Candy Dispenser with AI Handwave Recognition.

Note: This report contains only the parts regarding the programs using the AI handwave recognition interface and the design and training of the Convolutional Neural Network (CNN) used for the handwave classification.

A youtube video that shows the Candy Dispenser and AI Handwave Recognition in action: https://youtube.com/shorts/uVtKizIbAko?feature=share

Abstract

The candy dispenser uses a Convolutional Neural Network (CNN) which does a multi classification on images with the following classes: angle_1, angle_2, angle_3, angle_4, angle_5, angle_6, and noWave. The reasoning is that the motion of a handwave can be broken down into six angles and that identification of a handwave can be made by detecting a certain number of angle classes in their order. For this project, the system was set to determine a handwave was detected when three classes were detected. Further modifications could be made to criteria for determining handwaves such as number of classes identified and taking order into account for example in proper order: angle_1, angle_3, and angle_5 causing handwave identification but the order: angle_3, angle_1, and angle_5 doesn't cause handwave identification. The project as of the writing of this report can be seen as a good prototype and starting point for a future more improved model and system.

Overview of how it works.

The computer's camera is on and takes in 30 frames per second. A program on the computer takes the frames in and the CNN model does a multi-class classification on each frame. If three frames are classified as angle waves in a certain time period then the program sends a message to the Raspberry Pi which activates a motor that dispenses the candy. The Pi then communicates to the computer that the process of dispensing the computer is done and the cycle repeats.

Programs for Control System

There are two programs that operate the control system in this device, the main.py in appendix A and the theDemo.py on appendix B which is the interface between the Jupyter Notebook on the computer and the Raspberry Pi. It is through these two codes that the computer and the Pi communicate with each other. The Jupyter Notebook runs the live time display as seen in figure 1 and runs the model that determines what class a frame is. When the program detects three handwave angles it sends a message to the Pico that a handwave is detected through the code line: pico_serial.write(b'wavedetected\n') as seen in figure 2. The program reads the message and puts it in the variable called line with the code line: line = sys.stdin.readline().strip() . Then the statement checks if the line is the proper statement and activates the motor if it is. The connection process is done via the code block in figure 3 in that the communication is established on COM3 which is what is connecting the computer to PI. The AI model is saved on a .keras file which is loaded as seen in figure 4. A keras file contains the neural architecture and the neuron weights of the AI in a file that can be loaded via code. We configured the program to use TensorFlow Lite which allows our AI to process frames faster by having the calculations that the model does to make its predictions with int values instead of higher bit values like float. This allows the model to keep up with the camera's frames per second(fps) which is 30 frames per second.

Figure 1

```python
# === Handwave Logic ===
unique_angles = set(label for label in pred_history if label in angle_labels)
if len(unique_angles) >= 3:
    i += 1
    print(f"Handwave Detected! (Angles: {sorted(unique_angles)}) → Hello World! {i}")
    pico_serial.write(b'wavedetected\n')
    print("Sent: wavedetected")
```

Figure 2

```python
    line = sys.stdin.readline().strip()

    if line == 'wavedetected':
        run_motor()
        time.sleep(1)
```

Figure 3

```python
import cv2
import numpy as np
import tensorflow as tf
from collections import deque
import os
import serial
import time

# Replace 'COM3' with your actual Pico port if needed
pico_serial = serial.Serial('COM3', 115200, timeout=1)
time.sleep(2)  # Give time for Pico to initialize
```

Figure 4

CNN Design and Training

Note: Given the amount of programs used for the training and development of our CNN model, not all of the programs used for development are in the appendix of this report. Therefore more programs are available in this github link: https://github.com/JosephEsteves/handwaveRecognition which is in appendix D. The architecture for the model can be seen in figure 5. The CNN was trained to classify an image among seven classes: angle_1, angle_2, angle_3, angle_4, angle_5, angle_6, and noWave.

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 128, 128, 3)	0
conv2d (Conv2D)	(None, 128, 128, 32)	896
batch_normalization (BatchNormalization)	(None, 128, 128, 32)	128
conv2d_1 (Conv2D)	(None, 128, 128, 32)	9,248
batch_normalization_1 (BatchNormalization)	(None, 128, 128, 32)	128
max_pooling2d (MaxPooling2D)	(None, 64, 64, 32)	0
conv2d_2 (Conv2D)	(None, 64, 64, 64)	18,496
batch_normalization_2 (BatchNormalization)	(None, 64, 64, 64)	256
conv2d_3 (Conv2D)	(None, 64, 64, 64)	36,928
batch_normalization_3 (BatchNormalization)	(None, 64, 64, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 32, 32, 64)	0
flatten (Flatten)	(None, 65536)	0
dense (Dense)	(None, 128)	8,388,736
batch_normalization_4 (BatchNormalization)	(None, 128)	512
dense_1 (Dense)	(None, 7)	903

Figure 5

Curriculum Training

For training the model, we gradually made the dataset more complex and bigger as the model developed. The levels of difficulties. Can be listed bellow:

- Simple noWave gestures and simple background. Pictures were taken with white wall background. One person doing the waves noWave gestures.
- Different people and more complex environments. More ambiguous gestures.
- Different lighting levels, more complex environments and more ambiguous gestures.
- Motion included in pictures, different lighting levels, more complex environments and even more ambiguous gestures.

Training Parameters

Training parameters were gradually made to be more conservative as the model developed in that we increased the penalty for neural weight changes and decreased the learning rate. The code in appendix C has the latest training parameters as of the writing of this report. For all training sessions, batch = 2 was found to be best at preventing overfitting in live time field testing.

- Adam Optimizer = 0.0001 (learning rate)
- Batch size = 2 (how often the neurons are updated, 2 was set to avoid generalization)
- L2 regularization = 0.01 (bigger number penalizes neural weight changes)

Layer Freezing

Certain layers were frozen at different training sessions to focus more on developing particular layers of neurons. It was found that it improved performance to have the different layers at different layers of development.

Performance and Discussion

The CNN model has around a 90% accuracy and precision for all of the classes with the train, test, and validation test sets. The performance log and confusion matrix were done with our latest trained model and with the latest dataset generated as of the writing of this. The dataset contains a little over 1,400 images for every class in the training dataset. Data augmentation was done 10 times on the reference images to generate for the train, test, and validation dataset splits. From the plots, we notice that the accuracy curves exponentially increase towards a number, and the loss curves exponentially decrease towards a number. These exponential curves indicate that the model is learning instead of just guessing or overly preferring a particular class. In early training of the model, the model was just guessing (predictions were like making a coin toss), or the model would just classify everything as a single class.

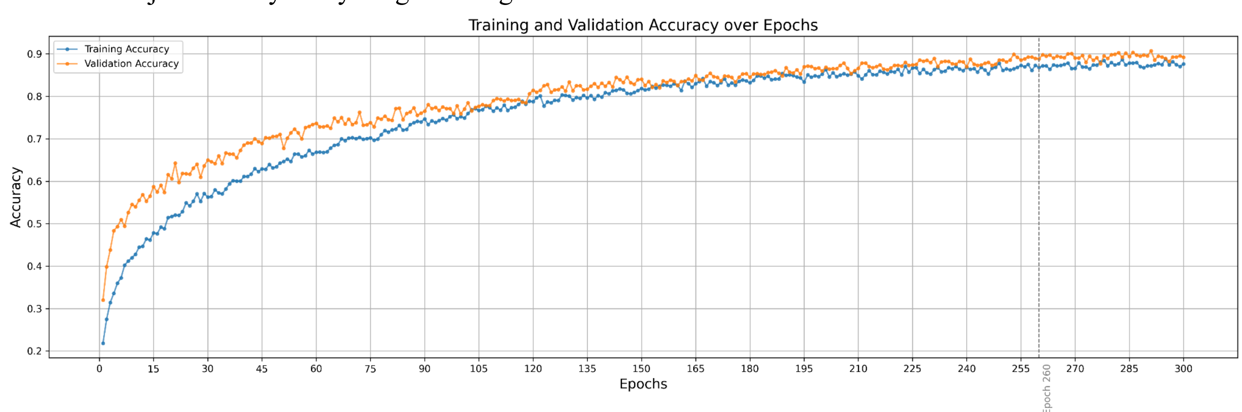

Figure 6

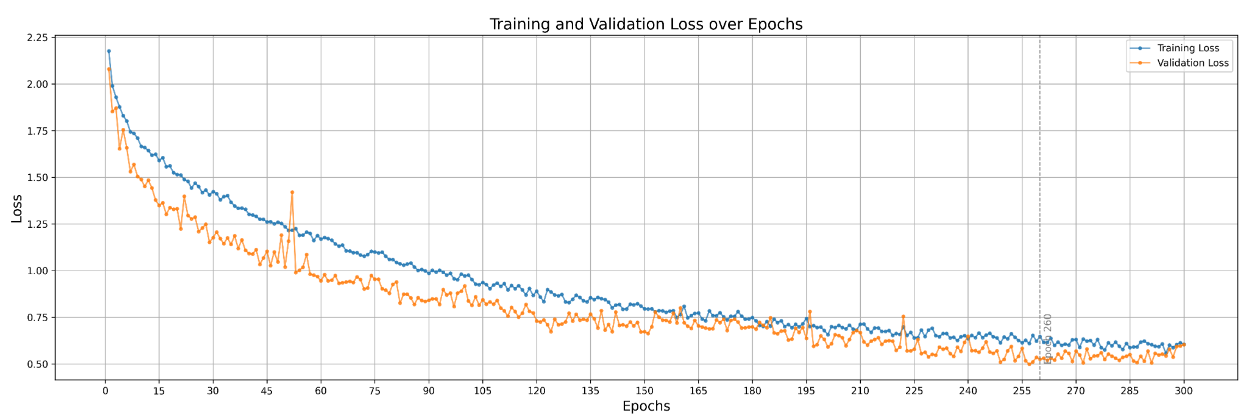

Figure 7

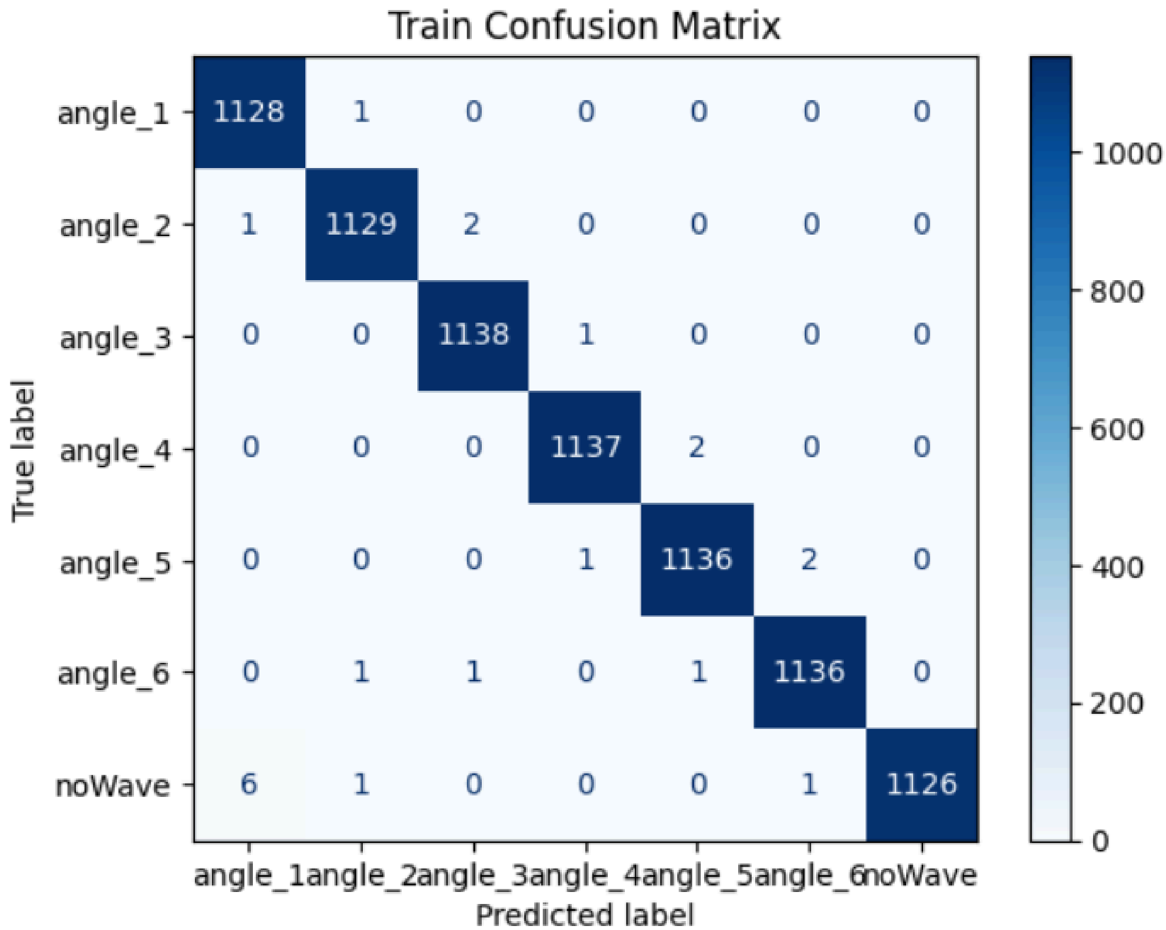
Train Confusion Matrix
True label
angle_1
angle_2
angle_3
angle_4
angle_5
angle_6
noWave
1128 1 0 0 0 0 0
1 1129 2 0 0 0 0
0 0 1138 1 0 0 0
0 0 0 1137 2 0 0
0 0 0 1 1136 2 0
0 1 1 0 1 1136 0
6 1 0 0 0 1 1126
Predicted label
1000
800
600
400
200
0

Figure 8

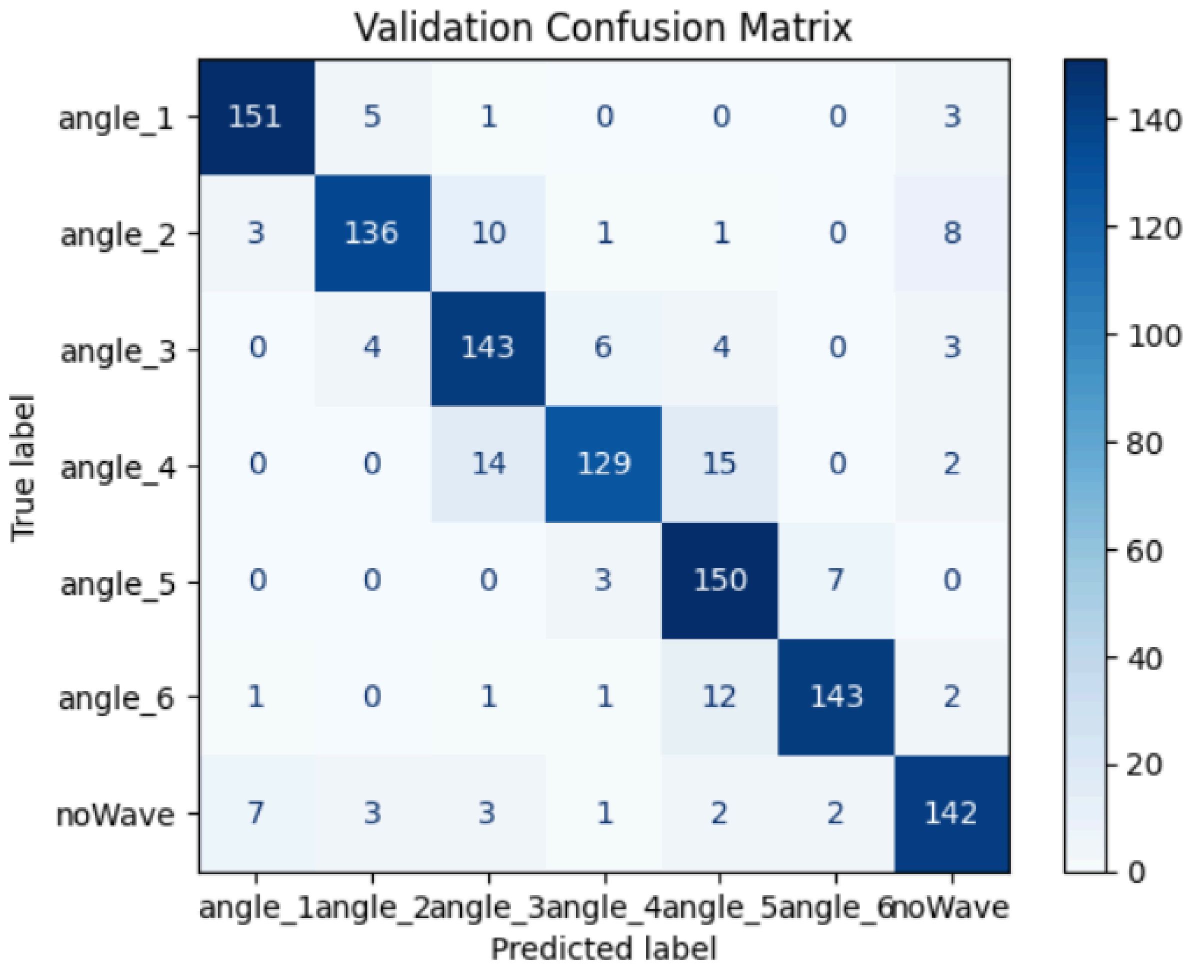
Validation Confusion Matrix
True label
angle_1
angle_2
angle_3
angle_4
angle_5
angle_6
noWave
151 5 1 0 0 0 3
3 136 10 1 1 0 8
0 4 143 6 4 0 3
0 0 14 129 15 0 2
0 0 0 3 150 7 0
1 0 1 1 12 143 2
7 3 3 1 2 2 142
angle_1 angle_2 angle_3 angle_4 angle_5 angle_6 noWave
Predicted label
0
20
40
60
80
100
120
140

Figure 9

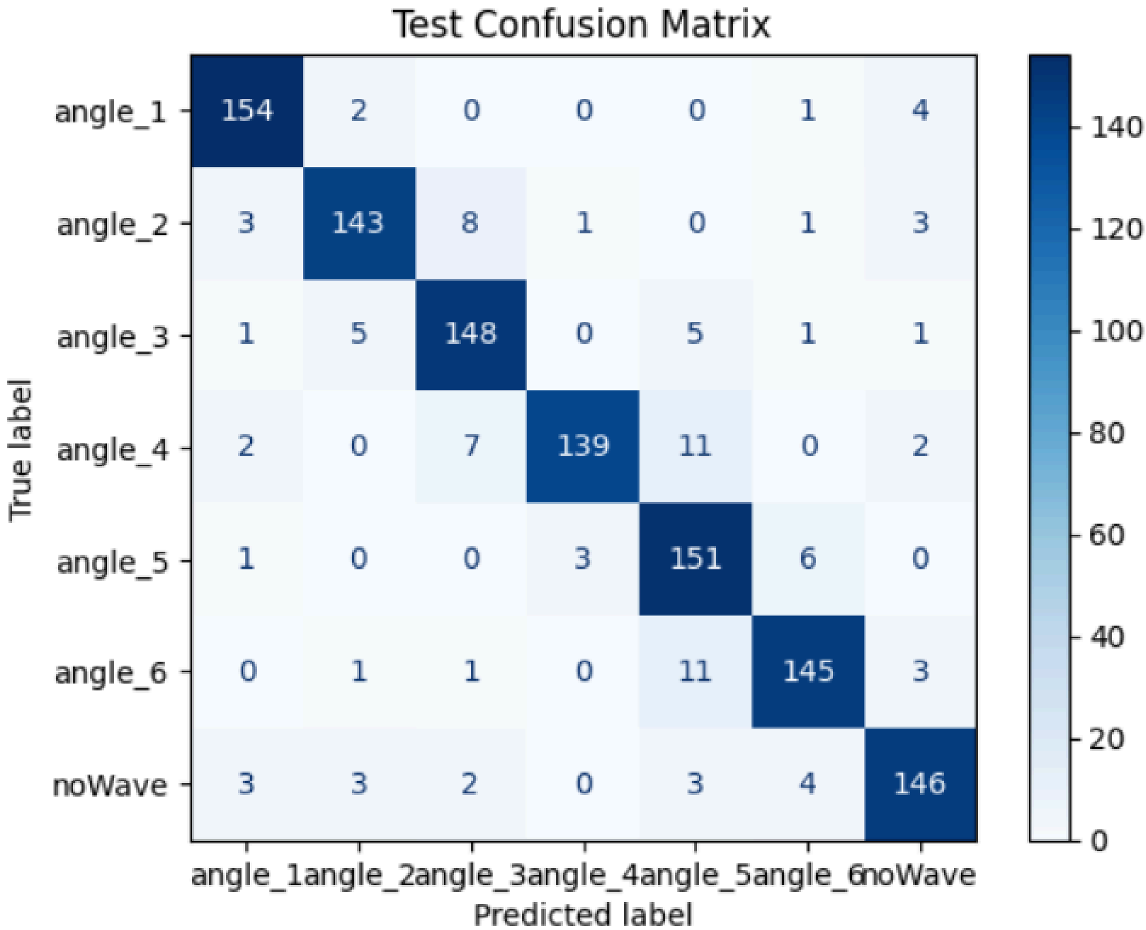

Figure 10

Conclusion

The curves in the figures indicate that the model is learning, but overfitting is a different story. Although validation, training, and test accuracy and precision are good, the performance varies drastically when doing the live time implementation. Which is why field testing is done with the live time display through the code in the appendix. The variety of performance in real-time is due to the fact that we are working with a small dataset and there are several various lighting environments. The biggest factor that we have noticed to have an effect on the model through field testing is the lighting environment, whether the light is natural, fluorescent, yellowish, and how uniform it is. The model's performance is best in darker environments, non-uniform lighting, and with natural or fluorescent light. A weakness with previous models and, less so with the current model, is that the model doesn't perform as well in uniform lighting environments. This can be accomplished in a small room, with only light from a ceiling overhead, and just a white wall background. This weakness is due to the fact that the model uses shadows to help detect angle waves, and the edges of the handwaves are harder to see in uniform lighting. This explains why the model works well in the conditions listed earlier, in that either we have more shadows in darker environments and more light scatter and variation in bigger and more complex environments, such as big rooms and outside. This project has shown us that for real-world applications of AI, we can't rely on performance metrics from the train, test, and validation datasets, and that field testing must be done to truly see how well the model performs.. Further work can be done by expanding the dataset and continuing to experiment with the training parameters, modifying the CNN architecture with add-ons such as L2, and freezing certain layers at certain points in training.

Appendix A: Code on the main.py inside the PI

```
###########################################################################################
from machine import Pin,Timer, ADC
import time
import sys


# Pin configuration
step_pin = Pin(18, Pin.OUT)  # Pin for STEP signal
dir_pin = Pin(19, Pin.OUT)   # Pin for DIR signal
ldr_pin = machine.ADC(28)  #  connect the voltage divider here
threshold_low = 1.5   # Adjust based on calibration (empty)
threshold_high = 1.8  # Adjust based on calibration (full)

# Motor characteristics
steps_per_revolution = 200   # Common for NEMA17 motors
pulse_delay = 10000          # Delay between steps in microseconds (100 Hz)
num_revolutions = 10         # Total revolutions per cycle

status = "unknown"
last_reported_status = None

print("Pico LDR sensor ready.")

def get_candy_status(voltage):
    if voltage < threshold_low:
        return "full"
    elif voltage > threshold_high:
        return "empty"
    else:
        return "initializing"

def rotate_motor(steps_count, direction, delay_time):
    """
    Function to rotate the motor in a specified direction.
    Arguments:
        steps_count: The total number of steps for the motor to take (int).
        direction: The direction of rotation (0 = counter-clockwise, 1 = clockwise).
        delay_time: Time delay between pulses in microseconds (int).
    """
    dir_pin.value(direction)   # Set direction (clockwise or counter-clockwise)
    time.sleep_us(1)           # Ensure proper setup time (more than 200ns)

    for _ in range(steps_count):
        step_pin.value(1)      # Generate a STEP pulse (high)
        time.sleep_us(1)       # Duration for high pulse (at least 1 Âµs)
        step_pin.value(0)      # STEP pulse goes low
        time.sleep_us(delay_time - 1)  # Adjust the delay to maintain correct frequency

def run_motor():
    """
    Main loop to rotate the motor forward and backward.
```

```
    """
    total_steps = steps_per_revolution * num_revolutions  # Calculate total steps for 10 revolutions
    print("Rotating clockwise for 10 revolutions...")
    rotate_motor(total_steps, 1, pulse_delay)
    print("Rotating counter-clockwise for 10 revolutions...")
    rotate_motor(total_steps, 0, pulse_delay)


while True:

    # === Read from LDR ===
    adc_val = ldr_pin.read_u16()  # 16-bit value (0–65535)
    time.sleep(1)
    voltage = (adc_val/65535)*3.3
    status = get_candy_status(voltage)

    # === Send status only if it changed ===
    if status != last_reported_status:
        print(f"status:{status}")
        last_reported_status = status

    line = sys.stdin.readline().strip()

    if line == 'wavedetected':
        run_motor()
        time.sleep(1)
##########################################################################################
```

Appendix B: Code used to interface tensorflow lite to do live time handwave detection and communicate with PI.

```
##########################################################################################
import cv2
import numpy as np
import tensorflow as tf
from collections import deque
import os
import serial
import time

# Replace 'COM3' with your actual Pico port if needed
pico_serial = serial.Serial('COM3', 115200, timeout=1)
time.sleep(2)  # Give time for Pico to initialize

# === CONFIG ===
keras_model_path = "P2gesture_Epoch60.keras"
tflite_model_path = "P2gesture_Epoch60.keras.keras.tflite"
frame_window = 15
image_size = (128, 128)
angle_labels = [f"angle_{i}" for i in range(1, 7)]
no_wave_label = "noWave"
all_labels = angle_labels + [no_wave_label]
```

```
# === TFLite Conversion (INT8) ===
if not os.path.exists(tflite_model_path):
    print("Converting Keras model to INT8 TFLite...")

    model = tf.keras.models.load_model(keras_model_path)

    def representative_data_gen():
        for _ in range(100):
            dummy_img = np.random.rand(*image_size, 3).astype(np.float32)
            dummy_img = np.expand_dims(dummy_img, axis=0)
            yield [dummy_img]

    converter = tf.lite.TFLiteConverter.from_keras_model(model)
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    converter.representative_dataset = representative_data_gen
    converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
    converter.inference_input_type = tf.uint8
    converter.inference_output_type = tf.uint8

    tflite_model = converter.convert()
    with open(tflite_model_path, "wb") as f:
        f.write(tflite_model)

    print(f" Saved quantized model to: {tflite_model_path}")
else:
    print(f" Found existing quantized model: {tflite_model_path}")

# === Load INT8 Model ===
interpreter = tf.lite.Interpreter(model_path=tflite_model_path)
interpreter.allocate_tensors()
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# === Camera and History ===
pred_history = deque(maxlen=frame_window)
cap = cv2.VideoCapture(0)
i = 0

# === Track candy level ===
candy_status = "Initializing"

def read_candy_status():
    global candy_status
    if pico_serial.in_waiting:
        try:
            line = pico_serial.readline().decode().strip()
            if line.startswith("status:"):
                candy_status = line.split(":")[1].capitalize()
        except Exception as e:
            print("Error reading Pico:", e)

read_candy_status()
print("INT8 Handwave + Candy Level Detector running... Press Q to quit.")
```

```
while cap.isOpened():
    read_candy_status()
    ret, frame = cap.read()
    if not ret:
        break

    # === Preprocess frame ===
    resized = cv2.resize(frame, image_size)
    input_scale, input_zero_point = input_details[0]['quantization']
    input_tensor = resized.astype(np.float32) / 255.0
    input_tensor = input_tensor / input_scale + input_zero_point
    input_tensor = np.clip(input_tensor, 0, 255).astype(np.uint8)
    input_tensor = np.expand_dims(input_tensor, axis=0)

    # === TFLite Inference ===
    interpreter.set_tensor(input_details[0]['index'], input_tensor)
    interpreter.invoke()
    output = interpreter.get_tensor(output_details[0]['index'])

    pred_label = all_labels[np.argmax(output)]
    pred_history.append(pred_label)


     # === UI overlay ===
    cv2.putText(frame, f"Prediction: {pred_label}", (10, 30),
                cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
    cv2.putText(frame, f"Candy: {candy_status}", (10, 60),
                cv2.FONT_HERSHEY_SIMPLEX, 0.9, (255, 0, 0), 2)
    cv2.imshow('Handwave + Candy Status', frame)
    # === Handwave Logic ===
    unique_angles = set(label for label in pred_history if label in angle_labels)
    if len(unique_angles) >= 3:
        i += 1
        print(f"Handwave Detected! (Angles: {sorted(unique_angles)}) → Hello World! {i}")
        pico_serial.write(b'wavedetected\n')
        print("Sent: wavedetected")
        read_candy_status()

        # Optional reply from Pico
        reply = pico_serial.readline().decode().strip()
        print("Pico replied:", reply)
    read_candy_status()

    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
###########################################################################################
```

Appendix C Transfer Learning

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D, Dense, Dropout
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import os
import json
import numpy as np
from tensorflow.keras.models import load_model
import cv2

from google.colab import drive
drive.mount('/content/drive')

# === Settings ===
image_size = (128, 128)
batch_size = 2
epochs = 100
num_classes = 7
dataset_dir =  '/content/drive/MyDrive/theBigDataset'

# === Load datasets ===
train_datagen = ImageDataGenerator(rescale=1.0/255)
val_datagen   = ImageDataGenerator(rescale=1.0/255)
test_datagen  = ImageDataGenerator(rescale=1.0/255)

train_gen = train_datagen.flow_from_directory(
    os.path.join(dataset_dir, "Train"),
    target_size=image_size,
    batch_size=batch_size,
    class_mode="categorical"  # since you're using softmax and 7 classes
)

print(train_gen.class_indices)
# {'angle_1': 0, 'angle_2': 1, ..., 'angle_6': 5, 'noWave': 6}

from collections import Counter

# === Step 1: Compute class counts
counter = Counter(train_gen.classes)
total = sum(counter.values())

# === Step 2: Initial inverse-frequency weighting
class_weight = {i: total / (len(counter) * count) for i, count in counter.items()}

# === Step 3: Boost wave classes
wave_boost_factor = 1.0
wave_class_indices = [0, 1, 2, 3, 4, 5]  # angle_1 to angle_6
for i in wave_class_indices:
    class_weight[i] *= wave_boost_factor

print(" Final class weights:", class_weight)
```

```
val_gen = val_datagen.flow_from_directory(
    os.path.join(dataset_dir, "Validation"),
    target_size=image_size,
    batch_size=batch_size,
    class_mode="categorical"
)

test_gen = test_datagen.flow_from_directory(
    os.path.join(dataset_dir, "Test"),
    target_size=image_size,
    batch_size=batch_size,
    class_mode="categorical",
    shuffle=False  # keep test order stable
)
```

```
class FullDatasetPredictionLogger(tf.keras.callbacks.Callback):
    def __init__(self, train_gen, val_gen, test_gen, log_dir="PredictionLogs"):
        super().__init__()
        self.train_gen = train_gen
        self.val_gen = val_gen
        self.test_gen = test_gen
        self.log_dir = log_dir
        os.makedirs(log_dir, exist_ok=True)

        self.logs = {
            "Train": [],
            "Validation": [],
            "Test": []
        }

    def _log_predictions_for_dataset(self, dataset_name, generator, epoch):
        # Predict for the entire dataset
        probs = self.model.predict(generator, verbose=0)
        true_labels = np.argmax(np.vstack([generator[i][1] for i in range(len(generator))]), axis=1)

        self.logs[dataset_name].append({
            "epoch": epoch + 1,
            "predictions": [
                {
                    "true_label": int(true_labels[i]),
                    "probs": probs[i].tolist()
                }
                for i in range(len(probs))
            ]
        })
```

```
        # Save to disk
        with open(os.path.join(self.log_dir, f"{dataset_name}_predictions.json"), 'w') as f:
            json.dump(self.logs[dataset_name], f, indent=2)

    def on_epoch_end(self, epoch, logs=None):
        self._log_predictions_for_dataset("Train", self.train_gen, epoch)
        self._log_predictions_for_dataset("Validation", self.val_gen, epoch)
        self._log_predictions_for_dataset("Test", self.test_gen, epoch)

from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping, Callback

es = EarlyStopping(monitor = "val_accuracy", min_delta = 0.01, patience = 10, verbose = 1)
class CustomModelCheckpoint(Callback):
    def __init__(self, filepath, save_freq):
        super(CustomModelCheckpoint, self).__init__()
        self.filepath = filepath
        self.save_freq = save_freq
    def on_epoch_end(self, epoch, logs=None):
        if (epoch + 1) % self.save_freq == 0:  # Save on specific iterations (1-indexed)
            self.model.save(self.filepath.format(epoch=epoch + 1))

model_cp = ModelCheckpoint(filepath =
'/content/drive/MyDrive/P2handwaveExperiments/P2gesture_Best.keras', monitor = "val_accuracy",
                           save_best_only = True,
                           save_weights_only = False,
                           verbose = 1)
# Define your custom checkpoint for specific iterations
specific_iteration_cp =
CustomModelCheckpoint(filepath='/content/drive/MyDrive/P2handwaveExperiments/P2gesture_Epoch{epoch
:02d}.keras',
                                      save_freq=1)

class HistorySaver(tf.keras.callbacks.Callback):
    def __init__(self, file_path):
        super().__init__()
        self.file_path = file_path
        self.history = []
        self.last_epoch = 0

        # Load existing history if file exists
        if os.path.exists(self.file_path):
            with open(self.file_path, 'r') as file:
                data = json.load(file)
                self.history = data.get('history', [])
                self.last_epoch = data.get('last_epoch', 0)

class SaveHistoryAnd90Acc(Callback):
    def __init__(self, history_path='training_history.json', save_path='gesture_90acc.keras',
acc_threshold=0.90):
        super().__init__()
        self.history_path = history_path
        self.save_path = save_path
        self.acc_threshold = acc_threshold
```

```
        self.history = []
        self.last_epoch = 0
        self.saved = False  # Tracks if model was already saved at threshold

        # Load existing history if it exists
        if os.path.exists(self.history_path):
            with open(self.history_path, 'r') as file:
                data = json.load(file)
                self.history = data.get('history', [])
                self.last_epoch = data.get('last_epoch', 0)

    def on_epoch_end(self, epoch, logs=None):
        # Save training history
        self.history.append({**logs, 'epoch': epoch + 1})
        with open(self.history_path, 'w') as file:
            json.dump({
                'history': self.history,
                'last_epoch': epoch + 1
            }, file, indent=4)

        # Check for 90% training accuracy
        acc = logs.get('accuracy')
        if acc is not None and not self.saved and acc >= self.acc_threshold:
            self.model.save(self.save_path)
            print(f"\n Saved model at epoch {epoch+1} (Training Accuracy: {acc:.4f}) → {self.save_path}")
            self.saved = True


class SaveHistoryAnd80Acc(Callback):
    def __init__(self, history_path='training_history.json', save_path='gesture_80acc.keras',
acc_threshold=0.80):
        super().__init__()
        self.history_path = history_path
        self.save_path = save_path
        self.acc_threshold = acc_threshold
        self.history = []
        self.last_epoch = 0
        self.saved = False  # Tracks if model was already saved at threshold

        # Load existing history if it exists
        if os.path.exists(self.history_path):
            with open(self.history_path, 'r') as file:
                data = json.load(file)
                self.history = data.get('history', [])
                self.last_epoch = data.get('last_epoch', 0)

    def on_epoch_end(self, epoch, logs=None):
        # Save training history
        self.history.append({**logs, 'epoch': epoch + 1})
        with open(self.history_path, 'w') as file:
            json.dump({
                'history': self.history,
                'last_epoch': epoch + 1
```

```
            }, file, indent=4)

        # Check for 80% training accuracy
        acc = logs.get('accuracy')
        if acc is not None and not self.saved and acc >= self.acc_threshold:
            self.model.save(self.save_path)
            print(f"\n Saved model at epoch {epoch+1} (Training Accuracy: {acc:.4f}) → {self.save_path}")
            self.saved = True

class SaveHistoryAnd80Acc(Callback):
    def __init__(self, history_path='training_history.json', save_path='gesture_80acc.keras',
acc_threshold=0.80):
        super().__init__()
        self.history_path = history_path
        self.save_path = save_path
        self.acc_threshold = acc_threshold
        self.history = []
        self.last_epoch = 0
        self.saved = False  # Tracks if model was already saved at threshold

        # Load existing history if it exists
        if os.path.exists(self.history_path):
            with open(self.history_path, 'r') as file:
                data = json.load(file)
                self.history = data.get('history', [])
                self.last_epoch = data.get('last_epoch', 0)

    def on_epoch_end(self, epoch, logs=None):
        # Save training history
        self.history.append({**logs, 'epoch': epoch + 1})
        with open(self.history_path, 'w') as file:
            json.dump({
                'history': self.history,
                'last_epoch': epoch + 1
            }, file, indent=4)

        # Check for 80% training accuracy
        acc = logs.get('accuracy')
        if acc is not None and not self.saved and acc >= self.acc_threshold:
            self.model.save(self.save_path)
            print(f"\n Saved model at epoch {epoch+1} (Training Accuracy: {acc:.4f}) → {self.save_path}")
            self.saved = True

class SaveHistoryAnd70Acc(Callback):
    def __init__(self, history_path='training_history.json', save_path='gesture_70acc.keras',
acc_threshold=0.70):
        super().__init__()
        self.history_path = history_path
        self.save_path = save_path
        self.acc_threshold = acc_threshold
        self.history = []
        self.last_epoch = 0
        self.saved = False  # Tracks if model was already saved at threshold
```

```
        # Load existing history if it exists
        if os.path.exists(self.history_path):
            with open(self.history_path, 'r') as file:
                data = json.load(file)
                self.history = data.get('history', [])
                self.last_epoch = data.get('last_epoch', 0)

    def on_epoch_end(self, epoch, logs=None):
        # Save training history
        self.history.append({**logs, 'epoch': epoch + 1})
        with open(self.history_path, 'w') as file:
            json.dump({
                'history': self.history,
                'last_epoch': epoch + 1
            }, file, indent=4)

        # Check for 70% training accuracy
        acc = logs.get('accuracy')
        if acc is not None and not self.saved and acc >= self.acc_threshold:
            self.model.save(self.save_path)
            print(f"\n Saved model at epoch {epoch+1} (Training Accuracy: {acc:.4f}) → {self.save_path}")
            self.saved = True

combined_callback = SaveHistoryAnd90Acc(
    history_path='/content/drive/MyDrive/PhandwaveExperiments/PtrainingHistory.json',
    save_path='/content/drive/MyDrive/P2handwaveExperiments/P2gesture_90acc.keras',
    acc_threshold=0.90
)

combined_callback1 = SaveHistoryAnd80Acc(
    history_path='/content/drive/MyDrive/PhandwaveExperiments/PtrainingHistory.json',
    save_path='/content/drive/MyDrive/P2handwaveExperiments/P2gesture_80acc.keras',
    acc_threshold=0.80
)

combined_callback2 = SaveHistoryAnd80Acc(
    history_path='/content/drive/MyDrive/PhandwaveExperiments/PtrainingHistory.json',
    save_path='/content/drive/MyDrive/P2handwaveExperiments/P2gesture_70acc.keras',
    acc_threshold=0.70
)
# Usage
file_path = '/content/drive/MyDrive/PhandwaveExperiments/PtrainingHistory.json'
history_saver = HistorySaver(file_path)
# Load the last completed epoch to start from there
start_epoch = history_saver.last_epoch


prediction_logger = FullDatasetPredictionLogger(train_gen, val_gen, test_gen)
callbacks = [model_cp, specific_iteration_cp, history_saver, prediction_logger,combined_callback,
combined_callback1, combined_callback2]

model = load_model( "/content/drive/MyDrive/P1handwaveExperiments/P1gesture_Epoch100.keras")
```

```
model.summary()
# === Compile the model ===
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.00001),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)


model.fit(
    train_gen,
    validation_data=val_gen,
    epochs=epochs,
    class_weight=class_weight,
    callbacks=callbacks,
    verbose=1
)

# === Evaluate on test set ===
test_loss, test_acc = model.evaluate(test_gen, verbose=1)
print(f"\n Final Test Accuracy: {test_acc:.4f}")

# === Save model ===
model.save("gesture_CNN.keras")
print(" Model saved as gesture_CNN.keras")
```

Appendix D GitHub Link for programs used for the project.
https://github.com/JosephEsteves/handwaveRecognition