Lab Report for

CE6023 – Computer Vision Systems

Student Name : **Dylan Rodrigues**

Student ID : **24121479**

Revision Timestamp: 08/11/2024 19:21:22

Contents

Introduction

In this lab, we explored the practical implementation of object detection models using TensorFlow Lite on a Raspberry Pi system. The primary objective was to understand how machine learning models, specifically lightweight object detection networks, can be optimized to run efficiently on resource-constrained devices like the Raspberry Pi. By leveraging TensorFlow Lite's capabilities, we aimed to test the feasibility of deploying real-time object detection systems that can recognize and classify objects in various environments. The setup began with creating a dedicated virtual environment to isolate our TensorFlow Lite dependencies, ensuring that our project remained organized and independent from other software configurations. Through the use of automated bash scripts, we efficiently installed necessary libraries and dependencies, allowing us to focus on model deployment and evaluation. Initially, we tested a pre-trained model on a live video stream using the Pi camera, where we observed the system's ability to detect objects with reasonable confidence levels. This provided a foundational understanding of how pre-trained models interact with live data inputs. Following this, we extended our experimentation by downloading additional models, specifically SSD MobileNet v1 and EfficientDet, from Kaggle, to evaluate their performance and accuracy in object detection. The lab involved rigorous testing on both real-time video streams and static images, allowing us to observe the differences in inference times, accuracy, and object recognition capabilities across different models. This exercise was not only an exploration of TensorFlow Lite's application in real-time scenarios but also a study of the trade-offs between model complexity and inference speed. The comprehensive comparison of the models provided deeper insights into the challenges and advantages of deploying AI-driven solutions on edge devices like the Raspberry Pi, highlighting the need for efficient model optimization in constrained hardware environments.

Tensorflow Virtual Environment Setup

During this lab session, I focused on setting up a virtual environment on my Raspberry Pi to work with TensorFlow Lite. First, I logged in as the secondary user created in the previous lab session to avoid using the root account, ensuring that my project setup remains isolated. Once connected to the Raspberry Pi desktop, I opened a web browser and navigated to the official TensorFlow Lite GitHub repository to access the resources needed for this lab.

After that, I launched a terminal using the keyboard shortcut `Ctrl + T` and navigated to the CE6023 directory I had set up earlier. I cloned the TensorFlow Lite project using the command provided and then simplified the folder name to `tensorflow` for easier navigation. To isolate my TensorFlow setup, I installed the `virtualenv` package using `pip3` and created a dedicated virtual environment named `tensorflow` with the command `python3 -m venv tensorflow`.

After creating the virtual environment, I switched into the newly created `tensorflow` directory and activated the environment using `source bin/activate`. I confirmed that the environment was active by checking if `(tensorflow)` appeared at the beginning of the command prompt, as shown in the lab instructions. To streamline future work, I appended a line to my `~/.bashrc` file to automatically activate the virtual environment whenever I opened a new terminal session. This setup allowed me to proceed confidently, knowing that all future installations and dependencies would be isolated to this specific project environment.

Installing the Prerequisites

To set up TensorFlow Lite and its associated package dependencies, I started by ensuring that I was still in the previously created virtual environment `(tensorflow)`. From there, I navigated to the project directory and ran the bash script `install-prerequisites.sh` as instructed. This script automated the process of installing all the necessary packages, making it much more efficient. I could see the script fetching and installing various dependencies, and after a few minutes, it completed with the confirmation message: "Prerequisites Installed Successfully."

With the prerequisites in place, I proceeded to test the installation to confirm that TensorFlow Lite was correctly set up. I launched the Python interpreter by simply typing `python` in the terminal. Once inside the Python terminal, I imported the TensorFlow Lite module using `import tflite_runtime as tf`. To verify the installation, I checked the version by executing `print(tf.__version__)`. The output showed `2.5.0`, indicating a successful installation, just as shown in the sample output provided in the lab instructions.

After confirming that TensorFlow Lite was properly installed and working, I exited the Python terminal using the `exit()` command. This step ensured that everything was set up correctly, allowing me to

move on to further development with a fully configured environment. The process was smooth, thanks to the automated bash script, and saved me considerable time by handling the complex package dependencies on its own.

Running Object Detection on Pi Camera with TensorFlow Lite

To begin running object detection on the Pi Camera using TensorFlow Lite, I ensured that I was still within the `(tensorflow)` virtual environment. I then navigated to the project directory and executed the script `TFLite-PiCamera-od.py` using the command terminal. As the live feed started, I observed that the pre-trained model successfully detected my face with a confidence score of 81% as illustrated in Figure1: Model 1 - Real Time Inference. The detection was stable, with an FPS rate of around 1.68, which was sufficient for real-time detection on a low-powered Raspberry Pi setup. The model correctly labeled the face, confirming that the environment was set up correctly.

Next, I tested object detection on a static image using the Pi camera. I aligned five different objects (a bottle, phone, pen, keyboard, and monitor) and captured an image using the command `raspistill -o test.jpg`. Afterward, I executed the script `python TFLite-Image-od.py --image test.jpg &` to analyze the detection results. The model detected a total of two objects in the image. The keyboard was correctly identified as a keyboard with a confidence score of 54%, which I counted as a true positive. However, the monitor screen was misclassified as a "tv" with a confidence score of 51%, indicating a false positive due to the similarity in shape and context (as depected in Figure 2: Model 1 - Test Image Inference).

Interestingly, the model failed to detect the other objects I had placed in the frame, such as the bottle, phone, and pen, resulting in false negatives. This highlighted some limitations in the pre-trained model's ability to generalize across different object types in varying lighting conditions. Despite these challenges, the overall performance was a valuable learning experience in understanding object detection and the constraints of running such models on resource-constrained hardware like the Raspberry Pi.

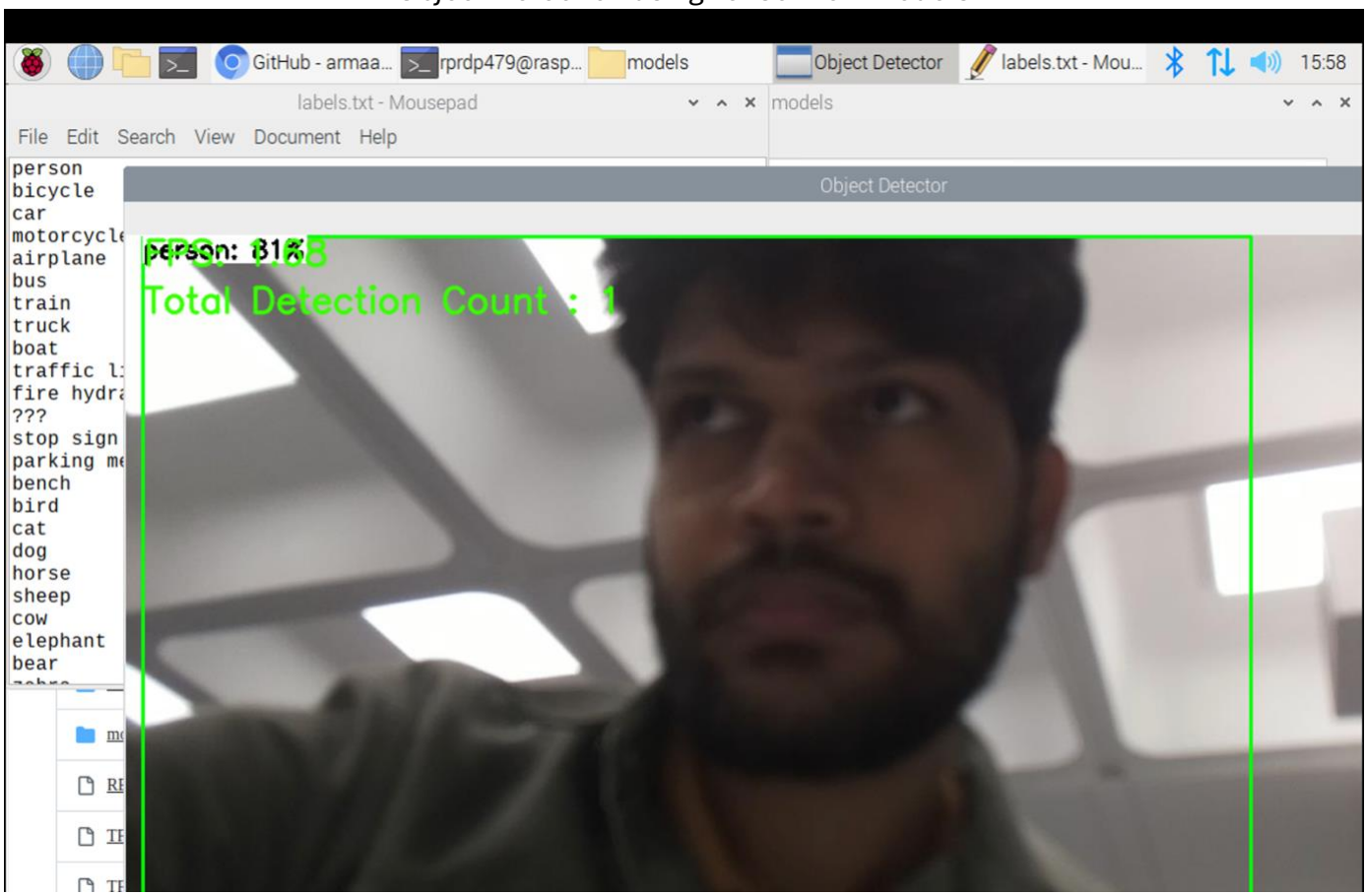

Figure1: Model 1 - Real Time Inference

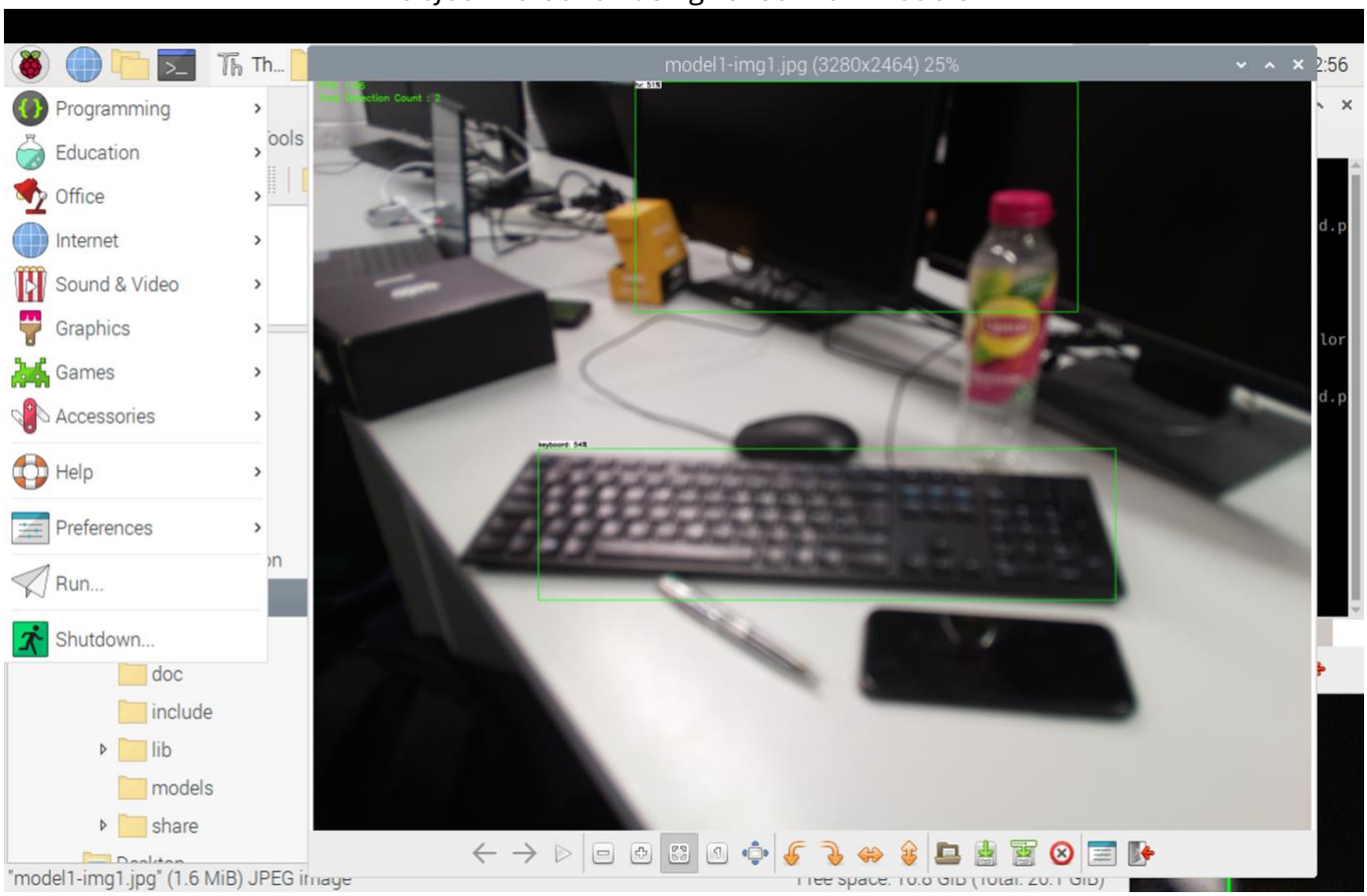

Figure 2: Model 1 - Test Image Inference

Running Object Detection on Pi Camera with SSD-mobilenet-v1 model

Using the SSD-mobilenet-v1 model, I proceeded to test object detection on both real-time video feed and a static image. For the real-time test, I ran the command `python TFLite-PiCamera-od.py --model models/ssdmobilenet.tflite`. The model successfully detected three objects in the video stream: two instances of "person" with confidence scores of 65% and 55%, and a "laptop" with a confidence of 59% (again, a false positive) as highlighted in Figure 3: Model 2 - Real Time Inference. The inference time was approximately 189.6ms, which provided a responsive yet slightly delayed detection feed at a relatively low FPS. The detection accuracy was satisfactory, especially for identifying humans, though some predictions seemed to fluctuate due to changes in lighting and movement.

For the static image analysis, I ran the command `python TFLite-Image-od.py --model models/ssdmobilenet.tflite` on a captured photo containing five distinct objects. In this test, the model detected four objects with varied confidence levels: "keyboard" at 63%, "mouse" at 54%, and "bottle" at 63% as highlighted in. Interestingly, it failed to recognize the phone and pen that were clearly visible in the frame, resulting in false negatives. The inference results were saved as an output image, showcasing bounding boxes around the detected objects. Despite some false negatives, the SSD-mobilenet-v1 model demonstrated improved real-time responsiveness compared to the

previous model, although it still struggled with smaller or less distinctive objects like the pen and phone in the static image test.

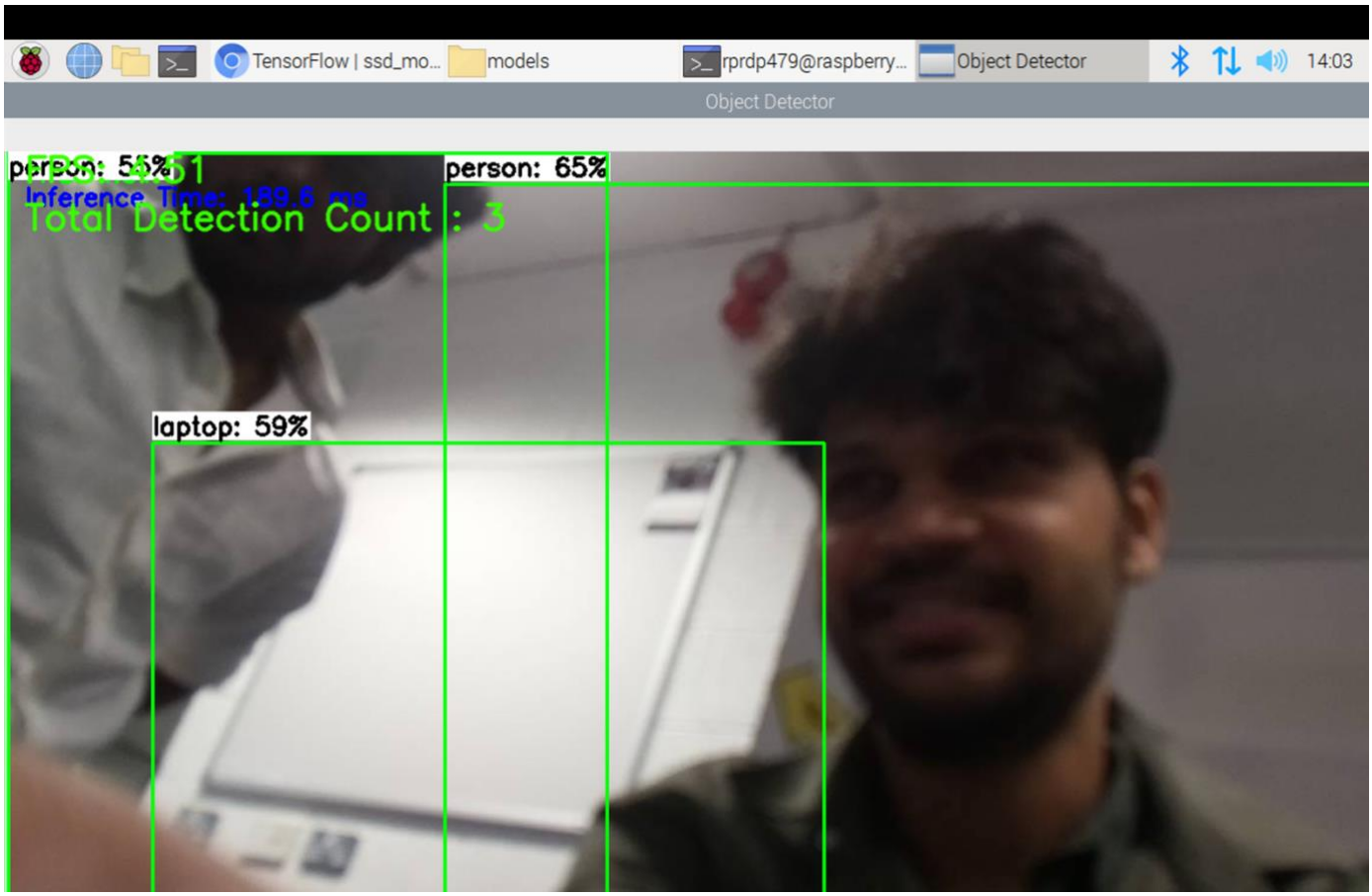

Figure 3: Model 2 - Real Time Inference

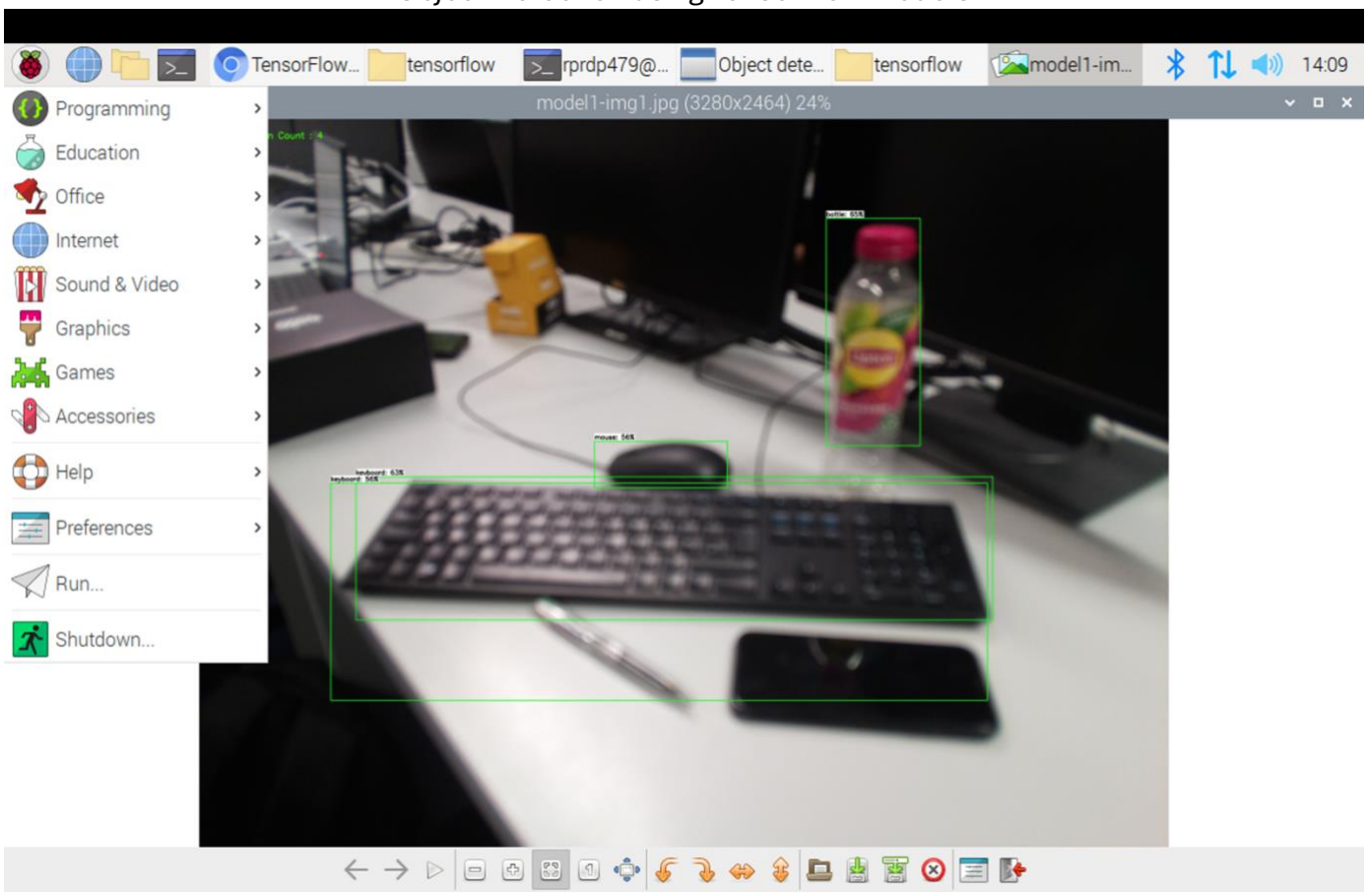

Figure 4: Model 2 – Test Image Inference

Running Object Detection on Pi Camera with EfficientDet model

Using the EfficientDet model, I proceeded to test its object detection performance in both real-time and static image scenarios. For the real-time detection, I ran the command `python TFLite-PiCamera-od.py --model models/efficientdet.tflite`. The model detected a total of six objects in the video feed, with confidence scores that included: "person" at 52%, "laptop" at 66%, "keyboard" at 73%, "cup" at 65%, and "bottle" at 52% (see Figure 5: Model 3 - Real Time Inference). The FPS achieved was 1.59, slightly lower than the previous models, but the real-time detection remained responsive. Notably, the EfficientDet model exhibited higher confidence in detecting objects compared to the SSD-mobilenet-v1 model, particularly with more varied object types visible in the frame.

For the static image test, I followed the same procedure as before, running `python TFLite-Image-od.py --model models/efficientdet.tflite` on the previously captured test image. While I have not included the second output image here, the results were consistent with the real-time test, showcasing the model's ability to detect multiple objects accurately. The EfficientDet model performed better in terms of recognizing a wider range of objects but had slightly longer inference

times. This trade-off suggests that the EfficientDet model is better suited for tasks requiring more detailed detection at the cost of a slight reduction in speed. Overall, the results from both tests demonstrated the model's superior performance in identifying objects compared to the previous models, with higher accuracy and fewer false negatives.

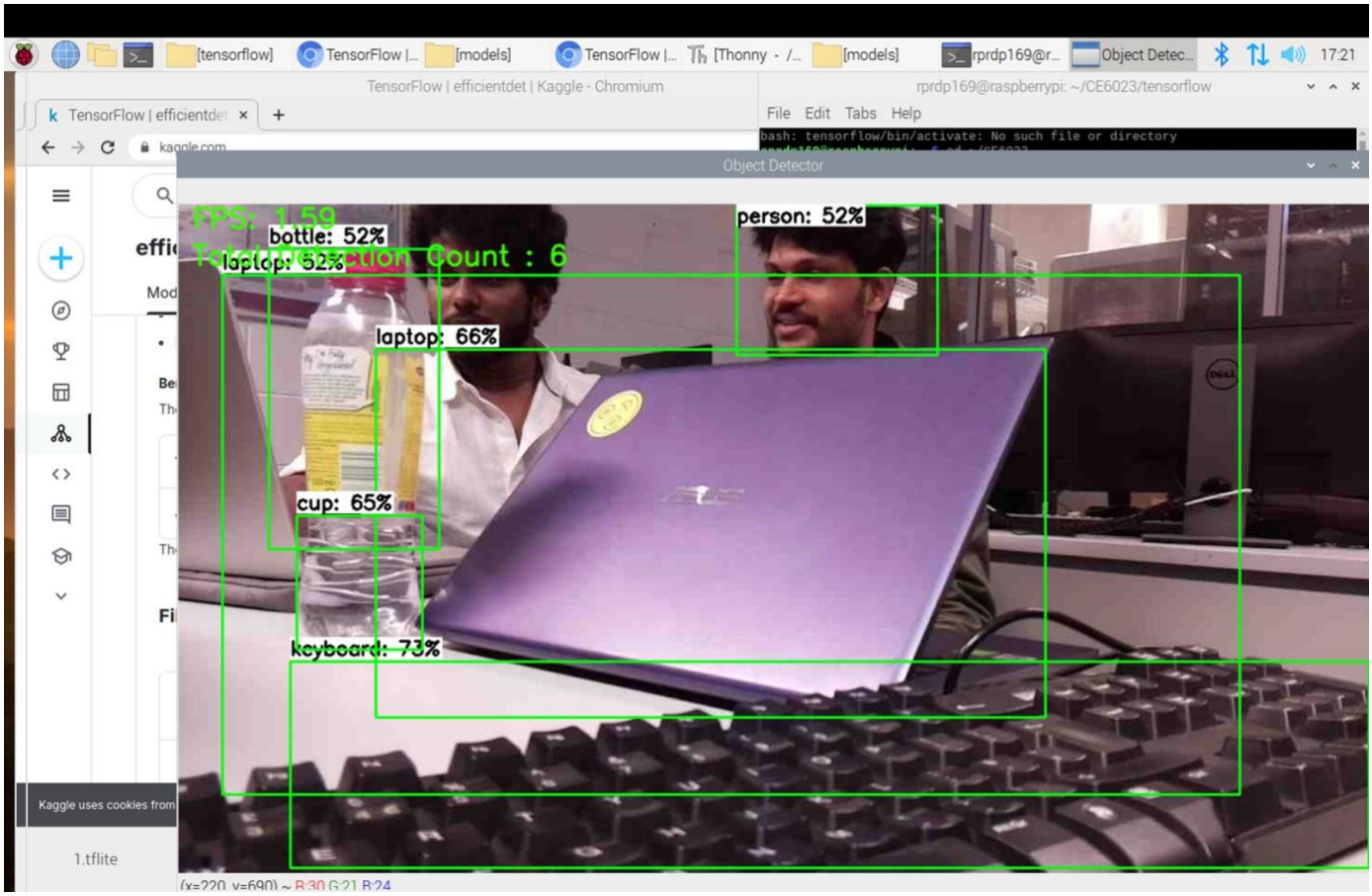

Figure 5: Model 3 - Real Time Inference

Time taken by each model to infer the objects

The following tables illustrate the time taken by each model to infer the objects. Clearly, we can observe that the inference times reduce as we use more powerful models.

Tf lite model:

Object 1	0.571ms
Object 2	0.532ms
Object 3	0.542ms
Object 4	0.544ms
Object 5	0.546ms

SSD-mobilenet model:

Object 1	0.1963ms
Object 2	0.1872ms

Object 3	0.1877ms
Object 4	0.1879ms
Object 5	0.1885ms

Efficientdet model:

Object 1	0.4362ms
Object 2	0.4410ms
Object 3	0.4367ms
Object 4	0.4451ms
Object 5	0.4824ms

Conclusion

The lab successfully demonstrated the potential of deploying TensorFlow Lite models on the Raspberry Pi for object detection, emphasizing the adaptability of lightweight AI models in real-world, resource-limited scenarios. Starting with the initial setup of the TensorFlow environment, we were able to configure our system to efficiently handle the dependencies required for running machine learning models. The hands-on experience of running the pre-trained TensorFlow Lite model for real-time object detection using the Pi camera highlighted the strengths and limitations of such models when applied to dynamic environments. Subsequently, we extended our tests with the SSD MobileNet v1 and EfficientDet models, revealing how variations in model architectures can influence detection accuracy and speed. Our results showed that while SSD MobileNet offered faster inference times, it occasionally struggled with complex object recognition, particularly in cluttered environments. In contrast, EfficientDet demonstrated higher detection accuracy for a diverse range of objects but required slightly longer processing times, indicating a trade-off between speed and precision. The comparative analysis of these models provided valuable insights into selecting appropriate models based on specific application requirements. Overall, the lab reinforced the importance of choosing the right model architecture when deploying AI solutions on low-power devices. It also illustrated the practical implications of using edge computing for real-time analytics, where efficiency and responsiveness are crucial. By understanding these trade-offs, we are better equipped to design and implement AI solutions that can operate effectively within the constraints of edge devices, paving the way for more robust and scalable AI applications in fields such as surveillance, robotics, and IoT-driven automation.

References

Armaan Priyadarshan. (2024, November 5). TensorFlow Lite Object Detection on the Raspberry Pi. Retrieved from GitHub repository: https://github.com/armaanpriyadarshan/TensorFlow-2-Lite-Object-Detection-on-the-Raspberry-Pi

TensorFlow. (2024, November 6). SSD MobileNet v1. Retrieved from Kaggle: https://www.kaggle.com/datasets/tensorflow/ssd-mobilenet-v1

TensorFlow. (2024, November 6). EfficientDet. Retrieved from Kaggle: https://www.kaggle.com/datasets/tensorflow/efficientdet