Report by: Rebecca Bayssari

Supervisor: Hervé Le Borgne

CentraleSupélec

rebecca.bayssari@student-cs.fr, herve.le-borgne@cea.fr

Android Application for Object Localization and Recognition in a Video Stream

Abstract

Object detection has seen significant advancements with the progress of artificial intelligence and computer vision. In this project, we develop an Android application for real-time object localization and recognition from a video stream. Leveraging the computational capabilities of smartphones, our approach aims to provide an accessible and efficient solution without requiring specialized equipment. This work explores the optimization of detection models for mobile deployment, focusing on speed and accuracy to ensure a seamless and intuitive user experience.

1. Introduction

Object detection is a fundamental challenge in computer vision, enabling applications in areas such as surveillance, robotics, autonomous vehicles, and augmented reality. The rise of deep learning has significantly improved the performance of object detection models, making real-time recognition more reliable and accessible.

This project focuses on developing an Android application that enables real-time object detection and recognition using a smartphone's camera. Given the widespread use of smartphones, they provide an ideal platform for deploying AI-driven applications without requiring additional specialized hardware. The goal is to integrate a robust deep learning model into a mobile application that can detect and classify objects in a live video stream efficiently.

By optimizing the model for mobile deployment, we ensure a balance between accuracy and processing speed, making it suitable for real-world applications. The challenges of this project include model compression, energy-efficient execution on mobile processors, and ensuring user-friendly interaction within the application.

2. State of the art

Object detection is an important task in computer vision that involves identifying and locating objects of interest in an image or video. Advances in this field are heavily based on neural networks, which are machine learning models inspired by the human brain. In particular, Convolutional Neural Networks (CNNs) have revolutionized object detection by automatically and hierarchically extracting features from images, leading to high accuracy and speed.

CNNs are made up of multiple layers of artificial neurons that filter and analyze input data step by step. The convolutional layers detect simple patterns, such as edges, textures, and shapes. These patterns are then combined in deeper layers to identify more complex objects. This ability to learn detailed and meaningful representations from raw data has made CNNs the foundation of modern object detection models.

Object detection models using CNNs can be divided into two main categories:

1. One-stage models
2. Two-stage models

2.1. One-stage models

One-stage models detect objects in a single step, predicting both the object category and its position directly from the image features. These models are generally faster and are best suited for real-time applications. Some well-known one-stage models include:

- YOLO (You Only Look Once): Developed by Joseph Redmon et al., YOLO is one of the most popular real-time object detection models. It divides the image into a grid and predicts object classes and bounding boxes at the same time.

- SSD (Single Shot MultiBox Detector): Introduced by Wei Liu et al., SSD improves accuracy while maintaining high speed. It uses different-sized convolution filters to detect objects of various sizes efficiently.
- RetinaNet: Proposed by Tsung-Yi Lin et al., RetinaNet introduces "Focal Loss," which gives more importance to difficult-to-classify examples, improving accuracy while keeping good processing speed.

2.2. Two-stage models

Two-stage models perform object detection in two separate steps:

1. Generate region proposals, which are areas likely to contain objects.

2. Refine and classify these regions.

These models are usually more accurate but slower than one-stage models. The most well-known two-stage models include:
- R-CNN (Regions with Convolutional Neural Networks): Developed by Ross Girshick et al., R-CNN is one of the first two-stage models. It generates region proposals using a selective search algorithm and then extracts features using CNNs.
- Fast R-CNN: Also developed by Ross Girshick, Fast R-CNN improves efficiency by integrating feature extraction directly into the detection process, reducing redundant computations.
- Faster R-CNN: Introduced by Shaoqing Ren et al., Faster R-CNN includes a Region Proposal Network (RPN) that generates region proposals directly within the detection network, significantly improving speed and accuracy.
- Mask R-CNN: An extension of Faster R-CNN, proposed by Kaiming He et al., Mask R-CNN adds an extra step for object segmentation, allowing the model not only to detect and classify objects but also to generate segmentation masks.

One-stage models like YOLO and SSD offer fast and efficient solutions for real-time applications. On the other hand, two-stage models like Faster R-CNN and Mask R-CNN provide higher accuracy and robustness, making them ideal for applications requiring precise object detection.

3. Approach

3.1. Presentation of YOLOv8

In the field of computer vision, the YOLO (You Only Look Once) series is well known for its efficiency and accuracy in real-time object detection. YOLOv8 is developed by Ultralytics. It brings significant improvements in both performance and ease of use compared to earlier versions like YOLOv5. This new version offers high accuracy and fast processing speed, making it stand out from previous versions thanks to its better optimizations.

3.2. Advantages and Performance of YOLOv8

YOLOv8 stands out for its optimal balance between accuracy and speed, making it especially suitable for demanding applications such as security surveillance, autonomous navigation, and object recognition in various industrial and urban environments. The improved accuracy is supported by fine-tuned network architecture, including better attention mechanisms and more advanced feature fusion strategies.

A comparison between YOLOv8 and previous versions like YOLOv7, YOLOv6.2.0, and YOLOv5.7.0 shows clear improvements in both accuracy and processing performance. As shown in the graph below, YOLOv8 achieves higher mean Average Precision (mAP) while maintaining or even reducing latency, demonstrating its enhanced efficiency.

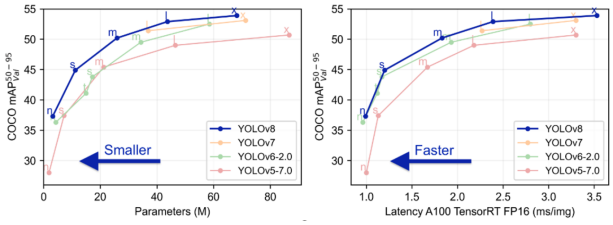

Figure 1. Performance comparison between YOLOv8 and its predecessors: accuracy and speed

Similarly, the recent evaluation of YOLOv8 on the Roboflow 100 benchmark demonstrated its excellent ability to generalize to new domains. This benchmark, derived from Roboflow Universe, includes 100 datasets representing various fields. YOLOv8 outperformed its predecessors, YOLOv5 and YOLOv7, showing fewer outliers and a higher overall mAP. Performance graphs confirm that YOLOv8 exhibits less variance and better average performance, consistently surpassing previous models across different Roboflow 100 categories.

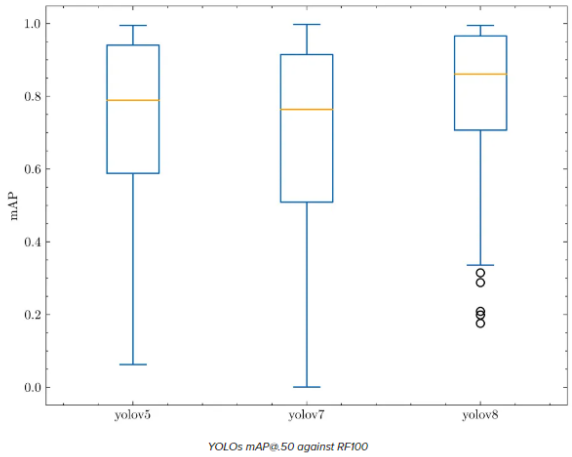

Figure 2. Analysis of accuracy on the RF100 benchmark: Comparison of YOLOv5, YOLOv7, and YOLOv8 versions

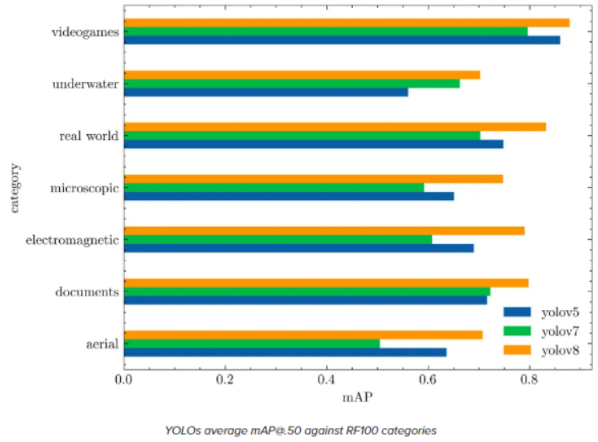

Figure 3. Accuracy evaluation of YOLO models in various application domains

3.3. Architecture of YOLOv8

The architecture of YOLOv8 has three main parts: the backbone, the neck, and the head. Each of them plays an important role in object detection.

- **Backbone**: The backbone of YOLOv8 uses CSP blocks to extract important details from images efficiently while reducing unnecessary calculations. It splits the input into two parallel paths: one path goes through several transformations with convolutions and residual connections, while the other path skips these steps and joins back at the end. The choice of parameters such as kernel size, stride, and padding is important because it controls the size and details of the output, directly affecting both accuracy and speed.

A key element in the backbone is the SPPF (Spatial Pyramid Pooling-Fast) block. This is an improved version of the traditional Spatial Pyramid Pooling (SPP) technique. The SPPF helps the network handle images of different sizes and captures details from multiple scales. It applies different types of pooling to the extracted features, creating a fixed-size output regardless of the input size. This is important for the next layers, which require input of a standard size.

- **Neck**: The neck is based on the FPN (Feature Pyramid Network) architecture. Enhances the features extracted from the backbone by using up-sampling operations. This means that it aligns feature maps of different resolutions to make sure that small and large objects can be detected effectively. The process creates a feature pyramid that helps in detecting objects at different levels of detail. Short-cut connections between the backbone and the neck, as well as between the neck and the head, help to transfer information efficiently and improve learning.

- **Head**: The head is responsible for predicting object classes and their bounding boxes. It processes the combined features of the neck feature pyramid to make the final predictions. This ensures fast and accurate object detection across different sizes.

YOLOv8 is an important step in object detection because it includes advanced techniques such as CSP blocks, SPPF, and C2F, along with optimized bottleneck structures. Each part of the model, from the backbone to the head, is designed to work together smoothly, providing high performance for real-time applications.

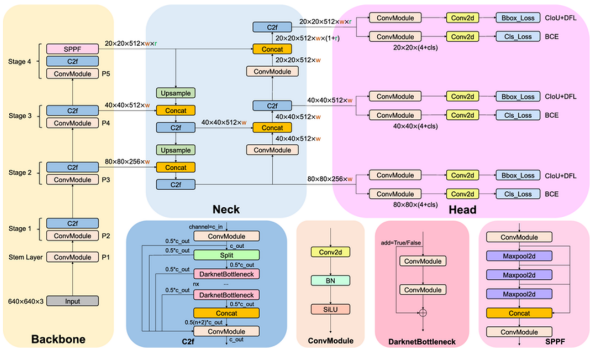

Figure 4. Detailed Architecture of YOLOv8: Components and Processing Flow

3.4. Integrating YOLOv8 with TensorBoard

Understanding and improving computer vision models like YOLOv8 from Ultralytics becomes easier when we closely examine how they are trained. Visualizing the training process helps us to see learning patterns, performance metrics, and overall behavior of the model.

Integrating YOLOv8 with TensorBoard makes this visualization and analysis process more effective, allowing for

better and more informed model adjustments.

TensorBoard is a TensorFlow visualization tool that is essential for machine learning experiments. It provides important tools to track machine learning models, including monitoring key metrics such as loss and accuracy, displaying model graphs, and showing weight and bias histograms over time. It also allows the projection of embeddings into lower-dimensional spaces and the display of multimedia data.

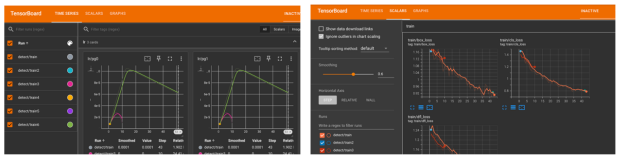

Figure 5. Visualization of Training Metrics with TensorBoard for YOLOv8

3.5. Conclusion

In summary, YOLOv8 stands out as a cutting-edge solution for object detection, combining impressive accuracy and speed with enhanced ease of use and accessibility through advanced visualization tools. Its growing popularity across various fields reflects its ability to meet the demands of modern computer vision applications, making YOLOv8 a preferred choice for researchers and professionals in the field.

4. Experience

4.1. Dataset

Now that I have chosen my model, I need to select the dataset for training.

The dataset I have chosen (see Figure 2.6) consists of 3,343 images of flowers, divided into 13 classes, which are as follows:

- Lantana commun
- Hibiscus
- Jatropha
- Souci
- Rose
- Champaka
- Chitrak
- Chèvrefeuille
- Guimauve indienne
- Melastome de Malabar
- Shankupushpam
- Lis araignée
- Tournesol

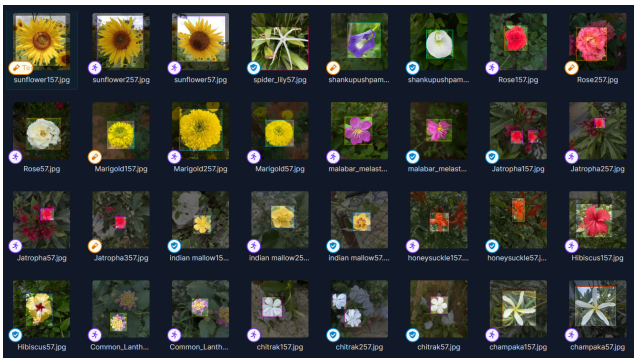

Figure 6. Sample from the Dataset

This dataset will be divided into three subsets:
The training set, which will be used to teach the model the characteristics of different flower classes.
The validation set, which helps evaluate the model's performance during training and adjust hyperparameters to prevent overfitting.
The test set, which is used to assess the model's final performance on unseen data to verify its ability to generalize.
To train the YOLO (You Only Look Once) model on this dataset, the data needs to be structured in a specific way(fig 6)

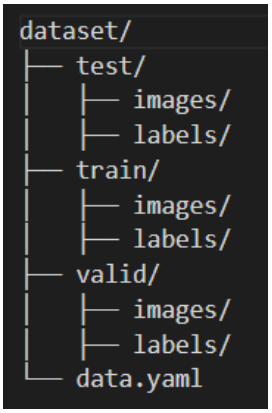

Figure 7. Structure of the Dataset

In the "labels" folders, there are '.txt' files that contain the image annotations. Each line corresponds to an object to be detected in the image. The first number (an integer) represents the class number to which the object belongs. The second and third numbers correspond to the normalized coordinates of the object's center. Finally, the fourth and fifth numbers indicate the object's width and height.

The presence of the YAML file is essential. This file contains the paths to the different image sets and their corresponding labels. It defines the configuration and structure of the data so that the model can use them correctly during training.

By analyzing the dataset, I noticed that the objects to be detected are all positioned in the center of the images and are relatively large (see Figure 8). This means that my model will only be able to detect objects that are mainly located in the center of the detection area.

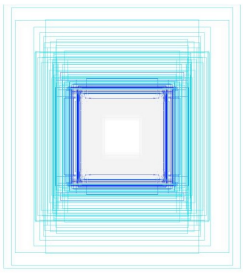

Figure 8. Bounding Box Locations in Images

This characteristic of the dataset has several implications for the performance of my YOLO model. Since all the objects to be detected are positioned in the center, the model may develop a dependency on this pattern. As a result, if objects are placed outside this central area during inference or if they are too small, the model might not detect them correctly.

To reduce this limitation, it could be beneficial to diversify the dataset by including images where objects appear in different positions and sizes within the frame. This would help the model learn to detect objects regardless of their location and size in the image, improving its robustness and generalization ability.

I found my dataset on Roboflow (at this link). Roboflow is a platform that allows downloading annotated image datasets that follow the structure required by various computer vision models, including the format compatible with YOLOv8 (see Figure 7). This structure is crucial to ensuring a smooth integration of the data into my training and inference pipeline.

Roboflow significantly accelerated my development process, saving me valuable time. I did not have to manually capture and annotate images, which would have required a lot of time and effort. However, it is important to note that relying on available datasets from Roboflow somewhat limits my flexibility and ability to customize the dataset according to specific needs.

4.2. Training Process

Now that I have both the model and the dataset, I can proceed with training. For this, I will use the Ultralytics package, which was developed to simplify the training and deployment of object detection models, particularly for YOLOv8.

I will train the model on my custom dataset, allowing it to adjust its parameters to better recognize the object classes of interest—in this case, flowers. During training, the model will refine its ability to detect different flower species by continuously learning from the dataset.

At the end of the training process, the model will generate a '.pt' file, which is a PyTorch file containing the learned parameters. To make the model more suitable for deployment, I will convert this '.pt' file into a '.tflite' format. The '.tflite' format is widely used for deploying object detection models on mobile devices due to its compatibility, efficiency, and optimized performance.

Since training deep learning models requires significant computational power, I utilized a GPU (Graphics Processing Unit). GPUs, with their parallel processing capabilities, can handle large amounts of data efficiently, significantly reducing training time. This allowed me to reach up to 100 epochs in a reasonable timeframe.

4.3. Performances

Once the model is trained (over 100 epochs), I can evaluate the performance of the selected weights (saved in the '.pt' file).

First, I analyze the training curves (see Figure 9). These curves represent different performance metrics across epochs. The performance is measured on the training set when the title contains 'train' and on the validation set when it contains 'val'.

4.3.1 Loss Functions

The box loss measures the error between the predicted bounding boxes and the actual boxes. This loss is computed using the bounding box regression loss function:

$$L_{\text{box}} = \lambda_{\text{coord}}(\hat{x}-x)^2+(\hat{y}-y)^2+\lambda_{\text{size}}(\hat{w}-w)^2+(\hat{h}-h)^2 \quad (1)$$

where: - $\hat{b} = (\hat{x}, \hat{y}, \hat{w}, \hat{h})$ are the model's predicted values for the bounding box center $(\hat{x}, \hat{y})$, width $(\hat{w})$, and height $(\hat{h})$.
- $b = (x, y, w, h)$ are the actual coordinates, width, and height.
- λ_{coord} and λ_{size} are hyperparameters that balance the different components of the loss.

The classification loss (L_{cl}) measures the error between the predicted object classes inside the bounding boxes and their actual classes. This loss is computed using the cross-entropy loss function, which compares the predicted class probabilities with the true class labels.

Let $\hat{p}_i$ be the probability predicted by the model for class i, and y_i be the actual class label (1 if the object belongs to class i, 0 otherwise). The classification loss is defined as:

$$L_{\text{cl}} = -\sum_{i=1}^{C} y_i \log(\hat{p}_i) \tag{2}$$

where C is the total number of classes.

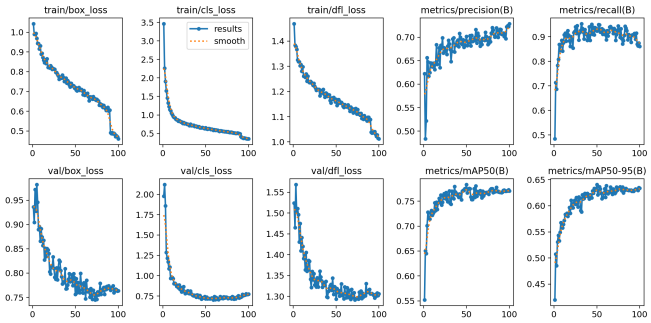

Figure 9. Training curves

4.3.2 Loss Functions and Performance Metrics

This formula strongly penalizes classification errors by increasing the loss when the predicted probability for the correct class is low.

The objectness loss (L_{df}) measures the error in predicting the presence or absence of objects within bounding boxes. This loss helps the model adjust its predictions to minimize false positives (predicting an object when there is none) and false negatives (not predicting an object when there is one). It is calculated using the binary cross-entropy loss function, as this is a binary classification problem.

Let $\hat{p}$ be the predicted probability of an object's presence in a bounding box, and y be its actual presence (1 if an object is present, 0 otherwise). The objectness loss is expressed as:

$$L_{\text{df}} = -[y \log(\hat{p}) + (1 - y) \log(1 - \hat{p})] \tag{3}$$

This formula penalizes incorrect predictions by increasing the loss when the predicted probability of an object's presence is high. It also penalizes missed detections when the predicted probability of an object's presence is low.

4.3.3 Precision and Recall

Precision is defined as the ratio of correct predictions to the total number of predictions made (both correct and incorrect). It measures the likelihood that a model's prediction is accurate.

Let TP (True Positives) be the number of correct predictions (the model predicts an object, and an object is actually present) and FP (False Positives) be the number of incorrect predictions (the model predicts an object, but there is none). Precision is calculated as:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4}$$

Recall is defined as the ratio of correct predictions to the total number of actual objects present in the dataset. It measures the ability of a model to detect all objects that should be classified.

Let FN (False Negatives) be the number of missed predictions (the model does not predict an object when there is one). Recall is calculated as:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5}$$

4.3.4 Mean Average Precision (mAP)

The mAP@50 metric represents the mean Average Precision (mAP) when the Intersection over Union (IoU) threshold is set to 0.5. In other words, a prediction is considered correct if the IoU between the predicted bounding box and the actual bounding box is greater than or equal to 0.5. The formula for mAP is:

$$\text{mAP} = \frac{1}{N} \sum_{i=1}^{N} AP_i \tag{6}$$

where: - N is the total number of classes, - AP_i is the Average Precision for class i.

The mAP@50-95 metric is the mean Average Precision computed across multiple IoU thresholds, ranging from 0.5 to 0.95 with a step size of 0.05. This provides a stricter and more detailed evaluation of the model's performance. The formula for mAP@50-95 is:

$$\text{mAP}_{50-95} = \frac{1}{10} \sum_{t=0.5}^{0.95} \text{mAP}_t \tag{7}$$

where mAP_t represents the mean Average Precision at a specific IoU threshold t.

By examining the training curves, I observed that errors steadily decreased in the train curves, indicating that the model successfully learned key patterns within the dataset. However, after around 80 epochs, the validation curves showed slight fluctuations, suggesting that the model had reached a stable point in its learning process. This indicates that further training might not significantly improve performance and could risk overfitting.

The confusion matrix highlights that the model performs well overall but still encounters challenges in distinguishing certain flower classes, particularly those that are visually similar. One noticeable issue is the misclassification

of Jatropha, which is likely due to incomplete annotations rather than actual model errors. After adjusting for annotation inconsistencies, the overall classification error remains around 10

Regarding model deployment, converting the trained model from .pt to .tflite results in a slight drop in performance, as expected. However, this trade-off is justified by the reduced memory size, making the model more suitable for deployment on mobile or edge devices. The mean Average Precision (mAP) and other key performance metrics remain at competitive levels, ensuring that the model maintains strong detection capabilities.

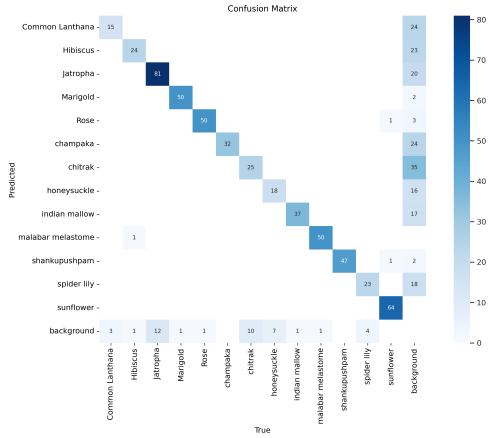

Figure 10. Confusion Matrix

4.4. Android Application Overview

The application is designed to process real-time video from the rear camera, analyze it using an object detection model, and display bounding boxes around detected objects. Users can select different models through a menu, allowing the detection of various object categories such as animals and flowers.

4.4.1 Development and Structure:

The app is developed in Kotlin, a modern programming language compatible with Java libraries. It consists of four main classes and an interface:

- **Main ACtivity**: Manages the application's lifecycle, initializes the camera, handles user interactions, and connects different components.
- **Detector**: Loads the .tflite model, processes video frames, and detects objects by converting images into tensors.
- **Overlay**: Defines the visual layer where detected objects are displayed using bounding boxes.
- **Box**: Handles the bounding box properties, ensuring correct positioning and formatting for the overlay.
- **DetectorListener** (Interface): Defines the actions to be performed when an object is detected.

4.4.2 Functionality and Workflow:

- **Camera Integration**: The cameraProviderProcess is initialized asynchronously using ListenableFuture, ensuring smooth operation without blocking the main UI thread. This prevents delays in launching the application while the camera is being configured. Once initialized, the camera captures frames in real-time, converts them into Bitmaps, and sends them for processing.
- **Image Processing and Model Inference**: To handle image analysis efficiently, a separate thread (imageAnalyzer) segments the video stream into individual images. Each image undergoes preprocessing, including resizing and format conversion, before being transformed into a TensorImage. This step ensures compatibility with the YOLOv8 model's expected input dimensions. The processed images are then passed to the model, which generates a vector containing bounding box coordinates, confidence scores, and class labels for detected objects.
- **Result Filtering and Display** To ensure accuracy, only objects with high confidence scores are retained. The function calculateIoU (Intersection over Union) is used to remove redundant detections and keep only the most relevant bounding boxes. The filtered bounding boxes are then converted into Box instances, which store their parameters before being visually displayed using the Overlay class. This allows detected objects to be clearly marked in the camera feed in real time.

4.4.3 Multi-Threading for Efficiency

To ensure smooth performance and prevent UI lag, the app utilizes multiple threads to manage different tasks efficiently. The **UI thread** is responsible for handling user interactions and updating the display, ensuring a seamless experience. **cameraExecutor** captures images from the camera and prepares them for processing, offloading this task from the main thread to avoid delays. Meanwhile, **model threads** are dedicated to tensor processing and model inference, allowing the application to analyze frames and detect objects in real-time without affecting responsiveness.

4.4.4 Model Management and Customization

The app supports multiple detection models that can be dynamically switched through a menu. Models and label files (.tflite and labels.txt) are stored in /assets/. The app retrieves model metadata, automatically adapting image processing settings to match the model's input requirements.

This application efficiently integrates real-time object detection with a smooth and responsive interface. By leveraging multi-threading, optimized TensorFlow Lite in-

ference, and a structured codebase, it ensures fast detection and minimal computational overhead, making it well-suited for mobile deployment.
This app's development was guided by concepts from this GitHub repository: (Click here)

5. Conclusion

The integration of YOLOv8 into this project allowed me to develop a high-performing object detection model trained on a custom dataset. Despite challenges related to dependency incompatibilities, I successfully converted the model into the .tflite format and integrated it into the mobile application. This ensured that the model could run efficiently on a mobile device while maintaining good detection performance.

The mobile application effectively combines real-time camera processing, image segmentation, model inference, and result visualization. By leveraging multi-threading and TensorFlow Lite, it maintains smooth performance with minimal computational overhead. The system is optimized to process video frames in real time, ensuring a responsive and efficient user experience.

If more time had been available, I would have added features such as saving detections, adjusting confidence thresholds, and customizing detection parameters. Additionally, with greater computational resources, training on a more diverse dataset would have improved the model's ability to generalize across different environments and object variations.

Overall, this project demonstrates the successful integration of YOLOv8-based AI into a real-world mobile application, balancing performance, efficiency, and adaptability while laying the groundwork for future enhancements.