ECE 445

SENIOR DESIGN LABORATORY

FINAL REPORT

Visual Chatting and Real-Time Acting Robot

Team #37

HAOZHE CHI
(haozhe4@illinois.edu)
MINGHUA YANG
(minghua3@illinois.edu)
JIATONG LI
(jl180@illinois.edu)
ZONGHAI JING
(zonghai2@illinois.edu)

TA: Enxin Song

May 31, 2024

Abstract

This project focuses on the development and implementation of an AI-enhanced robotic service system designed to assist blind individuals in navigating large public spaces and safely interacting with water dispensers. The system comprises three main components: the navigation system, the Raspberry Pi auxiliary system, and the PCB water dispenser system. The navigation system utilizes advanced large language models (LLMs) and large visual language models (LVLMs) to process visual inputs from a head-mounted camera and verbal commands from the microphone, providing real-time guidance and safety instructions. The Raspberry Pi auxiliary system integrates object detection, hardware control, and automation, leveraging a fine-tuned SSD-MobileNet-V2 model in TFLite format to achieve high accuracy in detecting and handling water bottles and cups. This system is connected with a UR3e robot arm, MegaPi controller, and various sensors to enable seamless automated operations. The PCB water dispenser system coordinates with the robot arm to ensure precise and safe water bottle refilling. Extensive testing confirms the system's reliability, efficiency, and effectiveness in real-world scenarios, demonstrating the feasibility of deploying complex AI models and robotics for practical applications in assistive technology.

Keywords: AI-Enhanced Robotics, LLM, LVLM, Raspberry Pi, Object Detection, SSD-MobileNet-V2, TFLite, UR3e Robot Arm, MegaPi, PCB, Real-Time Automation

Contents

1 Introduction

1.1 Problem Statement

Blind individuals often face significant difficulties when navigating unfamiliar environments, such as finding water dispensers in large public spaces. Additionally, there is a risk of injury from interacting with devices that dispense hot water. The emergence of large language models (LLMs) and large visual language models (LVLMs) offers a promising avenue for developing innovative solutions to these challenges.

1.2 Solution Overview

We developed an AI-enhanced robotic service system to assist blind individuals in safely accessing and interacting with water dispensers in public spaces. This system addresses the challenges of locating amenities and avoiding injury from hot water dispensers.

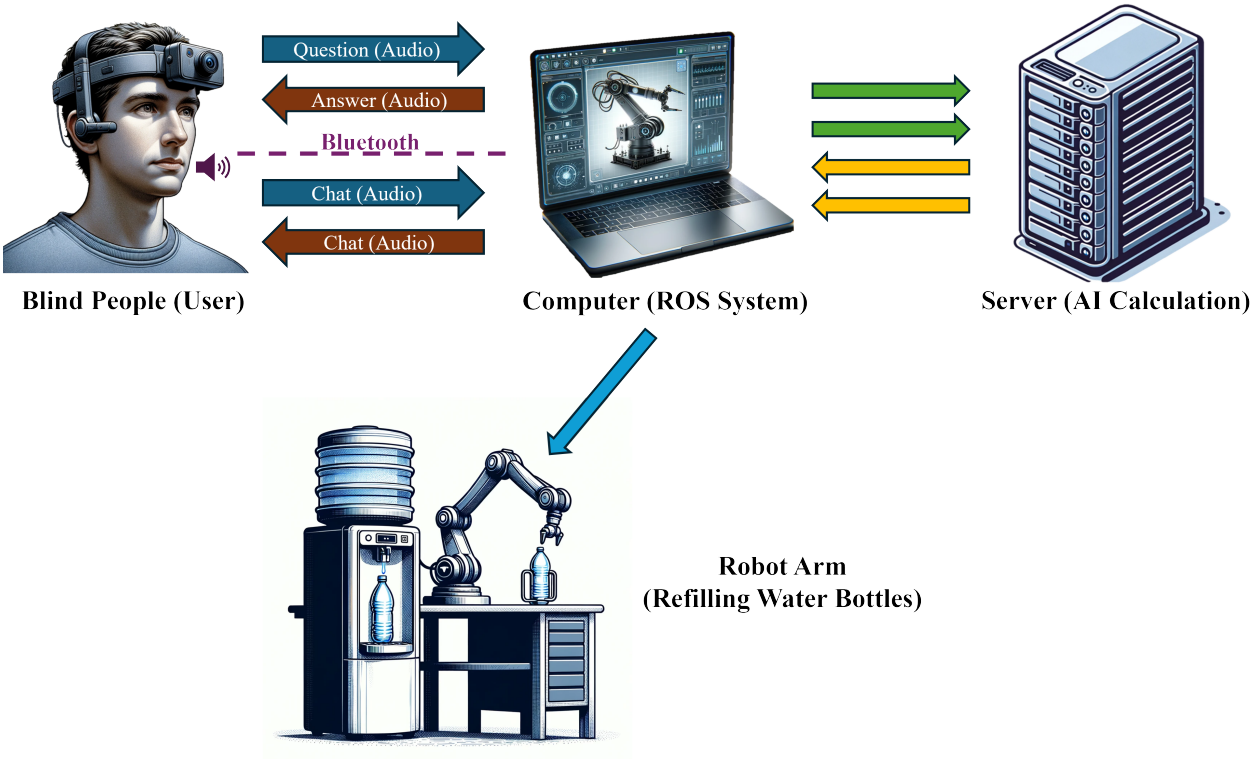


Figure 1: Visual Illustration of the AI-enhanced Robotic Service System

As shown in Figure 1, the system integrates advanced technologies, including large language models (LLMs) and large visual language models (LVLMs), to process visual inputs from a head-mounted camera and verbal commands via speech-to-text AI. It provides real-time guidance and safety instructions. To enhance user interaction with the automated bottle refilling facilities, we developed a Raspberry Pi Auxiliary system that offers auditory guidance through visual monitoring. The system's key components are:

- **Real-Time Visual and Verbal Input Processing and Communication:** A head-mounted camera and speech-to-text AI capture and analyze the user's surroundings and commands, with audio feedback and instructions delivered via a Bluetooth headset for clear communication.
- **Dynamic Guidance and Interaction:** The BLIP-2 model provides navigation assistance, warns of dangers, and instructs on interacting with a water dispenser.
- **Autonomous Assistance:** A Universal Robot Arm UR3e, controlled by the Robot Operating System and the Raspberry Pi Auxiliary System, autonomously refills the user's water bottle.

Operational Process When a blind individual approaches a water dispenser, the system follows these steps:

1. The Vision Language AI model guides the user to the water dispenser.
2. The Raspberry Pi Auxiliary System provides audio instructions to help the user place their bottle in the correct location.
3. The robot arm, following instructions from the Raspberry Pi system, securely grasps the bottle, fills it with water, and returns it to the user.

1.3 Block Diagram

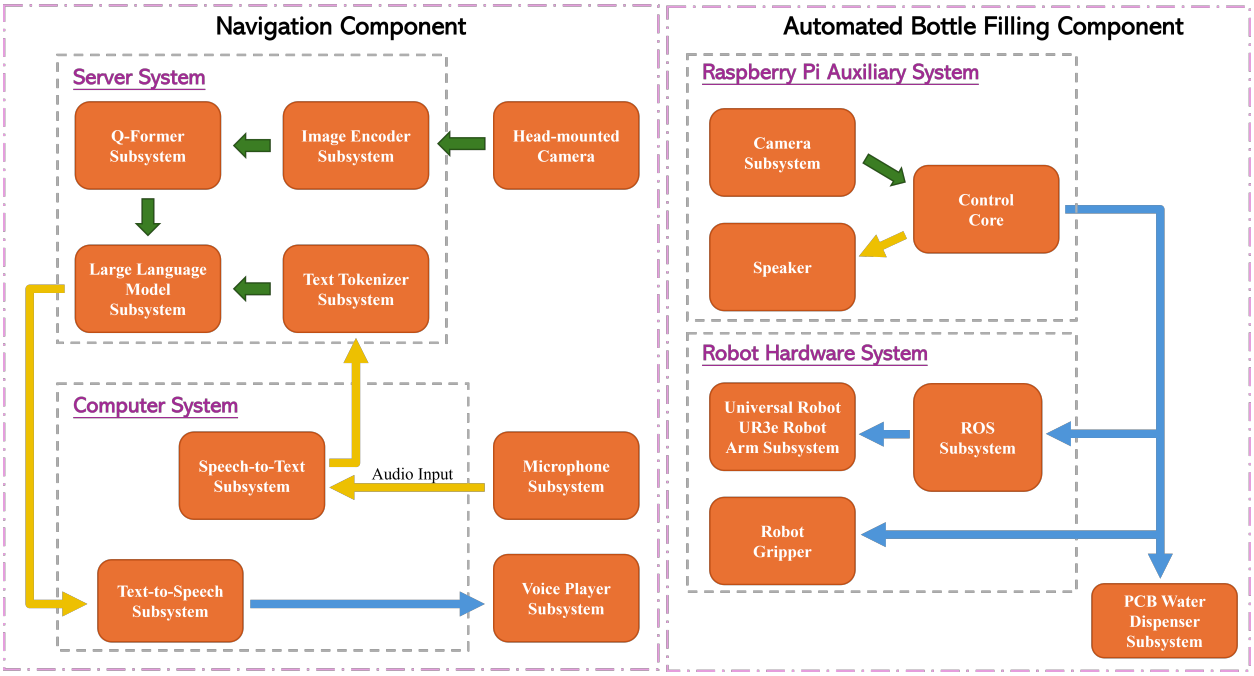


Figure 2: The navigation component processes visual and audio inputs, while the Raspberry Pi coordinates the automated filling process (Green arrow: visual flow; Yellow arrow: text flow; Blue arrow: instruction flow).

2 Design

Our AI-enhanced robotic service system has three main components, the Navigation System, the Raspberry Pi Auxiliary System, and the PCB Water Dispenser System.

2.1 Navigation System

The Navigation System consists of the following tasks:

1. Deploy speech-to-text and text-to-speech modules to process vocal inputs and provide guidance.
2. Implement a depth-map generation module to create depth maps from visual images for navigation algorithms to detect potential dangers.
3. Establish a stable connection between the personal computer and AI server for efficient real-time data transmission.
4. Deploy Large Visual Language Models (LVLM) on the AI server to provide real-time guidance and instructions.
5. Apply model acceleration techniques to reduce processing delays.
6. Design a real-time visual screenshot program using the Zoom App to transfer visual input to the AI server.

2.1.1 Speech-to-Text and Text-to-Speech Modules

We deploy speech-to-text and text-to-speech modules on a Mac OS PC using a VMware virtual machine running Ubuntu. For speech-to-text, we utilize the pre-trained ***silero***[1] model. The process involves recording voice using Python's pyaudio library, saving it as a wave file, and converting it to text with the silero model. For text-to-speech, we use the ***pyttsx3***[2] project. After receiving a JSON file from the AI server, the program reads the answer message, converts it to audio using pyttsx3, and plays it through Bluetooth devices.

2.1.2 Navigation Algorithm

The navigation algorithm prioritizes tasks as shown in Figure 3. The highest priority is to immediately alert the user to potential dangers detected through depth map analysis. If no danger is present, the system responds to the user's vocal inputs. If there are no vocal inputs, the system repeats the navigation route.

We generate depth maps using the NYU FCRN network, based on ResNet50 with additional depth blocks. Given the fixed route design, repeated guidance is also predefined.

[1]https://github.com/snakers4/silero-models
[2]https://pyttsx3.readthedocs.io/en/latest/

The LVLM alerts the user to objects in the depth map with a depth smaller than a set threshold, K.

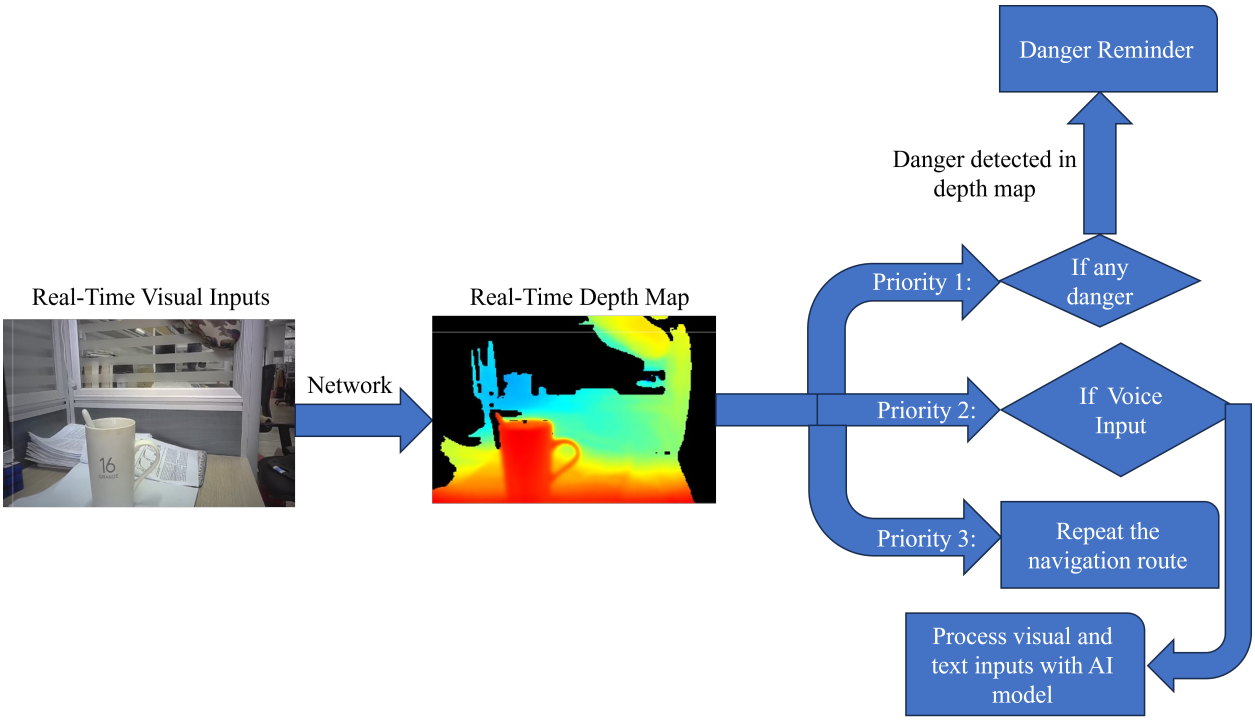


Figure 3: Navigation Algorithm Priorities

2.1.3 BLIP-2-Based Visual Language Model Deployment

We use the paramiko library in Python to establish a stable SSH connection and SFTP protocol for data transmission between our PC and the AI server. This ensures fast and stable communication, with new videos generated and sent to the server every second. Vocal questions from the user generate a JSON file sent to the server, where the LVLM processes it along with the latest video, and sends back a response.

We deploy the modified VideoChatGPT model, based on BLIP-2 [1], on the AI server. This model handles vision-language interaction by processing real-time visual and audio data from the user. The architecture of the BLIP-2-based model is shown in Appendix B.1 Figure 14. The VideoChatGPT model lacks a Q-former module for cross-modality processing, as shown in Appendix B.1 Figure 15. We adapted the Q-former structure from VideoLlama, replacing the spatial pooling stage to generate temporal features through attention mechanisms. The architecture of the modified VideoChatGPT is shown in Appendix B.1 Figure 16.

2.1.4 Apply Acceleration Module

To speed up LVLM inference, we integrated the flash-attention module into the Llama model, the LLM decoder. Flash-attention uses a tiling strategy to calculate attention

blocks in parallel, leveraging the decomposition of the softmax function. The detailed algorithm is shown in Appendix B.2 Figure 17.

As per [2], flash-attention significantly accelerates GPT-2 model training and optimizes GPU and CPU bandwidth usage. The layout of flash-attention is illustrated in Appendix B.2 Figure 18.

2.1.5 Design of Real-Time Screenshot Program

Blind users capture real-time visual information using an iPhone. To streamline the process, we connect the iPhone to a personal computer via Zoom and use a script to take screenshots every second. These images are converted into short videos and uploaded to the AI server.

We utilize the pyautogui library to automate screenshots, crop the informative Zoom screen area, and save the images. The moviepy library's ImageSequenceClip class generates short videos, which are then uploaded to the AI server via the SFTP protocol every second. Detailed code is provided in Appendix C.1.

2.2 Raspberry Pi Auxiliary System

The integration of the Raspberry Pi Auxiliary System into our project was driven by the need to address significant challenges observed in the initial design, which relied heavily on a Vision Language AI model for controlling all operations. Here, we outline the reasons behind adopting the Raspberry Pi system to enhance functionality and efficiency.

Initial Design and Challenges: Our initial setup tasked the Vision Language AI model with guiding interactions entirely, from bottle placement to operation of the robot arm and water dispenser. This centralized approach led to considerable processing delays due to the model's inherent latency and added complexity, impacting system responsiveness and precision.

Rationale for the Raspberry Pi Auxiliary System: The adoption of the Raspberry Pi Auxiliary System was aimed at mitigating these issues through several key improvements:

1. **Reduced Latency:** By decentralizing control, the Raspberry Pi significantly cuts down on overall system latency, facilitating near real-time interactions between the user and the robotic arm.
2. **Simplified Processes:** The Raspberry Pi manages direct control over the robot arm and water dispenser, simplifying operations and allowing the Vision Language AI model to focus on providing high-level navigational aid rather than managing minute details.

3. **Enhanced Precision and Interaction:** With dedicated object detection and text-to-speech models, the Raspberry Pi system provides precise localization of the water bottle and clear auditory instructions, improving the guidance provided to users.
4. **Improved System Efficiency and Usability:** These modifications ensure smoother, more convenient operations and enhance the user experience, particularly in facilitating effective interaction for visually impaired users.

Introducing the Raspberry Pi Auxiliary System was a strategic choice to optimize our project's structure, enhancing operational efficiency, responsiveness, and user interaction. This approach not only improved system performance but also better aligned with our objectives of supporting visually impaired individuals in public spaces.

2.2.1 System Architecture Overview

The Raspberry Pi Auxiliary System is strategically designed to enhance interaction convenience for visually impaired users. As shown in Appendix B.3 Figure 19, it integrates three cameras and a speaker to facilitate user interaction with the robot arm:

- **Two Web-Cameras:** Positioned to monitor the placement of the water bottle on the desk (both X and Y directions), ensuring it is within the operational range of the robot arm.
- **Pi-Camera:** Used to confirm the bottle's correct positioning at the water dispenser, ensuring accurate filling.
- **Speaker:** Provides real-time vocal instructions to guide the user in adjusting the bottle's placement.

2.2.2 Block Diagram of Overall Work Flow

The overall workflow of the Raspberry Pi Auxiliary System is illustrated in Appendix B.3 Figure 20. The process consists of six control steps:

1. The Raspberry Pi core control center captures visual input from camera-x and camera-y. It uses the object detection model to locate the position of the water bottle placed on the desk.
2. Based on the detected position of the water bottle, the control center outputs audio instructions to help the user move the bottle (right/left/forward/back) to the designated location so that the robot gripper can grasp it. (Steps 1 and 2 are repeated until the water bottle is correctly placed.)
3. The robot gripper grabs the water bottle.
4. The control center, through the ROS system, directs the robot arm to move the water bottle to the water dispenser.
5. The control center takes visual input from camera3, mounted on the water dispenser, to verify the presence of a bottle at the dispenser, ensuring accurate filling.

6. If the check in the previous step is successful, the control center sends a start signal to the PCB board to begin filling the bottle.

After filling, the robot arm moves the water bottle back to the user, and the robot gripper releases the bottle. Finally, the control center outputs an audio message indicating that the user can now pick up the bottle.

2.2.3 Water Bottle Object Detection Task

Model Selection For the task of detecting water bottles, we selected the **SSD-MobileNet-V2** model [3] because it offers an optimal balance between speed and accuracy, which makes it well suited for applications in near real time on our edge device, the Raspberry Pi. SSD-MobileNet-V2 is designed specifically for mobile devices and edge devices, combining the Single Shot Multi-Box Detector (SSD) framework [4] with MobileNetV2 [5]. MobileNetV2 builds on its predecessor, MobileNetV1 [6], by introducing an inverted residual structure with linear bottlenecks. This enhancement significantly improves computational efficiency by maintaining crucial depth information for performance while reducing model size and computational cost. This efficiency is crucial in reducing latency and ensuring that the system can operate in real time, which is essential for user interaction.

Model Training and Finetuning For the water bottle detection task, we selected the pretrained SSD-MobileNet-V2 model from the **TensorFlow 2 Object Detection Model Zoo**[3] [7].

This pretrained model is initially trained on the COCO dataset [8], which includes a wide variety of objects, allowing the model to learn robust feature representations. Utilizing a pretrained model provides several benefits for our project. Firstly, it reduces the computational resources and time required for training from scratch. The SSD-MobileNet-V2 model is particularly advantageous due to its efficiency and accuracy, making it suitable for deployment on resource-constrained devices such as the Raspberry Pi.

Motivation for Finetuning and Transfer Learning Method The primary task of our project is to detect water bottles. Although the pretrained SSD-MobileNet-V2 model is effective, fine-tuning it specifically for water bottle detection can significantly enhance its performance for this particular task. Fine-tuning involves adjusting the pretrained model on a smaller, task-specific dataset, which in our case, is a custom dataset of water bottles.

We used the transfer learning method to fine-tune the pretrained model on our custom water bottle dataset. Transfer learning leverages the knowledge gained from the large-scale COCO dataset and applies it to our specific task, enabling the model to adapt to the nuances of water bottle detection. This approach is more efficient than training a model

[3]TensorFlow2-Object-Detection-Model-Zoo

from scratch, as it requires less data and computational power while providing better performance due to the pre-learned features from the larger dataset.

Custom Water Bottle Dataset As shown in Figure 4, my custom water bottle dataset consists of 260 images of water bottles and cups, collected from the Web and daily life environments. These images represent common bottles and cups that users are likely to carry, ensuring the relevance and applicability of the model.

Using the labelling tool **Labelimg**[4] [9] from GitHub, we manually labeled all the images by drawing bounding boxes around the water bottles and cups and assigning the appropriate labels. Once labeled, the dataset was randomly split into training, validation, and test sets. We then created a Labelmap and TFRecords, which are required by TensorFlow for the training process.

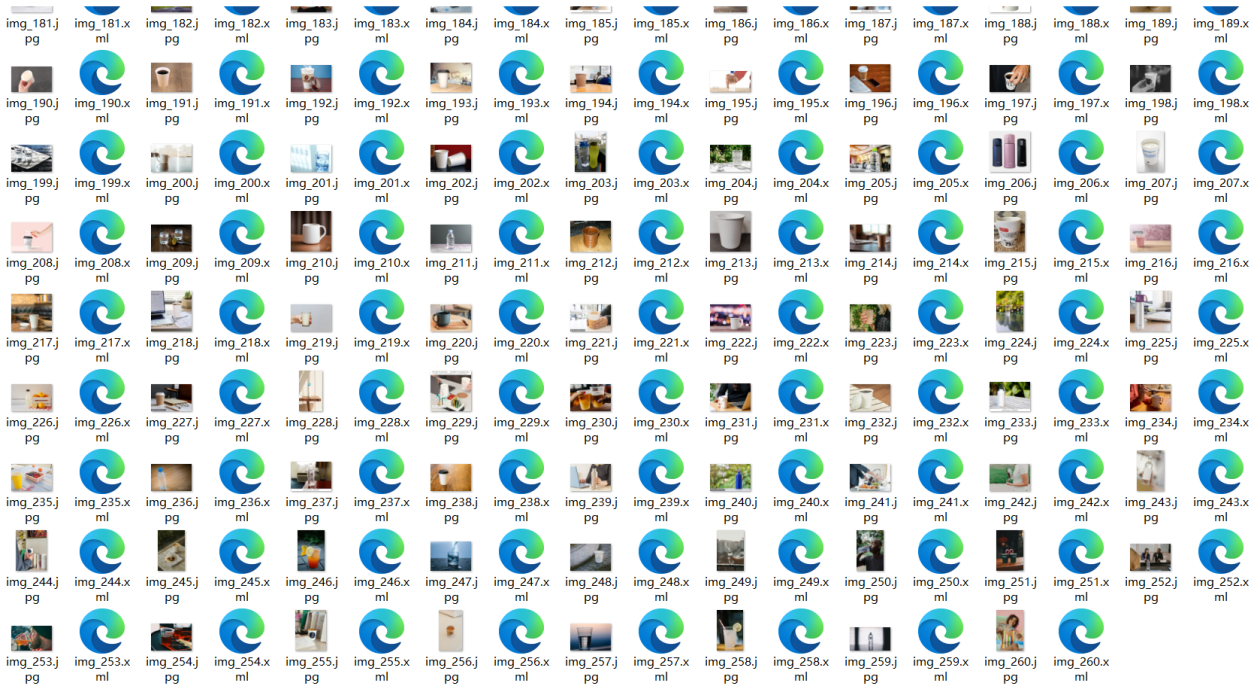


Figure 4: Screenshot of My Custom Water Bottle Dataset

Model Deployment After the fine-tuned model is trained, the next crucial step is to export the model graph, which contains information about the architecture and weights, to a TensorFlow Lite (TFLite) format. This conversion is essential for several reasons, particularly when deploying models on edge devices such as the Raspberry Pi. The basic outline of the inference script is shown in Appendix C.3.

The primary motivation for converting the model to the TFLite format is to optimize it for performance on resources-constrained devices. The TFLite format offers several benefits over the standard TensorFlow (TF) model format:

[4]https://github.com/HumanSignal/labelImg

1. **Reduced Model Size**: TFLite models are significantly smaller in size compared to their TF counterparts. This reduction is achieved through various optimization techniques such as quantization, which helps in reducing the model footprint and making it suitable for devices with limited storage.
2. **Faster Inference**: TFLite models are optimized for speed. They are designed to perform efficient inference on devices with limited computational power. This optimization ensures that the model can run in real time, providing quick responses, which is critical for applications requiring immediate feedback, such as water bottle detection.
3. **Lower Latency**: By converting to TFLite, the inference latency is minimized. This is particularly important for interactive applications where user experience depends on how quickly the system can process input and provide output.
4. **Energy Efficiency**: TFLite models are designed to be energy-efficient, making them ideal for battery-powered edge devices. Reduced energy consumption ensures that these devices can operate longer without frequent recharges.

Bottle Position Detection Algorithm In our water bottle detection task, determining whether the detected water bottle is correctly positioned within the frame is crucial for providing accurate guidance to the user. The Algorithm shown in Appendix C.4 was developed to judge the position of the detected water bottle relative to the center of the screen. There are two key concepts used in the algorithm:

Central Position Threshold The *central_position_threshold* is a predefined value that determines the allowable range around the center of the screen within which the water bottle is considered correctly positioned. If the center of the detected water bottle's bounding box lies within this threshold, the bottle is deemed to be in the center. This threshold helps in reducing sensitivity to minor movements and ensures that the detection is robust.

Position Stability Threshold The *position_stability_threshold* is another predefined value that ensures the detected position of the water bottle is stable over a series of frames. This stability check prevents the system from reacting to transient or noisy detections, thereby providing a more reliable indication of the bottle's position.

2.2.4 Robot Gripper Control

The robot gripper used to grab the water bottle is the Makeblock Robot Gripper, mounted on the UR3e Robot Arm as shown in Figure 5. This gripper is operated through the MegaPi [10] controller, which is programmed and interfaced with a Raspberry Pi. The setup and control of the robot gripper involve several steps to ensure precise manipulative operations. The detailed code is shown in Appendix C.5.

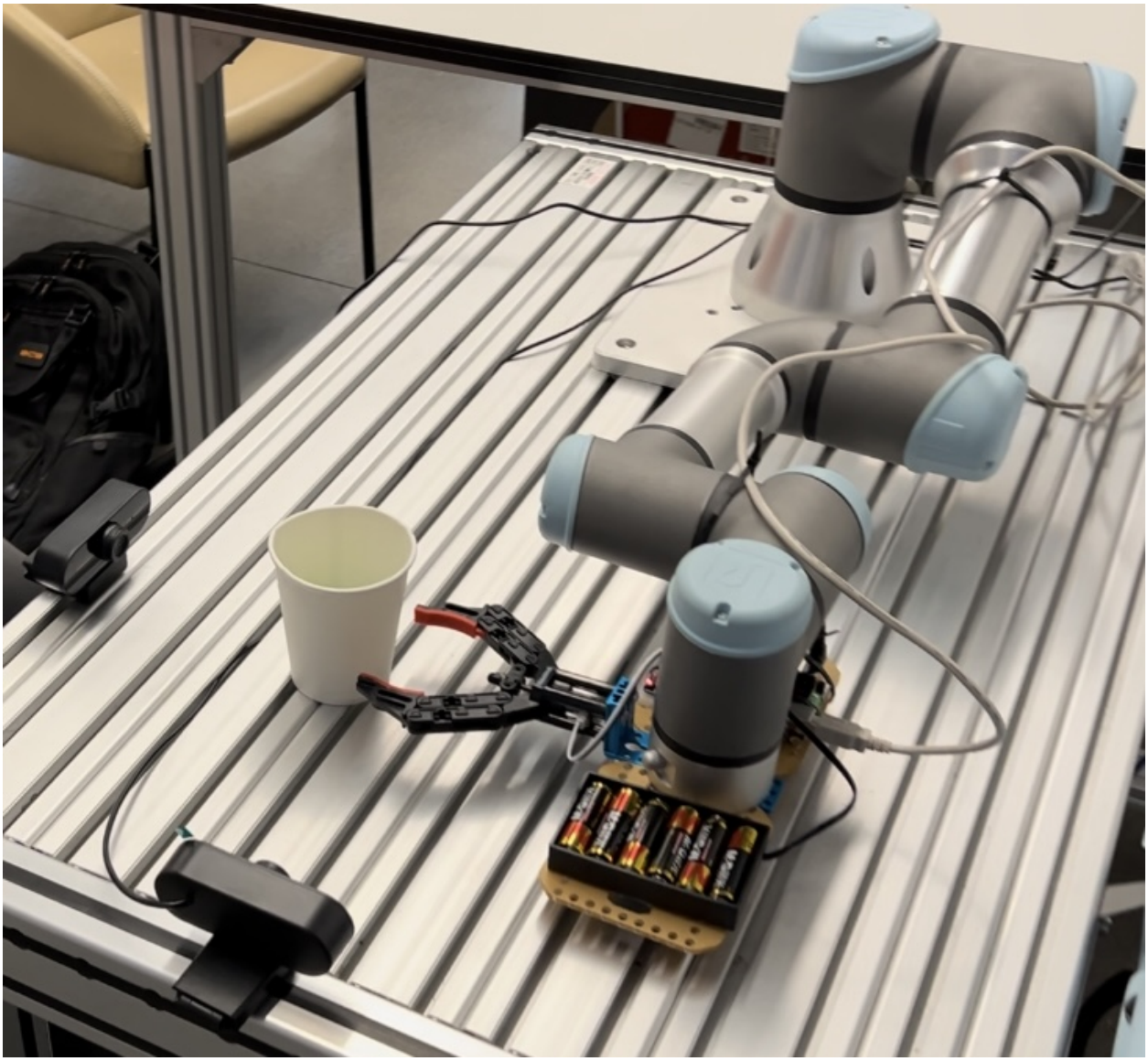


Figure 5: The robot gripper is mounted on the robot arm, with two cameras controlled by the Raspberry Pi to detect the position of the water bottle.

2.2.5 Communication Setup with ROS System

To control the UR3e robot arm [11], a robust communication setup involving a Virtual Machine (VM) and the Robot Operating System (ROS) is established. This setup allows for the control of the robot arm from a Raspberry Pi via SSH, providing a streamlined method of sending instructions remotely.

Setup and Control Procedures

1. **Virtual Machine Configuration**: An Ubuntu VM (version 20.04) is set up on a PC using virtualization software. This VM serves as the host for the ROS environment

and interfaces with the UR3e robot arm.

2. **ROS Installation**: ROS Noetic [12], which is compatible with Ubuntu 20.04, is installed on the Ubuntu VM. This version of ROS provides the necessary packages and libraries to interface with the robot hardware and execute control commands.
3. **Network Configuration**: To ensure seamless communication between the VM and the robot arm, the VM's network is configured to use a bridged adapter. This setup utilizes the Realtek PCIe GbE Family Controller, allowing the VM to connect directly to the same network as the host machine and the robot arm.
4. **Physical Connection**: The UR3e Robot Arm is connected to the PC using a network cable. This direct connection minimizes latency and ensures reliable communication.
5. **Automated Control Script**: On the Raspberry Pi, a shell script is developed to automate the control of the robot arm. This script uses SSH to securely connect to the Ubuntu VM and execute ROS commands. The SSH setup allows remote and secure command transmission from the Raspberry Pi to the VM.
6. **Enable SSH Communication**: Enable the hotspot on the PC, then set up a new bridge network with Microsoft Wi-Fi Direct Virtual Adapter #2 on VM. Connect both the Raspberry Pi and the VM to the PC hotspot to ensure that they are under the same subnet.

The main ROS code to control the robot arm is shown in Appendix C.2.

2.2.6 Model Selection for Text-to-Speech

For converting text outputs into audible instructions, we chose the **eSpeak**[5] module. eSpeak is known for its simplicity, lightweight design, and wide language support, making it suitable for real-time speech synthesis on constrained devices. Its compact size and efficient speech generation capabilities ensure minimal impact on the system's resources, aligning with the need for a fast response time in our application.

2.2.7 Communication Setup with PCB Water Dispenser System

The communication between the Raspberry Pi and the PCB water dispenser system is established through GPIO pins, as shown in Appendix B.3 Figure 21 [13]. This setup is crucial for indicating the operational status of the water dispenser system using LEDs and for managing the water dispensing process based on camera detections.

LED Indicators The PCB water dispenser board is equipped with two LEDs:

- **Green LED**: Indicates that the entire water dispenser system is operational. This LED turns on when the system is powered up and remains on during normal operation.

[5]https://espeak.sourceforge.net/

- **Red LED**: Indicates that hot water is being dispensed. This LED turns on only when the camera mounted on the water dispenser detects that a water bottle is placed at the dispenser. The LED turns off once the filling process is complete.

GPIO Pin Configuration The communication with the LEDs is managed through GPIO pins on the Raspberry Pi. Specifically, two pins are utilized:

- **Green LED Pin**: Connected to GPIO pin 27.
- **Red LED Pin**: Connected to GPIO pin 17.
- **Ground Pin**: Connected to GPIO Ground pin (No.6 in the layout diagram).

These pins control the LEDs on the PCB board, providing a straightforward and low-latency interface to manage the water dispensing actions based on the camera's detections. The detailed code is shown in Appendix C.6.

Bottle Detection and Red LED Control As shown in Figure 6, the system checks if a water bottle is placed at the dispenser using the *Detect_Bottle* function. If a bottle is detected, the red LED is turned on to indicate that hot water is being dispensed. The LED remains on during the filling process and turns off once the process is complete. If no bottle is detected, the system provides an appropriate audio message, and then the robot arm will move back to the original position.

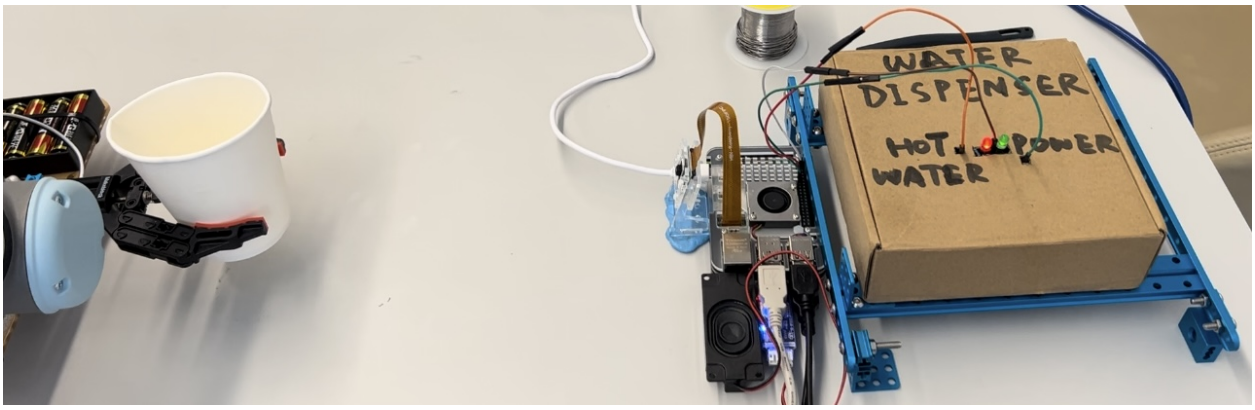


Figure 6: The camera3 detects that a water bottle is placed at the water dispenser, then the red LED is turned on to indicate that hot water is now being dispensed.

2.2.8 Shell Script for the Whole Raspberry Pi Auxiliary System

To integrate the above all components of the Raspberry Pi Auxiliary System and achieve automatic control, a comprehensive shell script is used. This script orchestrates the actions of the camera detection, robot gripper, UR3e robot arm, and LED indicators on PCB board to provide a seamless and automated process for water bottle handling. The following is an introduction to the shell script and its functionalities.

Overview of the Shell Script The shell script is designed to perform the following sequence of operations:

1. Reset the robot arm to its initial position.
2. Turn on the green power LED light to indicate that the system is operational.
3. Wait for the camera to detect the water bottle.
4. Grab the water bottle using the robot gripper.
5. Move the robot arm with the bottle to the water dispenser.
6. Check if the water bottle is correctly placed at the dispenser and fill it with water.
7. Move the robot arm back to the user.
8. Release the water bottle.

Using a shell script to control each component together provides several benefits, including streamlined automation, reduced manual intervention, and improved synchronization between components. And the detailed implementation is shown in Appendix C.7.

2.3 PCB Water Dispenser System

The design details of our PCB board for the simulated water dispenser are provided in Appendix B.4. The board features two LED lights and several gate-controlled logic components. Based on inputs from the Raspberry Pi Auxiliary System, the green light indicates the system is powered on, and the red light indicates hot water is being dispensed.

PCB Board Design Figure 7 and Figure 8 show the PCB design, which includes voltage and signal inputs from the Raspberry Pi, with NAND gates and an inverter controlling the LEDs. The key components are:

- **Voltage and Signal Inputs:** Provided by the Raspberry Pi's external GPIO, these inputs control the LEDs.
- **74LS00 NAND Gates:** The 74LS00 chip (U1A and U2A) uses NAND logic to control the red (D1) and green (D2) LEDs. The green LED indicates system power, and the red LED indicates hot water dispensing.
- **74LS04 Inverter:** The 74LS04 chip (U3A) inverts the signal from the NAND gate to control the green LED, ensuring it is on when the system is powered and off when hot water is dispensed.
- **Voltage Regulator:** The LM1117-5.0 provides a stable 5V supply to the circuit.
- **LED Indicators:** The red LED (D1) shows hot water dispensing, while the green LED (D2) shows the system is powered.

The circuit operates with the external GPIO input driving the NAND gates U1A and U2A, which control the LEDs. U2A's output drives the red LED, while U1A's output is inverted by U3A to drive the green LED. The voltage regulator maintains a stable 5V supply.

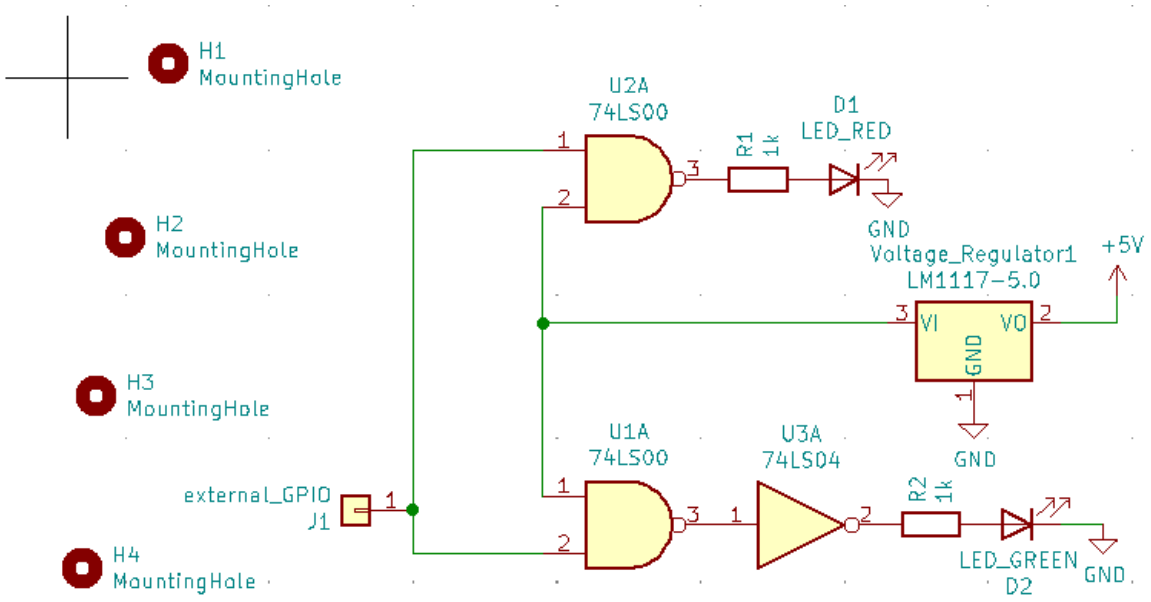


Figure 7: PCB Schematics

符号:封装分配

1	D1	-	LED_RED	: LED_THT:LED_Rectangular_W5.0mm_H5.0mm
2	D2	-	LED_GREEN	: LED_THT:LED_Rectangular_W5.0mm_H5.0mm
3	H1	-	MountingHole	: MountingHole:MountingHole_3.2mm_M3
4	H2	-	MountingHole	: MountingHole:MountingHole_3.2mm_M3
5	H3	-	MountingHole	: MountingHole:MountingHole_3.2mm_M3
6	H4	-	MountingHole	: MountingHole:MountingHole_3.2mm_M3
7	J1	-	external_GPIO	: Connector_PinHeader_2.54mm:PinHeader_1x01_P2.54mm_Horizontal
8	R1	-	1k	: Resistor_SMD:R_0805_2012Metric_Pad1.20x1.40mm_HandSolder
9	R2	-	1k	: Resistor_SMD:R_0805_2012Metric_Pad1.20x1.40mm_HandSolder
10	U1	-	74LS00	: TTL:SOIC127P600X175-14N
11	U2	-	74LS00	: TTL:SOIC127P600X175-14N
12	U3	-	74LS04	: TTL:DIP794W45P254L1969H508Q14
13	Voltage_Regulator1	-	LM1117-5.0	: Package_TO_SOT_SMD:SOT-223

Figure 8: PCB Footprints Details

3 Verification

3.1 Performance of the LVLM

To verify the effectiveness of the modified VideoChatgpt model, we conduct comprehensive experiments on Video QA datasets. We also quantatively test the inference delay of model to show the effectiveness of the added acceleration module flash-attention.

Model	Size	MSVD	MSRVTT	ActivityNet
FrozenBiLM [14]	1B	32.2	16.8	24.7
VideoChat [15]	7B	56.3	45.0	26.5
Video-ChatGPT [16]	7B	64.9	49.3	35.2
Video-LLaMA [17]	7B	51.6	29.6	12.4
Video-LLaVA [18]	7B	70.7	59.2	45.3
Ours	7B	65.7	51.6	34.1

Table 1: Short Video Question Answering Results (Accuracy) on MSVD, MSRVTT, and ActivityNet Dataset

Model	Size	Time
VideoChatgpt	7B	5.6s
VideoChatgpt + flash-attention	7B	4.1s
Ours	7B	5.9s
Ours + flash-attention	7B	3.8s

Table 2: Inference Time Comparison

We conduct experiments on short video QA datasets MSVD, MSRVTT and NextQA. MSVD is a short video dataset collected by Microsoft company that contains 1970 short video clips in total. The average length of each video clip is 10 seconds. These clips are from Youtube and cover various themes like animals, music, sports and travelling. Each short video clip has a text describing the contents in the video. The Video QA file is generated from these video descriptions. MSRVTT dataset is also collected by Microsoft, containing 10,000 short video clips. The average length here is 20 seconds, which means it's longer compared to MSVD. MSRVTT covers more topics and show more variety of contents. Furthermore, MSRVTT include more scene switches, which make the video clips harder to understand. ActivityNet dataset is a benchmark that contains many types

of human activities. ActivityNet dataset provides various types of videos for advance video understanding, including temporal action reasoning and common scene understanding. It comprises of different motion information for further action recognition and analysis.

The modified VideoChatgpt model shows competitive results in QA tasks, as shown in Table 2. First, we can see that flash-attention reduces the total time delay by 2.1 seconds. Furthermore, we can see that flash-attention module only reduces original VideoChatgpt inference time by 1.5 seconds. This is because I add Q-former module to original model, and flash-attention module can significantly reduce the processing time of Q-former as well. Second, it's shown in Table 1 that the modified VideoChatgpt achives better accuracy than original model in MSVD dataset QA task, and shows competitive result with respect to Video-LLaVA. Moreover, the modified VideoChatgpt also shows better accuracy in MSRVTT dataset QA task. Lastly, although it does not show superior result in ActivityNet dataset QA task, it still has high accuracy compared to other baselines. The reason of lower accuracy might due to the flexibility of questions in ActivityNet, as more complex reasoning questions are included. In general, the performance of modified VideoChatgpt is impressive and inspiring. This also shows the effectiveness of Q-former structure in BLIP-2 architecture.

3.2 Performance of the SSD-MobileNet-V2 model

The fine-tuning training results of the SSD-MobileNet-V2 model are illustrated through several key figures that show the behavior of different loss functions and the rate of learning during the training process.

3.2.1 Loss Figures

The training process is evaluated using four types of loss functions as shown in Figure 9, each representing a different aspect of the model's performance.

1. **Classification Loss**: The first figure shows the classification loss, which measures the model's accuracy in classifying detected objects. This loss represents the error in predicting the correct class labels for the detected objects. A decrease in classification loss over time indicates that the model is improving in its ability to correctly classify objects.

2. **Localization Loss**: The second figure shows the localization loss, which measures the accuracy of the bounding box predictions. This loss accounts for the differences between the predicted bounding boxes and the ground truth boxes. A decrease in localization loss signifies that the model is getting better at predicting the precise locations of the objects.

3. **Regularization Loss**: The third figure shows the regularization loss, which helps to prevent overfitting by adding a penalty for large weights. This loss component ensures that the model generalizes well to unseen data. A smooth decrease in regularization loss indicates effective regularization, contributing to better generalization.

4. **Total Loss**: The fourth figure shows the total loss, which is the sum of the classification, localization, and regularization losses. This combined loss gives an overall measure of the model's performance during training. A consistent decrease in total loss suggests that the model is learning effectively, balancing classification accuracy, localization precision, and regularization.

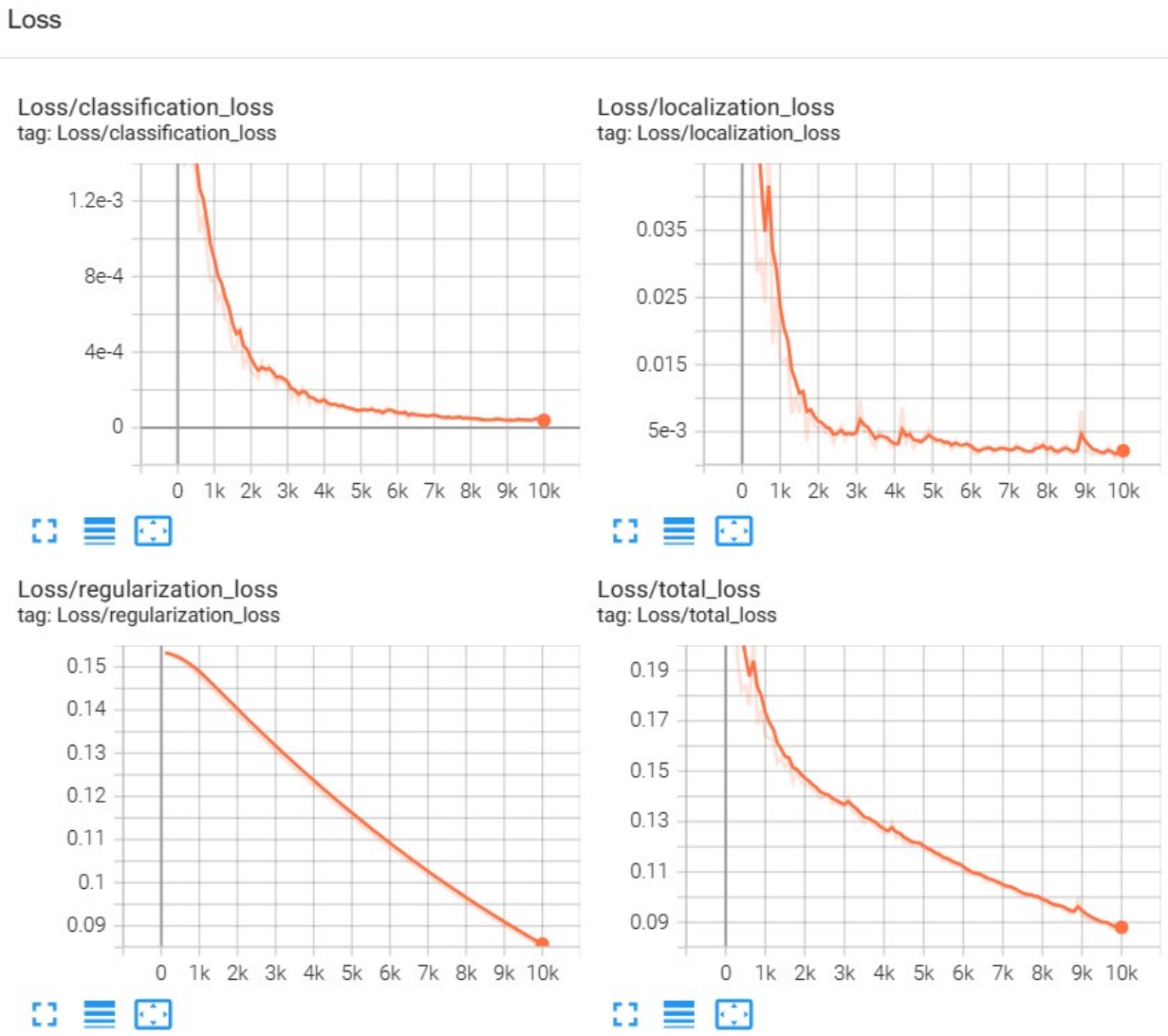


Figure 9: Four Types of Loss Functions During Training Progress

3.2.2 Learning Rate Figure

The learning rate in Figure 10 shows how the learning rate changes over the training process. The learning rate controls the step size during gradient descent optimization. Initially, the learning rate increases to allow rapid learning, then it stabilizes and gradually decreases to fine-tune the model's weights. The shape of the curve indicates that the training process starts with aggressive learning and transitions to fine-tuning as training progresses.

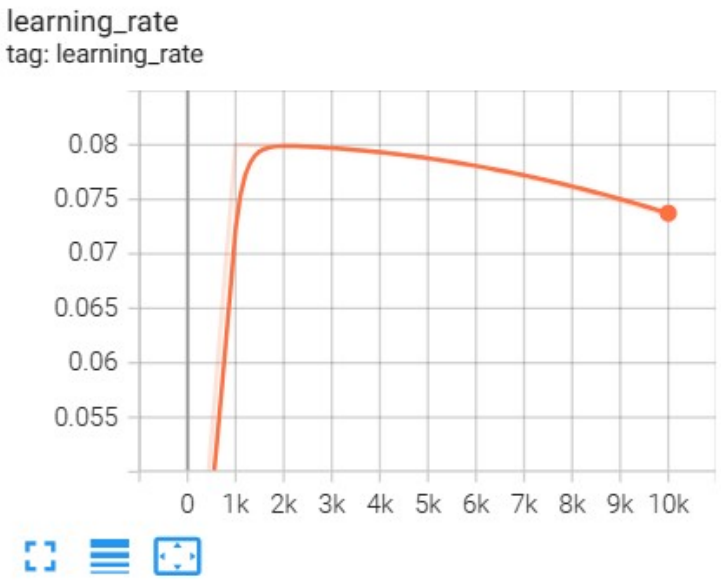


Figure 10: Learning Rate During Training Progress

3.2.3 Training Process Analysis

The general training process can be analyzed from the above loss functions and learning rate figures:

- **Early Stage**: The initial part of the training shows a steep decline in classification and localization losses, indicating a rapid improvement in both classifying objects and predicting their locations. This is supported by a high initial learning rate that facilitates quick learning.
- **Mid Stage**: As training progresses, the decrease in losses starts to slow down, showing that the model is entering a phase of fine-tuning. The learning rate stabilizes and then gradually decreases, indicating a shift from rapid learning to careful adjustment of the model's parameters.
- **Late Stage**: In the later stages of training, the losses continue to decrease at a slower rate, and the learning rate decreases further. This suggests that the model is making smaller and more precise adjustments to improve performance without overfitting.

Overall, the figures demonstrate a successful training process in which the model improves significantly in terms of classification and localization, with effective regularization to ensure generalization. The learning rate adjustments contribute to a balanced training process, optimizing both speed and accuracy.

3.2.4 Test Results and Analysis

After converting the fine-tuned SSD-MobileNet-V2 model to TensorFlow Lite (TFLite) format, I evaluated its performance on my custom test dataset discussed in Section 2.2.3.

Two examples of inference results and the mAP results on the test dataset are analyzed in the following two sections.

Inference Results Two example inference results are shown in Figure 11, captured in the actual lab environment where we have set up our robot arm and the water dispenser. In both test images, cups are placed in front of the robot arm. The green bounding boxes accurately identify the position and label of the cups. These results demonstrate the TFLite model's high accuracy in detecting and labeling the cups. The bounding boxes are well-aligned with the objects, indicating the model's ability to accurately locate and classify the items.

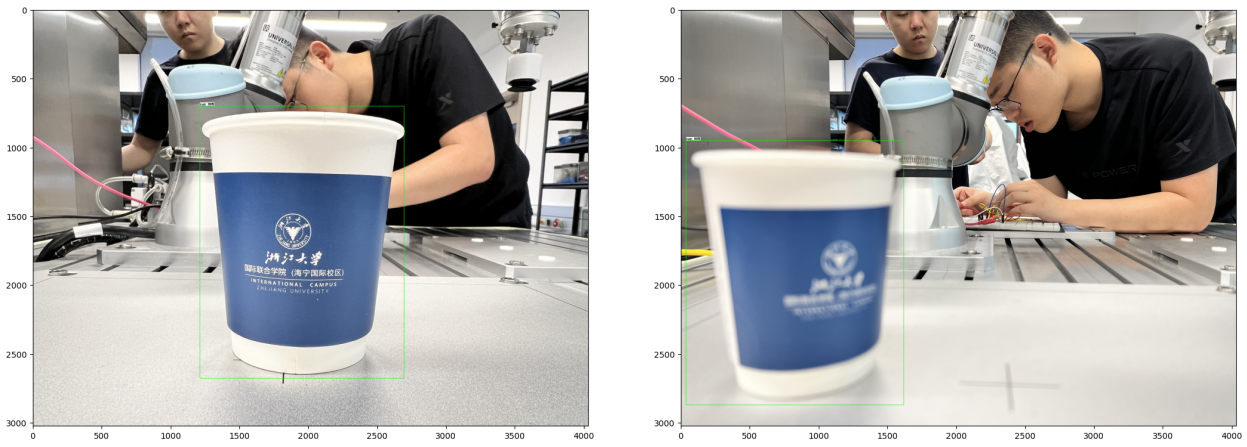


Figure 11: Example Inference Results of the TFLite Model on the Custom Test Dataset

Mean Average Precision (mAP) Results Mean Average Precision (mAP) is a standard metric used to evaluate the performance of object detection models. It considers both the precision and the recall of the detections.

- **Precision** is the ratio of true positive detections to the total number of detections made by the model.
- **Recall** is the ratio of true positive detections to the total number of actual objects present in the images.

The mAP is calculated by averaging the Average Precision (AP) for each class. AP is computed as the area under the Precision-Recall (PR) curve.

- **Precision:**

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

- **Recall:**

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

- **Average Precision (AP):**

$$\text{AP} = \int_0^1 \text{Precision}(\text{Recall})\, d\text{Recall}$$

- **Mean Average Precision (mAP):**

$$\text{mAP} = \frac{1}{N} \sum_{i=1}^{N} \text{AP}_i$$

COCO Metric for mAP @ 0.50:0.95: The mAP results for the TFLite model are determined using the **mAP calculator**[6] tool [19]. They are reported using the COCO metric for mAP @ 0.50:0.95 [20], as shown in Figure 12. This metric is the average AP across different Intersection over Union (IoU) thresholds ranging from 0.50 to 0.95 in increments of 0.05. It provides a comprehensive evaluation of the model's performance by considering varying levels of detection overlap.

```
***mAP Results***

Class           Average mAP @ 0.5:0.95
---------------------------------------
bottle          75.63%
cup             89.11%

Overall         82.37%
```

Figure 12: mAP Results for the TFLite Model on the Custom Test Dataset

- **Bottle Detection**: The model achieved an average mAP of **75.63%** for detecting bottles. This indicates a high level of accuracy, but also suggests room for improvement, possibly due to the variability in bottle appearances.
- **Cup Detection**: The model performed exceptionally well in detecting cups, with an average mAP of **89.11%**. This high accuracy demonstrates the model's robustness in recognizing and localizing cups.
- **Overall Performance**: The overall mAP of **82.37%** reflects the model's strong performance across both classes. This high mAP score indicates that the fine-tuned SSD-MobileNet-V2 model is highly effective for the water bottle detection task.

In conclusion, the TFLite model exhibits excellent detection capabilities, with high mAP scores indicating reliable performance for both bottles and cups. The conversion to TFLite format did not compromise accuracy, making it suitable for deployment on edge devices such as the Raspberry Pi.

[6] https://github.com/Cartucho/mAP

3.3 Performance of the PCB Board

We conducted tests to analyze the behavior of the LEDs in our circuit, focusing on their brightness at different voltages.

Red LED The red LED begins to shine brightly at approximately 2.2 V and reaches full brightness at around 3.0 V, indicating efficient operation within this voltage range.

Green LED The green LED, however, remains dimmer even with increased voltage.

3.3.1 Attempts to Improve Brightness

To enhance the green LED's brightness, we replaced the LED and reduced the resistance in its branch from 1000 Ω to 680 Ω. These changes did not significantly improve its brightness, making the green LED less visible during demonstrations in dim lighting.

3.3.2 Possible Causes and Future Steps

The difference in brightness may be due to varying forward voltage requirements and efficiencies between the LEDs. Future steps to balance the brightness include:

- Using LEDs with matched forward voltage and efficiency.
- Implementing precise current control.
- Exploring alternative circuit configurations.

These optimizations aim to achieve a balanced and effective LED display in our circuit.

4 Cost Analysis

Our fixed labor salary is estimated to be \$10/hour, and 50 hours for each person. The total labor costs for all partners:

$$4 \cdot \$10/\text{hour} \cdot 2.5 \cdot 50 \text{ hours} = \$5000$$

The costs of all parts in our project are shown in Table 3.

Part	**Cost**
Personal Computer (Macbook)	\$1200
Bluetooth Headset and iPhone Camera	\$1000
Raspberry Pi System	\$150
PCB Board (with Control Lights)	\$20
Robot Arm and AI Server (Borrow from ZJUI)	\$0

Table 3: Cost of Each Part

The grand total costs: \$5000 + \$1200 + \$1000 + \$150 + \$20 = \$7370.

5 Conclusion

In summary, our AI-enhanced robotic service system successfully addresses the challenges faced by blind individuals in navigating public spaces and safely interacting with water dispensers. This project has demonstrated the effective integration of advanced technologies, including large language models (LLMs), large visual language models (LVLMs), and robotics, to create a practical and reliable solution.

The key accomplishments of our project include:

- **Effective Real-Time Visual and Verbal Input Processing:** The head-mounted camera and LVLM work seamlessly to capture and analyze the user's surroundings and commands, providing real-time guidance.
- **Dynamic and Accurate User Guidance:** The BLIP-2 model provides precise navigation assistance, warning of potential dangers and instructing the user on interacting with water dispensers.
- **Autonomous Bottle Refilling:** The Universal Robot Arm UR3e, controlled by the Robot Operating System and instructed by the Raspberry Pi Auxiliary System, performs autonomous water bottle refilling, ensuring safety and efficiency.
- **Clear and Effective User Communication:** Audio feedback and instructions delivered through a Bluetooth headset guarantee clear communication, enhancing the overall user experience.

Our system has undergone rigorous testing, confirming its reliability and effectiveness in real-world scenarios. The navigation component, powered by LVLM, has proven capable of guiding users accurately to water dispensers, while the automated bottle filling component ensures a safe and efficient interaction with the dispensers.

In conclusion, the project demonstrates a significant step forward in assistive technology for blind individuals. This project showcases the potential of integrating AI and robotics to improve accessibility and safety for visually impaired individuals in public spaces. With further refinements and potential scalability, this system could be deployed widely to enhance the independence and quality of life for blind individuals, making public spaces more accessible and user-friendly.

5.1 Ethics and Safety

In developing our robotics project, we adhere to the IEEE Code of Ethics [21] to ensure ethical practice and safety.

5.1.1 Ethics

Privacy (ACM 1.7: Respect the Privacy of Others) We implement strict protocols to protect personal information collected by the robot, ensuring only authorized access and secure storage to maintain privacy and trust.

Fairness (IEEE: Avoiding Real or Perceived Conflicts of Interest) To prevent AI bias, we train our robots with diverse data, encompassing various demographics and cultural backgrounds, ensuring impartial decision-making.

Being Open (ACM 1.2: Avoid Harm) We maintain transparency by documenting algorithms, data inputs, and decision processes, making this information accessible to stakeholders to build trust and confidence.

Professional Development (ACM 2.6) Our team commits to ongoing learning and adaptation of ethical standards to keep our robotics practices responsible and up-to-date with societal impacts.

5.1.2 Safety

Avoiding Accidents (IEEE: Priority to Public Welfare) Our robots are equipped with advanced sensors and emergency stops to prevent collisions, prioritizing the safety of individuals and property.

Staying Secure (ACM 3.7: Recognize the Need to Protect Personal Data) We employ strong security measures, including encryption, access controls, and continuous monitoring, to protect our robots from cyber threats.

Dealing with Mistakes (ACM 2.5 & IEEE: Acknowledge and Correct Mistakes) Our robots have fail-safe mechanisms and real-time monitoring to detect and address anomalies, ensuring prompt intervention and continuous safe operation.

Responsibility (IEEE) We commit to the responsible deployment of robotics, regularly assessing and mitigating negative impacts to ensure our innovations benefit society and the environment.

Whistleblowing (ACM 1.4) We encourage and protect whistleblowing to maintain high ethical standards, addressing any misuse or misconduct involving our robots promptly.

References

[1] J. Li, D. Li, S. Savarese, and S. Hoi, "Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models," *arXiv preprint arXiv:2301.12597*, 2023.

[2] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," 2022.

[3] Y.-C. Chiu, C.-Y. Tsai, M.-D. Ruan, G.-Y. Shen, and T.-T. Lee, "Mobilenet-ssdv2: An improved object detection model for embedded systems," in *2020 International conference on system science and engineering (ICSSE)*. IEEE, 2020, pp. 1–5.

[4] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*. Springer, 2016, pp. 21–37.

[5] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[6] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[7] TensorFlow, "Tensorflow 2 detection model zoo," 2021, accessed: 2024-05-25. [Online]. Available: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md

[8] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*. Springer, 2014, pp. 740–755.

[9] Tzutalin, "Labelimg," 2015, accessed: 2024-05-25. [Online]. Available: https://github.com/HumanSignal/labelImg

[10] Makeblock, "About megapi - programming guide," 2024, accessed: 2024-05-25. [Online]. Available: https://support.makeblock.com/hc/en-us/articles/12963818051991-About-MegaPi#6.%20Programming%20Guide

[11] U. Robots, "Ur3e robot arm technical details," 2024, accessed: 2024-05-25. [Online]. Available: https://www.universal-robots.com/media/1802780/ur3e-32528_ur_technical_details_.pdf

[12] O. S. R. Foundation, "Ros noetic installation on ubuntu," 2024, accessed: 2024-05-25. [Online]. Available: http://wiki.ros.org/Installation/Ubuntu

[13] R. P. Foundation, "Raspberry pi documentation - computers," 2024, accessed: 2024-05-25. [Online]. Available: https://www.raspberrypi.com/documentation/computers/raspberry-pi.html

[14] A. Yang, A. Miech, J. Sivic, I. Laptev, and C. Schmid, "Zero-shot video question answering via frozen bidirectional language models," 2022.

[15] K. Li, Y. He, Y. Wang, Y. Li, W. Wang, P. Luo, Y. Wang, L. Wang, and Y. Qiao, "Videochat: Chat-centric video understanding," 2024.

[16] M. Maaz, H. Rasheed, S. Khan, and F. S. Khan, "Video-chatgpt: Towards detailed video understanding via large vision and language models," 2023.

[17] H. Zhang, X. Li, and L. Bing, "Video-llama: An instruction-tuned audio-visual language model for video understanding," 2023.

[18] B. Lin, Y. Ye, B. Zhu, J. Cui, M. Ning, P. Jin, and L. Yuan, "Video-llava: Learning united visual representation by alignment before projection," 2023.

[19] J. Cartucho, "map: Mean average precision for object detection," 2020, accessed: 2024-05-25. [Online]. Available: https://github.com/Cartucho/mAP

[20] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," 2015, accessed: 2024-05-25. [Online]. Available: https://cocodataset.org/#detection-eval

[21] Institute of Electrical and Electronics Engineers. (2016) IEEE Code of Ethics. Accessed on: 2024-03-07. [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8.html

Appendix A Team Photo and Introductions

Figure 13 was taken in the ZJUI Intelligent Robotics Laboratory of our team and Prof. Liangjing Yang. We are grateful to Prof. Liangjing Yang and the ZJUI Intelligent Robotics Laboratory for providing the UR3e Robot Arm and Robot Gripper for our project.

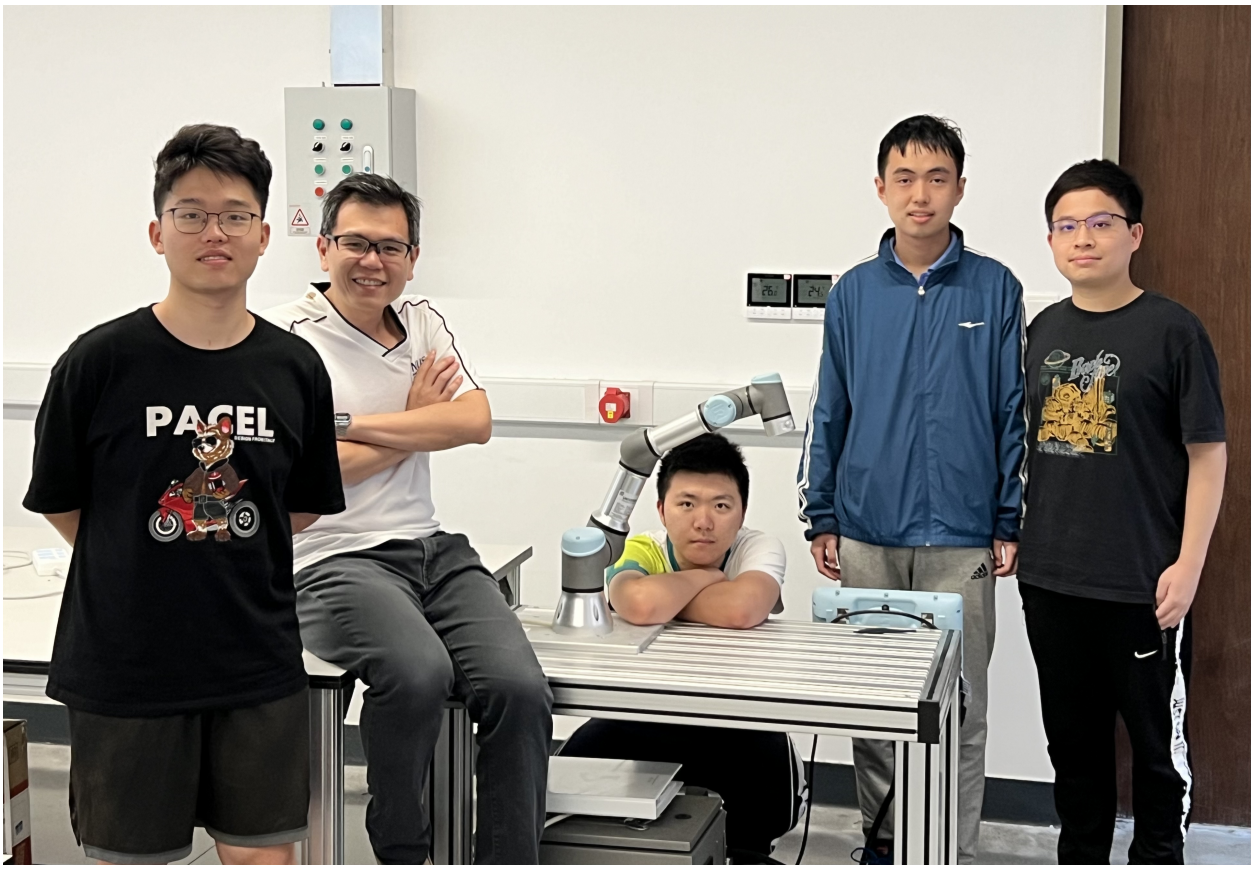


Figure 13: Team Photo

Team Members (From left to right):
Jiatong Li; Prof. Liangjing Yang; Zonghai Jing; Haozhe Chi; Minghua Yang

Appendix B Supplementary Figures

B.1 Architecture of LVLMs

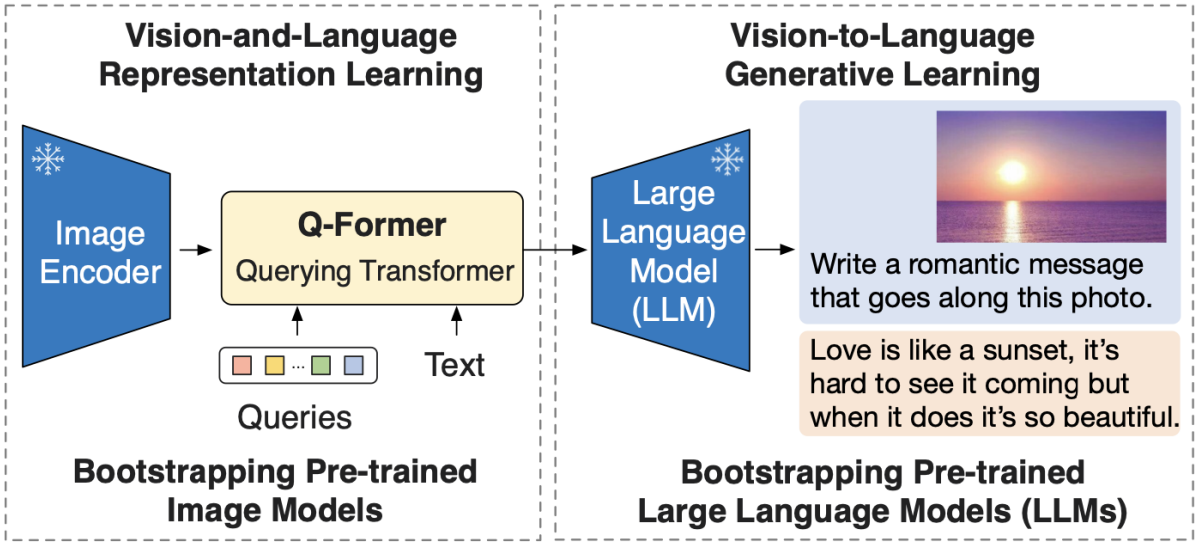


Figure 14: BLIP-2-based Model Architecture

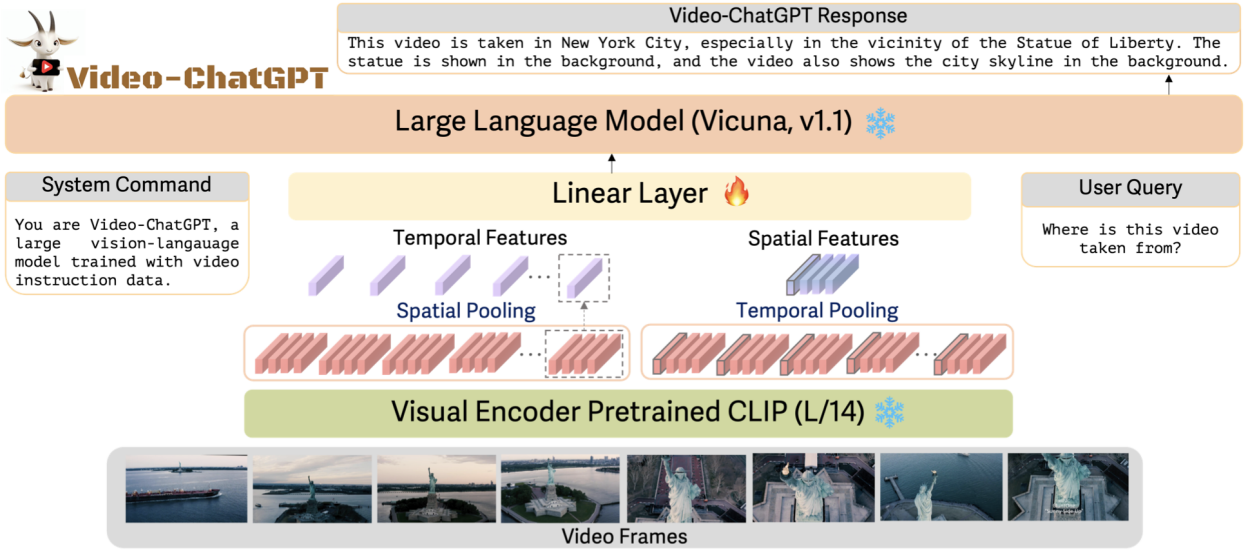


Figure 15: Video-ChatGPT Model Architecture

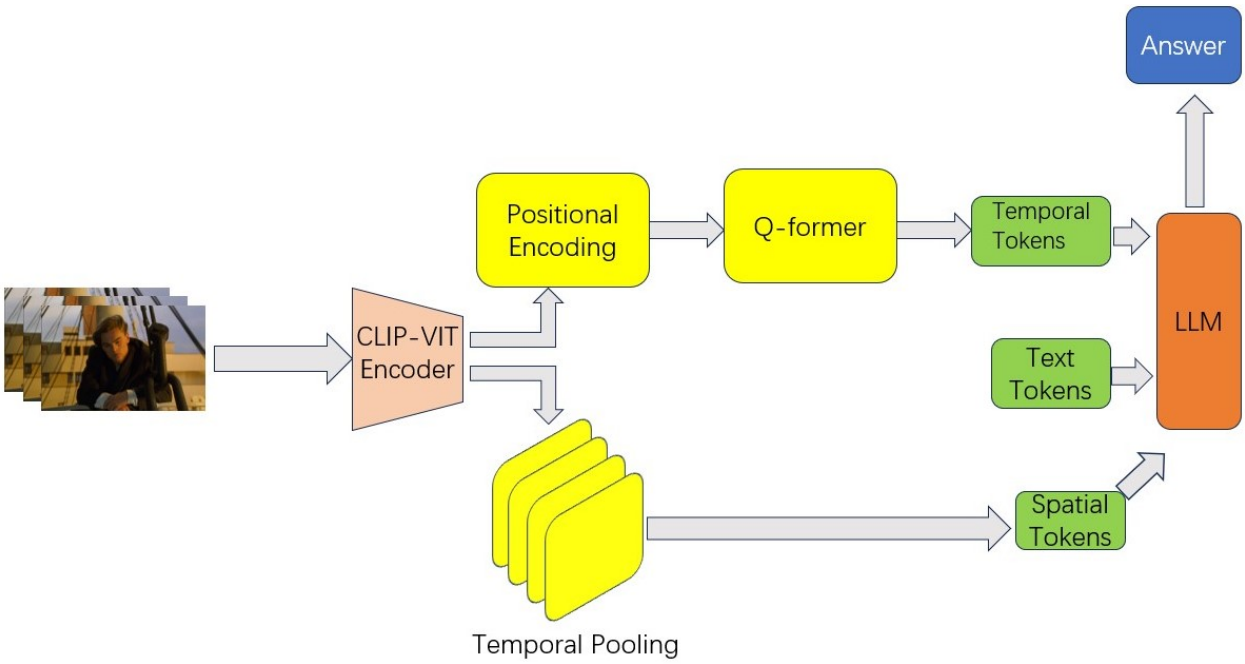


Figure 16: Modified Video-ChatGPT Model Architecture

B.2 Flash-Attention Module

Algorithm 1 FLASHATTENTION

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

Set block sizes $B_c = \left\lceil \frac{M}{4d} \right\rceil, B_r = \min\left(\left\lceil \frac{M}{4d} \right\rceil, d\right)$.

Initialize $\mathbf{O} = (0)_{N \times d} \in \mathbb{R}^{N \times d}, \ell = (0)_N \in \mathbb{R}^N, m = (-\infty)_N \in \mathbb{R}^N$ in HBM.

Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

Divide $\mathbf{O}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, divide ℓ into T_r blocks $\ell_i, \ldots, \ell_{T_r}$ of size B_r each, divide m into T_r blocks $m_1, \ldots, m_{T_r}$ of size B_r each.

for $1 \le j \le T_c$ **do**

  Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.

  for $1 \le i \le T_r$ **do**

    Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.

    On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i \mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $\tilde{m}_{ij} = \text{rowmax}(\mathbf{S}_{ij}) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_{ij} = \exp(\mathbf{S}_{ij} - \tilde{m}_{ij}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\tilde{\ell}_{ij} = \text{rowsum}(\tilde{\mathbf{P}}_{ij}) \in \mathbb{R}^{B_r}$.

    On chip, compute $m_i^{\text{new}} = \max(m_i, \tilde{m}_{ij}) \in \mathbb{R}^{B_r}$, $\ell_i^{\text{new}} = e^{m_i - m_i^{\text{new}}} \ell_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\ell}_{ij} \in \mathbb{R}^{B_r}$.

    Write $\mathbf{O}_i \leftarrow \text{diag}(\ell_i^{\text{new}})^{-1}(\text{diag}(\ell_i) e^{m_i - m_i^{\text{new}}} \mathbf{O}_i + e^{\tilde{m}_{ij} - m_i^{\text{new}}} \tilde{\mathbf{P}}_{ij} \mathbf{V}_j)$ to HBM.

    Write $\ell_i \leftarrow \ell_i^{\text{new}}$, $m_i \leftarrow m_i^{\text{new}}$ to HBM.

  end for

end for

Return $\mathbf{O}$.

Figure 17: Flash-attention Algorithm

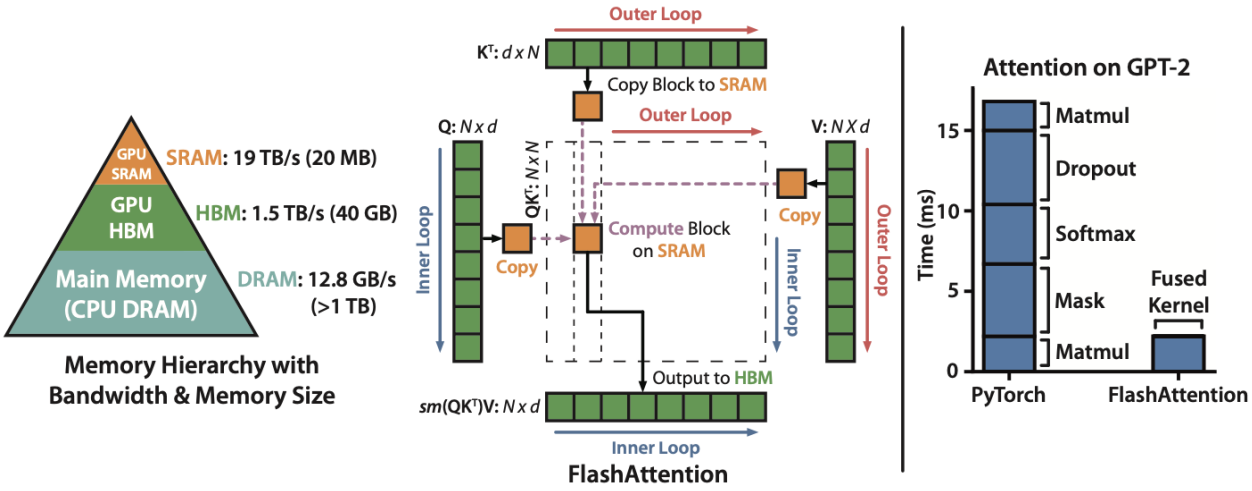


Figure 18: Flash-attention Layout

B.3 Raspberry Pi Auxiliary System

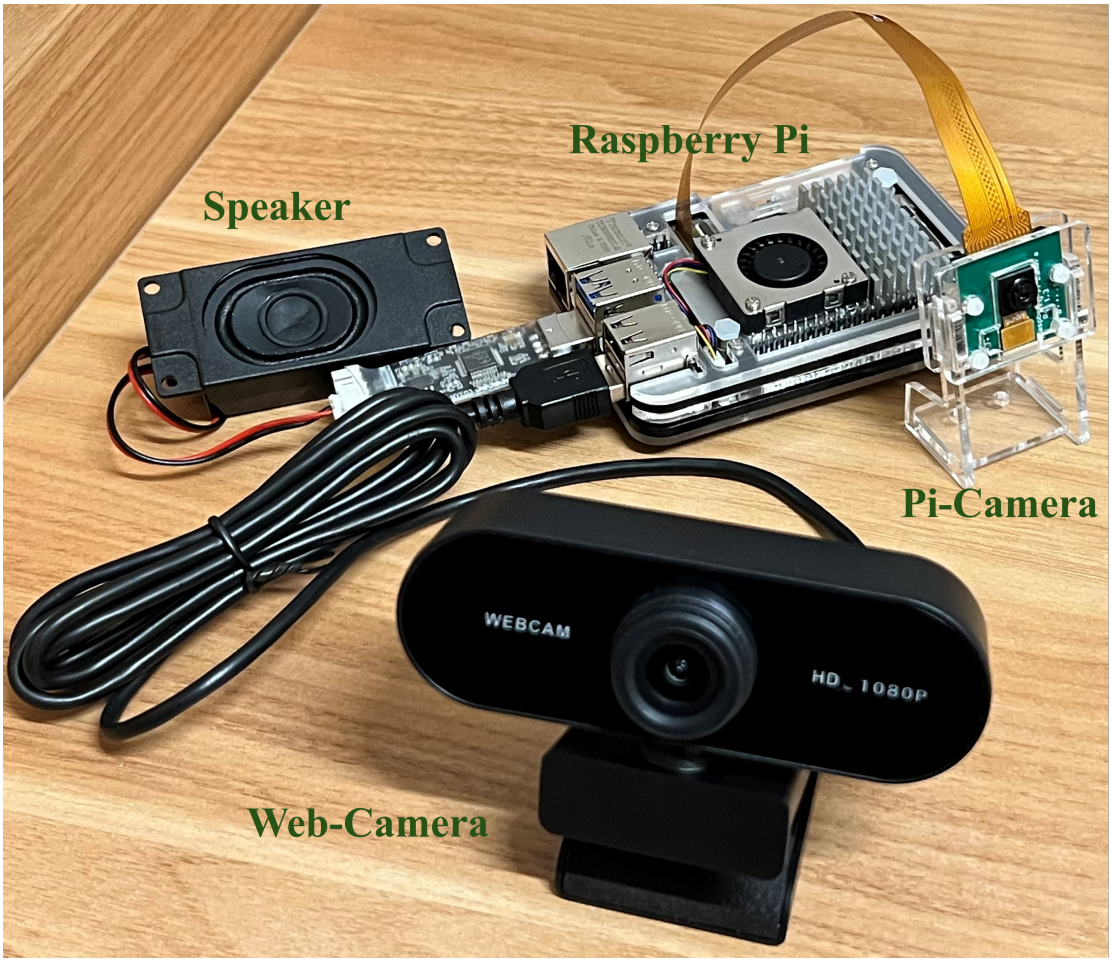


Figure 19: Architecture Overview of the Raspberry Pi Auxiliary System

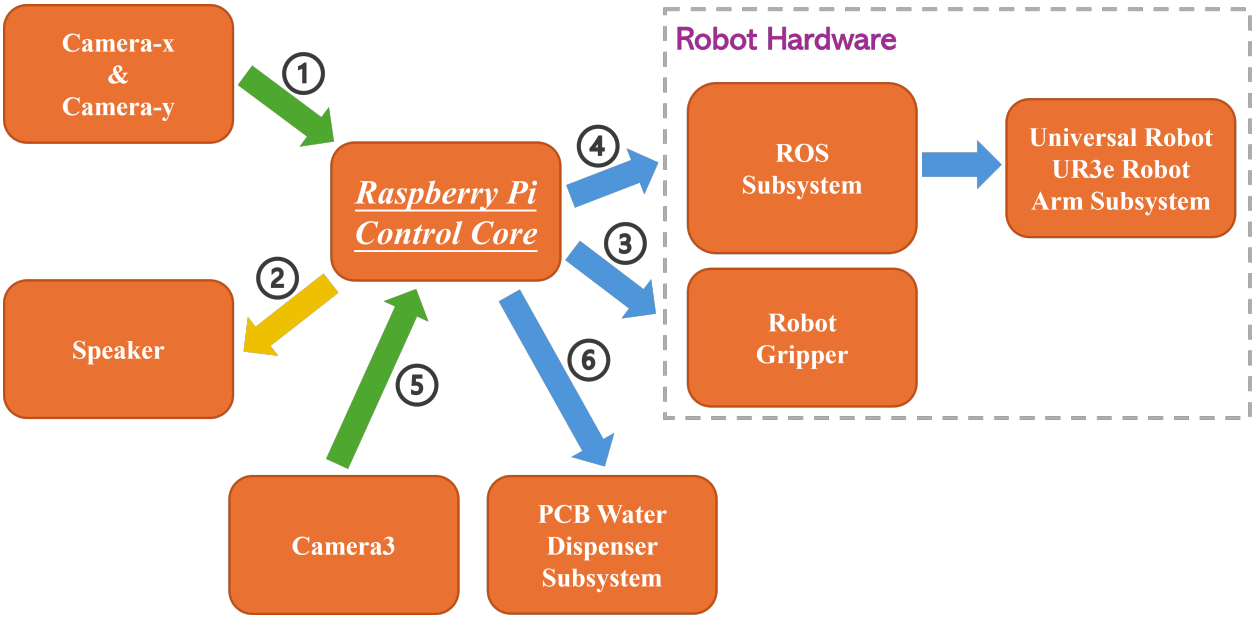


Figure 20: Overall Workflow Block Diagram of the Raspberry Pi Auxiliary System

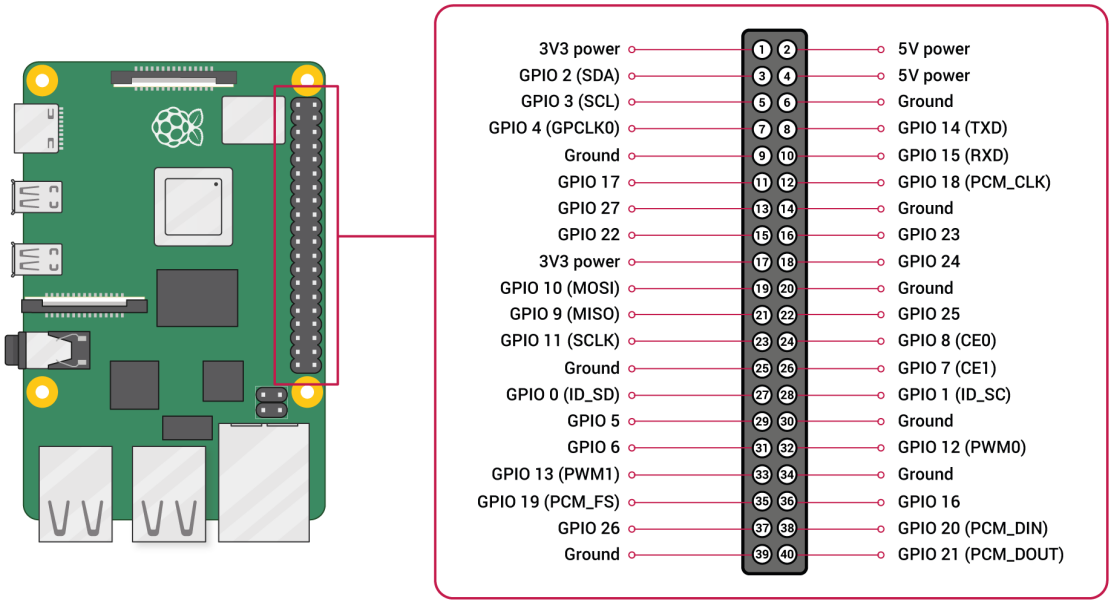


Figure 21: Raspberry Pi GPIO 40-Pin Layout

B.4 PCB Board Design

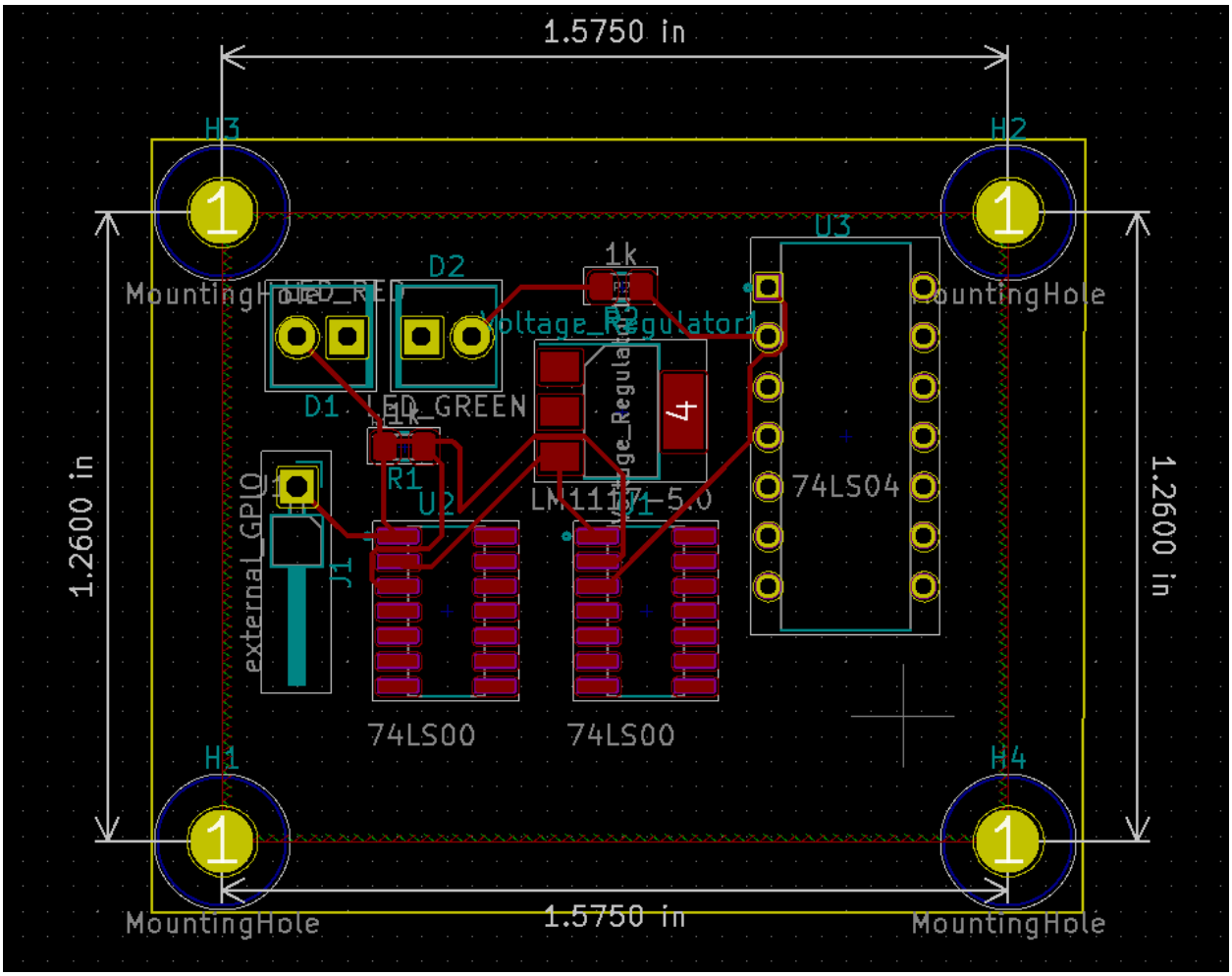
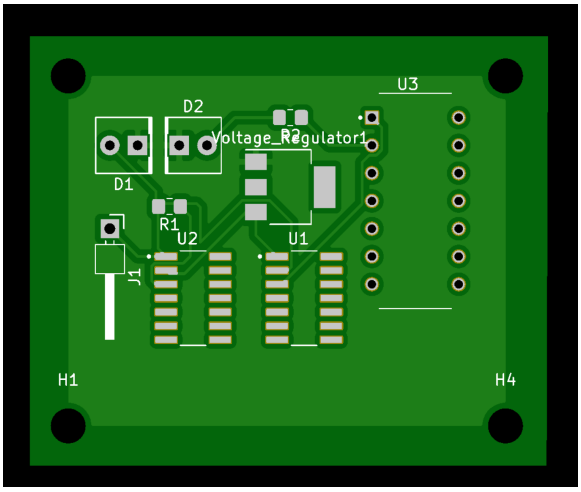
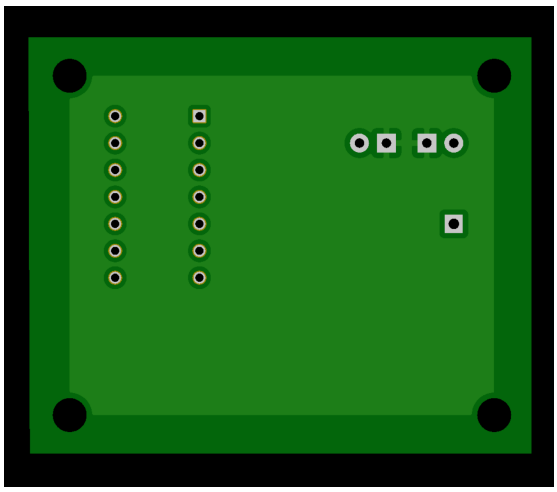


Figure 22: PCB Schematics in KiCad



(a) PCB Front



(b) PCB Back

Appendix C Code and Algorithm

C.1 Code for Taking Screenshots

```
while True:
    time.sleep(1)
    demo_image = pyautogui.screenshot()
    demo = demo_image.crop((375, 393, 2000, 1600))
    #print(demo_image.size)
    demo.save("./demo.png")
    img_files = ['./demo.png']
    duration = 10.0
    fps = 30
    clip = ImageSequenceClip(img_files, fps=fps, durations=
        ↪ duration)
    clip.write_videofile('demo.mp4', fps=fps)
    sftp.put(filevideopth, remotevideopth)
    sftp.put(fileimgpth, remoteimgpth)
    #time.sleep(1)
```

C.2 Main ROS Code to Control the Robot Arm

```
def move_arm(pub_cmd, loop_rate, dest, vel, accel):
    global thetas
    global SPIN_RATE

    error = 0
    spin_count = 0
    at_goal = 0

    driver_msg = command()
    driver_msg.v = vel
    driver_msg.a = accel
    driver_msg.destination = dest
    driver_msg.io_0 = current_io_0
    pub_cmd.publish(driver_msg)

rospy.init_node('ECE445')

# Initialize publisher for ur3/command with buffer size of 10
pub_command = rospy.Publisher('ur3/command', command, queue_size
    ↪ =10)
sub_position = rospy.Subscriber('ur3/position', position,
    ↪ position_callback)
```

```
sub = rospy.Subscriber('/ur3/gripper_input', gripper_input,
   ↪ gripper_input_callback)
```

C.3 Inference Script for TFLite Model

```
from tflite_runtime.interpreter import Interpreter
import numpy as np
import cv2

# Load the TFLite model and allocate tensors
interpreter = Interpreter(model_path="model.tflite")
interpreter.allocate_tensors()

# Get input and output tensors
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Load the label map
with open('labelmap.txt', 'r') as f:
   labels = [line.strip() for line in f.readlines()]

# Function to perform inference
def detect_water_bottle(image_path, frame):
   # Load and preprocess each frame form camera
   input_data = np.expand_dims(frame, axis=0)

   # Set input tensor
   interpreter.set_tensor(input_details[0]['index'], input_data)

   # Run the model
   interpreter.invoke()

   # Class index of detected objects
   classes = interpreter.get_tensor(output_details[classes_idx]['
      ↪ index'])[0]

   # Get output label
   Object_name = labels[int(classes[i])]
```

C.4 Bottle Position Detection Algorithm

Algorithm 1 Bottle Position Detection

Require: Video stream from camera; screen center *screen_center_x*; central position threshold *central_position_threshold*; position stability threshold *position_stability_threshold*

Initialize *object_positions* $\leftarrow \{\}$, *object_stability_count* $\leftarrow \{\}$
Start video stream from camera
while True **do**
    Capture frame from video stream
    Perform object detection on the frame
    for each detected object **do**
        if object is a bottle or cup **then**
            Calculate center coordinates $(center\_x, center\_y)$ of the bounding box
            Get object identifier *object_id*
            if *object_id* in *object_positions* **then**
                if Position is stable **then**
                    Increment stability count *object_stability_count*[*object_id*]
                else
                    Update position and reset stability count
                    object_positions[*object_id*] $\leftarrow (center\_x, center\_y)$
                    object_stability_count[*object_id*] $\leftarrow 1$
                end if
            else
                Initialize position and stability count
                object_positions[*object_id*] $\leftarrow (center\_x, center\_y)$
                object_stability_count[*object_id*] $\leftarrow 1$
            end if
            {if the object has been in the same position for 8 consecutive frames}
            if *object_stability_count*[*object_id*] ≥ 8 **then**
                if *center_x* $<$ *screen_center_x* $-$ *central_position_threshold* **then**
                    position $\leftarrow right$
                else if *center_x* $>$ *screen_center_x* $+$ *central_position_threshold* **then**
                    position $\leftarrow left$
                else
                    position $\leftarrow center$
                end if
                if *position* $== center$ **then**
                    Output audio instruction: "Hold your *object_id* in the current position and wait for the next instruction"
                    Stop video stream and return success
                else
                    Output audio instruction: "Move your *object_id position* a bit"
                end if
                Reset stability count *object_stability_count*[*object_id*] $\leftarrow 0$
            end if
        end if
    end for
    if exit condition met **then**
        Break
    end if
end while
Stop video stream and clean up

C.5 Code for Robot Gripper

```
from time import sleep
from megapi import MegaPi

class Gripper:
    def __init__(self, usb_port='/dev/ttyUSB0', motor_port=4,
        ↪ motor_speed=100, run_time=1):
        """
        Initialize the gripper.
        :param usb_port: Specify the USB-port connection to the
            ↪ MegaPi.
        :param motor_port: specify the motor port on MegaPi.
        :param motor_speed: Specify the speed of the motor.
        :param run_time: Specify the time (in seconds) to run the
            ↪ motor for open/close operations.
        """
        self.bot = MegaPi()
        self.bot.start(usb_port)
        self.motor_port = motor_port
        self.motor_speed = motor_speed
        self.run_time = run_time

        # Ensure the motor is stopped initially.
        self.bot.motorRun(self.motor_port, 0)
        sleep(1) # Wait for the MegaPi to initialize properly.

    def release(self):
        """
        Open the gripper.
        """
        self.bot.motorRun(self.motor_port, -(self.motor_speed))
        sleep(self.run_time)
        self.stop()
        return 1

    def grab(self):
        """
        Close the gripper.
        """
        self.bot.motorRun(self.motor_port, self.motor_speed)
        sleep(self.run_time)
        self.stop()
        return 1
```

```
    def stop(self):
        """
        Stops any gripper motion.
        """
        self.bot.motorRun(self.motor_port, 0)
        sleep(1) # Pause to allow the motor to fully stop.

    def close(self):
        """
        Closes the serial connection and cleans up.
        """
        self.bot.close()
        self.bot.exit(0, 0)
```

C.6 Raspberry Pi GPIO Control

```
import time
import gpiod

LED_GREEN_PIN = 27
LED_RED_PIN = 17

def Led_green_ON(LED_GREEN_PIN):
  chip = gpiod.Chip('gpiochip4')
  led_line_green = chip.get_line(LED_GREEN_PIN)
  led_line_green.request(consumer="LED", type=gpiod.
    ↪ LINE_REQ_DIR_OUT)
  led_line_green.set_value(1)

def Led_green_OFF(LED_GREEN_PIN):
  chip = gpiod.Chip('gpiochip4')
  led_line_green = chip.get_line(LED_GREEN_PIN)
  led_line_green.request(consumer="LED", type=gpiod.
    ↪ LINE_REQ_DIR_OUT)
  led_line_green.set_value(0)
  time.sleep(1)
  led_line_green.release()

### Check if the water bottle is placed at the water dispenser
if (Detect_Bottle()):
   print("Detect␣bottle␣at␣the␣water␣dispenser...")
   chip = gpiod.Chip('gpiochip4')
   led_line_red = chip.get_line(LED_RED_PIN)
```

```
    led_line_red.request(consumer="LED", type=gpiod.
        ↪ LINE_REQ_DIR_OUT)
    led_line_red.set_value(1)

    # Start filling the bottle (dispensing hot water)
    play_audio_vlc("filling_bottle.mp3")

    led_line_red.set_value(0)
    led_line_red.release()
else:
    print("No bottle at the water dispenser!")
```

C.7 Shell Script Implementation

```
#!/bin/bash

# SSH into the Ubuntu VM to control the Robot Arm (Reset position
    ↪ )
ssh chris@192.168.137.119 << EOF
source /opt/ros/noetic/setup.bash
source ~/catkin_robot_arm/devel/setup.bash
rosrun RobotArm_pkg_py RobotArm_reset.py
logout
EOF
echo "Robot arm reset successful"

# Function to handle cleanup
cleanup() {
    echo "Cleaning up..."
    python3 water_dispenser_led.py OFF
    echo "LED cleanup complete. Exit!"
    exit 0 # Exit the script
}

# Catch the SIGINT (Ctrl+C), calling cleanup
trap cleanup SIGINT

# Setup holding state of water dispenser (Green LED ON)
python3 water_dispenser_led.py ON
echo "Water dispenser is ready..."

while true; do

    # Detect water bottle and then grab
```

```
    echo "Start␣detecting␣the␣location␣of␣water␣bottle..."
    usb_port=$(ls /dev/ttyUSB*)
    echo "USB␣port:␣$usb_port"
    camera1_index=$(v4l2-ctl --list-devices | grep -A 1 'XWF␣1080P
    ↪ ' | tail -n 1)
    echo "Camera␣1␣Index:␣$camera1_index"
    camera2_index=$(v4l2-ctl --list-devices | grep -A 1 'USB2.0
    ↪ _CAM2' | tail -n 1)
    echo "Camera␣2␣Index:␣$camera2_index"
    python3 detect_grab.py --webcam1 $camera1_index --webcam2
    ↪ $camera2_index --usb_port $usb_port

    # SSH into the Ubuntu VM to control the Robot Arm (forward to
    ↪ the water dispenser)
    ssh chris@192.168.137.119 << EOF
source /opt/ros/noetic/setup.bash
source ~/catkin_robot_arm/devel/setup.bash
rosrun RobotArm_pkg_py RobotArm_forward.py
logout
EOF
    echo "Robot␣arm␣is␣moving␣toward␣the␣water␣dispenser..."

    # Start the water dispenser
    python3 check_bottle_led.py

    # SSH into the Ubuntu VM to control the Robot Arm (back to the
    ↪  user)
    ssh chris@192.168.137.119 << EOF
source /opt/ros/noetic/setup.bash
source ~/catkin_robot_arm/devel/setup.bash
rosrun RobotArm_pkg_py RobotArm_back.py
logout
EOF
    echo "Robot␣arm␣is␣moving␣back␣to␣the␣user..."

    # Release the water bottle
    python3 release_bottle.py --usb_port $usb_port
    echo "Water␣bottle␣is␣released..."
    echo "Finish"

    sleep 60
done
```