 9, pp. 141 861–141 875, 2021.

[12] H.-K. Jung and G.-S. Choi, "Improved yolov5: Efficient object detection using drone images under various conditions," *Applied Sciences*, vol. 12, no. 14, p. 7255, 2022.

[13] J.-H. Kim, N. Kim, Y. W. Park, and C. S. Won, "Object detection and classification based on yolo-v5 with improved maritime dataset," *Journal of Marine Science and Engineering*, vol. 10, no. 3, p. 377, 2022.

[14] Z. Li, "Road aerial object detection based on improved yolov5," in *Journal of Physics: Conference Series*, vol. 2171, no. 1. IOP Publishing, 2022, p. 012039.

[15] B. Mahaur and K. Mishra, "Small-object detection based on yolov5 in autonomous driving systems," *Pattern Recognition Letters*, vol. 168, pp. 115–122, 2023.

[16] A. Siouras, K. Stergiou, P. Karlsson, and S. Moustakidis, "Hybrid object detection methodology combining altitude-dependent local deep learning models for search and rescue operations," *Journal of Control and Decision*, pp. 1–11, 2022.

[17] M. Goel, S. Dargar, S. Ghatak, N. Verma, P. Chauhan, A. Gupta, N. Vishnumolakala, H. Amuru, E. Gambhir, R. Chhajed, M. Jain, A. Jain, S. Garg, N. Narwade, N. Verhwani, A. Tiwari, K. Vashishtha, and G. Bagler, "Dish detection in food platters: A framework for automated diet logging and nutrition management," 2023.

[18] G. Wang, H. Ding, Z. Yang, B. Li, Y. Wang, and L. Bao, "Trc-yolo: A real-time detection method for lightweight targets based on mobile devices," *IET Computer Vision*, vol. 16, no. 2, pp. 126–142, 2022.