# Final Lab Report: Embedded AI on Microcontroller

Binary Gesture Classification: The Hand Wave

Tytus Felbor

December 13, 2025

## Contents

# 1 Introduction: Embedded AI in Robotics

In the rapidly growing and constantly evolving field of robotics and IoT, the ability to process data locally known as Embedded/Edge AI is becoming more and more critical. Unlike cloud-based AI, where data is sent to a central server for processing, Embedded AI performs algorithms directly on the microcontroller (MCU). For robotics, this approach offers several distinct advantages:

- **Latency:** Robots often require real-time response times. Waiting for network transmission to the cloud introduces unacceptable delays.
- **Bandwidth Efficiency:** Streaming high-frequency sensor data consumes significant bandwidth. Processing locally sends only the *inference* (the result), dramatically reducing data traffic.
- **Privacy and Security:** Sensitive behavioral data remains on the device.
- **Power Consumption:** Transmitting data via radio (Wi-Fi/LoRa/BLE) often consumes more energy than performing calculations on efficient processors like the Cortex-M4.

In this third and final lab, we utilized the RFThings-AI Dev Kit (STM32L476RGT6) to demonstrate a complete Embedded AI pipeline. The objective was to build a binary classification model capable of detecting a specific human gesture: **a hand wave**. The pipeline involved capturing motion data, training a neural network to distinguish "Waving" from "Not Waving," and deploying the model for real-time inference.

# 2 The Importance of Data Quality

One of the central lessons of this lab is that an AI model is only as good as the data it is trained on. Unlike previous labs where we used curated datasets (MNIST, UCI HAR), here a new challenge was faced - creating our own custom dataset.

## 2.1 Quality vs. Quantity

While having a large quantity of data is important for deep learning, the *quality* is crucial. In the context of accelerometer data:

- **Noise:** The sensor must be calibrated to handle drift.
- **Labeling Accuracy:** Inaccurate labeling (e.g., marking a random movement as a "Wave") confuses the loss function and degrades performance.

## 2.2 Diversity and Generalization (The "False Positive" Challenge)

To create a robust model, the data must represent the diversity of the real world and rigorously define what the gesture is *not*. A significant observation during the testing phase was the issue of similarity in gestures. It was observed that the initial training data lacked sufficient diversity in the "Negative" class regarding opposing movements. Specifically:

- **The Upside-Down Wave:** When the board was inverted and a wave was performed, the accelerometer patterns were similar enough that the model classified it as positive.
- **The Circle Gesture:** Moving the hand in a circular motion generated sinusoidal patterns on the X and Y axes that mimicked the rhythm of a wave, leading to False Positives.

**Analysis of Limitations:** These false positives occurred because the negative class of the training data focused on random movements and idleness, but did not explicitly include these similar gestures which resulted in model's confusion. In the final demo, the model correctly identified standard waves but failed to reject these similar but contrary motions.

**Proposed Solution:** To fix this in a future iteration, one would need to adopt an "Adversarial Data Collection" strategy. This involves specifically performing circles and inverted waves during the data collection phase and explicitly labeling them as **Negative (Class 0)** in the `MicroAIGUI`. This would force the neural network to learn the subtle feature distinctions between a true wave and a circle.

# 3 Course Progression: Lab 1 to Lab 3

The embedded AI course has followed a structured trajectory designed to build understanding layer by layer:

1. **Lab 1 (Environment Setup):** We introduced the toolchain—Docker, TensorFlow, and Arduino IDE—and verified hardware connections.
2. **Lab 2 (Model Architecture & Conversion):** We focused on CNN theory, trained on standard datasets, and learned deployment via manual coding and *MicroAI* tools.
3. **Lab 3 (End-to-End Application):** We applied the full pipeline to a custom binary problem. We generated data for a "Hand Wave," preprocessed it, trained an architecture, and deployed it for real-time gesture detection.

# 4 Lab 3 Breakdown and Implementation

## 4.1 Part I: Data Acquisition

The goal was to collect raw accelerometer data using the STM32 board. We used the `lab4_collect_attached.ino` sketch to stream X, Y, Z acceleration data over the serial port.

For data collection, the board was held in the user's hand. The `serial_client.py` script forwarded this data to the `MicroAIGUI`, where we annotated specific windows:

- **Class 1 (Positive):** A distinct Hand Wave gesture performed while holding the board.
- **Class 0 (Negative):** This included standing still (idleness) as well as various random movements (e.g., walking, typing, or handling the board without waving).

The goal was to obtain a balanced dataset to prevent model bias. The final dataset consisted of:

- **Positive Class (Wave):** 1,328 entries
- **Negative Class (No Wave):** 1,289 entries

This near 50/50 split ensures the model does not favor either class simply due to frequency.

Table 1: Snippet of collected accelerometer data (CSV format).

| T (ms) | Ax | Ay | Az | CLASS |
|---|---|---|---|---|
| 50431.00 | -0.03 | 1.10 | -0.29 | Negative |
| 50482.00 | 0.06 | 1.06 | -0.07 | Negative |
| 50533.00 | -0.03 | 0.98 | -0.10 | Negative |
| | *... (omitted data) ...* | | | |
| 182694.00 | -1.06 | 1.95 | -0.57 | Positive |
| 182744.00 | -1.04 | 2.00 | -1.30 | Positive |

## 4.2 Part II: Model Training (PolyHAR)

A Jupyter Notebook was utilized to process the collected data. The implementation included:

### 4.2.1 Step 1: Windowing

Since accelerometer data is a continuous stream, we sliced the data into windows of size 32. This provides the temporal context necessary for the CNN to detect the rhythmic pattern of a wave.

```
SIZE = 32
CLASSES = 2
windowscount = np.ceil(x_full.shape[0]/SIZE).astype(int)
x_full = np.resize(x_full, (windowscount, SIZE, x_full.shape[-1]))
```

Listing 1: Windowing Code

### 4.2.2 Step 2: Model Architecture

We defined a 1D Convolutional Neural Network. 1D convolutions are ideal for time-series data as they extract features across the time axis.

```
model = Sequential()
model.add(Input(shape=(SIZE, 3)))
model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
model.add(MaxPool1D(pool_size=2))
model.add(Conv1D(filters=32, kernel_size=3, activation='relu'))
model.add(MaxPool1D(pool_size=2))
model.add(Flatten())
model.add(Dense(units=CLASSES))
model.add(Activation('softmax'))
```

Listing 2: CNN Architecture

### 4.2.3 Step 3: Quantization and C-Code Generation

To run efficiently on the Cortex-M4, we converted the model to fixed-point arithmetic using the `kerascnn2c` library. We used a Q7.9 format (16-bit integers), which balances precision and range.

```
res = kerascnn2c.Converter(
    output_path=Path('polyhar_output_fixed'),
    fixed_point=9,          # Q7.9 format
    number_type='int16_t',  # 16-bit quantization
    long_number_type='int32_t',
    number_min=-(2**15),
    number_max=(2**15)-1
).convert_model(copy.deepcopy(model))
```

Listing 3: MicroAI Conversion

## 4.3 Part III: Real-time Prediction (Deployment)

The generated `polyhar_model_fixed.h` file contained the weights and the C-structure of the network. We integrated this header into the `inference_attached.ino` sketch. The implementation logic on the board followed these steps:

1. **Initialization:** Initialize the IMU (ICM 20948).
2. **Buffering:** Fill a rolling buffer with 32 samples of accelerometer data.
3. **Normalization:** Z-score normalization ($X_{norm} = \frac{X-\mu}{\sigma}$) was implemented in C and provided to us in order to match the training data distribution.
4. **Inference:** Upon filling the buffer, the CNN function predicts the class probabilities.
5. **Actuation:** If the output class index is 1 (Positive/Wave), we toggle the LED.

```
    // Run inference
    cnn(inputs, outputs);

    // Get output class
    label = 0;
    if (outputs[1] > outputs[0]) {
        label = 1; // Class 1 represents the "Wave" gesture
    }
    static long inference_count = 0;
    inference_count++;

    // Serial protocol: ID, Count, Label, Probability
    Serial.print("2,");
    Serial.print(inference_count);
    Serial.print(",");
    Serial.print(label);
    Serial.print(",");
    Serial.print(outputs[label]);

    if (label == 1) {
      digitalWrite(PIN_LED, HIGH);
```

```
22        Serial.println("\t|\tWave Detected");
23      } else {
24        digitalWrite(PIN_LED, LOW);
25        Serial.println("\t|\tNo Wave");
26      }
```

Listing 4: Arduino Inference Logic

# 5 Results and Analysis

## 5.1 Training Performance

The model was trained the model for 50 epochs. As seen in the training logs, the model converged with a categorical accuracy of approximately 87% on the training set. The loss curve showed a steady decrease, indicating successful learning without significant overfitting.

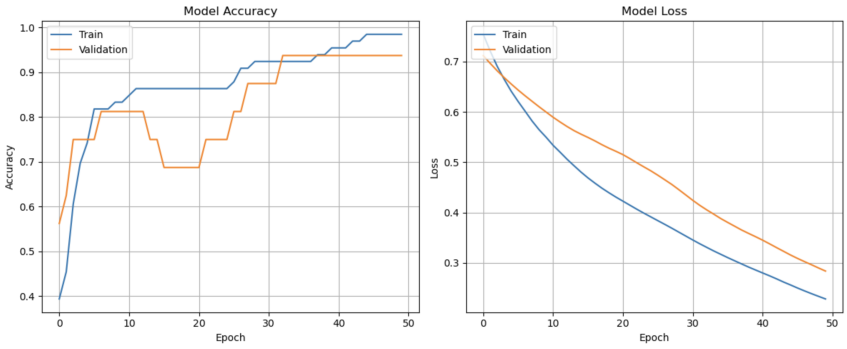


Figure 1: Training Accuracy and Loss curves.

## 5.2 Hardware Inference Observations

Upon deploying the model to the STM32 board, the binary classification performed well for standard wave gestures. The LED successfully triggered when waving the board held in hand.

However, we observed the following limitations during the final demo:

- **False Positives:** The model incorrectly identified circular hand motions and inverted waves as "Class 1 (Wave)."
- **Cause:** The accelerometer signatures for these movements are mathematically similar to a wave, and the "Negative" dataset (which contained random movements) did not contain these specific examples.

Despite these edge cases, the system demonstrated robust detection for the intended gesture and successfully ignored general random motion.

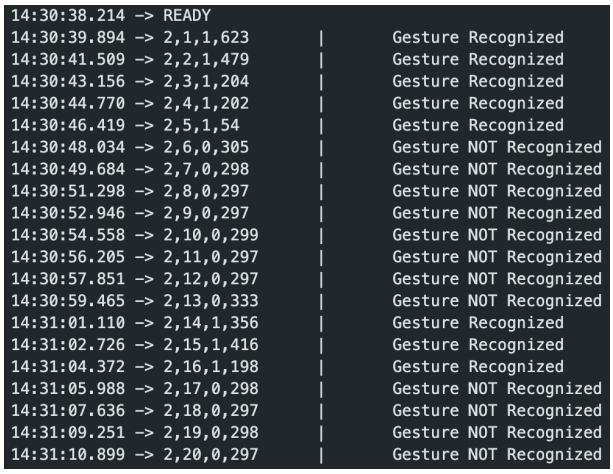


Figure 2: STM32 Board performing real-time inference.

# 6 Conclusion

Lab 3 successfully demonstrated and taught us the end-to-end workflow of Embedded AI. By collecting our own dataset for a binary "Hand Wave" classifier, we gained insight into the difficulties of signal processing and data quality in robotics. We observed that defining what is *not* a gesture is just as important as defining what *is*. We successfully trained a lightweight 1D CNN and converted it to fixed-point C code, highlighting the potential of running sophisticated AI algorithms on resource-constrained microcontrollers.