VCU College of Engineering

Semester Project: Real-Time Hand Grip Classification using ML

By

Yarah Al-Fouleh, Senem Keceli, and Sam Lewis

Date Submitted:

4/29/25

Table of Contents

---

1. Abstract
2. Background and Literature Review
3. Hypothesis
4. Methods
    a. Materials
    b. Software
    c. Data Collection
    d. Model Training
    e. Model Implementation
    f. Testing
5. Results
6. Discussion
    a. Problems Overcome
    b. Future Work
    c. Applications
    d. Contributions
7. References
8. Appendix A
9. Appendix B
10. Appendix C
11. Appendix D

1. Abstract

The objective of this project is to develop a portable device that can classify and predict hand grip types in real time using surface electromyography signals (sEMG). Through the data collection of 6 hand grip types, a machine learning model was trained to accurately classify and predict the muscle activity patterns. The setup includes a single bipolar EMG sensor to collect the electrical signals of the muscle contractions. The model was implemented on a LilyGo ESP32 T-Display, making a compact and wearable, real-time, machine learning-powered EMG device.

2. Background and Literature Review

Muscles in the human body are controlled by electrochemical signalling. The process begins in the brain where a plane of movement is formed, then a corresponding signal or series of signals are sent through the spinal cord. From there the action potential travels through the nerve tissue to the target motor group, where a percentage of the muscle fibers in the group are activated and contract to create movement. In the forearm, there are several different muscle groups that are responsible for controlling the digits of the hand. The groups that manipulate each of the four digits in terms of finger flexion are the flexor digitorium superficialis (FDS) and the flexor digitorum profundus (FDP), respectively controlling them at the proximal interphalangeal and distal interphalangeal joints [2].

Surface electromyography (sEMG) is a non-invasive detection, recording, and interpretation of the electrical activity of muscles at rest and during activity. The muscle activity is detected by using a series of adhesive electrodes placed on the skin above of the target muscle group [1]. The positive and negative leads are placed along the muscle belly and the ground lead is placed on an area with no muscle activity, such as the back of the hand. When an action potential initiates muscle activity, a potential difference is created with intensity based on how much of the muscle group is activated, which can then be read by the reading instrument. Surface electromyography is used as an aid to diagnose neuromuscular disorders, control systems in prosthetics, measurements in physical/occupational therapy, and computer communication [1].

In order to record sEMG signals, there are steps that follow. Electrodes are placed on the skin to detect the muscle contraction signals. The signals are then processed, amplified, and filtered to be digitized and analyzed using microcontrollers or computers [8]. The analysis of these signals are achieved by extracting features [4,11]. These features essentially will describe what is happening in the signal. The two main types of features are time domain features and frequency domain features. The time domain features are important in terms of understanding how the signal behaves over time. A time series based raw signal is taken from the sensor to then be used to extract time domain features [3]. The frequency domain features help understand how often the signal changes. This is especially useful in comparing different types of muscle movement [6].

Machine learning is a very useful way to interpret signals and make predictions after data collection from an sEMG [6]. Machine learning is a subset group of artificial intelligence in which computers/machines are able to imitate the way in which humans learn [7]. A neural network is a subset of machine learning. It is essentially a machine learning program/model that makes decisions and processes information in a similar manner to the human brain. These neutral networks consist of nodes in which each one learns to analyze different parts of the data [8]. There are different types of models that have been used for classification, including SVM, Random Forest, LTSM, and DSVM. [4,5,9] Computations have also been done locally on an attached computer or through remote cloud methods. [10]

In this project, we will be focusing on detecting and classifying six different hand grips: cylindrical, spherical, hook, tip/precision, lateral, and palmar. These six different hand grips represent a range of everyday actions of the hand that are important to assess hand functionality. The neural networks will be trained on the different EMG signals from the different grips, and learn to tell the difference between each one. The project consists of the training of a quantized neural network. This is a neural network that was reduced in order to operate on a microcontroller.

3. Hypothesis

Prior research has demonstrated that both hand position and grip can be classified by ML algorithms, and that digit position can be determined by readings from a single bipolar sEMG. This can be extended to the idea that a singular device with one bipolar electrode can accomplish both of these tasks. [8]

Our hypothesis is as follows: Utilizing machine learning on data collected from a single bipolar EMG paired with embedded software on a portable microcontroller, can accurately predict different types of hand grips with at least 85% accuracy.

4. Methods

a. Materials

The Muscle BioAmp BisCute is a bipolar EMG sensor used to collect the electrical signals produced by muscle contraction. Bipolar EMG refers to having two electrodes placed along the target muscle, alongside a reference electrode. One of the advantages of this device is its compact design, which made it ideal for portability and integration into our setup. The Muscle BioAmp BisCute was delivered as a PCB board containing resistors, capacitors, and a quad op amp.

On the input side of the PCB, there is a network of capacitors and resistors that form a high-pass filter. This section is responsible for reducing low-frequency noise before the signal reaches the amplifier. At the center of the PCB is the quad op amp, four operational amplifiers built into one chip. This component is essentially the core of the board. It performs three main tasks: Amplification of

the small signals from muscle activity, buffering to preserve signal integrity, and filtering to reduce noise. Toward the output side of the board, additional resistors and capacitors form a low-pass filter to remove high-frequency noise. There are also two capacitors located near the power lines that help smooth out the voltage supply to prevent fluctuations that could distort the signal.

The input side of the board includes pins for the inverting positive, inverting negative, and reference electrodes, which connect directly to surface electrodes. The output side provides connections for ground, signal output, and voltage, which allows us to connect the EMG board directly to a microcontroller.

To receive clean muscle contraction signals, we used disposable adhesive surface electrodes combined with conductive electrode gel. The gel was applied to the skin to improve conductivity and ensure a stronger signal during data collection. After applying the gel, the electrodes were placed on the targeted muscle group, specifically the flexor digitorum superficialis, to measure activity during hand grips.

For data collection and model deployment, we used an ESP32 chip programmed through the Arduino IDE. During training data collection, a regular ESP32 board was used. For model implementation, we used the LilyGo ESP32 T-Display, which has an integrated LCD screen. Both microcontrollers are similar in functionality, but we opted for the LilyGo for implementation because its built-in display simplified setup and allowed us to turn the system into a more compact and wearable, real-time, machine learning-powered EMG device.

b. Software

When deciding to use an ESP32, the Arduino IDE was the decision we chose for our data collection process and implementation of the TensorFlow model onto our microcontroller. The Arduino IDE contains the ESP32 board, along with a large amount of libraries that we could use during our implementation process. For training the TensorFlow model, we opted to use Google Colab, as it was a platform that provided familiarity with our team.

c. Data Collection

Data collection was done at the beginning of this project using the ESP32 chip within the Arduino IDE. If we had decided against collecting our own data, we would have to ensure the dataset we used was also collected from the same EMG and ESP32 as us. Additionally, we would have to mimic the electrode placements exactly, which is difficult to verify without images. Due to these constraints, we opted to collect our own training data for our grip classification device. This allowed for consistency during training, implementation, and verification. During our data collection process, we utilized 2 channels (+ and -

inverting) on the FDS). Because we decided to use only one EMG (one muscle group), we had to choose the muscle group wisely. The FDS is the muscle that moves the non-thumb fingers, which allows us to maximize the differences between each grip's raw signal using only one muscle group.

The code utilized for data collection is displayed in Appendix A. The code was written in the Arduino IDE and uploaded to our ESP32 microcontroller. We included a Butterworth Bandpass filter within the code to omit any high or low frequency noise that was not a part of our muscle simulation. This code allows for sampling up to 5,000 data points. After being uploaded to the microcontroller, the test subject held 1 grip at a 45° angle with consistent grip strength from samples 0 - 5,000. Using a sampling frequency of 500 Hz, this process provided a continuous signal of 5,000 points for that singular grip, which was displayed in the serial monitor on the Arduino IDE. Once the cycle was complete, the values were pasted into a labeled Excel spreadsheet. The data collection was completed 6 different times with different objects for the following grips [10]:

1. Cylindrical - Screwdriver
2. Spherical - Ball of crumpled up paper
3. Tip/Precision - End of a dupont wire
4. Palmar - A pen
5. Hook - A backpack
6. Lateral - Pokemon cards

Once data was collected for all 6 grips, we had 6 corresponding Excel files that correlated with each grip. Our team ensured that each dataset had exactly 5,000 samples to avoid bias toward one grip. Each Excel file was titled with the type of grip and converted to a .csv file for processing.

d. Model Training

Due to familiarity with Google Collab, the team made the decision to train our ML model using this platform. The type of model we decided to create was a TensorFlow model. Using a TensorFlow model allowed us to avoid classifications using decision trees. Before deciding, we had trained two separate models: a TensorFlow model and a RandomForest model. Because our model will be implemented into a microcontroller, we remained concerned about memory issues. RandomForest models remained very large in memory due to large decision trees, so we made the decision to use a TensorFlow model. The full TensorFlow model is displayed in Appendix B.

The 6 .csv datasets were first uploaded into the code and put into one list. Putting all of the datasets into 1 list allowed for easier preprocessing of data. After uploading the data into Google Collab, we had to go through and extract time and frequency domain features. Feature extraction is necessary when working with

continuous, raw EMG signals. Raw EMG signals are difficult to classify without features due to the high variability, noise, and lack of structured patterns in the signal. Time domain features are important as they provide information about signal amplitude, duration, and energy, which are key for identifying muscle contractions. Frequency domain features are important as they provide information about muscle degradation over time, fatigue, and overall frequency content of the signal. Without both time and frequency domains, it would be difficult, if not impossible, for the model to correctly classify. When training our model, we used a total of 24 features, which included 12 features for each channel. Obtaining separate features for each channel is important because each channel captures muscle activity from different areas, and combining their unique characteristics improves the model's ability to differentiate between grip types.

After preparing and obtaining our feature values, we used a window size to ensure that we are treating our signal like a continuous signal. Because we collected data using 5,000 continuous points, we have to ensure that we are not treating this signal like individual points. Defining a window size allows us to choose how many individual signals we want within one sample while training. During our model, we opted to use a window size of 50, so 50 continuous signals were grouped into one sample. We defined a sliding window of 10, which allowed us to move through the entire 5,000 row data set in increments of 10 rows at a time. After defining our window and sliding window sizes, we were able to prepare our feature dataset. When building our features dataset, we looped through the full-length dataset and extracted the 24 features for each 50-row window, storing them as a single sample and shifting by 10 rows for the next sample.

The final step of data preparation consists of normalizing the data, so the mean is centered around 0, and the standard deviation is centered around 1. These normalization techniques are used because raw EMG signals can vary greatly. Large variations in numbers may introduce bias to the model, and it could begin to provide higher importance to bigger numbers. Bigger numbers do not mean more importance, and we want to minimize the bias, which is why we normalize our dataset before we begin the training process.

The dataset was then ready to be split into a training and testing set. The dataset was randomly split between training and testing sets, with 80% going into the training set and 20% going into the testing set. The testing set remains unseen from the training set to avoid overfitting. An overfitted model would display accurate results during testing, however, if the model was exposed to new data, it would show inaccurate results. To further avoid overfitting, we added an early stopping mechanism. A telltale sign of an overfit model is having validation loss increase or plateau. Over epochs, the validation loss should continue to decrease.

The early stopping mechanism stopped the code from running whenever the validation loss began to plateau. We utilized class weights to further avoid overfitting or biasing our model. Class weights were utilized to provide a higher reward for grips that may be harder to classify. This avoids the model from only classifying one thing because it's "easy." An optimizer was also utilized to dynamically adjust these class weights as the model runs through.

The final part of our model consisted of building a 3-layer neural network. Neural networks learn patterns in complex datasets by mimicking how the human brain processes and interprets information. To prevent overfitting, we included a dropout mechanism that randomly drops 20% of the neurons during training. This helps the model avoid becoming too reliant on specific pathways and encourages it to learn more general, robust features. Another important component of our neural network was the ReLU (Rectified Linear Unit) activation function, which introduces non-linearity into the model. ReLU allows the network to better capture complex relationships in the data by zeroing out negative values and maintaining positive ones, improving training efficiency and performance.

As previously mentioned, we were concerned about the memory that this model may take up on our microcontroller. Because of this, we quantized our model to use 8-bit integers, reducing both the model size and computational load. This is especially important for the ESP32 because it has limited RAM and flash storage, and it does not have a dedicated floating-point unit (FPU). Floating-point operations require more memory and processing power, which can slow down performance or cause the device to crash. Quantizing the model to integers helps avoid these issues by enabling faster, more efficient computation on hardware like the ESP32.

The quantized model was then exported as a .tflite file. However, .tflite files cannot be directly uploaded to the Arduino IDE. To implement the model onto the LilyGo TTGO board, we had to convert the .tflite file into a .h file, a C-style integer array that can be embedded in the firmware. This conversion was done using the MacBook Terminal with the xxd command. After converting to a .h file, the model size increased significantly, reaching over 200 kB. Despite this jump in size, it was still small enough to be uploaded onto the ESP32-based microcontroller, so the team did not see a need to re-quantize the model a second time.

e. Model Implementation

When doing the implementation process, we first had to upload our board and necessary libraries. The LilyGo TTGO required an older version of the ESP32 board package, specifically version 2.0.9, which was the version used for successful model deployment. As for libraries, we used both the TFT_eSPI library

and the TensorFlowLite_ESP32 library. The TFT_eSPI library was used to activate and display classification results on the LCD screen. However, this library includes a User_Setup file that was entirely commented out by default. Because of this, we manually created a User_Setup file, which can be found in Appendix C.

The TensorFlowLite_ESP32 library was essential for processing our model in real-time on the microcontroller. It allowed us to run the quantized .h model file directly on the ESP32 hardware. All constants that were defined during training were kept consistent throughout the implementation. Keeping consistent values ensures reliable and accurate performance. If the model is not implemented in the same way it was trained, it becomes very difficult for the model to produce valid predictions. Even minor differences in how features are handled can lead to poor performance.

In addition to feature consistency, we had to implement the quantization constants and parameters. While models are typically trained using 32-bit floating-point values, quantized models use only 8-bit integers, which must be defined explicitly in the microcontroller code. These constants and parameters tell the ESP32 how to scale and normalize incoming data in the exact same way the training data was normalized. This ensures that the live data falls within the expected range of the model. The quantization parameters (scale and zero-point) were printed during the training script, while the constants were extracted after the model was quantized.

As previously mentioned, the goal of this implementation was to mimic the training process as closely as possible. To achieve this, we extracted the same exact features from the incoming real-time data as we did during preprocessing. For frequency domain features, we opted to manually calculate the Fast Fourier Transform (FFT) because the ArduinoFFT library is not compatible with our specific microcontroller. We also implemented the Butterworth bandpass filter used during data collection directly into the real-time signal processing pipeline to preserve the same input conditions.

Once the model and features were fully processed, the final step was implementation onto the LilyGo TTGO. One of the most visible outcomes of our real-time classification was the LCD screen display, which was controlled using the TFT_eSPI library. A function called updateDisplay() was written to handle visual output. This function dynamically displays the name of the predicted grip, the model's confidence level as a percentage, and the incoming EMG signal waveform in real time.

The display only updates when there is a significant change, either when the predicted grip changes or the confidence level shifts by more than 5%. This reduces unnecessary screen flickering and keeps the display clean. The top section

of the screen shows the current grip type and confidence score, while the bottom section displays a rolling waveform of the raw EMG signal. To do this, the signal is scaled and mapped to the screen dimensions, allowing the user to visually confirm muscle activity as it happens.

The setup() function initializes the screen and the model. Because the TFT display's backlight is disabled by default, the code first sets up the backlight pin. The display is cleared, text color and size are defined, and a loading message is displayed. If the model fails to load, a static error message is printed.

Inside the loop(), the EMG signal is continuously sampled. After signal filtering, envelope extraction, and buffering, a window of EMG data is used to extract features. These features are then standardized using previously stored mean and scale values, and quantized based on the input scale and zero-point (required due to 8-bit integer quantization). Once the model outputs a prediction, the class with the highest score is identified. If the score exceeds a predefined confidence threshold, the display is updated with the predicted grip name. If not, the label “Uncertain” is shown. If the signal was too weak or noisy to be valid, the display shows “No Signal.” The full implementation code can be found in Appendix D.

f. Testing

When testing our full design, we remained consistent with our electrode placements on the FDS to preserve signal reliability. The test subject was instructed to randomly pick up various objects corresponding to different grip types used during training. This randomized testing approach ensured that the model’s classifications were not simply overfitting to a pattern or order, but instead could generalize across unpredictable grip sequences. To track model performance, we manually tallied the number of accurate classifications during testing. Rather than testing under controlled or repetitive sequences, this method aimed to simulate more realistic, everyday use where the model must identify grips in real time, without prior knowledge of the user’s intentions. By observing when the LCD displayed the correct grip classification, we were able to determine whether the model was genuinely interpreting the EMG signal or making inaccurate guesses.

The randomness of the grips paired with real-time display feedback provided a clear and intuitive way to verify model accuracy without needing to log results externally. If the model confidently predicted the correct grip (above our threshold), it was marked as a success. If it labeled the grip as “Uncertain” or produced an incorrect classification, it was noted accordingly. Over several trials, this method allowed us to get a strong sense of real-world performance and responsiveness.

5. Results

The results from our model training showed high levels of accuracy (Figure 1). There were no signs of overfitting, as seen by the continuous decrease in validation loss throughout the training process (Figure 2). This consistent drop in validation loss across epochs confirmed that the model was learning meaningful patterns rather than memorizing the training data.

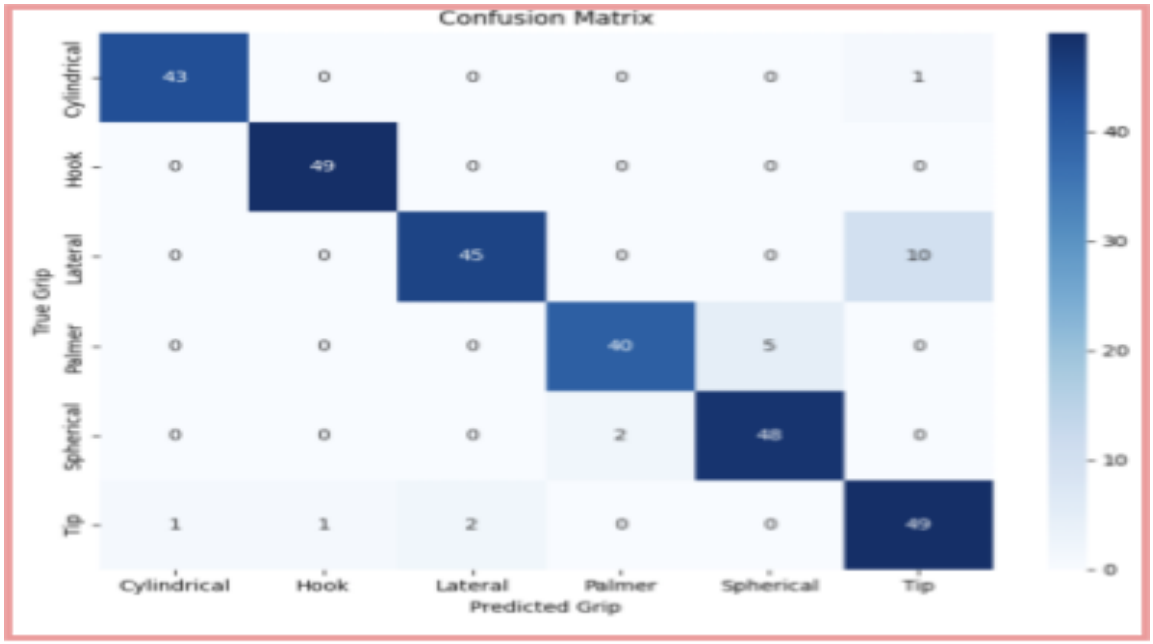

Figure 1. This confusion matrix displays a visual representation of the TensorFlow model's performance.

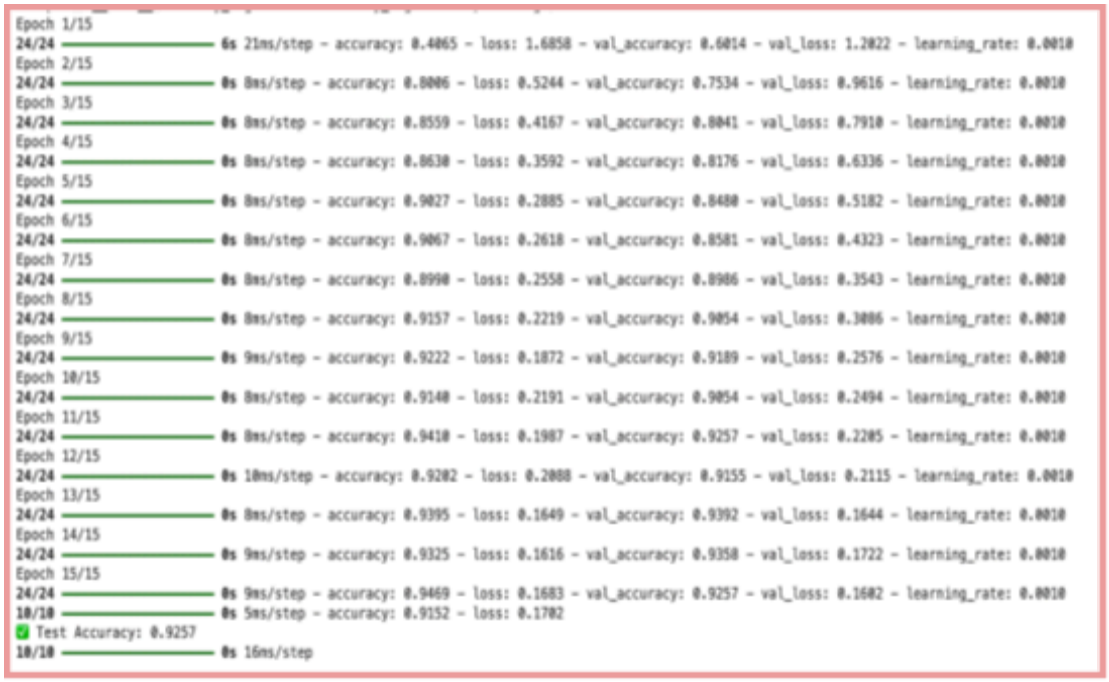

Figure 2. The TensorFlow model running through each epoch. As it goes through more epochs, the accuracy increases, while the loss decreases.

When reviewing these figures, it becomes clear that the model generalizes well. The lack of overfitting gives us confidence that the model can classify grip types accurately and reliably, even when presented with new, unseen data. This was a crucial indicator that the model would be suitable for real-time implementation.

Since the model needed to be quantized before being uploaded to the microcontroller, we also tested the performance of the quantized version to ensure that the accuracy held up. The quantized model continued to perform well and produced highly accurate results (Figure 3), which reassured us that the conversion process didn't significantly impact the model's effectiveness.

Figure 3. The accuracy of our quantized, .tflite model.

When implementing our model into the Arduino IDE, we performed a manual check to ensure that the implemented model was not randomly classifying grips. We found that the model could correctly identify grips such as spherical, cylindrical, palmar and tip but often struggled to differentiate hook and lateral grips between the others.

6. Discussion

a. Problems Overcome

Grip classification using only one EMG sensor has historically never yielded accurate results, which posed a major challenge for us since we intentionally chose to work with just one EMG placed on the flexor digitorum superficialis. Most projects in this space avoid this approach for a reason, but we wanted to explore whether we could still achieve reliable performance with a minimalistic setup.

Another challenge came during the quantization and model conversion phase. When quantizing and downloading the .tflite file, it was crucial that we followed the correct steps to convert it into a .h file using the Terminal. Any mistake in the bashing or conversion process could easily break the implementation. Quantization parameters also became a key issue. Without properly implementing these values in the C++ code, the model will not interpret or scale the input data correctly. These parameters are not optional—they are essential for the model to function as intended during real-time inference.

We also ran into issues with the User_setup.h file in the TFT_eSPI library. By default, the file was completely commented out, and it didn't seem to work with the LilyGo TTGO board for unknown reasons. To fix this, we had to go into the library folder and create our own custom User_setup file tailored to our display settings.

Additionally, we learned that the ArduinoFFT library was extremely finicky and not compatible with our ESP32 board. This forced us to manually calculate the FFT to extract our frequency domain features in real time.

Our very first challenge, though, was the one we knowingly accepted: the decision to use only one EMG channel. We were aware that most similar projects use multiple channels to improve accuracy, but we wanted to test the limits of what could be done with a simpler and more accessible setup.

Since our device was operating independently from a computer or powerful processor, one of our biggest limitations was collecting and processing signals then feeding them into our classification algorithm with limited memory and speed. The ESP32 was a powerful alternative to the Arduino, but further testing showed that the on board memory was not sufficient to support any deeper networks effectively. We overcame this by optimizing our code to use less memory and managing which libraries we used during our code drafting process.

During testing, we also experienced hardware failures; our LM324 Quad Op amp failed and rendered our device unusable during our final optimization process. We were not affected by this due to proper recording of testing and trials

during our project process, and could still validate our hypothesis based on our documentation.

b. Future Work

For the future work of this project we plan to optimize both the hardware and software components to improve the classification accuracy. Our main goal is to improve upon our classification of the hook grip. During data collection, the hook grip was collected with a heavy object which in turn appeared to activate the bicep muscles significantly more than the forearm muscles. This led to variability and inaccuracy in the hook grip classification. To address this issue, we plan to collect data again for the hook grip utilizing a lightweight object that activates the forearm muscles and to better isolate that specific muscle group.

Our project consists of a single bipolar EMG sensor. Most literature typically uses more than 1 EMG to yield accurate results, and there was minimal literature supporting accurate results with only 1 EMG. Successfully receiving accurate results with only 1 EMG is a novel idea and has implications to be a cheap, simple, and effective tool for many different applications. We plan upon expanding this project with the goal of contributing to the existing body of knowledge. We believe that this project has a lot of future potential. We are excited to continue developing and building upon this classification system as it will be useful for various applications, particularly in embedded systems.

c. Applications

Real-time hand grip classification can be utilized in the medical field to assist patients who have undergone finger amputations. In these cases, our device, which classifies hand grips in real time using a TensorFlow model, can be used to collect muscle activity from remaining functional muscle groups in the forearm. These readings can then be implemented into prosthetic control systems or rehabilitation devices to enable movement that reflects the user's intent, improving both motor function and quality of life. By recognizing distinct grip patterns through muscle signals, this system helps bridge the gap between muscle intention and prosthetic action.

In addition to prosthetics, this device has potential applications in diagnosing and monitoring neuromuscular disorders. By analyzing deviations from normal grip patterns or delays in activation, clinicians could use the data to detect early signs of conditions such as ALS, muscular dystrophy, or nerve damage. The continuous and real-time nature of the readings also makes it suitable for long-term tracking of disease progression or rehabilitation outcomes.

In a non-medical context, this technology can be implemented in computer interfacing methods. Creating immersive and natural controls for computer

systems has been a challenge for many who set out to improve the ergonomics of different systems, and there are dozens of specific situations where natural human movement is a superior method of control to keyboard and mouse, or existing expensive VR technology.

In some cases, it's been hypothesized that accurate computer interpretation can be used to translate sign language into text or speech, which would be a large step forward in terms of accessibility.

d. Contributions

Throughout this project, we split up the work based on our strengths. Yarah and Sam have strengths in hardware, while Senem has strengths in coding. However, toward the end of this project it became clear that we needed all of our brains to debug the model implementation code. For the background research, we all did our own research to get a background and agree on a muscle group. To start this project, we were required to solder the EMG components onto the PCB board, which was done by Yarah. Sam and Yarah went back to fix a few soldering errors. We were all a part of the data collection process, with Senem being the test subject. Sam developed the testing method. For the ML model training, Senem did the training, testing and quantization of the TensorFlow model within Google Collab. For the final implementation onto the Arduino IDE, Senem made the base code. However, that code did not work and took a lot of debugging from all of us (plus AI) to help us get a functioning code for our model implementation. For the report, Yarah wrote about the materials, abstract and background. Sam also contributed to the background. Senem wrote about the code, testing and results. Yarah and Senem split up the discussion section evenly.

References

[1] “Error | BlueCross BlueShield of Tennessee,” Bcbst.com, 2017. https://www.bcbst.com/mpmanual/
[2] T. Wall, “Muscles in The Finger,” *JOI Jacksonville Orthopaedic Institute*, Jul. 09, 2020. https://www.joionline.net/library/muscles-in-the-finger/
[3] M. R. Azhar, “Extracting Time-Domain and Frequency-Domain Features from a Signal — Python implementation 1/2,” Medium, Oct. 11, 2023. https://medium.com/@rehanmbl/extracting-time-domain-and-frequency-domain-features-from-a-signal-python-implementation-1-2-d36148c949ba (accessed Apr. 30, 2025).
[4]S. Chatterjee *et al.*, “EMG Signal Acquisition and Processing for Feature Extraction And Detection of Disease.” Accessed: Apr. 30, 2025. [Online]. Available: https://jet-m.com/wp-content/uploads/130-JETM8266.pdf
[5] X. Wang, V. Liesaputra, Z. Liu, Y. Wang, and Z. Huang, “An in-depth survey on Deep Learning-based Motor Imagery Electroencephalogram (EEG) classification,” *Artificial intelligence in medicine*, vol. 147, pp. 102738–102738, Jan. 2024, doi: https://doi.org/10.1016/j.artmed.2023.102738.
[6] F. Di Nardo, A. Nocera, A. Cucchiarelli, S. Fioretti, and C. Morbidoni, “Machine Learning for Detection of Muscular Activity from Surface EMG Signals,” *Sensors*, vol. 22, no. 9, p. 3393, Apr. 2022, doi: https://doi.org/10.3390/s22093393.
[7] IBM, “What Is Machine learning?,” *Ibm.com*, Sep. 22, 2021. https://www.ibm.com/think/topics/machine-learning
[8] IBM, “What is a Neural Network?,” *IBM*, Oct. 06, 2021. https://www.ibm.com/think/topics/neural-networks
[9]X. Bao, Y. Zhou, Y. Wang, J. Zhang, X. Lü, and Z. Wang, “Electrode placement on the forearm for selective stimulation of finger extension/flexion,” *PLOS ONE*, vol. 13, no. 1, p. e0190936, Jan. 2018, doi: https://doi.org/10.1371/journal.pone.0190936.
[10] A. Phinyomark, A. N. Ghaffari, and C. L. Scheme, “sEMG for Basic Hand Movements Data Set,” UCI Machine Learning Repository, 2012. [Online]. Available: https://archive.ics.uci.edu/dataset/313/semg+for+basic+hand+movements
[11] Y. Himath Kumar, J. Kumar, and Poonam Sheoran, “Integration of cloud computing in BCI: A review,” *Biomedical Signal Processing and Control*, vol. 87, pp. 105548–105548, Jan. 2024, doi: https://doi.org/10.1016/j.bspc.2023.105548.

Appendix A: Data Collection Code

```
#define SAMPLE_RATE 500
#define BAUD_RATE 115200
#define INPUT_PIN 33
#define BUFFER_SIZE 128
#define MAX_SAMPLES 5000  // Stop after 5000 samples

int circular_buffer[BUFFER_SIZE];
int data_index = 0;
int sum = 0;
unsigned long sample_count = 0;  // Track number of samples collected
bool collecting = true;          // Flag to stop data collection

void setup() {
  Serial.begin(BAUD_RATE);
  analogReadResolution(12);  // ESP32 supports up to 12 bits (4096 levels)
}

void loop() {
  if (!collecting) return;

  static unsigned long last_sample_time = 0;
  unsigned long now = millis();

  if (now - last_sample_time >= 1000 / SAMPLE_RATE) {
    last_sample_time = now;

    int sensor_value = analogRead(INPUT_PIN);
    int signal = EMGFilter(sensor_value);
    int envelop = getEnvelop(abs(signal));
    Serial.print(signal);
    Serial.print(",");
    Serial.println(envelop);

    sample_count++;
    if (sample_count >= MAX_SAMPLES) {
      collecting = false;
      Serial.println("Data collection complete.");
```

```
    }
  }
}


int getEnvelop(int abs_emg) {
  sum -= circular_buffer[data_index];
  sum += abs_emg;
  circular_buffer[data_index] = abs_emg;
  data_index = (data_index + 1) % BUFFER_SIZE;
  return (sum / BUFFER_SIZE) * 2;
}

// Band-Pass Butterworth IIR filter
float EMGFilter(float input) {
  float output = input;

  {
    static float z1, z2;
    float x = output - 0.05159732 * z1 - 0.36347401 * z2;
    output = 0.01856301 * x + 0.03712602 * z1 + 0.01856301 * z2;
    z2 = z1;
    z1 = x;
  }

  {
    static float z1, z2;
    float x = output - -0.53945795 * z1 - 0.39764934 * z2;
    output = 1.00000000 * x + -2.00000000 * z1 + 1.00000000 * z2;
    z2 = z1;
    z1 = x;
  }

  {
    static float z1, z2;
    float x = output - 0.47319594 * z1 - 0.70744137 * z2;
    output = 1.00000000 * x + 2.00000000 * z1 + 1.00000000 * z2;
    z2 = z1;
    z1 = x;
  }
```

```
 {
  static float z1, z2;
  float x = output - -1.00211112 * z1 - 0.74520226 * z2;
  output = 1.00000000 * x + -2.00000000 * z1 + 1.00000000 * z2;
  z2 = z1;
  z1 = x;
 }

 return output;
}
```

Appendix B: TensorFlow Model Code

```
# Import libraries
from google.colab import drive
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.utils import shuffle, class_weight
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.utils import to_categorical
from scipy.fft import fft, fftfreq

from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf

# Mount your Drive
drive.mount('/content/drive')

# Define grip names and file paths
grip_files = {
   'Tip': '/content/drive/MyDrive/Data/Tip.csv',
```

```
    'Spherical': '/content/drive/MyDrive/Data/Spherical.csv',
    'Palmer': '/content/drive/MyDrive/Data/Palmer.csv',
    'Cylindrical': '/content/drive/MyDrive/Data/Cylindrical.csv',
    'Hook': '/content/drive/MyDrive/Data/Hook.csv',
    'Lateral': '/content/drive/MyDrive/Data/Lateral.csv',
}

# Feature extraction
def extract_features(signal, fs=500):
    features = {}
    features['MAV'] = np.mean(np.abs(signal))
    features['RMS'] = np.sqrt(np.mean(signal**2))
    features['VAR'] = np.var(signal)
    features['IEMG'] = np.sum(np.abs(signal))
    features['ZC'] = ((signal[:-1] * signal[1:]) < 0).sum()
    features['WL'] = np.sum(np.abs(np.diff(signal)))
    diff1 = np.diff(signal)
    diff2 = np.diff(diff1)
    features['SSC'] = np.sum((diff1[:-1] * diff1[1:] < 0) & (np.abs(diff2)
> 0.01))
    features['WAMP'] = np.sum(np.abs(np.diff(signal)) > 0.02)
    features['LOG_D'] = np.exp(np.mean(np.log(np.abs(signal) + 1e-6)))

    # Frequency-domain features
    N = len(signal)
    freqs = fftfreq(N, 1 / fs)
    fft_vals = np.abs(fft(signal))
    positive_freqs = freqs[:N // 2]
    positive_fft = fft_vals[:N // 2]
    total_power = np.sum(positive_fft)
    features['Total_Power'] = total_power
    features['MNF'] = np.sum(positive_freqs * positive_fft) / total_power
    features['MDF'] = positive_freqs[np.where(np.cumsum(positive_fft) >=
total_power / 2)[0][0]]

    return features

# Sliding window parameters
window_size = 50
step_size = 10
```

```
# Build feature dataset
feature_rows = []

for grip_label, file_path in grip_files.items():
   df = pd.read_csv(file_path, header=None)
   ch1 = df.iloc[:, 0].values
   ch2 = df.iloc[:, 1].values

   for i in range(0, len(ch1) - window_size + 1, step_size):
       window1 = ch1[i:i + window_size]
       window2 = ch2[i:i + window_size]

       feat1 = extract_features(window1)
       feat2 = extract_features(window2)

       combined = {f'EMG1_{k}': v for k, v in feat1.items()}
       combined.update({f'EMG2_{k}': v for k, v in feat2.items()})
       combined['Grip'] = grip_label
       feature_rows.append(combined)

# Convert to dataframe
features_df = pd.DataFrame(feature_rows)

# Encode grip labels
label_encoder = LabelEncoder()
y_labels = label_encoder.fit_transform(features_df['Grip'])
y = to_categorical(y_labels, num_classes=6)

# Prepare features
X = features_df.drop(columns='Grip').values
X, y = shuffle(X, y, random_state=42)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.2, random_state=42)

# Compute class weights
```

```
class_weights = class_weight.compute_class_weight('balanced',
classes=np.unique(y_labels), y=y_labels)
class_weights_dict = {i: class_weights[i] for i in
range(len(class_weights))}

# Build model
model = Sequential([
   Dense(200, activation='relu', input_dim=X_train.shape[1]),
   BatchNormalization(),
   Dropout(0.2),

   Dense(80, activation='relu'),
   BatchNormalization(),
   Dropout(0.2),

   Dense(64, activation='relu'),
   BatchNormalization(),
   Dropout(0.2),

   Dense(6, activation='softmax')
])

# Compile
model.compile(optimizer=Adam(learning_rate=0.001),
loss='categorical_crossentropy', metrics=['accuracy'])

# Callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5,
min_lr=1e-4)

# Train
history = model.fit(
   X_train, y_train,
   validation_data=(X_test, y_test),
   epochs=15,
   batch_size=50,
   class_weight=class_weights_dict,
   callbacks=[early_stopping, reduce_lr]
```

```
)

# Predict labels
y_pred_probs = model.predict(X_test)
y_pred = np.argmax(y_pred_probs, axis=1)
y_true = np.argmax(y_test, axis=1)


# Evaluate
loss, acc = model.evaluate(X_test, y_test)
print(f"✅ Test Accuracy: {acc:.4f}")


# Convert to TFLite with full integer quantization
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.target_spec.supported_ops =
[tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8

# Representative dataset generator
def representative_dataset():
   for i in range(min(100, len(X_train))):
       yield [X_train[i:i+1].astype(np.float32)]

converter.representative_dataset = representative_dataset
tflite_quant_model = converter.convert()

# Save the quantized model
with open("grip_model_quant.tflite", "wb") as f:
   f.write(tflite_quant_model)


from google.colab import files
files.download("grip_model_quant.tflite")

# Load and evaluate quantized model
# Load and evaluate quantized model
interpreter = tf.lite.Interpreter(model_path="grip_model_quant.tflite")
```

```
interpreter.allocate_tensors()

input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Get quantization parameters
input_scale = input_details[0]['quantization'][0]
input_zero_point = input_details[0]['quantization'][1]

# Evaluate on test data
correct = 0
total = len(X_test)

print("\nQuantization parameters:")
print(f"Input scale: {input_scale}")
print(f"Input zero point: {input_zero_point}")

for i in range(total):
   # Quantize the input data
   input_data = X_test[i].astype(np.float32)
   input_data_quantized = np.array(input_data / input_scale +
input_zero_point, dtype=np.int8)
   input_data_quantized = np.expand_dims(input_data_quantized, axis=0)

   interpreter.set_tensor(input_details[0]['index'], input_data_quantized)
   interpreter.invoke()
   output_data = interpreter.get_tensor(output_details[0]['index'])

   # Dequantize the output
   output_scale = output_details[0]['quantization'][0]
   output_zero_point = output_details[0]['quantization'][1]
   output_data_dequantized = (output_data.astype(np.float32) -
output_zero_point) * output_scale

   if np.argmax(output_data_dequantized) == np.argmax(y_test[i]):
       correct += 1

quant_acc = correct / total
print(f"⚡ Quantized Model Accuracy: {quant_acc:.4f}")
```

```
# Print quantization parameters for C++ code
print("\nFor C++ code:")
print(f"Input scale: {input_scale}")
print(f"Input zero point: {input_zero_point}")
print(f"Output scale: {output_scale}")
print(f"Output zero point: {output_zero_point}")


# Confusion matrix
cm = confusion_matrix(y_true, y_pred)
grip_names = label_encoder.classes_

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=grip_names,
yticklabels=grip_names)
plt.xlabel('Predicted Grip')
plt.ylabel('True Grip')
plt.title('Confusion Matrix')
plt.tight_layout()
plt.show()
```

Appendix C: User_setup code

```
#define USER_SETUP_INFO "User_Setup"

// Driver
#define ST7789_DRIVER

// Screen dimensions
#define TFT_WIDTH  135
#define TFT_HEIGHT 240

// Pin configuration for TTGO T-Display
#define TFT_MISO -1
#define TFT_MOSI 19
#define TFT_SCLK 18
#define TFT_CS   5
#define TFT_DC   16
#define TFT_RST  23
#define TFT_BL   4    // Display backlight control pin

// Fonts to load
#define LOAD_GLCD   // Font 1. Original Adafruit 8 pixel font needs ~1820 bytes in FLASH
#define LOAD_FONT2  // Font 2. Small 16 pixel high font, needs ~3534 bytes in FLASH, 96 characters
#define LOAD_FONT4  // Font 4. Medium 26 pixel high font, needs ~5848 bytes in FLASH, 96 characters
#define LOAD_FONT6  // Font 6. Large 40 pixel font, needs ~2666 bytes in FLASH, only characters 1234567890:-.apm
#define LOAD_FONT7  // Font 7. 7 segment 48 pixel font, needs ~2438 bytes in FLASH, only characters 1234567890:.
#define LOAD_FONT8  // Font 8. Large 75 pixel font needs ~3256 bytes in FLASH, only characters 1234567890:-.
#define LOAD_GFXFF  // FreeFonts. Include access to the 48 Adafruit_GFX free fonts FF1 to FF48 and custom fonts

#define SMOOTH_FONT

// SPI Frequency
#define SPI_FREQUENCY  40000000
```

```
// Read speed
#define SPI_READ_FREQUENCY  20000000

// Touch screen speed (not used for this display)
#define SPI_TOUCH_FREQUENCY  2500000
```

Appendix D: Implementation Code

```
#include <Arduino.h>
#include <TFT_eSPI.h>
#include <TensorFlowLite_ESP32.h>
#include "tensorflow/lite/micro/all_ops_resolver.h"
#include "tensorflow/lite/micro/micro_error_reporter.h"
#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/schema/schema_generated.h"
#include "grip_model_s.h"
#include <math.h>

#define EMG_PIN 33
#define SAMPLING_FREQ 500
#define WINDOW_SIZE 64  // Changed to power of 2 for FFT
#define NUM_FEATURES 24
#define DISPLAY_BL_PIN 4
#define ENVELOPE_BUFFER_SIZE 128
#define DISPLAY_UPDATE_MS 5000
#define TFT_BLACK 0x0000
#define TFT_WHITE 0xFFFF
#define TFT_CYAN  0x07FF
#define TFT_DARKGREY 0x7BEF
#define EMG_THRESHOLD 50
#define CONFIDENCE_THRESHOLD 0.6f
#define OVERLAP 25
#define SLIDING_BUFFER_SIZE (WINDOW_SIZE * 2)

// Quantization parameters
const float INPUT_SCALE = 0.033870019018650055f;
const int8_t INPUT_ZERO_POINT = -34;
const float OUTPUT_SCALE = 0.00390625f;
const int8_t OUTPUT_ZERO_POINT = -128;

// Scaler parameters
const float scaler_mean[NUM_FEATURES] = {
    75.893293, 93.695606, 14231.133197, 3794.664651, 15.235887, 4049.460013,
    19.481519, 48.675403, 47.793002, 9381.111688, 99.732806, 95.346102,
    150.446747, 150.567069, 62.273538, 7522.337366, 0.000000, 48.588710,
    4.631720, 19.104503, 150.324950, 7999.607066, 4.225365, 0.000000
};

const float scaler_scale[NUM_FEATURES] = {
    59.042009, 73.892020, 22690.784478, 2952.100455, 3.390095, 3586.373565,
```

```
    3.020037, 0.610295, 41.171668, 8717.177357, 6.709998, 11.072275,
    114.137582, 114.251664, 189.316755, 5706.879113, 1.000000, 39.077779,
    4.277597, 10.216482, 114.023770, 6151.288014, 2.487493, 1.000000
};

// Grip class labels
const char* grip_names[] = {
    "Tip", "Spherical", "Palmer", "Cylindrical", "Hook", "Lateral"
};
const int num_grip_classes = sizeof(grip_names) / sizeof(grip_names[0]);

// 3. Class definitions
class CircularBuffer {
private:
    float data[SLIDING_BUFFER_SIZE];
    int head = 0;
    bool is_full = false;

public:
    void push(float value) {
        data[head] = value;
        head = (head + 1) % SLIDING_BUFFER_SIZE;
        if (head == 0) is_full = true;
    }

    bool isFull() const { return is_full; }

    const float* getData() const { return data; }

    void getWindow(float* window) {
        if (!window) return;
        int start = (head - WINDOW_SIZE + SLIDING_BUFFER_SIZE) %
SLIDING_BUFFER_SIZE;
        for (int i = 0; i < WINDOW_SIZE; i++) {
            window[i] = data[(start + i) % SLIDING_BUFFER_SIZE];
        }
    }

    bool hasNewWindow() const {
        static int last_head = 0;
        int samples_since_last = (head - last_head + SLIDING_BUFFER_SIZE) %
SLIDING_BUFFER_SIZE;
        if (samples_since_last >= (WINDOW_SIZE - OVERLAP)) {
            last_head = head;
```

```
            return true;
        }
        return false;
    }
};
// Global variables
TFT_eSPI tft = TFT_eSPI();
CircularBuffer emg_raw_buffer;
CircularBuffer emg_env_buffer;
int envelope_buffer[ENVELOPE_BUFFER_SIZE] = {0};
int envelope_index = 0;
int envelope_sum = 0;
bool model_loaded = false;

// TFLite globals
constexpr int kTensorArenaSize = 32 * 1024;
uint8_t tensor_arena[kTensorArenaSize];
tflite::MicroErrorReporter micro_error_reporter;
tflite::AllOpsResolver resolver;
tflite::MicroInterpreter* interpreter;
TfLiteTensor* input;
TfLiteTensor* output;

// Performance monitoring
struct PerformanceMetrics {
    unsigned long last_inference_time = 0;
    int inference_count = 0;
    float avg_inference_time = 0;
} metrics;

// Feature extraction
float calculateMAV(const float* buffer, int size);
float calculateRMS(const float* buffer, int size);
float calculateVAR(const float* buffer, int size);
float calculateIEMG(const float* buffer, int size);
int calculateZC(const float* buffer, int size, float threshold = 0.01f);
float calculateWL(const float* buffer, int size);
int calculateSSC(const float* buffer, int size, float threshold = 0.01f);
int calculateWAMP(const float* buffer, int size, float threshold = 0.02f);
float calculateLOG_D(const float* buffer, int size);

// FFT and frequency domain
unsigned int bitReverse(unsigned int x, int bits);
void fft(float* real, float* imag, int N);
```

```
void calculateFrequencyFeatures(const float* signal, int N, float fs, float& totalPower, float& mnf,
float& mdf);

// Signal processing
float EMGFilter(float input);
int calculateEnvelope(int abs_emg);
void extractFeatures(const float* raw_buffer, const float* env_buffer, float* features);

// Display and model
void updateDisplay(const char* grip_name, float confidence, const float* signal);
bool initializeModel();
bool isSignalValid(const float* buffer, int size);

// 6. Function implementations
// Feature extraction implementations
float calculateMAV(const float* buffer, int size) {
    float sum = 0;
    for (int i = 0; i < size; i++) {
        sum += abs(buffer[i]);
    }
    return sum / size;
}

float calculateRMS(const float* buffer, int size) {
    float sum = 0;
    for (int i = 0; i < size; i++) {
        sum += buffer[i] * buffer[i];
    }
    return sqrt(sum / size);
}

float calculateVAR(const float* buffer, int size) {
    float mean = 0;
    for (int i = 0; i < size; i++) {
        mean += buffer[i];
    }
    mean /= size;

    float variance = 0;
    for (int i = 0; i < size; i++) {
        float diff = buffer[i] - mean;
        variance += diff * diff;
    }
    return variance / (size - 1);
```

```
}

float calculateIEMG(const float* buffer, int size) {
    float sum = 0;
    for (int i = 0; i < size; i++) {
        sum += abs(buffer[i]);
    }
    return sum;
}

int calculateZC(const float* buffer, int size, float threshold) {
    int count = 0;
    for (int i = 0; i < size - 1; i++) {
        if ((buffer[i] > threshold && buffer[i + 1] < -threshold) ||
            (buffer[i] < -threshold && buffer[i + 1] > threshold)) {
            count++;
        }
    }
    return count;
}

float calculateWL(const float* buffer, int size) {
    float sum = 0;
    for (int i = 1; i < size; i++) {
        sum += abs(buffer[i] - buffer[i-1]);
    }
    return sum;
}

int calculateSSC(const float* buffer, int size, float threshold) {
    int count = 0;
    for (int i = 1; i < size - 1; i++) {
        float diff1 = buffer[i] - buffer[i-1];
        float diff2 = buffer[i+1] - buffer[i];
        if ((diff1 > 0 && diff2 < 0) || (diff1 < 0 && diff2 > 0)) {
            if (abs(diff1) >= threshold && abs(diff2) >= threshold) {
                count++;
            }
        }
    }
    return count;
}int calculateWAMP(const float* buffer, int size, float threshold) {
    int count = 0;
    for (int i = 0; i < size - 1; i++) {
```

```
        if (abs(buffer[i+1] - buffer[i]) > threshold) {
            count++;
        }
    }
    return count;
}

float calculateLOG_D(const float* buffer, int size) {
    float sum = 0;
    for (int i = 0; i < size; i++) {
        sum += log(abs(buffer[i]) + 1e-6);
    }
    return exp(sum / size);
}

// FFT implementations
unsigned int bitReverse(unsigned int x, int bits) {
    unsigned int result = 0;
    for (int i = 0; i < bits; i++) {
        result = (result << 1) | (x & 1);
        x >>= 1;
    }
    return result;
}

void fft(float* real, float* imag, int N) {
    int bits = log2(N);
    for (int i = 0; i < N; i++) {
        int j = bitReverse(i, bits);
        if (i < j) {
            float temp_real = real[i];
            float temp_imag = imag[i];
            real[i] = real[j];
            imag[i] = imag[j];
            real[j] = temp_real;
            imag[j] = temp_imag;
        }
    }

    for (int step = 2; step <= N; step <<= 1) {
        float angle = -2 * PI / step;
        float wr = cos(angle);
        float wi = sin(angle);
```

```
        for (int i = 0; i < N; i += step) {
            float w_real = 1;
            float w_imag = 0;

            for (int j = 0; j < step/2; j++) {
                int a = i + j;
                int b = i + j + step/2;

                float temp_real = w_real * real[b] - w_imag * imag[b];
                float temp_imag = w_real * imag[b] + w_imag * real[b];

                real[b] = real[a] - temp_real;
                imag[b] = imag[a] - temp_imag;
                real[a] = real[a] + temp_real;
                imag[a] = imag[a] + temp_imag;

                float temp = w_real * wr - w_imag * wi;
                w_imag = w_real * wi + w_imag * wr;
                w_real = temp;
            }
        }
    }
}

void calculateFrequencyFeatures(const float* signal, int N, float fs, float& totalPower, float& mnf,
float& mdf) {
    float* real = new float[N];
    float* imag = new float[N];
    float* power = new float[N/2];

    if (!real || !imag || !power) {
        Serial.println("Memory allocation failed");
        if (real) delete[] real;
        if (imag) delete[] imag;
        if (power) delete[] power;
        return;
    }

    for (int i = 0; i < N; i++) {
        real[i] = signal[i];
        imag[i] = 0;
    }

    fft(real, imag, N);
```

```
    totalPower = 0;
    for (int i = 0; i < N/2; i++) {
        power[i] = sqrt(real[i]*real[i] + imag[i]*imag[i]);
        totalPower += power[i];
    }

    float freqSum = 0;
    for (int i = 0; i < N/2; i++) {
        float freq = i * fs / N;
        freqSum += freq * power[i];
    }
    mnf = freqSum / totalPower;

    float cumPower = 0;
    int mdfIndex = 0;
    for (int i = 0; i < N/2; i++) {
        cumPower += power[i];
        if (cumPower >= totalPower/2) {
            mdfIndex = i;
            break;
        }
    }
    mdf = mdfIndex * fs / N;

    delete[] real;
    delete[] imag;
    delete[] power;
}

// Signal processing implementations
float EMGFilter(float input) {
    static float z1a, z2a, z1b, z2b, z1c, z2c, z1d, z2d;
    float output = input;

    float x = output - 0.05159732f * z1a - 0.36347401f * z2a;
    output = 0.01856301f * x + 0.03712602f * z1a + 0.01856301f * z2a;
    z2a = z1a; z1a = x;

    x = output - -0.53945795f * z1b - 0.39764934f * z2b;
    output = x + -2.0f * z1b + z2b;
    z2b = z1b; z1b = x;

    x = output - 0.47319594f * z1c - 0.70744137f * z2c;
```

```
    output = x + 2.0f * z1c + z2c;
    z2c = z1c; z1c = x;

    x = output - -1.00211112f * z1d - 0.74520226f * z2d;
    output = x + -2.0f * z1d + z2d;
    z2d = z1d; z1d = x;

    return output;
}
int calculateEnvelope(int abs_emg) {
    envelope_sum -= envelope_buffer[envelope_index];
    envelope_sum += abs_emg;
    envelope_buffer[envelope_index] = abs_emg;
    envelope_index = (envelope_index + 1) % ENVELOPE_BUFFER_SIZE;
    return (envelope_sum / ENVELOPE_BUFFER_SIZE) * 2;
}

void extractFeatures(const float* raw_buffer, const float* env_buffer, float* features) {
    int idx = 0;

    // EMG1 features (raw signal)
    features[idx++] = calculateMAV(raw_buffer, WINDOW_SIZE);
    features[idx++] = calculateRMS(raw_buffer, WINDOW_SIZE);
    features[idx++] = calculateVAR(raw_buffer, WINDOW_SIZE);
    features[idx++] = calculateIEMG(raw_buffer, WINDOW_SIZE);
    features[idx++] = calculateZC(raw_buffer, WINDOW_SIZE);
    features[idx++] = calculateWL(raw_buffer, WINDOW_SIZE);
    features[idx++] = calculateSSC(raw_buffer, WINDOW_SIZE);
    features[idx++] = calculateWAMP(raw_buffer, WINDOW_SIZE);
    features[idx++] = calculateLOG_D(raw_buffer, WINDOW_SIZE);

    float totalPower1, mnf1, mdf1;
    calculateFrequencyFeatures(raw_buffer, WINDOW_SIZE, SAMPLING_FREQ, totalPower1,
mnf1, mdf1);
    features[idx++] = totalPower1;
    features[idx++] = mnf1;
    features[idx++] = mdf1;

    // EMG2 features (envelope signal)
    features[idx++] = calculateMAV(env_buffer, WINDOW_SIZE);
    features[idx++] = calculateRMS(env_buffer, WINDOW_SIZE);
    features[idx++] = calculateVAR(env_buffer, WINDOW_SIZE);
    features[idx++] = calculateIEMG(env_buffer, WINDOW_SIZE);
    features[idx++] = calculateZC(env_buffer, WINDOW_SIZE);
```

```
    features[idx++] = calculateWL(env_buffer, WINDOW_SIZE);
    features[idx++] = calculateSSC(env_buffer, WINDOW_SIZE);
    features[idx++] = calculateWAMP(env_buffer, WINDOW_SIZE);
    features[idx++] = calculateLOG_D(env_buffer, WINDOW_SIZE);

    float totalPower2, mnf2, mdf2;
    calculateFrequencyFeatures(env_buffer, WINDOW_SIZE, SAMPLING_FREQ, totalPower2,
mnf2, mdf2);
    features[idx++] = totalPower2;
    features[idx++] = mnf2;
    features[idx++] = mdf2;
}

void updateDisplay(const char* grip_name, float confidence, const float* signal) {
    static char last_grip[20] = "";
    static float last_conf = -1;

    if (strcmp(last_grip, grip_name) != 0 || abs(last_conf - confidence) > 0.05f) {
        tft.fillRect(0, 0, tft.width(), 40, TFT_BLACK);
        tft.setCursor(0, 0);
        tft.println(grip_name);
        tft.print(int(confidence * 100));
        tft.println("%");

        strcpy(last_grip, grip_name);
        last_conf = confidence;
    }

    tft.fillRect(0, 120, tft.width(), 60, TFT_BLACK);
    tft.drawLine(0, 150, tft.width(), 150, TFT_DARKGREY);

    if (signal) {
        for (int i = 0; i < WINDOW_SIZE - 1; i++) {
            int x1 = map(i, 0, WINDOW_SIZE - 1, 0, tft.width());
            int x2 = map(i + 1, 0, WINDOW_SIZE - 1, 0, tft.width());
            int y1 = map(constrain(signal[i], -2000, 2000), -2000, 2000, 180, 120);
            int y2 = map(constrain(signal[i + 1], -2000, 2000), -2000, 2000, 180, 120);
            tft.drawLine(x1, y1, x2, y2, TFT_CYAN);
        }
    }
}

bool initializeModel() {
    const tflite::Model* model = tflite::GetModel(grip_model_s_tflite);
```

```
    if (model->version() != TFLITE_SCHEMA_VERSION) {
        Serial.println("Model schema mismatch!");
        return false;
    }

    interpreter = new tflite::MicroInterpreter(
        model, resolver, tensor_arena, kTensorArenaSize, &micro_error_reporter);

    if (interpreter->AllocateTensors() != kTfLiteOk) {
        Serial.println("Failed to allocate tensors!");
        return false;
    }

    input = interpreter->input(0);
    output = interpreter->output(0);

    if (input->dims->data[1] != NUM_FEATURES) {
        Serial.println("Input dimension mismatch!");
        return false;
    }

    return true;
}

bool isSignalValid(const float* buffer, int size) {
    float sum = 0;
    for(int i = 0; i < size; i++) {
        sum += abs(buffer[i]);
    }
    float avg = sum / size;
    return avg > EMG_THRESHOLD;
}

void setup() {
    Serial.begin(115200);

    pinMode(DISPLAY_BL_PIN, OUTPUT);
    digitalWrite(DISPLAY_BL_PIN, HIGH);
    tft.init();
    tft.setRotation(1);
    tft.fillScreen(TFT_BLACK);
    tft.setTextColor(TFT_WHITE, TFT_BLACK);
    tft.setTextSize(2);
```

```
    if (!initializeModel()) {
        tft.println("Model init failed!");
        while (1) delay(1000);
    }

    model_loaded = true;
    tft.println("Ready for EMG");
}

void loop() {
    static unsigned long last_sample_time = 0;
    static float window_buffer[WINDOW_SIZE];
    static float env_window_buffer[WINDOW_SIZE];
    unsigned long now = millis();

    if (now - last_sample_time >= 1000 / SAMPLING_FREQ) {
        last_sample_time = now;

        int raw_value = analogRead(EMG_PIN);
        float filtered = EMGFilter(raw_value);
        int env = calculateEnvelope(abs(filtered));

        emg_raw_buffer.push(filtered);
        emg_env_buffer.push(env);

        if (emg_raw_buffer.isFull() && emg_raw_buffer.hasNewWindow()) {
            emg_raw_buffer.getWindow(window_buffer);
            emg_env_buffer.getWindow(env_window_buffer);

            if (isSignalValid(window_buffer, WINDOW_SIZE)) {
                float features[NUM_FEATURES];
                extractFeatures(window_buffer, env_window_buffer, features);

                for(int i = 0; i < NUM_FEATURES; i++) {
                    float standardized = (features[i] - scaler_mean[i]) / scaler_scale[i];
                    input->data.int8[i] = (int8_t)(standardized / INPUT_SCALE +
INPUT_ZERO_POINT);
                }

                if (interpreter->Invoke() == kTfLiteOk) {
                    float max_score = -1;
                    int prediction = -1;

                    for (int i = 0; i < num_grip_classes; i++) {
```

```
                float score = (output->data.int8[i] - OUTPUT_ZERO_POINT) * OUTPUT_SCALE;
                if (score > max_score) {
                    max_score = score;
                    prediction = i;
                }
            }

            if (max_score >= CONFIDENCE_THRESHOLD) {
                updateDisplay(grip_names[prediction], max_score, window_buffer);
            } else {
                updateDisplay("Uncertain", max_score, window_buffer);
            }
        }
    } else {
        updateDisplay("No Signal", 0, window_buffer);
    }
  }
}
```