Project Report on

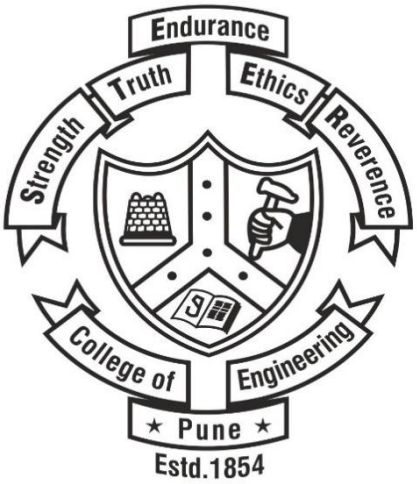
PICOBOO - SECURITY SYSTEM WITH MOTION TRACKING USING THERMAL IMAGING AND DEEP LEARNING

by

Anway Pimpalkar **111907066**
Kushagra Shrivastava **111907074**
Ishita Rathor **111807046**

Under the guidance of

Dr. Shrinivas Mahajan
Head of Department, Electronics and Telecommunications, COEP



COEP

DEPARTMENT OF

ELECTRONICS AND TELECOMMUNICATION ENGINEERING

COLLEGE OF ENGINEERING PUNE

2021 – 2022

Abstract

This report explores the development of security systems on embedded systems catering to low-budget, high reliability applications.

The "*PicoBoo*" system we developed is built on the foundations of thermal imaging and deep neural networks. The application is built and then deployed on a Raspberry Pi 4 Model B for testing and prototyping. We first collected data from a high-end thermal imaging camera, cleaned the data and made a usable training dataset of human images from it. To implement a model which enables tracking of humans in a thermal camera video stream, we used a reliable neural network architecture and gained a high accuracy coupled with a low latency on it. These detections were sent to a real-time cloud database, which was accessed through a web-based app showing whether a human was detected in the frame or not. This web application was deployed to a PaaS application, which can be easily accessed over web browser.

Contents

1. Introduction

Thermal imaging cameras are handheld electronic devices that detect heat energy and have an integrated visual display. A heat sensor is coupled to a specific type of lens in a thermal camera, which is then customized to work with regular image-capture technology. This enables engineers to quickly discover areas of high temperature or sources of lost heat energy in building inspections, such as overheating components or potential thermal insulation gaps. Warmer components or regions will appear as reds, oranges, and yellows on a color thermographic display, while cooler sections will appear as purples and blues (green usually indicates areas that are roughly at room temperature). Thermal cameras are excellent for finding heat sources in very dark or otherwise occluded conditions because they measure infrared radiation rather than visible light.

Home protection and family safety are the primary purposes of a home security system. We are deploying a security system in a given area that identifies, tracks and alerts a user of the presence of another human in the area. Using thermal imaging and deep learning, the presence and movement of a human can be detected in the surveillance-enabled environments. The owner will be notified of the human presence via a central alarm and a web app.

1.1 Literature Review

On a thorough survey of the existing security systems currently available in the market today, we observed that maximum systems are based on RGB imaging, not thermal. RGB imaging based systems have the drawback that they require adequate light from the visible spectrum to be always available in order to function optimally. This can be fixed using thermal imaging, which works in the IR spectrum even in the absence of visible light.

The current technology utilizes either of two main types of camera systems to capture data:

i. Active IR Cameras
ii. Thermal Cameras

Active IR systems use short wavelength infrared light to illuminate an area of interest. Some of the infrared energy is reflected back to a camera and interpreted to generate an image. Thermal imaging systems use mid or long wavelength IR energy. Thermal imagers are passive, and only sense differences in heat. These heat signatures (usually black (cold) and white (hot)) are then displayed on a monitor. Because thermal imagers operate in longer infrared wavelength regions than active IR, they do not see reflected light, and are therefore not affected by oncoming headlights, smoke, haze, dust, etc. This makes thermal imaging steeply preferable, but expensive as well. Such systems which work on thermal imaging are expensive and are not deployable to common uses such as shops, low-key management, and storage facilities, etc. There is a strong need of affordable yet reliable systems in the thermal security domain. The model we have built is different from other existing thermal security cameras as it is built using a microcontroller. It is cost effective. It works even in the dark which improves our security as it will be functional 24/7.

1.2 Aim & Problem Statement

To deploy a security system at a given area which identifies, tracks, and alerts a user of the presence of another human in the area.

1.3 Objectives

The objectives of the project are as follows:

i. Detecting a human using thermal imaging and machine learning.
ii. Tracking the motion of the human to follow it.
iii. Alert the user of the presence of another human – through a central alarm and a mobile app.

1.4 Technical Specifications

While building the project, we aimed to reach these benchmark specifications:

i. Camera Temperature Measurement Accuracy: ± 2° C
ii. Camera Field of View: 25° x 19°
iii. Accuracy of the Human Detection model: 85% for 75% IoU threshold.
iv. Latency of Human Detection model: < 0.2 s (minimum 5FPS)
v. Power Consumed by the Raspberry Pi: < 5 Watts

2. Hardware and Software Design

In this section, we have discussed the hardware and software components as well as concepts which we have used to build our security system with motion tracking using thermal imaging and deep learning.

2.1 Hardware Components

In this project, we have used available state-of-the-art equipment to build a prototype of the security system. They are detailed in this subsection.

2.1.1 FLIR T420bx Thermal Imaging Camera

The FLIR T420bx Thermal Imaging Camera is a state-of-the-art thermal imaging camera designed and manufactured by Teledyne FLIR *(Wilsonville, Oregon, United States)*. It has a temperature range of -20°C to 350°C and is cased in a portable handheld package.

Figure 2.1: FLIR T420bx Thermal Imaging Camera (Courtesy: Teledyne FLIR)

The camera is powered by Multi-Spectral Dynamic Imaging (MSX) technology which adds visible spectrum definition to IR images in real time for excellent thermal detail. It can provide a Frame Rate of upto 60Hz for video recordings, which we used during our data collection step. By default, it is equipped with a 25° lens. It supports uncompressed colorized video streaming over USB.

Figure 2.2: FLIR T420 bx Thermal Imaging Camera Used for the Project.

2.1.2 Raspberry Pi 4 Model B (2GB RAM Variant)

The Raspberry Pi is a low cost, credit-card sized computer that plugs into a computer monitor or TV and uses a standard keyboard and mouse. It is an extremely capable device that enables people of all ages to explore computing, and to learn how to program in languages like Scratch and Python. It's capable of doing everything you'd expect a desktop computer to do, from browsing the internet and playing high-definition video, to making spreadsheets, word-processing, and playing games. The Raspberry Pi 4 is a tiny single-board computer, which means that all its components, from the memory to the USB ports, fit on one PCB without add-on cards or accessories. It measures 2.2 by 3.4 inches and stands about 0.6 inch tall.

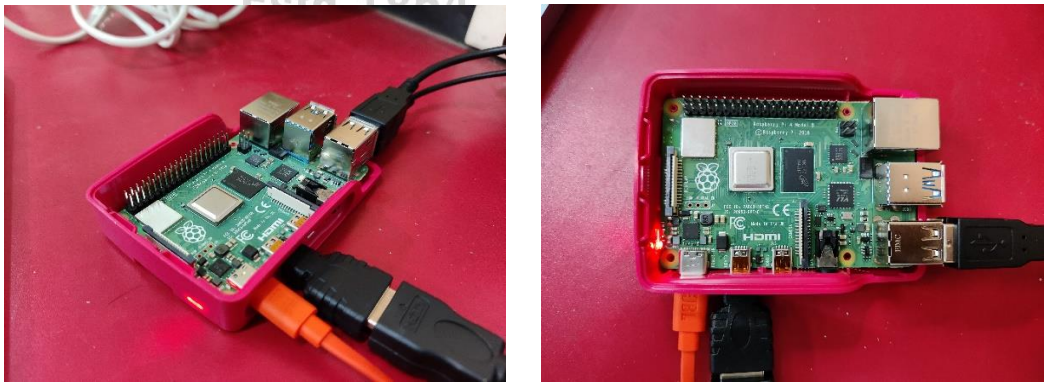


Figure 2.3: Raspberry Pi 4 Model B used in our project.

The specifications of the Raspberry Pi 4 Model B are as shown in Table 2.1.

Table 2.1: Raspberry Pi 4 Model B Specifications

Feature	Specification
Chip	BCM2711
CLK	1.5 GHz
RAM	2 GB
Flash Memory	16 GB (via microSD Card)
CPU	5V / 3.3V
GPU	VideoCore VI
USB 2.0	2
USB 3.0	2
HDMI v2.0	2
Ethernet	1GBps

2.2 Software Components

2.2.1 Google Firebase Realtime Database

The Firebase Realtime Database is a cloud-hosted NoSQL database that lets you store and sync data between your users in real-time. Firebase is a Backend-as-a-Service (BaaS). It provides developers with a variety of tools and services to help them develop quality apps, grow their user base, and earn profit. It supports authentication using passwords, phone numbers, Google, Facebook, Twitter, and more. The Firebase Authentication (SDK) can be used to manually integrate one or more sign-in methods into an app. Data is synced across all clients in real-time and remains available even when an app goes offline. Firebase Hosting provides fast hosting for a web app; content is cached into content delivery networks worldwide. The application is tested on virtual and physical devices located in Google's data centres. Notifications can be sent with firebase with no additional coding.

2.2.2 Heroku

Heroku is a container-based cloud Platform as a Service (PaaS). Developers use Heroku to deploy, manage, and scale modern apps. Our platform is elegant, flexible, and easy to use,

offering developers the simplest path to getting their apps to market. Heroku lets developers scale applications instantly. The simple way to scale applications makes working with Heroku easy and convenient. Projects created in Heroku are bound to repositories in GitHub. Heroku is known for running apps in dynos – which are just virtual computers that can be powered up or down based on how big your application is.

2.2.3 Front-End Web

The Front-End side is also called as the "Client-Side" of the application which includes everything the user sees and experiences. It includes text, navbars, colour-styles, images, buttons, etc. These help the user to understand and interact with the webpage.

Figure 2.4: Main Technologies used to Create the Front-End Web system.

2.2.4 Back-End Web

The Back-End side is also called as the "Server Side" of application. This side of the code doesn't come in contact of the user, but it manages and makes sure that everything on the client side works perfectly. Working with Backend involves storing/arranging data, writing APIs, creating libraries and working with the server requests.

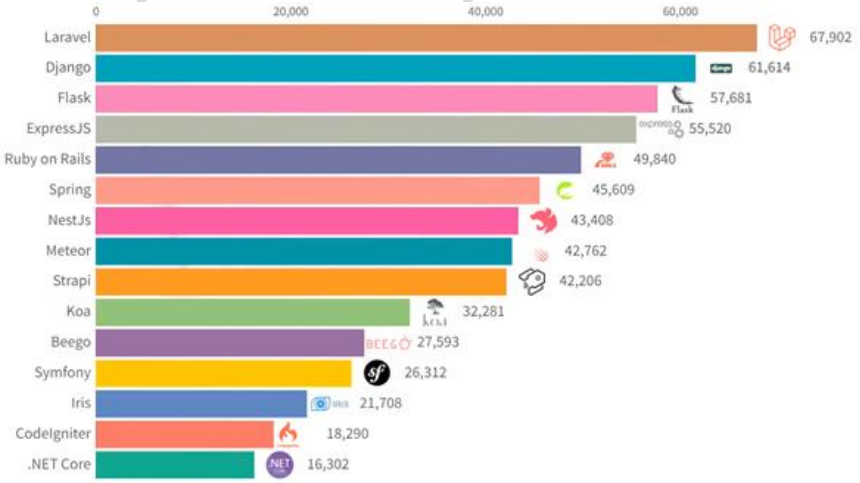


Figure 2.5: Most Popular Back-End Frameworks of January 2022.

2.3 Object Detection using Deep Learning on Embedded Systems

Machine learning is the science of getting computers to act without being explicitly programmed. These processes require a lot of computing power and memory to be present in the system on which it runs. A CNN model once compiled is in the order of MBs if not GBs. A typical GPU consumes power in the order of Watts. Deploying such models to embedded microcontrollers is a challenge because of the severe power and memory constraints.

The solution is to make use of various quantization methods on the models to shrink down the size and power requirements to be deployable to embedded microcontrollers. This is done by using various methods such as quantization, pruning, etc. Some accuracy is lost in the process due to the reduces computes. The key is to find the balance between the size, latency, and accuracy for any given application.

TensorFlow provides multiple APIs to streamline this process of shrinking down models to make them deployable to embedded systems. For this reason, TensorFlow has been used extensively throughout this project for building and deploying neural networks to our Raspberry Pi.

Object detection is a computer vision technique that works to identify and locate objects within an image or video. Specifically, object detection draws bounding boxes around these detected objects, which allow us to locate where said objects are in (or how they move through) a given scene.

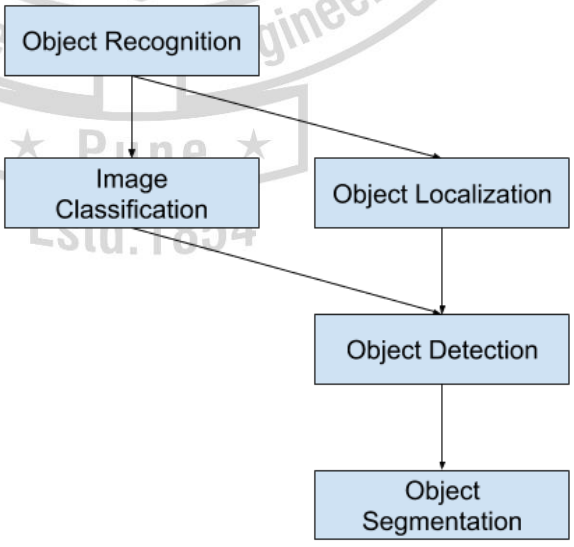


Figure 2.6: Basic Object Detection Algorithm.

To implement Object Detection, *EfficientNet* is a convolutional neural network architecture and

scaling method that uniformly scales all dimensions of depth/width/resolution using a compound coefficient. Unlike conventional practice that arbitrary scales these factors, the *EfficientNet* scaling method uniformly scales network width, depth, and resolution with a set of fixed scaling coefficients. The compound scaling method is justified by the intuition that if the input image is bigger, then the network needs more layers to increase the receptive field and more channels to capture more fine-grained patterns on the bigger image. We have used the quantized version of this architecture, *EfficientNet-Lite* for our model.

3. System Architecture

This section elaborates the methodologies employed to build our security system based on thermal imaging.

3.1 System Overview

The main aim of the system is to be robust and serve as a working prototype for a system which can be deployed in genuine security scenarios. The entire prototype is based on a Raspberry Pi 4 Model B, which has an array of features in-built into the board, of which we can take maximum advantage. A generic block diagram of the entire proposed system is represented in Figure 3.1.

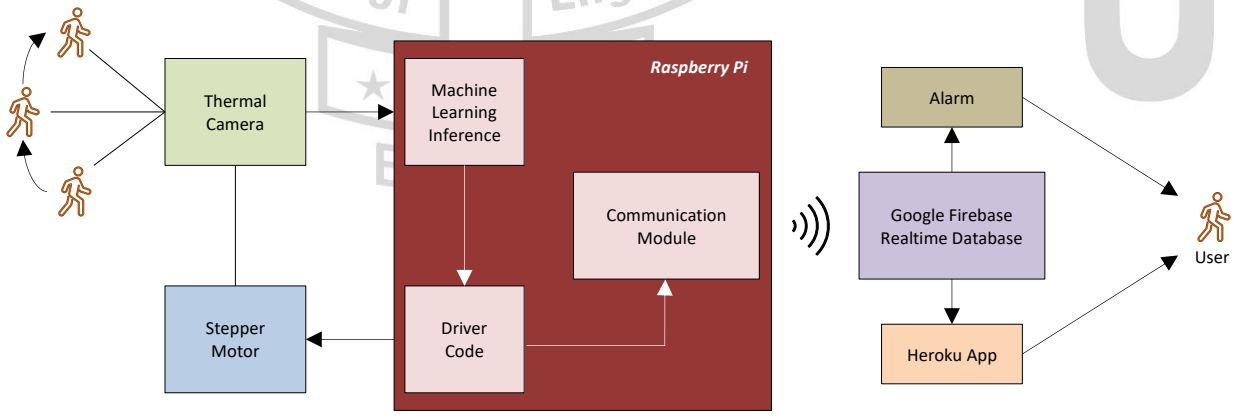


Figure 3.1: Generic Block Diagram for Implementation

The project is a three-part system encompassing the following subsystems:

i. Human detection and tracking using a thermal imaging camera using *TensorFlow Lite*.

ii. Web interface to show the status of the security system and stream the thermal camera stream to the application/interface.
iii. Drive a stepper motor to follow the motion of the human detected in the frame based on the position of the human in the frame.

To implement the project, we selected components and controllers based on their availability, suitability, feasibility, cost, and ease of use. Most components were available within our college laboratories, which we used to avoid unnecessary expenses.

3.2 Human Detection and Tracking

To implement this, we worked with *TensorFlow*. *TensorFlow* is a free and open-source software library for deep learning. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks. We used the concept of object detection to complete the human detection and tracking subsystem. We can summarize our work related to human detection and tracking as the following steps:

i. Collecting data using the *FLIR T420 bx Thermal Imaging Camera*.
ii. Cleaning the data, labeling it, and building a dataset suitable to train a detection model.
iii. Building a machine learning model capable of detecting humans in the frames.

3.2.1 Collecting Data and Building a Dataset

At the foundation of every artificially intelligent system is the data on which it is trained. Hence, we built a strong dataset which can later be diversified to be annotated into more than one type of objects. The data was collected using the *FLIR T420 bx Thermal Imaging Camera* using two methods, either by taking pictures and videos on the camera itself and extracting it from the SD card, or by connecting the camera to a computer and accessing the stream via the *FLIR Thermal Studio*.

Since the system we developed was for human intruder detection, we took images and videos focusing on two main labels:

i. Human subjects.
ii. Backgrounds

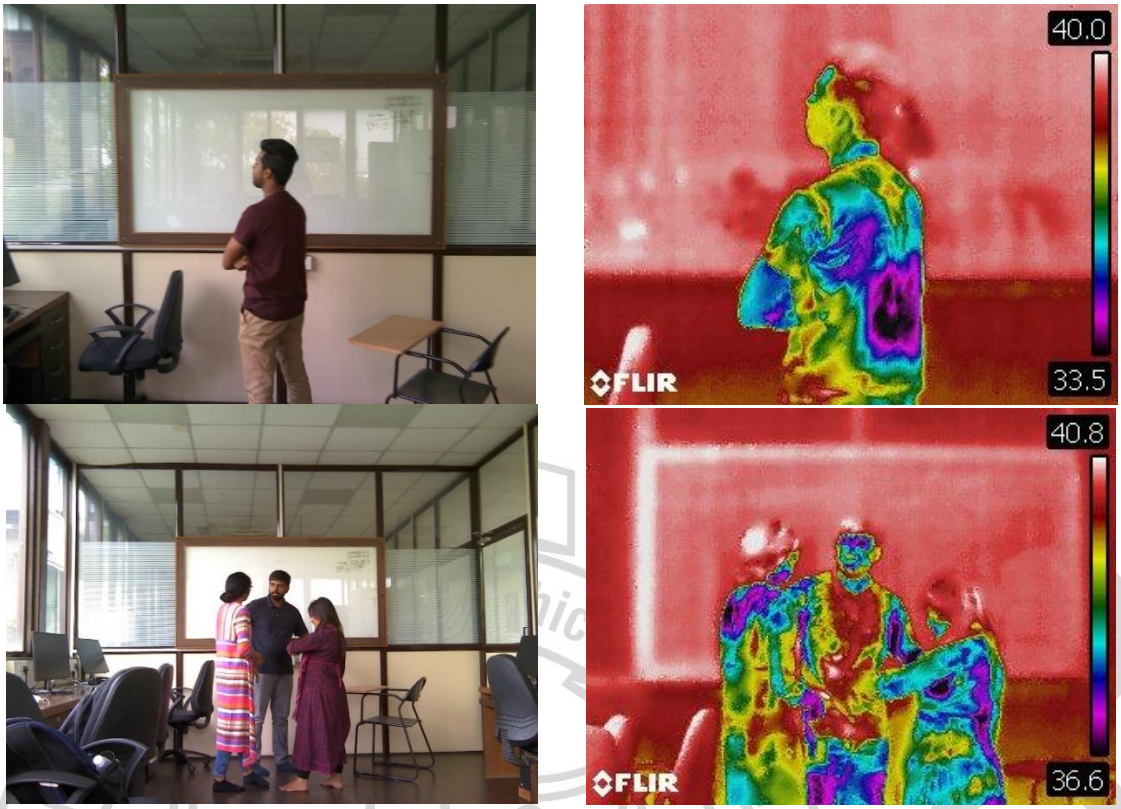


Figure 3.2: Sample images collected using the FLIR T420 bx Thermal Imaging Camera.

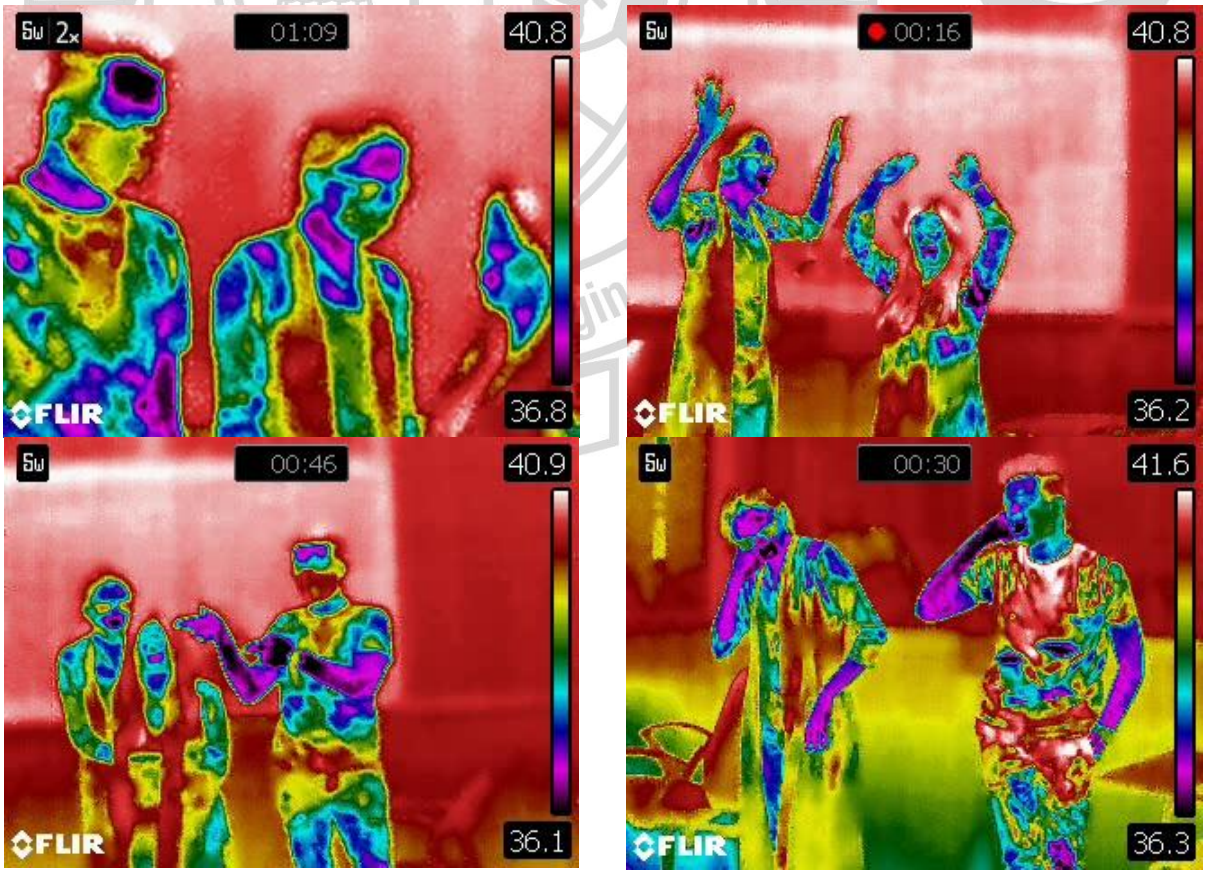


Figure 3.3: Images with multiple human subjects in a single frame.

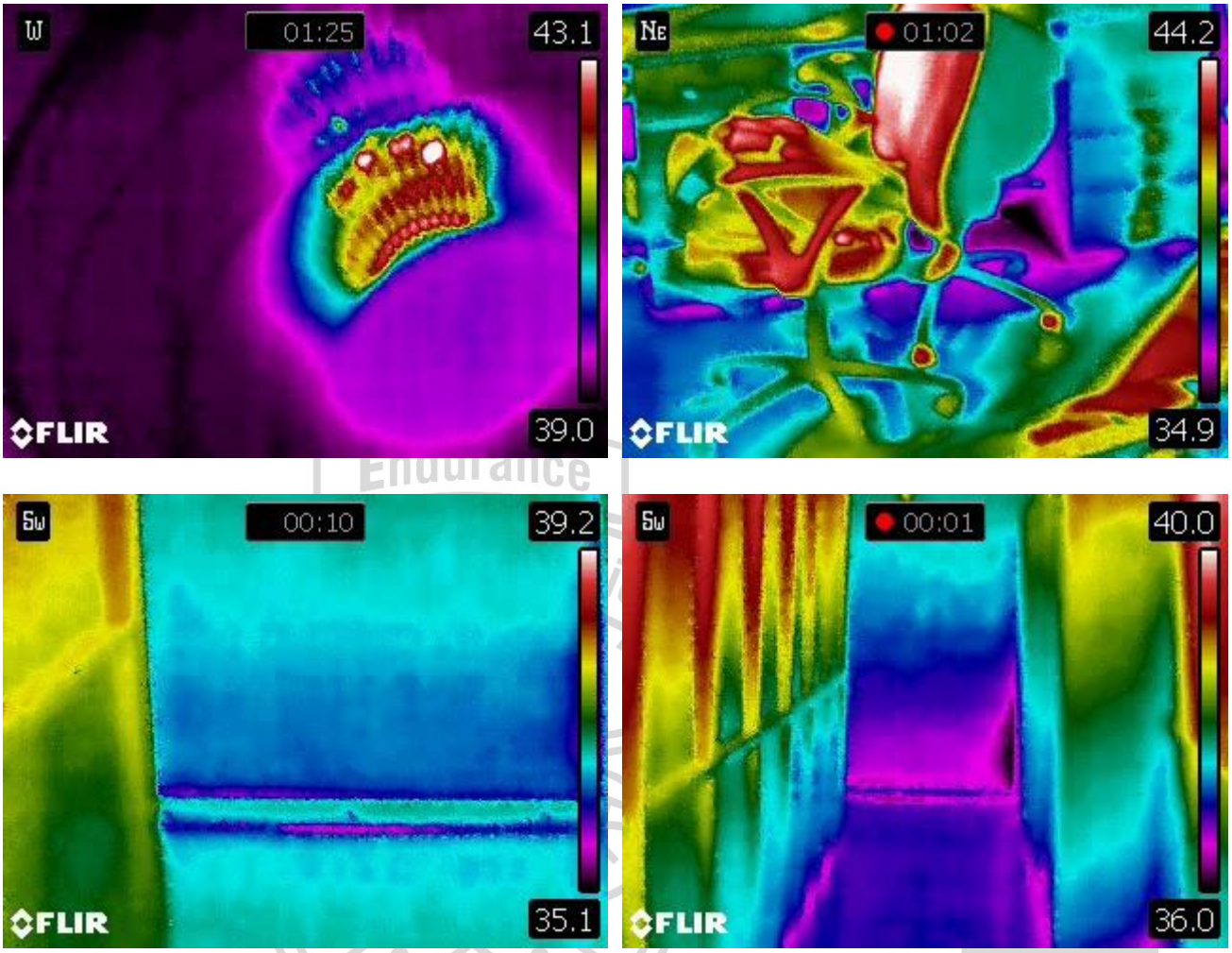
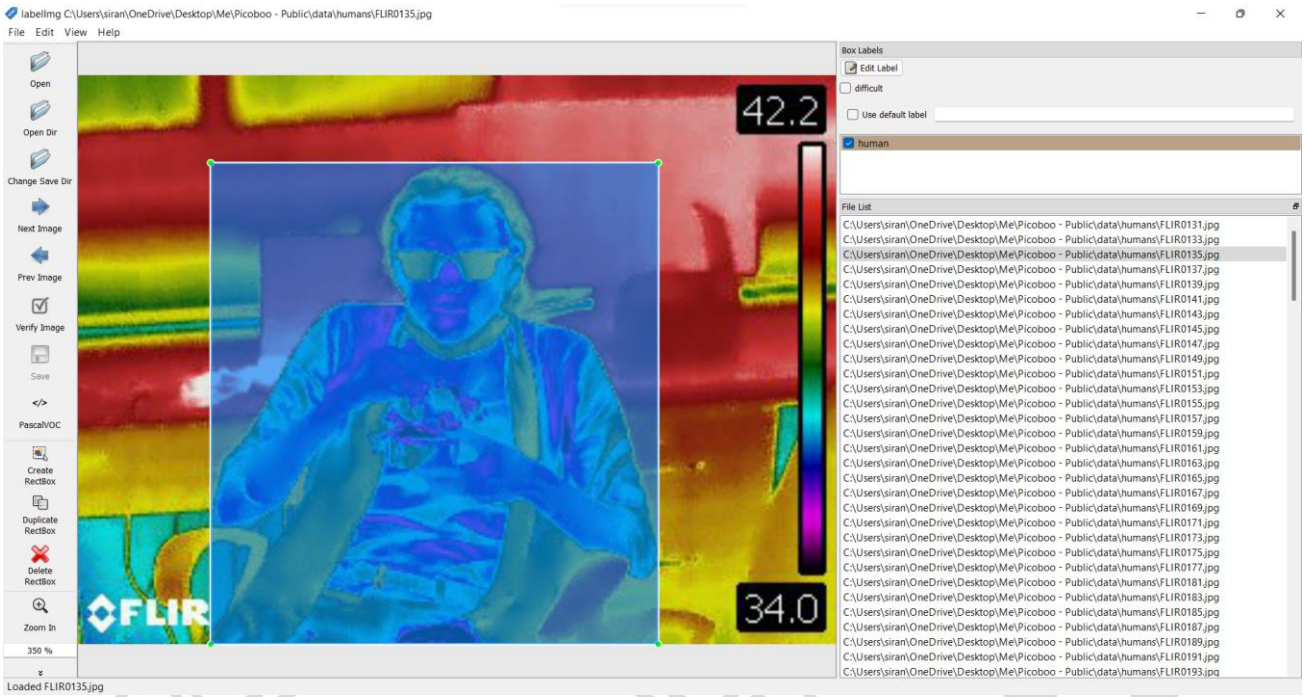


Figure 3.4: Images with no humans (only background).

A total of 90 photos and videos of human subjects, and 2 videos of background views were recorded. The videos were then split frame-by-frame using a Python script to create images which can be used to train the model. After this split and some preliminary data cleaning, we had a total of 6,067 images with human subjects and 4,937 instances of backgrounds.

Considering the time it would have taken to train our model, we worked with a small portion of this data to build the model, of 184 instances of humans, and 222 instances of backgrounds.

To label this data, we used a free open-source data labeler, *LabelImg*. The software allows for easy rectangular bounding of subjects in images, with annotations saved in XML files. This software had a limitation, it cannot add empty annotations to images with no human subjects. To fix this, we used a Python script to generate the annotations automatically for the background images from the dataset as discussed later in the Results section of the report.



C: > Users > siran > OneDrive > Desktop > Me > Picoboo - Public > data > humans > FLIR0135.xml

```xml
<annotation>
    <folder>humans</folder>
    <filename>FLIR0135.jpg</filename>
    <path>C:\Users\siran\OneDrive\Desktop\Me\Picoboo\FLIR T420br Data - Training Data\humans\FLIR0135.jpg</path>
    <source>
        <database>Unknown</database>
    </source>
    <size>
        <width>320</width>
        <height>240</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>human</name>
        <pose>Unspecified</pose>
        <truncated>1</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>56</xmin>
            <ymin>37</ymin>
            <xmax>245</xmax>
            <ymax>240</ymax>
        </bndbox>
    </object>
</annotation>
```

Figure 3.5: Annotations creating using LabelImg for the dataset.

In the data, we left the additional information captured as is, such as the temperature scale, the FLIR watermark and the current temperature reading. We hoped to create a model that would learn to ignore these features as background features. A summary of the created dataset is given in Table 3.1.

Table 3.1: Dataset specifications.

Human Instances Captured	6,067
Background Instances Captured	4,937
Human Instances Used for Training	184 (3.03%)
Background Instances Used for Training	222 (4.4%)
Total Size	39.6 MB

3.3.2 Object Detection Model using EfficientNet-Lite

The Object Detection model must have two key features – high accuracy and low latency. With extensive use of *TensorFlow*, we were able to build multiple models which took excellent advantage of the *TensorFlow Lite* library and performed well under limited power and memory environments such as the Raspberry Pi.

In May 2019, Google released a family of image classification models called *EfficientNet*, which achieved state-of-the-art accuracy with an order of magnitude of fewer computations and parameters. They looked to optimize these models for deployment at the edge, and in 2020 they launched *EfficientNet-Lite* which runs on *TensorFlow Lite* and is designed for performance on mobile CPU, GPU, and EdgeTPU. *EfficientNet-Lite* brings the power of *EfficientNet* to edge devices and comes in five variants, allowing users to choose from the low latency/model size option *(EfficientNet-Lite0)* to the high accuracy option *(EfficientNet-Lite4)*. Below are how the quantized *EfficientNet-Lite* models perform compared to similarly quantized version of some popular image classification models.

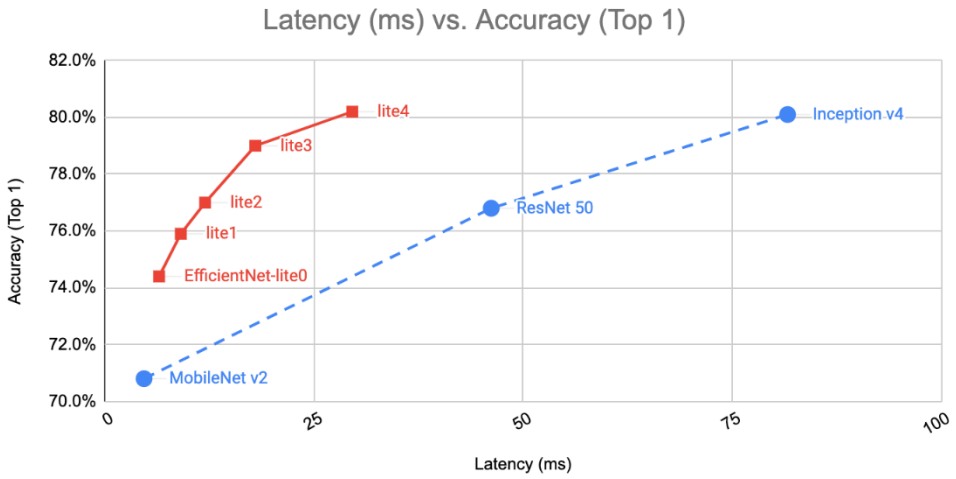


Figure 3.6: Comparison of EfficientNet-Lite models to different Image Classification models. (Courtesy: Google TensorFlow)

EfficientNet has a reputation for achieving high accuracy with minimal parameters and FLOPS (Floating Point Operations Per Second). It is suitable for use with the Raspberry Pi, which has limited processing power. We implemented transfer learning using the learned weights of *EfficientNet* from the ImageNet dataset since both FER-2013, and the ImageNet are image classification datasets. The architecture of the *EfficientNet-Lite0* network consists of 1x1 Convolution, Average Pooling, Convolution, Dense Connections, Dropout, Inverted Residual Block, Batch Normalization, and ReLU6.

To train our models, we used this *EfficientNet-Lite* series and checked their performance for our use-case. We started with the *EfficientNet-Lite0* baseline model using the *TFLite Model Maker*. The *TFLite Model Maker* library simplifies the process of adapting and converting a *TensorFlow* neural-network model to particular input data when deploying this model for on-device ML applications. After training the model on the *TFLite Model Maker*, it had to be quantized post-training to shrink it down. This created a flatbuffer for the model in TensorFlow Lite. Post-training quantization is a conversion technique that can reduce model size while also improving CPU and hardware accelerator latency, with little degradation in model accuracy.

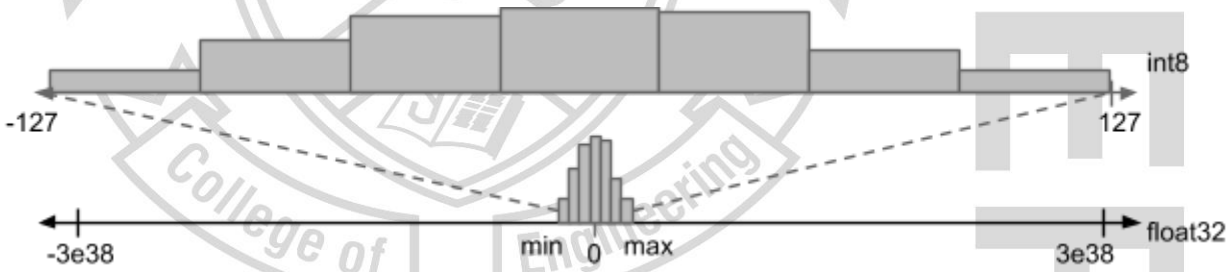


Figure 3.6: Quantization Techniques. (Courtesy: Google TensorFlow)

Table 3.2: Comparison of Model Hyperparameters.

Hyperparameter	EfficientNet-Lite0	EfficientNet-Lite2
Learning Rate	0.0090 to 2.2931e-05	0.0140 to 4.6005e-05
Batch Size	8	16
Epochs	20	20
Size of Model	4,341 KB	7,212 KB
Training Time (No Hardware Acceleration)	3,977 secs	11,123 secs

The accuracy of Object Detection Models is measured using two main metrics:

i. Average Precision
ii. Average Recall

Precision measures how accurate is your predictions. i.e. the percentage of your predictions are correct. Recall measures how good you find all the positives.

Average Precision is defined as the area under the precision-recall curve (PR curve). IoU is a good way of measuring the amount of overlap between two bounding boxes or segmentation masks. If the prediction is perfect, IoU = 1, and if it completely misses, IoU = 0. A degree of overlap will produce a IoU value between those two. This IoU can be kept fixed, for example, at 50% or 75%, which are called AP50 and AP75, respectively. When this is the case, it is simply the AP value with the IoU threshold at that value. The idea is similar for Average Recall as well.

Ideally, we must choose a model with maximum mAP and mAR with minimum latency. This is discussed at length in the Results section of the report.

Based on the detections returned by the inference model, two actions are triggered:

i. Rotation of the stepper motor, turning in the direction of the detections made in the image.
ii. Update the detections record in the *Google Firebase Realtime Database*.

3.3.3 Updating Detections Count to Google Firebase Realtime Database

Google Firebase has provided extensive APIs through which one can connect to the database through a multitude of languages and setups, including Python. The Realtime Database is excellent for backend server services which require data updating and recalling in real time applications such as ours.

3.3.4 Human Detection Pseudocode

The algorithm which was followed to run inference on the video stream is as follows:

```
continuously run:
            start capturing video input from the camera
            set visualization parameters
            initialize the object detection model
            continuously capture images from the camera:
                        run inference
                        if a human is detected in the image:
                                    update firebase count
                        draw key points and edges on input image
                        calculate and show the FPS
                                    .
```

3.3 Web App

The web application we built has the basic functionality required to run the project, as described below.

3.3.1 Front-End Using Vanilla HTML, CSS, and JS

In our application, we used HTML5 (Hypertext Markup Language) to make the structure of our site and to create a link between web pages. To style our website we used CSS (Cascading Style Sheets) which focuses on the presentation of the document and deals with text-font/style, background-colour, spacing and similar styling objects.

With the help of HTML and CSS we were able to make a static page, but to make the page Dynamic, we used JavaScript. JS is used to execute complex actions and enables user interaction with the web page. In our case, JS was used to change the state of the page (by changing the HTML content and CSS styling) and inform the user whether they are safe or not.

3.3.2 Back-End Using Node.JS

For our application we used Node.JS to set up our Local Host server, manage the Back-End Data and work with Real-Time Firebase data. Node.JS is a free JavaScript Open-Source cross-platform for server-side programming that allows users to build network applications quickly. The definition of Node.js as mentioned in its official documentation – "Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices."

Figure 3.7: Snapshot of NPM dependencies used.

Along with Node.JS, we used the Express.JS framework – which is a lightweight open-source web application that helps structure web applications on the server-side into a more organized MVC architecture.

Express JS is one of the most used Backend Framework and the reason behind it is its pre-defined libraries which reduces the complex programming to build efficient APIs. It has made programming in Node.JS effortless.

Figure 3.8: A snapshot of Console Window showing Detections.

As all of this was done Locally on LocalHost:3000, we wanted to shift it to a server so that anyone can access it. So, we used the services provided by Heroku and Git to shift our server from Local Host to Heroku's Server.

4. Results, Analysis and Conclusion

The implementation of this project was carried out by deploying the architecture we constructed to a Raspberry Pi 4 Model B and connecting it to cloud over Wi-Fi. The observations and results, along with an instance of failure are listed thoroughly below.

4.1. Human Detection and Tracking Analysis

As discussed, we used the *EfficientNet-Lite* family for classification and tracking of humans using a thermal imaging camera. The *EfficientNet-Lite0 model* gave a *mean Average Precision (mAP)* of 67.7% on the validation dataset. Thanks to the TensorFlow Model Optimization Toolkit, we easily quantized the model via integer-only post-training quantization. This reduced the model size by 4x and improved inference speed by 2x. The mAP post-quantization was 69.09%, surprisingly higher than our pre-quantization mAP.

Initially, the model was trained only on the data of humans, not background instances. XML files are generally generated only when a bounding box is present, hence no files were generated for the background instances by *LabelImg*. This went unnoticed when we trained our model initially, which was an error on our fault. The result of this error was noteworthy since the model learnt to recognize the background images as humans as well. We fixed this by writing a Python script which generated empty XML files for all background images, and then trained the model on the complete dataset. The model stopped detecting false positives in the background thereafter.

When the model ran on a given video stream, it ran at approximately 6 FPS. This was acceptable given our memory constraints and latency expectations. In an attempt to increase the accuracy, we also trained the model using *EfficientNet-Lite2.* This took considerably longer to train, but gave a higher mAP of 75.3%, which reduced to 72.6% post-quantization. Upon running

inferences on our video stream, the frame rate dropped to approximately 2 FPS, which was impractical for our application.

The sizes of the models were considerably small. The *EfficientNet-Lite0* was a total of 4.2 MB, and the *EfficientNet-Lite2* was a total of 7.0 MB on the disk. This was an expected model size, considering the extensive architecture of the networks as observed on Netron. The complete data for the training of the models is given in Table 4.1.

Table 4.1: Comparison of Human Tracking Model Accuracies.

Metric	**Parameter**	**EfficientNet-Lite0**	**EfficientNet-Lite2**
Outputs	Size of Model	4,341 KB	7,212 KB
	Training Time (no hardware acceleration)	3,977 secs	11,123 secs
Pre Quantization Metrics	mean Average Precision (mAP)	0.6770454	0.75306594
	Average Precision with 50% IoU	0.9646272	0.9970297
	Average Precision with 75% IoU	0.8850385	0.91438556
	Average Precision with Medium Objects	0.08974359	0.2025
	Average Precision with Large Objects	0.70565706	0.7883744
	Average Recall with Medium Objects	0.5	0.3
	Average Recall with Large Objects	0.75555557	0.84444445
	Average Recall with Max 1 Object	0.5	0.5736842
	Average Recall with Max 10 Objects	0.74210525	0.81578946
Post Quantization Metrics	mean Average Precision (mAP)	0.69065887	0.7267763
	Average Precision with 50% IoU	0.9431037	0.9970297
	Average Precision with 75% IoU	0.8883388	0.93584985
	Average Precision with Medium Objects	0.044444446	0.22777778
	Average Precision with Large Objects	0.7259452	0.7535018
	Average Recall with Medium Objects	0.4	0.7
	Average Recall with Large Objects	0.76111114	0.7888889
	Average Recall with Max 1 Object	0.52105266	0.5473684
	Average Recall with Max 10 Objects	0.74210525	0.7842105

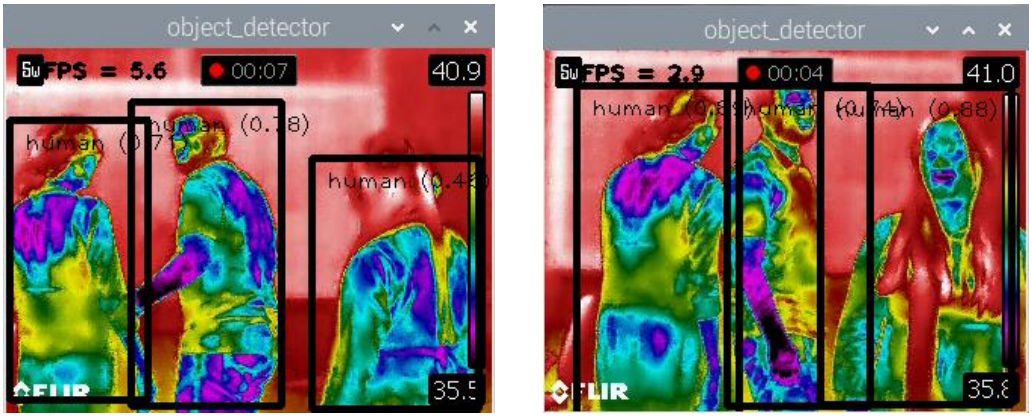


Figure 4.1: Image Outputs of EfficientNet-Lite0 (left) and EfficientNet-Lite2 (right).

After using both the models, we decided to proceed with the *EfficientNet-Lite0* model. Using OpenCV to drive the video stream into the inference system, the model was deployed to the Raspberry Pi. The output images for both the models are shown in Figure 4.1. As seen in the top-left corner of the images, the latency of the *EfficientNet-Lite0* model is much lesser than the *EfficientNet-Lite2* model.

Using the API provided for Python, we send the number of humans detected in the frame to the Firebase Realtime Database, which was immediately reflected in the database as shown in Figure 4.2.

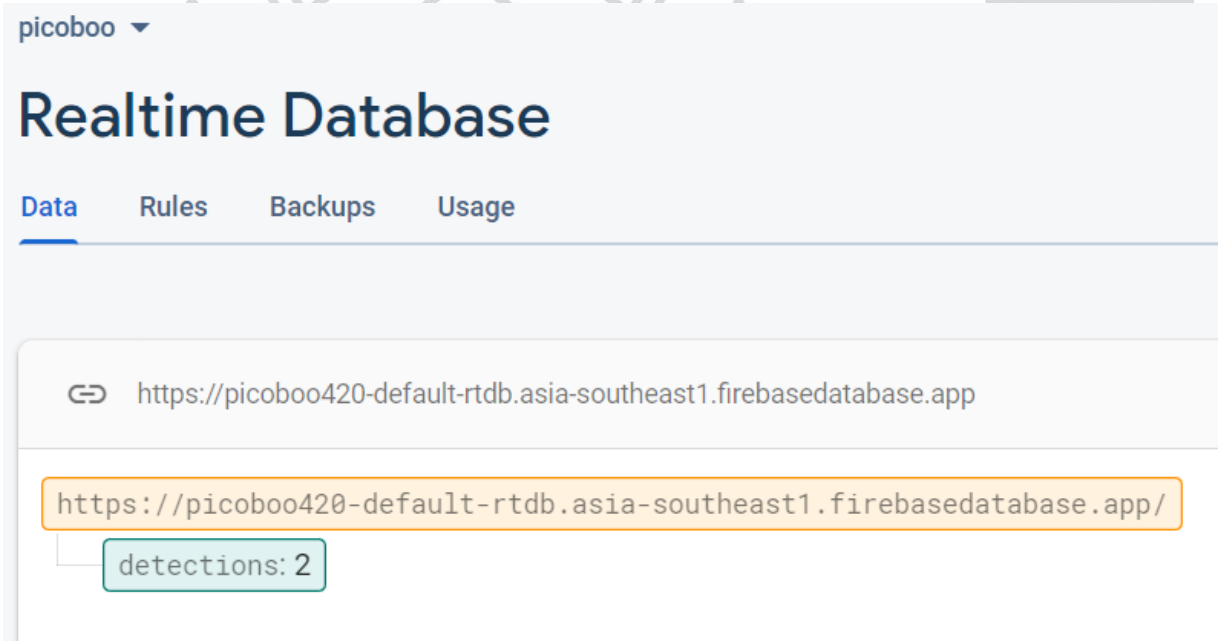


Figure 4.2: Updating detections count in Google Firebase Realtime Database

4.2. Access to Security Status Through Web App

If the number of detections on the Firebase Realtime Database is more than zero, then the page indicates that the security system is not safe and sends an alert of the intruder. This is triggered immediately due to the event handler which is set off as soon as there is a change in the Database reference point. Figures 4.3 and 4.4 show the changes in the page.

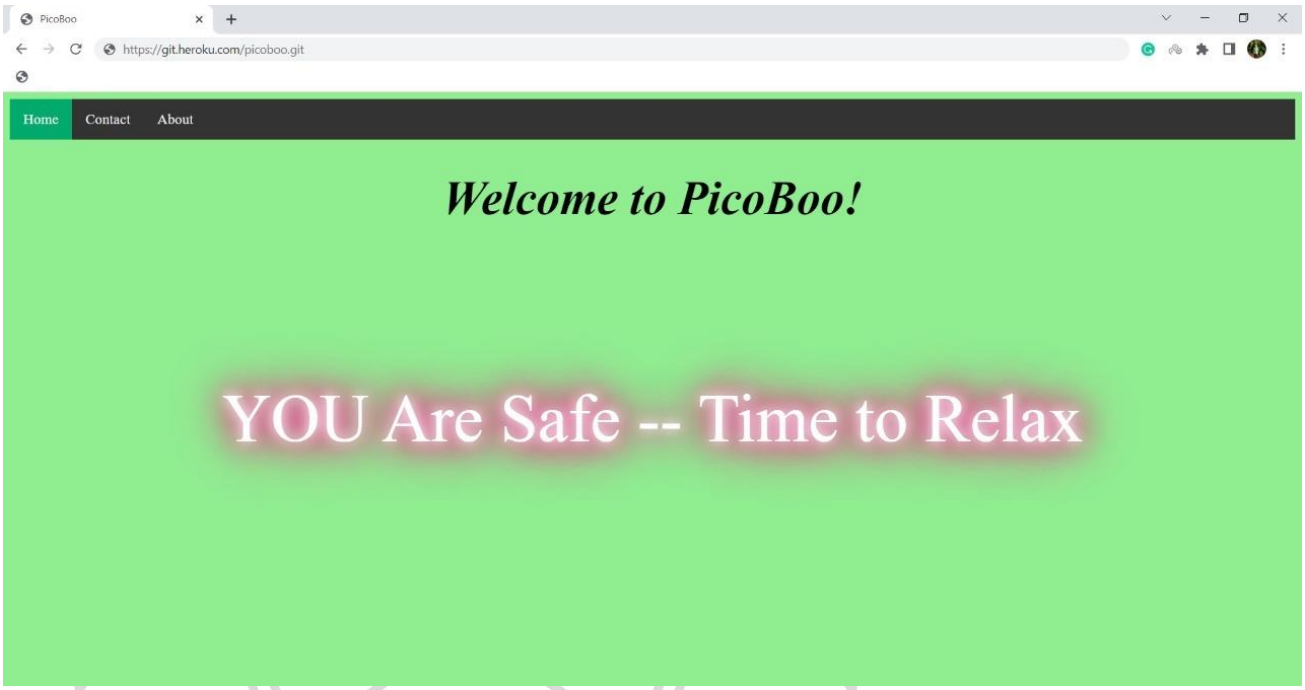


Figure 4.3: Webpage when there is an intruder present in the frame.

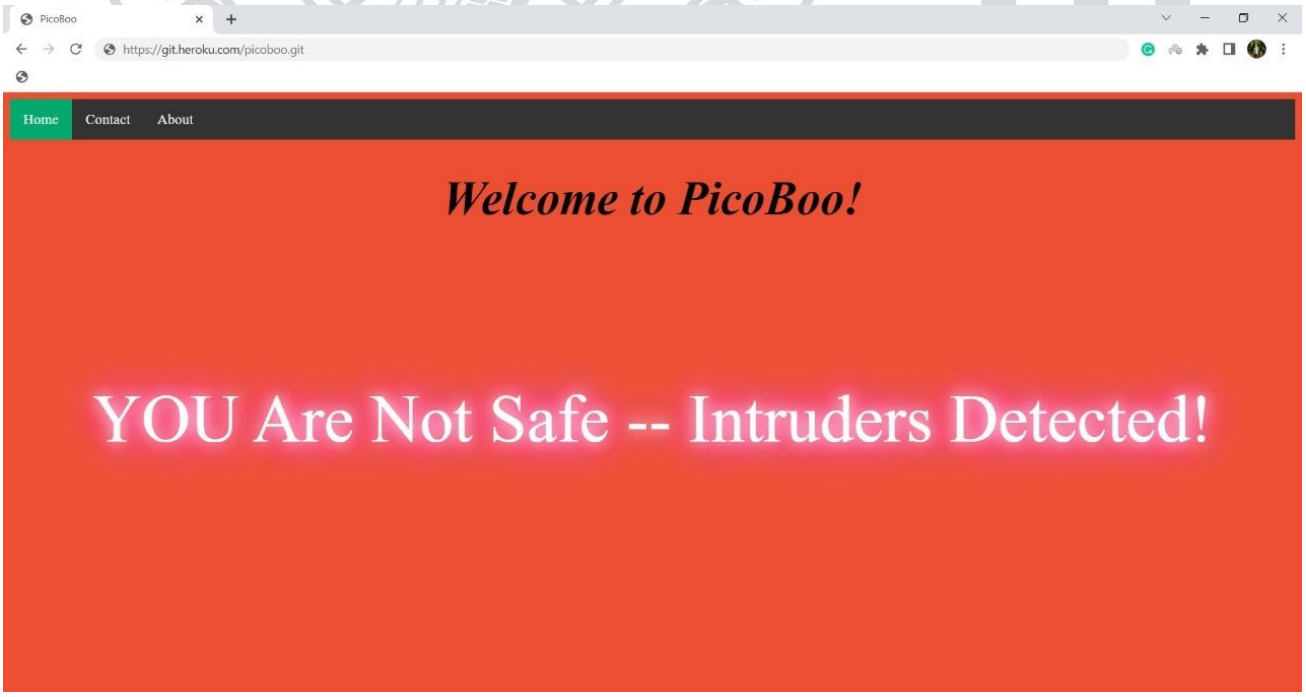


Figure 4.4: Webpage when there is no intruder present in the frame.

4.3 Bill of Materials (BoM) and Budget

All the components used for this project were either available in our college laboratories or could be sourced easily by us. They are listed down in Table 4.2.

Table 4.2: Bill of Materials.

Sr.	Component	Market Price (₹)	Availability	Effective Cost (₹)
1	FLIR T420 bx Thermal Imaging Camera	6,81,259	Sourced from the Center of Excellence of COEP	0
2	Raspberry Pi 4 Model B (2GB RAM Variant)	4,399	Sourced from the team	0
3	Raspberry Pi Peripherals	500	Sourced from the team	0
	TOTAL	**6,86,158**		**0**

4.4 Conclusion

PicoBoo – Security System with Motion Tracking using Thermal Imaging and Deep Learning is a working prototype of a security system that can be created to be deployed in areas where night-time security is a concern. Thermal imaging provides an accurate way of detecting intruders regardless of outside condititions and is hence the optimal way to build surveillance systems. We have gathered our own data and implemented deep learning frameworks to help build the crux of the system on which it works. We achieved a decent accuracy on training and validation as well, with a workable latency. The detections on our system were promptly reflected in our NoSQL cloud database which was captured and displayed by an app. A proper working model of such a system can be built which follows the motion of users and follows and tracks them through a room as well. Such innovations can be counted as further implementations of our project.

References

[1] Quoc V. Le, Mingxing Tan, "*EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks*", September 2020.

[2] Mohd Nadhir Ab Wahab, Anthony Tan Zhen Ren, Amril Nazir, Mohd Halim Mohd Noor, Muhammad Firdaus Akbar, Ahmad Sufril Azlan Mohamed "*EfficientNet-Lite and Hybrid CNN-KNN Implementation for Facial Expression Recognition on Raspberry Pi*", September 2021

[3] Cheng Chen, Sindhu Chandra, Yufan Han, Hyungjoon Seo, "*Deep Learning-Based Thermal Image Analysis for Pavement Defect Detection and Classification Considering Complex Pavement Conditions*", December 2021

[4] Olivier Janssensm, Rik Van de Walle, Mia Loccufier Sofie Van Hoecke, "*Deep Learning for Infrared Thermal Image Based Machine ,Health Monitoring*", July 2017

[5] EfficientNet-Lite Usage:
https://blog.tensorflow.org/2020/03/higher-accuracy-on-vision-models-with-efficientnet-lite.html

[6] Raspberry Pi 4 Model B Datasheet:
https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf

[7] FLIR T420 bx Datasheet:
https://www.globaltestsupply.com/pdfs/cache/www.globaltestsupply.com/t420bx/datasheet/t420bx-datasheet.pdf

[8] FLIR T420 bx User Manual:
https://www.globaltestsupply.com/pdfs/cache/www.globaltestsupply.com/t420bx/manual/t420bx-manual.pdf

[9] Heroku Node.js:
https://devcenter.heroku.com/articles/deploying-nodejs

[10] Linking Client Side and Server Side Data:
https://stackoverflow.com/questions/3922994/share-variables-between-files-in-node-js

[11] eJS Installation:
https://www.npmjs.com/package/ejs

[12] eJS to Node.js:
https://www.digitalocean.com/community/tutorials/how-to-use-ejs-to-template-your-node-application

[13] eJS Usage:
https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs