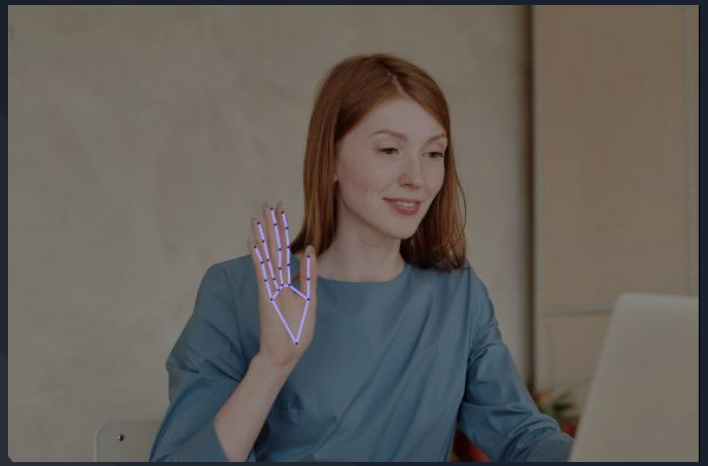
SIGN LANGUAGE RECOGNITION
&
SPEECH SYNTHESIS DEVICE



PROBLEM STATEMENT

Sign language is a means of communication for individuals with hearing impairments.However, a barrier arises when communication with those who do not understand sign language is required. Similarly, individuals who rely on spoken language may struggle to understand sign language.

The objective of this project is to bridge the communication gap people with hearing/speech impairments face by developing a device capable of both recognizing sign language and converting them to speech.

LITERATURE SURVEY

With the help of literature survey done we realized the basic steps in hand gesture recognition are :-

- Data acquisition :- 1. Use of sensory devices

  2. Vision based approach

- Data preprocessing
- Feature extraction
- Gesture classification

ABOUT ESP32

The ESP32 is a series of low-cost, low-power system-on-chip microcontrollers with integrated Wi-Fi and dual-mode Bluetooth capabilities, produced by Espressif Systems.

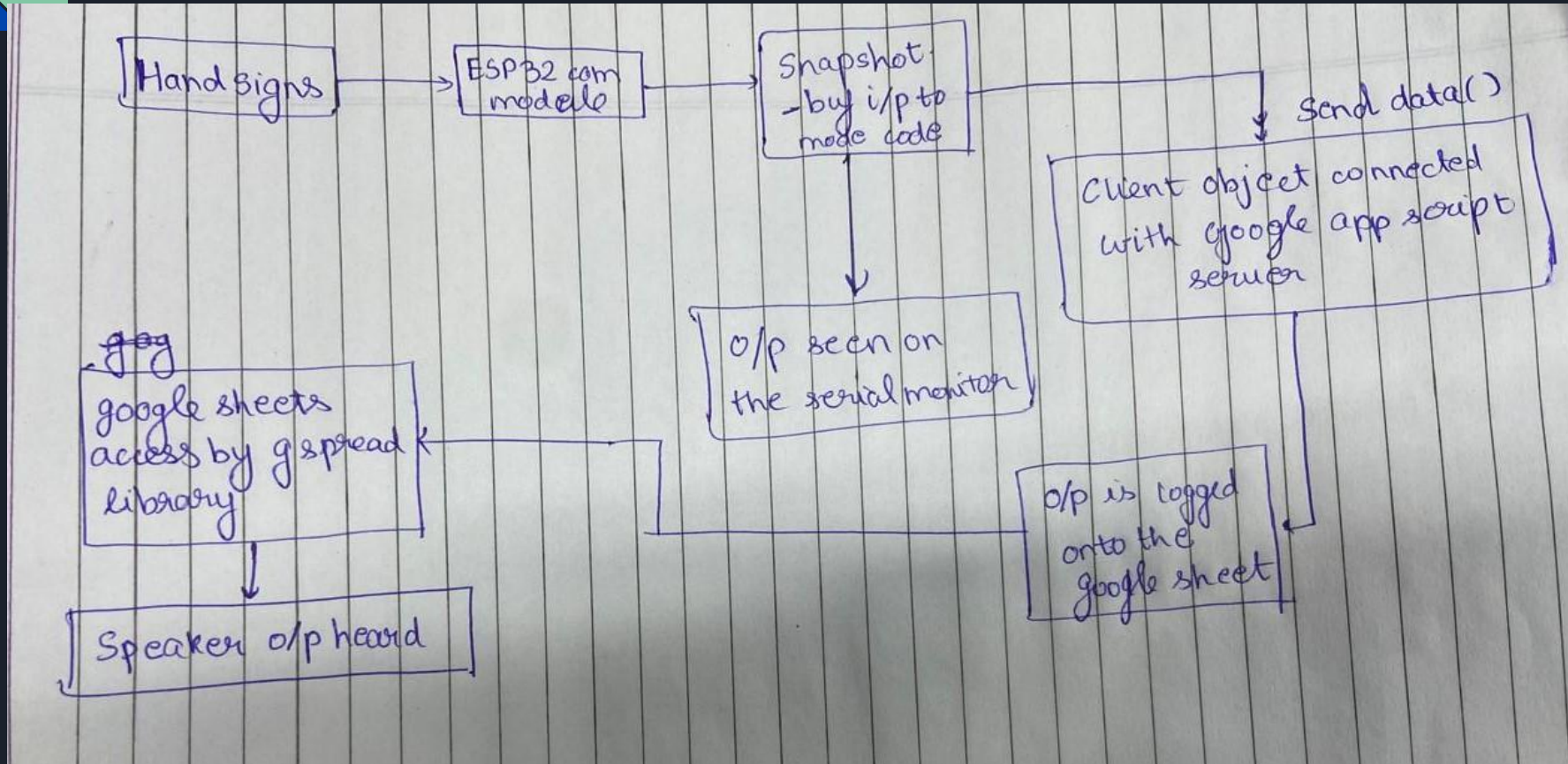
It is widely used in Internet of Things (IoT) projects due to its versatility, low power consumption, and robust connectivity features.

Dual-core processor: The ESP32 features a dual-core Tensilica LX6 processor, which allows for parallel processing and improved performance.

Wi-Fi and Bluetooth: It supports both Wi-Fi 802.11 b/g/n and Bluetooth v4.2 BR/EDR and BLE (Bluetooth Low Energy) connectivity.

GPIO pins: The ESP32 provides a wide array of general-purpose input/output (GPIO) pins, which can be used for various purposes such as digital input/output, analog input, PWM, I2C, SPI, etc.

Block Diagram with speech synthesis

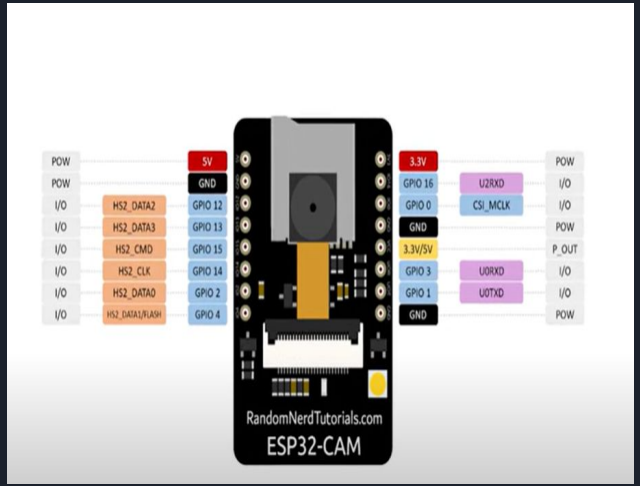


KEY CHALLENGES

Gesture Recognition: Being able to accurately recognize and interpret sign language gestures in various contexts and environments.

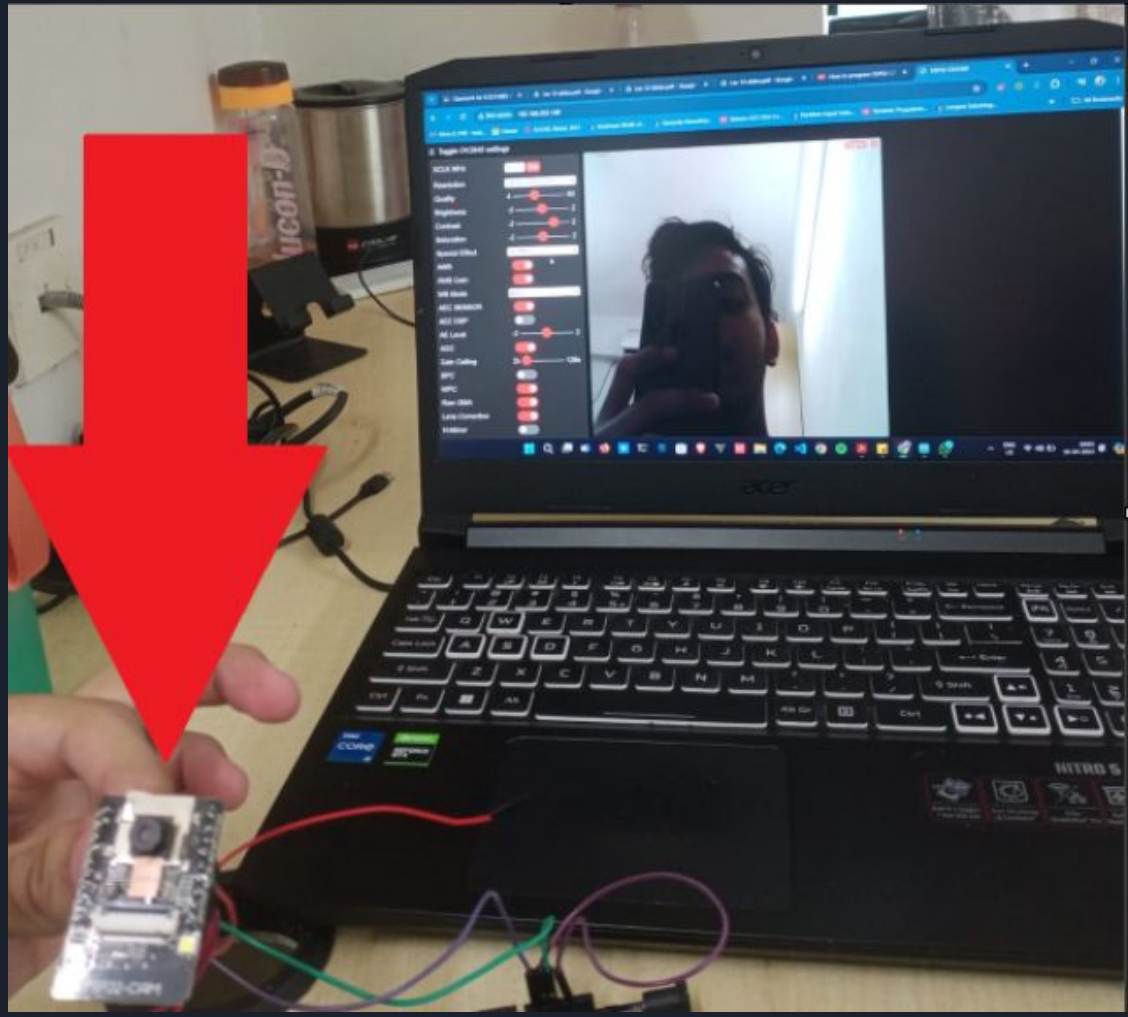
Data Collection and Annotation: Acquiring a dataset of sign language gestures and annotating them for training machine learning models.

Speech Synthesis: convert recognized sign language gestures into spoken language, maintaining clarity and coherence.



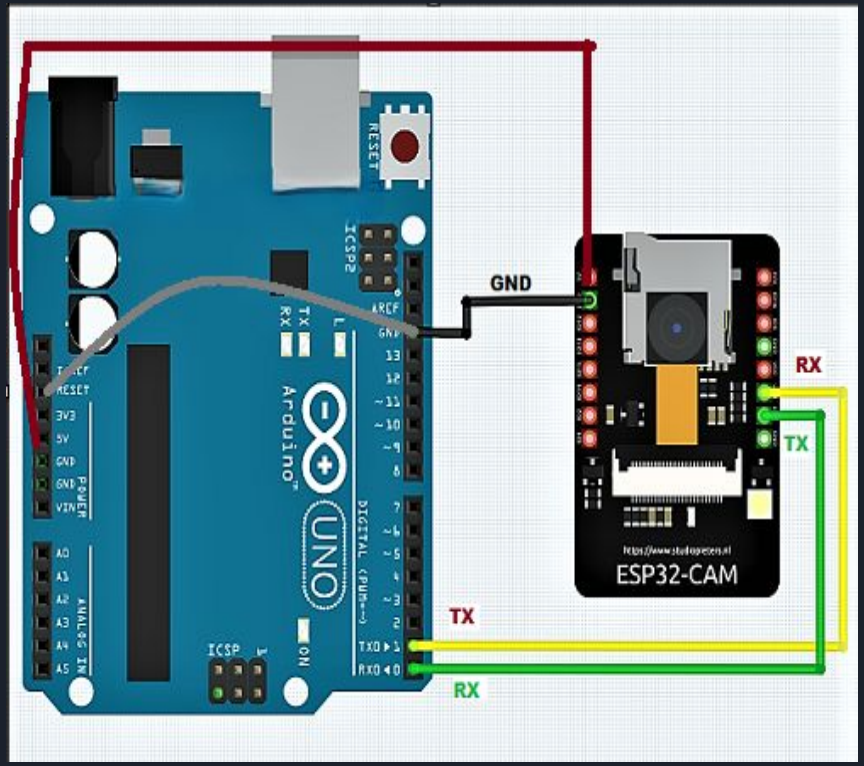
Getting the live feed through wifi:

We opted for the ESP32-CAM module due to its exceptional efficiency in delivering high-quality streams, crucial for our dataset creation of various hand gestures. Utilizing the Arduino IDE as our programming medium, we uploaded the cam_code library onto the ESP32-CAM module. Leveraging the built-in Wi-Fi module, the module established a web server, enabling seamless retrieval of live streams.

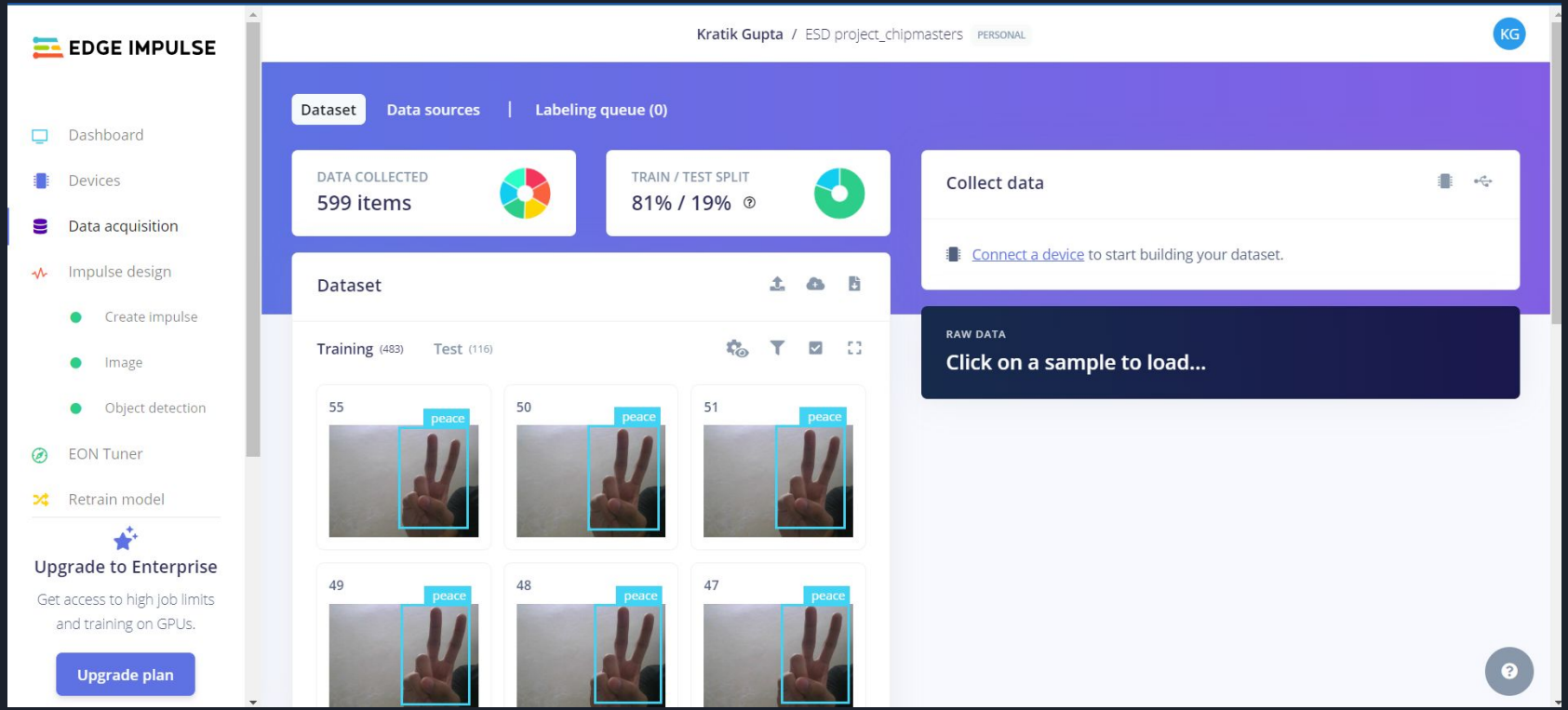


PROGRAMMING THE ESP32-CAM BOARD

The arduino is used to establish a serial connection between computer and ESP32 board . This conversion is necessary because the ESP32 communicates using UART (serial communication), whereas your computer communicates via USB. When you upload code from the Arduino IDE to the ESP32, the Arduino handles the transfer of the compiled program (in binary format) from your computer to the ESP32's flash memory. Also ,it has the serial monitor feature which can detect any errors that occur during the cam streaming.



CREATING DATASET FROM EDGE IMPULSE + TRAINING THE MODEL (6 labels)



Model generated through edge impulse before midsem VS after midsem

Model

Model version: Quantized (int8)

Last training performance (validation set)

F1 SCORE
100.0%

Confusion matrix (validation set)

	BACKGRO	BANG BA	GOOD JOE	LOSER	OK	CALL ME	PEACE
BACKGROUN	100%	0%	0%	0%	0%	0%	0%
BANG BANG	0%	100%	0%	0%	0%	0%	0%
GOOD JOB	0%	0%	100%	0%	0%	0%	0%
LOSER	0%	0%	0%	100%	0%	0%	0%
OK	0%	0%	0%	0%	100%	0%	0%
CALL ME	0%	0%	0%	0%	0%	100%	0%
PEACE	0%	0%	0%	0%	0%	0%	100%
F1 SCORE	1.00	1.00	1.00	1.00	1.00	1.00	1.00

On-device performance

Engine: EON™ Compiler

INFERENCING ...
1082 ms.

PEAK RAM USA...
235.6K

FLASH USAGE
64.5K

Model

Model version: Quantized (int8)

Last training performance (validation set)

F1 SCORE
36.4%

Confusion matrix (validation set)

	BACKGROUND	ONE	TWO
BACKGROUND	99.7%	0.1%	0.2%
ONE	66.7%	33.3%	0%
TWO	71.4%	0%	28.6%
F1 SCORE	1.00	0.40	0.33

On-device performance

Engine: EON™ Compiler

INFERENCING ...
959 ms.

PEAK RAM USA...
235.5K

FLASH USAGE
64.3K

collect_imgs.py M

collect_imgs.py > collect_data_for_class

```python
import os
import cv2
DATA_DIR = './data'
if not os.path.exists(DATA_DIR):
    os.makedirs(DATA_DIR)
number_of_classes = 3
dataset_size = 100
def collect_data_for_class(class_dir):
    if not os.path.exists(class_dir):
        os.makedirs(class_dir)
    print('Collecting data for class {}'.format(class_dir))
    print('Press "Q" to start collecting images.')
    while True:
        ret, frame = cap.read()
        cv2.putText(frame, 'Ready? Press "Q" ! :)', (100, 50), cv2.
        FONT_HERSHEY_SIMPLEX, 1.3, (0, 255, 0), 3,
                    cv2.LINE_AA)
        cv2.imshow('frame', frame)
        if cv2.waitKey(25) == ord('q'):
            break
    counter = 0
    while counter < dataset_size:
        ret, frame = cap.read()
        if not ret:
            print("Error: Couldn't capture frame.")
            break
```

Working on Edge Impulse

Edge Impulse is a platform that enables developers to create intelligent sensing devices using machine learning. It provides tools and infrastructure to collect, process, and analyze sensor data, as well as to build, train, and deploy machine learning models directly onto microcontrollers, which are low-power embedded devices.

One of the key features of Edge Impulse is its focus on edge computing, which means that the machine learning models are deployed directly onto the devices where the data is being collected, rather than relying on sending data to the cloud for processing. This enables real-time analysis and decision-making without requiring a constant internet connection, making it suitable for applications where low latency and privacy concerns are important, such as in IoT (Internet of Things) devices, wearables, and embedded systems.

Objective

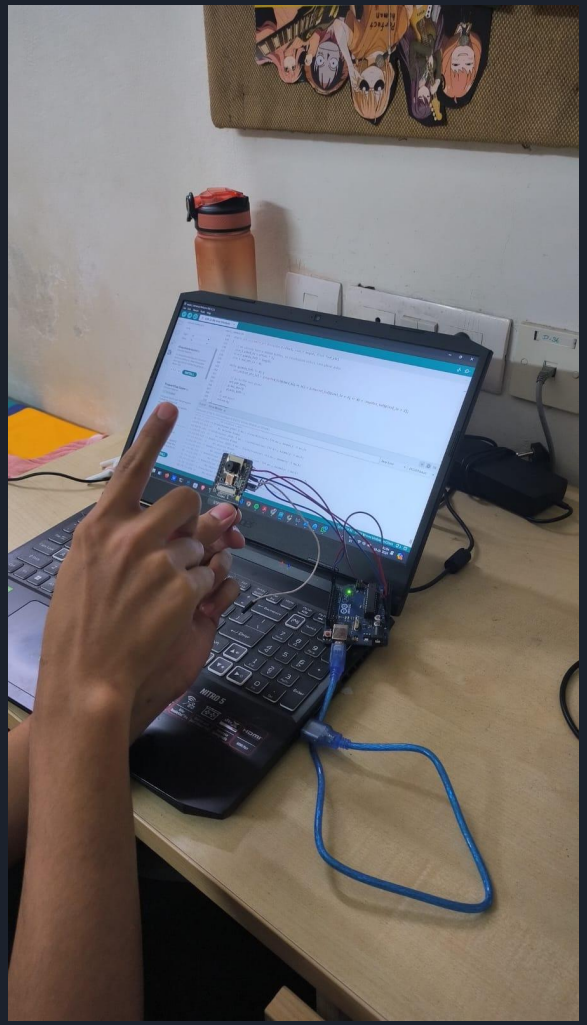
1.Learn how to program and get live feed from a esp32 cam module.

2.Collect and curate a basic dataset

3.Test the dataset model through importing the library created by the tool “EDGE IMPULSE “ by collecting custom data.

RESULTS

Upon deploying our trained model onto the target device (ESP32 CAM MODEL AI THINKER), we focused on recognizing hand gestures for 6 sign language gestures. Each instance of the machine learning model's execution on the CAM module results in distinct outcomes. When the camera detects an object unfamiliar to the model, it promptly displays 'no object detected' on the output terminal. Conversely, when presented with familiar gestures, such as the sign for '1', the model accurately reflects this in real-time on the output terminal. The results showed that the model could accurately recognize and interpret the gestures in real-time. When an unfamiliar object was detected, the output terminal displayed 'no object detected'. Google Script used above puts the particular hand signal in the spreadsheet.



esp32_camera.ino

```
 */

/* Includes ---------------------------------------------------------------- */
#include <ESD_project_inferencing.h>
#include "edge-impulse-sdk/dsp/image/image.hpp"

#include "esp_camera.h"

// Select camera model - find more camera models in camera_pins.h file here
// https://github.com/espressif/arduino-esp32/blob/master/libraries/ESP32/examples/C

// #define CAMERA_MODEL_ESP_EYE // Has PSRAM
#define CAMERA_MODEL_AI_THINKER // Has PSRAM

#if defined(CAMERA_MODEL_ESP_EYE)
#define PWDN_GPIO_NUM    -1
#define RESET_GPIO_NUM   -1
#define XCLK_GPIO_NUM     4
```

Output | Serial Monitor

Not connected. Select a board and a port to connect automatically.

```
23:48:18.248 -> Predictions (DSP: 8 ms., Classification: 544 ms., Anomaly: 0 ms.):
23:48:18.248 ->     No objects found
23:48:18.924 -> Predictions (DSP: 8 ms., Classification: 544 ms., Anomaly: 0 ms.):
23:48:18.959 ->     No objects found
23:48:19.679 -> Predictions (DSP: 8 ms., Classification: 544 ms., Anomaly: 0 ms.):
23:48:19.679 ->     No objects found
23:48:20.376 -> Predictions (DSP: 8 ms., Classification: 544 ms., Anomaly: 0 ms.):
23:48:20.376 ->     one (0.523438) [ x: 64, y: 32, width: 8, height: 8 ]
23:48:21.065 -> Predictions (DSP: 8 ms., Classification: 544 ms., Anomaly: 0 ms.):
23:48:21.065 ->     No objects found
23:48:21.769 -> Predictions (DSP: 8 ms., Classification: 544 ms., Anomaly: 0 ms.):
23:48:21.769 ->     No objects found
23:48:22.481 -> Predictions (DSP: 8 ms., Classification: 544 ms., Anomaly: 0 ms.):
23:48:22.481 ->     No objects found
```

```
void setup()
{
    Serial.begin(115200);
    while (!Serial);
    Serial.println("Edge Impulse Inferencing ");
    if (ei_camera_init() == false) {
        ei_printf("Failed to initialize Camera!\r\n");
    }
    else {
        ei_printf("Camera initialized\r\n");
    }

    WiFi.begin(ssid, password);

    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }

    Serial.println("");
    Serial.println("Wi-Fi connected successfully!");
    Serial.print("IP Address: ");
    Serial.println(WiFi.localIP());
    client.setInsecure();
    ei_printf("\nStarting continious inference in 7 seconds...\n");
    ei_sleep(7000);

}
```

Serial Communication Setup:

We set up serial communication at a baud rate of 115200 to facilitate debugging. After ensuring the serial port is ready, we print "Edge Impulse Inferencing" to the serial monitor.

Camera Initialization:

We attempt to initialize the camera. If successful, we print a message indicating successful initialization; otherwise, we print an error message.

Wi-Fi Connection:

We connect to the Wi-Fi network using provided credentials. We wait until the connection is established, periodically indicating progress with dots. Once connected, we print the device's IP address.

Configuration and Inference Start:

We configure the client for an insecure connection, then signal the start of continuous inference after a 7-second delay.

This setup function initializes serial communication, camera, and Wi-Fi connection, preparing for continuous inference.

```
WiFiClientSecure client;
void sendData(String hand_sign, float accuracy) {
  Serial.println("==========");
  Serial.print("connecting to ");
  Serial.println(host);
  if (!client.connect(host, httpsPort)) {
    Serial.println("connection failed");
    return;
  }else{Serial.println("connection goody good");}
  String hand =  String(hand_sign);
  String percent =  String(accuracy);
  String encodedSign = urlEncode(hand);
  String encodedAccuracy = urlEncode(percent);
  String url = "/macros/s/" + GAS_ID + "/exec?hand_sign=" + encodedSign + "&accuracy=" + encodedAccuracy;
  Serial.print("requesting URL: ");
  Serial.println(url);
  client.print(String("GET ") + url + " HTTP/1.1\r\n" +
               "Host: " + host + "\r\n" +
               "User-Agent: BuildFailureDetectorESP8266\r\n" +
               "Connection: keep-alive\r\n\r\n");
  Serial.println("request sent");
  while (client.connected()) {
    String line = client.readStringUntil('\n');
    Serial.println(line);
    if (line == "\r") {
      Serial.println("headers received");
      break;
    }
  }
  String line = client.readStringUntil('\n');
  if (line.startsWith("{\"state\":\"success\"")) {
    Serial.println("esp successfull!");
  } else {
```

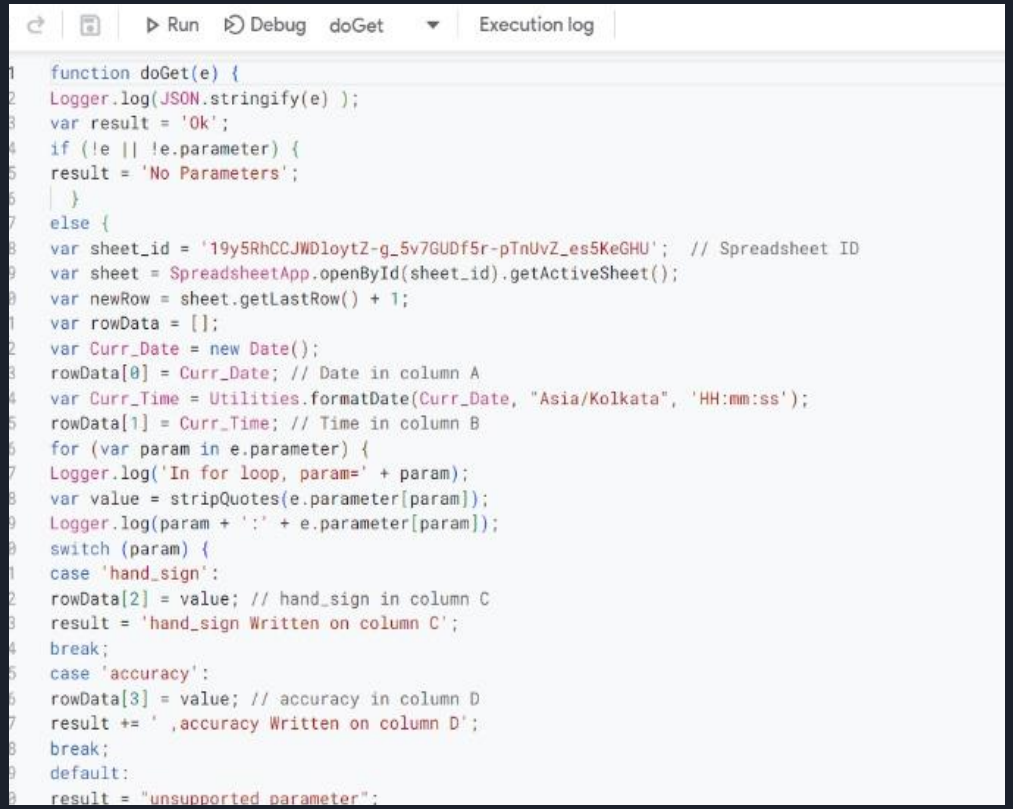
We have a function called sendData designed to send data to the Google Script server. Initially, we attempt to connect to the server specified by the variables host and httpsPort. If the connection fails, we print an error message and exit the function. On successful connection, we construct a URL with parameters hand_sign and accuracy, then send a GET request to the Google Script server. After sending the request, we wait for a response and print it to the serial monitor. If the response indicates success, we print "esp successful!"

```
#if EI_CLASSIFIER_OBJECT_DETECTION == 1
    bool bb_found = result.bounding_boxes[0].value > 0;
    for (size_t ix = 0; ix < result.bounding_boxes_count; ix++) {
        auto bb = result.bounding_boxes[ix];
        if (bb.value == 0) {
            continue;
        }
        ei_printf("    %s (%f) [ x: %u, y: %u, width: %u, height: %u ]\n", bb.label, bb.value, bb.x, bb.y, bb.width, bb.height);
        sendData(bb.label,bb.value*100);

    }
```

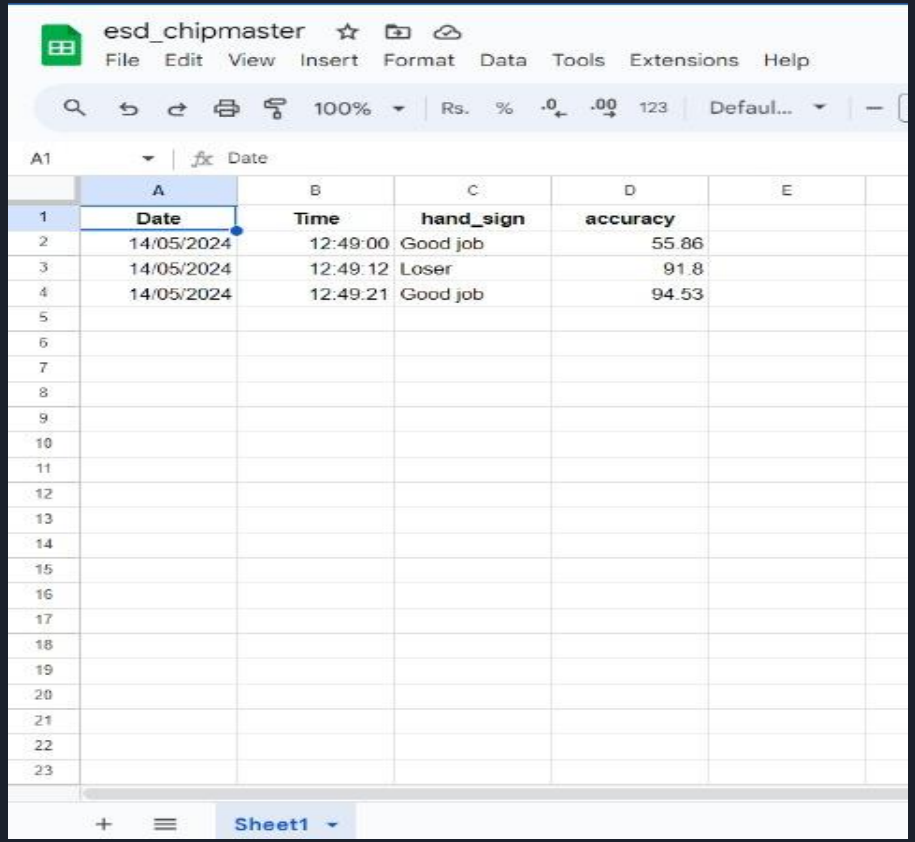
Here, the sendData function is being called to transmit data to the Google Script server. Specifically, it's sending information about the bounding box found during object detection. The data being transmitted includes:

bb.label: The label or class name associated with the detected object.
bb.value * 100: The confidence score of the detected object, scaled to a percentage for better readability.



This Google Apps Script function `doGet(e)` handles incoming GET requests and writes the received data to a Google Sheets spreadsheet.

1. **Logging Incoming Request:** The function first logs the incoming request `e` as a JSON string for debugging purposes.
2. **Handling Parameters:** It checks if there are parameters in the request (`e.parameter`). If there are no parameters, it sets the result to 'No Parameters'.
3. **Accessing Spreadsheet:** It then opens the target Google Sheets spreadsheet using its ID (`sheet_id`) and retrieves the active sheet.
4. **Writing Data to Spreadsheet:** Next, it finds the next empty row in the spreadsheet (`newRow`) and prepares the data to be written. It records the current date and time, then iterates through each parameter received in the request. For each parameter, it extracts the value and writes it to the corresponding column in the spreadsheet.



	A	B	C	D	E
1	Date	Time	hand_sign	accuracy	
2	14/05/2024	12:49:00	Good job	55.86	
3	14/05/2024	12:49:12	Loser	91.8	
4	14/05/2024	12:49:21	Good job	94.53	
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					
21					
22					
23					

The data in this spreadsheet is written by the Google Script. Each row contains information such as the date, time, hand sign, and accuracy, which are being received from the camera module and then processed and written into the spreadsheet using the Google Script function `doGet(e)`.

gui.py 3 X last_processed_row.txt

gui.py > ...

```python
import gspread
from oauth2client.service_account import ServiceAccountCredentials
import pyttsx3
import time
scope = ['https://spreadsheets.google.com/feeds',
         'https://www.googleapis.com/auth/drive']
creds = ServiceAccountCredentials.from_json_keyfile_name('C:/Users/91977/Desktop/ESD_project/esdchipmaster-e6b56107d039.json', scope)
client = gspread.authorize(creds)
# Access the Google Sheet by its ID
sheet_id = '19y5RhCCJWDloytZ-g_5v7GUDf5r-pTnUvZ_es5KeGHU'
sheet = client.open_by_key(sheet_id).sheet1
try:
    with open("last_processed_row.txt", "r") as file:
        last_processed_row1 = int(file.read().strip())
except FileNotFoundError:
    last_processed_row1 = 1

# Convert text to speech
engine = pyttsx3.init()

while True:
    print("running")
    column_c_data = sheet.col_values(3, value_render_option='UNFORMATTED_VALUE')[last_processed_row1 - 1:]

```

This Python script utilizes the gspread library to interact with Google Sheets and the pyttsx3 library to convert text to speech. Here's a breakdown of what it does:

Import Libraries: Import necessary libraries including gspread, oauth2client, pyttsx3, and time.

Authentication: Authenticate with Google Sheets using Service Account Credentials obtained from a JSON keyfile.

Access Google Sheet: Open the Google Sheet by its ID and select the first sheet (`sheet1`).

Retrieve Last Processed Row: Attempt to read the last processed row number from a text file named "last_processed_row.txt". If the file doesn't exist, set the last processed row to 1.

Text-to-Speech Conversion: Continuously loop and check for new data in column C of the Google Sheet, starting from the last processed row. For each non-empty cell in column C, convert the text to speech using pyttsx3 and play it. Update the last processed row index accordingly.

Save Last Processed Row: Write the index of the last processed row to the "last_processed_row.txt" file.

Wait Between Iterations: Pause for 2 seconds before checking for new data again.

This script effectively reads new entries from column C of the Google Sheet and converts them into speech using text-to-speech conversion, providing an auditory output of the data.

PROJECT OUTRO

In conclusion, the development of our sign language recognition and speech synthesis device marks a significant milestone in bridging the communication gap between individuals proficient in sign language and those who rely on spoken language. Through robust algorithms, innovative technologies, and collaborative efforts, we have successfully created a device that promotes inclusivity and accessibility in various facets of life.

As we look ahead, we envision further enhancements and applications for our device, including integration with augmented reality, expansion of language support, and adaptation for specialized domains such as education and healthcare.