mMyoHMI: A Flexible Low-Cost Mobile Platform for EMG-based Gesture Recognition

A Thesis submitted to the faculty of
San Francisco State University
In partial fulfillment of
the requirements for
the Degree

Master of Science

In

Electrical and Computer Engineering

by

Amir Modan

San Francisco, California

December 2022

Certification of Approval

I certify that I have read mMyoHMI: A Flexible Low-Cost Mobile Platform for EMG-based Gesture Recognition by Amir Modan, and that in my opinion this work meets the criteria for approving a thesis submitted in partial fulfillment of the requirement for the degree Master of Science Electrical and Computer Engineering at San Francisco State University.

_____

Xiaorong Zhang, Ph.D.
Associate Professor,
Thesis Committee Chair

_____

Zhuwei Qin, Ph.D.
Assistant Professor

Abstract

Surface electromyographic (sEMG) signal is a bioelectric signal that measures muscle activities by surface electrodes placed on the skin and is effective for representing movement intentions. Myoelectric-controlled human-machine interfaces (HMIs) utilize sEMG signals from the user's muscles to identify what the user is doing physically and can relay this action to control an output device such as a prosthetic limb or a gesture-controlled application. The aim of this project is to enhance a previously developed Android-based mobile computing platform named mMyoHMI, to provide a portable, low-cost, and flexible solution for sEMG pattern recognition-based myoelectric-controlled applications. The mMyoHMI platform seamlessly integrates a variety of interfacing and signal processing modules from data acquisition through pattern recognition to real-time evaluation and control. The enhanced MyoHMI app improves the input interface module to make it flexible to collect sEMG signals from many different types of sensors via Bluetooth Low Energy (BLE) and has made the first attempt to integrate deep learning methods on a mobile device for real-time sEMG pattern classification.

Acknowledgements

I would like to thank my family and professors for their undying support without which, this project would not have been possible.

Table of Contents

List of Tables

List of Figures

List of Appendices

Introduction

All living organisms rely on their muscular system to perform basic actions necessary for survival. However, individuals with a compromised muscular system, such as those who have survived a stroke, often experience a drastic reduction in their quality of life due to not being able to perform these actions. To regain at least partial muscular control, patients will often undergo intensive rehabilitation programs which require them to perform basic gestures repeatedly such as flexing a wrist. These people can benefit from a human-machine interface (HMI) like the "MyoHMI" platform presented by Donovan et al. which utilizes surface electromyographic (sEMG) signals from the user's muscles to identify what the user is doing physically by using pattern classification methods [1]. MyoHMI can be especially useful to physiotherapists who must often monitor their patients for improvements in muscular function. Furthermore, gestures classified by MyoHMI can even be relayed to an output device such as a Virtual Reality (VR) Headset presented by Muñoz et al., a Prosthetic Arm presented by Bhavani et al., or an assistive vehicle presented by Naber et al., allowing for advanced rehabilitation methods [2-4].

Surface electromyographic (sEMG) signal is a bioelectric signal that measures muscle activities by surface electrodes placed on the skin and is effective for representing movement intentions. Myoelectric-controlled HMIs are typically non-invasive meaning that the EMG signals are measured from the surface of the skin and can be worn/removed outside of a lab environment. While wearability is not always a primary concern such as in the interface designed by Ceolini et al., user interfaces must often be intuitive enough so that they could be used in day-

to-day life and by those who may not have a background in Engineering [5]. This is especially true in a rehabilitation environment. Additionally, the proposed system must be flexible enough so that it could be used in many applications with a low budget.

The aim of this project is to enhance a previously developed Android-based mobile myoelectric-controlled HMI platform named *mMyoHMI*, to provide a portable, low-cost, and flexible solution for sEMG pattern recognition-based myoelectric-controlled applications. The mMyoHMI platform seamlessly integrates a variety of interfacing and signal processing modules from data acquisition through pattern recognition to real-time evaluation and control. The enhanced mMyoHMI app improves the input interface module to make it flexible to collect sEMG signals from many different types of sensors via Bluetooth Low Energy (BLE) and has made the first attempt to integrate deep learning methods on a mobile device for real-time sEMG pattern classification. The Myoware Muscle Sensor used in the mMyoHMI prototype should theoretically be substitutable with other EMG devices or even a high-density (HD) EMG grid of sensors. This allows mMyoHMI to be applicable towards high-budget research while, on the other hand, being able to provide casual consumers with the same features in a low-cost package. Finally, mMyoHMI will be made open to the public and does not rely on proprietary software such as MATLAB which requires a license to use. For this reason, we designed our interface on Android's open-source operating system, allowing anyone with an Android device to utilize our software.

Related Works

Several HMIs have already been developed for the purposes of facilitating human-machine interactions. For instance, BioPatRec is a commonly used HMI presented by Ortiz-Catalan et al. offering many of the same functionalities as mMyoHMI [7]. It is also completely open-source, meaning that developers can adapt it into their own software. For this reason, it has been used in many research papers relating to EMG-based pattern recognition such as in the system proposed by Mastinu et al. [8]. However, the HMI runs on MATLAB, meaning that a MATLAB license is required to use the HMI, severely reducing the breadth of consumers the software appeals to. Bhat et al. presents an open HMI which, like mMyoHMI, allows the user to diagnose and monitor a possible movement disorder through EMG-based gesture recognition [9]. While the wearable device designed alongside the HMI shows promise with its flexible form, users of the HMI would be restricted to this device only. Omama et al. demonstrates how computer vision can be integrated with EMG into an Android-based HMI for high-accuracy hand gesture classification [10]. While the user is free to choose between a variety of different inputs, the interface still does not grant the user flexibility over characteristics such as number of channels. Zhu et al. presents an HMI that can interface with an off-the-shelf smartwatch for real-time gesture recognition [11]. However, the system proposed lacks relevance in the field of physical rehabilitation since the HMI relies solely on inertial signals rather than EMG which gives insight into the user's muscular system. A variety of different EMG-based classifiers are explored in the system proposed by Deng et al., though classification is performed offline, reducing its practicality in a consumer environment [12]. The HMI presented by Zea et al. can

evaluate the performance of machine-learning models after creating them, allowing for accurate gesture recognition [13]. However, this must be done manually, limiting its usability to a research environment only. mMyoHMI will aim to rectify all of the aforementioned drawbacks in previous works to provide users with a system that is low-cost, open, and non-invasive, while offering flexibility and modularity.

Current Limitations

While the current version of MyoHMI already supports the features mentioned, it is currently only available on PC's and only supports the discontinued Myo Armband by Thalmic Labs. The fact that MyoHMI can only be accessed from a computer severely reduces its usability in a consumer environment. Therefore, the goal of this research is to develop "mMyoHMI", a low-cost, flexible mobile platform that offers all of the features provided by its PC version, only on the open-source Android environment.

Additionally, changes will be made to the structure of the software to allow compatibility with any Bluetooth Low Energy (BLE) device rather than the Myo Armband. The consumer ready Myo Armband is easily worn/removed with little effort from the user, and even delivers 8 channels of EMG data allowing for several gestures that can be classified into. However, there are some major caveats to using the Myo Armband:

1. The Armband is rather expensive, costing about $200. Consumers with a lower budget will not be able to use mMyoHMI if they cannot afford the Myo Armband. By restricting compatibility to the Myo Armband, we are losing the consumer base who are looking for a cost-effective solution to gesture recognition.
2. The Armband samples data at only 200Hz. Most medical engineering papers suggest sampling in the kHz range for surface EMG signals as the frequency of EMG signals ranges from 50 Hz to 500 Hz.
3. The Armband has been discontinued and no longer receives support from its creator, Thalmic Labs. We cannot design a platform whose components are already limited in supply. In order

to keep mMyoHMI from becoming irrelevant in the market, we will have to support compatibility for at least one alternative that will continue to be supported and used in the foreseeable future.

To address the three issues mentioned above, we will be updating the Android version of mMyoHMI to support the Myoware Muscle Sensor. At a fraction of the price at $37.50 and still in production, the Myoware Muscle Sensor will serve our lower-budget target audience well, allowing us to gain the edge over competing platforms utilizing more expensive equipment. But more importantly, expanding our interface in this manner will mean that we will have to make several aspects of our interface dynamic, such as how many channels to expect or at what rate to receive data. Theoretically then, any device with any number of channels and sampling rate should be compatible with mMyoHMI so long as it uses the same communication protocol, Bluetooth Low Energy (BLE). Note that we will use four Myoware Muscle Sensors in order to maintain the HMI's ability to classify EMG data into several different gestures. While mMyoHMI is intended to work with as few as just one channel, accurate gesture classification would be limited to only a few gestures, so it would be difficult to analyze the full performance of mMyoHMI with such a data acquisition system (DAS).

mMyoHMI is already capable of classifying gestures using various conventional machine learning algorithms including Linear Discriminant Analysis (LDA), Support Vector Machine (SVM), Logistic Regression, Decision Tree, Multi-Layer Perceptron (MLP), K-Nearest Neighbors (KNN), and Adaboost. The main assumption of EMG pattern recognition is that each movement task has a consistent pattern of muscle activation that can be characterized by a set of features. These features should be reproducible across trials of the same task and discriminative

between different tasks. The real-time EMG pattern recognition procedure typically consists of three major phases: data segmentation, feature extraction, and pattern classification. Multi-channel EMG signals collected from the subject's muscles are the system input. The data segmentation module segments the EMG data stream into a continuous sequence of analysis windows. For each analysis window, the feature extraction module extracts features that characterize the EMG signals. The features extracted from individual input channels are concatenated into a feature vector and then fed into the pattern classification module for movement intent classification. EMG pattern classification typically consists of two phases: training and testing. The training phase uses mathematical methods to identify a decision boundary (classifier) that can maximally distinguish the features of different movement tasks. The training phase is usually a time-consuming process because it needs to process a large amount of data collected from all the movement tasks that are of interest. In the testing phase, the classifier is employed to process the data of each analysis window to make a prediction of the movement intent. The decisions will then be processed to generate commands to control external applications such as a prosthetic hand. Recently, with the advancement of deep learning (DL) technologies, applying DL methods to sEMG-based movement intent recognition has gained increasing interest in the research community. Various DL methods such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and hybrid CNN-RNN methods have been exploited for sEMG-based gesture recognition and have shown promising results in studies including the ones proposed by Geng et al., Quivira et al., Coˆte´-Allard et al., and Shen et al. [15-19]. Compared to conventional non-DL methods commonly used for EMG pattern recognition, DL algorithms have the advantage of automatically extracting EMG features

without the cumbersome manual feature engineering step and are especially effective in processing sEMG signals collected from 1-dimentional (1D) or 2D sensor arrays. However, a key challenge to the deployment of DL methods in sEMG-controlled HMI applications is the high computational cost associated with DL algorithms especially when the HMI applications need to be implemented on resource-constrained mobile platforms. In addition, the DL models are typically required to be trained with a large dataset which might be hard to obtain by individual users. Fortunately, a new technology, “TensorFlow Lite”, has been developed by Google which allows deep learning models to be trained offline, then finetuned on the mobile device to match the user’s specific setup.

Gesture recognition can aid in the rehabilitation of people whose muscular functions have been compromised, such as amputees and those who have recently suffered a stroke. These people can benefit from a HMI like mMyoHMI which can give them a good insight into how much progress they have made in regaining muscle control. The benefit is multiplied if the user has access to either a VR headset or a prosthetic device such as the Hackberry Arm. The former can be used to perform more entertaining tasks as a form of rehabilitation, such as VR Table Tennis and other games. Fortunately, VR Headsets are becoming increasingly accessible with extremely cheap headsets costing less than $10. A prosthetic arm can be used to perform even real-life activities, theoretically substituting the function of an actual arm assuming accurate and swift classification. However, the price for such a device suggests that this feature will still have lower consumer accessibility.

By expanding the mMyoHMI app so that it supports dynamic configurations, we are broadening our target audience to include anyone in search of an EMG-based HMI. While casual

users will be able to use the app with as few as a single EMG sensor, researchers can also use the app for performing high-accuracy experiments utilizing even an EMG grid consisting of hundreds of channels.

While gesture recognition platforms involving surface EMG signals have already been developed, these platforms generally involve laboratory equipment not suited for a commercial setting due to increased difficulty in setting up such a system. On the other hand, platforms have been developed for the average consumer in mind, taking advantage of wearable EMG devices like the Myo Armband to perform gesture recognition. However, these Interfaces do not offer precise and accurate results as their DAS's typically boast fewer channels of data and at a lower sampling rate. What mMyoHMI offers over competing platforms is flexibility, the ability to service the entire spectrum of potential users. The user can also specify other components of the interface, such as which gestures to use for classification and which features to be used.

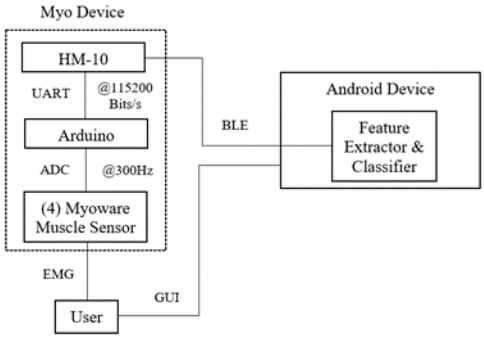
Design and Implementation of mMyoHMI

Interface Design

Hardware Design

As per the system architecture shown in Figure 1, the user will be an integral part of the system, providing EMG data through up to 4 different channels and interacting with mMyoHMI's GUI to access the data and the gestures they are classified into. Unlike with the Myo Armband, we will have to commence sampling of the EMG data using our own microcontroller. Like the Myo Armband, the data must also be sent using the BLE protocol, though we will have to program the microcontroller ourselves to do this.

Figure 1. MyoHMI System Architecture



This diagram displays the relationship between the Android Device, the sensor worn by the user, and the user themself.

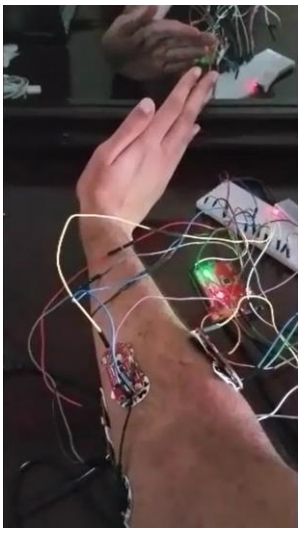
Firstly, we will use an Arduino board as our microcontroller due to its ease of use and its large user database. More importantly, the Arduino provides both 3.3V and 5V source pins, so we will have more freedom when deciding which peripherals will need to be connected as opposed to other microcontrollers such as the Tiva Board which only provides 3.3V. For instance, the BLE module we will be using to transmit data to mMyoHMI operates in a range

between 3.6-6V, a range that a 3.3V pin does not fall under. The Myoware muscle sensors themselves operate at 2.9-5.7V so either power source can be used.

A BLE module will be necessary as our low-cost microcontroller does not come with BLE built in. BLE (Bluetooth Low Energy) is a data transmission protocol that sacrifices low latency at short ranges for lower power consumption. Because we will want the user to keep the wearable device on for long periods of time, a low-power option would be ideal for such a system. We will connect this device to the TX/RX UART pins on the Arduino Board.

In order to build a robust apparatus, we soldered wires to each of the Myoware Sensors, so the chances of loose wires shorting together are minimal. Each sensor will require a connection to VDD, VSS, and an analog pin for sending data to. There are two data pins on the Myoware sensor that we can read data from. The most commonly used data pin sends data that has been both rectified and integrated for a smooth signal. However, we will utilize the third pin which will send the raw EMG data as we will want to remain consistent with the raw EMG data that is already sent from the Myo Armband. Additionally, we can tune the gain of the sensor directly by adjusting an on-board potentiometer. To ensure that we can accurately make predictions from an ample set of gestures, we will utilize a total of 4 Myoware sensors meaning that 4 channels of EMG data will be transmitted. Our physical setup is seen in Figure 2.

Figure 2. Physical Setup



The user performs a Wave-Out Gesture while wearing the 4 Myoware Muscle Sensors.

It should be emphasized, however, that the hardware design we have specified above is arbitrary and any component or even the system itself can be interchanged with other components. Only for the purposes of simplicity and convenience have we chosen the design above over, say, an EMG grid or a Tivaware Microcontroller. This means that wearability of the peripheral device is NOT a priority as the focus of our project is the HMI rather than the device itself.

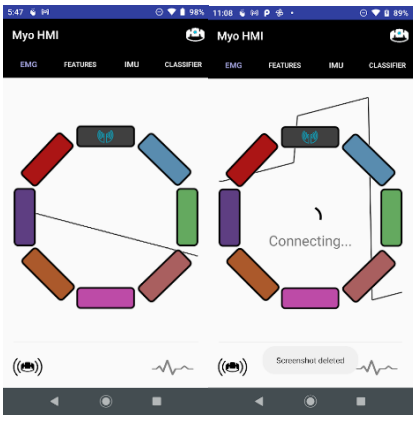
Software Design

The Myoware Sensors output data that ranges from 0 to 1023, requiring 10 bits to store the data. However because the Arduino IDE will only allow us to store this data in either a single byte (8 bits) or a short (16 bits), we would have to use 16 bits to store 10 bits of data, essentially wasting 6 bits of data. Also, mMyoHMI is already configured to take in data that is 8 bits long (1 byte) as is the case with the data sent by the Myo Armband, so data being stored in any other

format will likely be misinterpreted by the app. For these reasons, we plan to perform an arithmetic shift right by 2 bits, dividing the amplitude of the raw signal by 4 so the sample can fit into 1 byte rather than two. We would also have to insert a vertical offset in the data so that it ranges from -128 to 127 rather than 0 to 255. While this may reduce the resolution of the data, only one byte will be needed to send each sample to mMyoHMI, enhancing the data rate that the BLE module will transmit at.

Because our BLE module will send data at a baud rate of 115200 Bits/Second (14,400 Bytes/Second); With each sample occupying one byte, we should be able to comfortably achieve 300Hz transmission rate even with a Start and Stop Byte added. Whether or not data is fully being transmitted with no gaps present can be confirmed by transmitting a simple sawtooth wave instead of the actual EMG data which is harder to analyze for such irregularities. An example showing data that is being transmitted fully compared to one that exhibits data loss can be seen on the right in Figure 3.

Figure 3. Plots of Complete Transmission at 300Hz and Data Loss at 2kHz



Sawtooth Wave is transmitted without gaps when sampled at 200 Hz but is missing data when sampled at 2kHz.

The Myoware Sensors will need to be sampled by the Arduino at 300Hz. This can be achieved using a periodic interrupt that will perform an interrupt service routine (ISR) each time the Timer is reset. The ISR will use the Serial.write command from the Hardware Serial Library on the Arduino to encode the sampled data from each of the 4 channels as 4 Bytes.

Hardware Experimentation

Implementation Issues

We had faced many roadblocks in implementation of the peripheral, mostly dealing with data transmission from the BLE module to mMyoHMI. More specifically, having mMyoHMI display all data received from the BLE module without any data gaps at high frequencies was a real challenge that we have still not addressed fully.

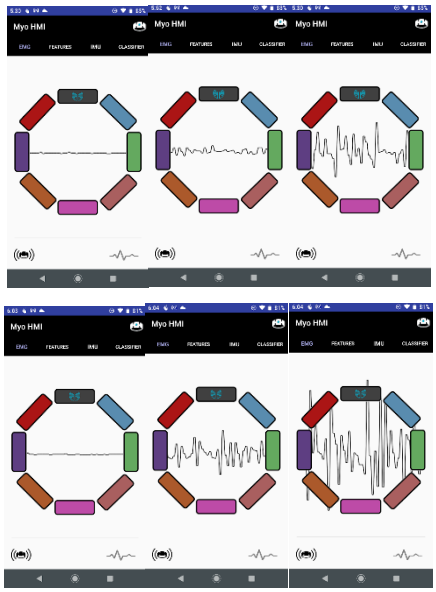
One bug that we had included early in the design that we had not realized until recently was the usage of Serial.write vs. Serial.print to send serial data from the microcontroller to the BLE module. The former will send data in Byte format, so Serial.write(123) would send a single byte of data containing bits: 01111011. On the other hand, the latter would be sending the data in String format, so Serial.print(123) would actually send the ASCII representation of characters (not numbers) ‘1’, ‘2’, ‘3’, whose bit representation would be 00110001 00110010 00110011. Using this option, we had severely impeded data transmission by sending 3 bytes of data when only 1 byte was needed. Nonetheless, fixing the bug was relatively simple as Serial.write was able to send data in the byte format we desire.

Another bug we had included, though mitigated early on, was the usage of the SoftwareSerial library instead of HardwareSerial. While the two are very similar in their function of sending UART data to a peripheral, the former will do so using digital I/O while the latter will

use dedicated hardware to send the data. While SoftwareSerial allows us to send data through any digital pin available on the Arduino, this software approach cannot reach the 115200 Bits/Second rate our BLE module will allow for, with maximum Baud Rate achieving about 38400 Bits/Second assuming no other processes are running. However, the use of the dedicated hardware pins on the Arduino will allow us to achieve our target baud rate, though only the Tx/Rx pins can be used for this purpose.

One minor issue we noticed with the Myoware Sensor was that, even with gain set at maximum, the sensor was still rather unsensitive to any gestures we were performing. This is because the signal generated by the sensor was affected by the internal resistance of the Arduino microcontroller. By buffering the analog output of the sensor using an operational amplifier, this issue can easily be mitigated. Alternatively, one can simply amplify the data using software, i.e. performing an arithmetic bit shift left for each digital sample if an operational amplifier is not available. Note that the latter technique will reduce the resolution of the data as the least significant bit of the sample would always be 0. Recall that the 10-bit ADC data would ultimately have to be divided anyways in order to fit in the 8-bit element of the BLE buffer, so preserving the resolution of the data is not a priority. Therefore, we decided to use bit-shifting to amplify the data instead of an operational amplifier to reduce hardware costs. The effects of amplification for different intensities can be seen below in Figure 4.

Figure 4. Plots showing Relaxation, Flexion, and Maximum Flexion without and with Amplification



A series of plots show the user performing varying levels of flexion. The second row is amplified by 2x.

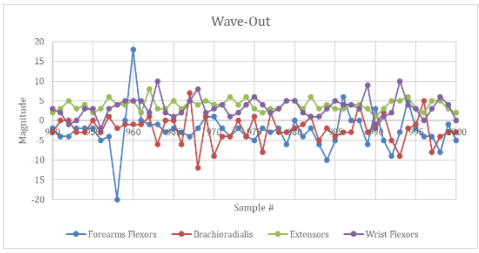
One final issue from the peripheral side that seemingly cannot be mitigated without the use of additional hardware is the overflowing of the 20-Byte buffer that is being sent from the BLE module to mMyoHMI. When sending data at low speeds, the BLE module is able to send all data without any gaps due to the fact that the buffer is not completely filled with data. However at higher frequencies in the kHz range, the Arduino will attempt to send more than 20 Bytes of data before the BLE module has the chance to send this buffer to mMyoHMI, resulting in some of the data to be overwritten even before it is sent. This will result in gaps in the data being sent to mMyoHMI. One may ask; “How come the Myo Armband is able to send 8 channels of data at 200Hz?”. This is because the Myo Armband sends data simultaneously in 4 separate buffers, effectively making use of hardware parallelism to boost its throughput. Else, one would see that each buffer is actually sending less data than the amount we are sending from

our BLE module. In order to boost the maximum throughput of our Arduino, we would have to use additional BLE modules so that we will be able to send data in parallel. Note, that this increase in performance comes at a tradeoff in cost and power consumption, so we may not follow this approach. We could also purchase a different BLE module that will allow us to increase the size of the 20-Byte buffer as our current module does not allow for sizes greater than 20.

Anticipated Results

When sending data from the peripheral device to mMyoHMI, it is expected that no data is lost during transmission. Transmission speed is optimized by using the highest baud rate possible on our microcontroller which is 115,200 bits/second. Additionally, the EMG data sampled by the microcontroller should be normalized such that it can fit inside a single byte for the sake of BLE transmission. This was achieved using arithmetic bit-shifting, a faster alternative to dividing by multiples of 2. When sending EMG data to the app, we would expect the output to be similar to what is shown in Figure 5.

Figure 5. Plot showing EMG data for a Wave-Out



Expected sEMG output for a Wave-Out Gesture

Design of the Graphic User Interface

Plotter

We will also have to make changes to mMyoHMI in order to remove some of the features that were available only to the Myo Armband. For instance, we will no longer have to read IMU data since our Myoware Muscle Sensor does not support such features. However, most changes will take place in classes having to do with Feature Extraction and Classification, as these classes are programmed to work specifically with the 8-Channel Myo Armband.

Data will be read from the BLE module in a 20-byte buffer called a GATT (Generic Attribute) Characteristic rather than as a continuous stream of data sent by a classic Bluetooth Module. Most wearable devices including the Myo Armband feature BLE over classic Bluetooth communication as its lower power consumption is ideal for products that will be worn possibly for many hours at a time. Therefore, we will prioritize supporting BLE communication though we may consider bringing support for classic Bluetooth in the future.

Because our interface is expected to be compatible with any number of channels, the current layout supporting up to 8 channels of data plotting will be impractical, particularly for applications involving more than 8 sensors. Additionally, the app currently allows the user to view only one stream of EMG data at a time. For this reason, we will redesign the layout of the EMG plotter such that it can accommodate more sensors.

For our updated layout, we will plot up to four channels of data simultaneously to allow the user to visually compare channels with minimum effort. By default, the first four channels will be plotted. Each plotter will have a corresponding drop-down menu which enables the user to plot their desired channel.

Feature Extraction

mMyoHMI allows for the user to enable/disable several features that can be used to classify gestures from EMG data. While it is difficult for us human beings to make sense of these features, Machine Learning Classifiers rely on them for training and to accurately make predictions on unclassified data. Whereas raw EMG data is too random to make any inferences off of, feature data like that shown in Figure 6 tend to remain relatively constant, enough for the computer to train itself to recognize further data. Of course, the right features which remain constant for select classes must be chosen if the classifier is to be reliable. Mathematically derived features, such as a waveform's Mean Absolute Value (MAV), are quite effective for determining the behavior of EMG data. The user may select which features should be used for classification by interacting with the HMI's checkboxes. Additionally, the Feature Extractor is able to adapt to support a variable number of channels based on what the user enters.

Figure 6. MyoHMI Feature Display

Features like the ones shown above are extracted for each channel and remain relatively constant.

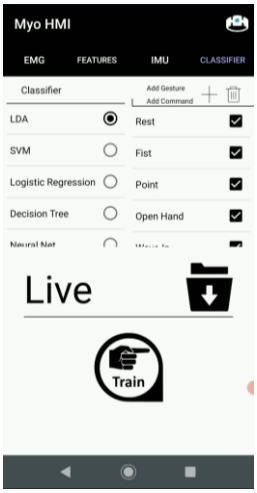
Classification

To train the classifier, the user must perform a set of gestures displayed by the screen. Since the classifier will undergo supervised learning, each feature used to train the classifier will

be associated with a label, i.e., the gesture that is being displayed at that time. By doing so, the classifier can infer a model which relates the features being extracted from the EMG data to the set of gestures. After the training is complete and the model is created, the model will make predictions on whatever EMG data the user provides.

mMyoHMI boasts a high degree of flexibility by allowing the user to select whichever classifier they would like to use, shown in Figure 7. For instance, the computationally simple Linear Discriminant Analysis can be used for situations in which hardware resources are limited whereas the more complex Neural Network can be used when accuracy is prioritized. The user can also specify which gestures will be trained. A default list of gestures is given, though the user can deselect any one of them or even add their own gestures to the list.

Figure 7. Classifier



The user may select between several different classifiers to use as well as gestures which are to be classified.

App Validation

Regarding the Android application, most of the groundwork was already completed in previous works. However, both of these versions of mMyoHMI only supported the Myo Armband as a peripheral device. As a result, the classes describing the retrieval and processing of the EMG data were designed only to support devices with 8 channels, like the Myo Armband. In order to achieve a high degree of flexibility as we described in Section I, we would have to update these classes to support any number of channels. For this reason, we added an additional feature to the app's device selection screen that prompts the user to enter the number of channels they will be sending. Whatever is entered in the box seen in Figure 8 will be used to create instances of the BLE connection, the Plotter, the Feature Calculator, and the Classifier classes.

Figure 8. Device Selection Screen

Before training a classifier, the user must first select a device to connect to and enter the number of channels to be used.

Another bug from the app side that was left over from the most previous version was that connection to a Hackberry arm, also seen in Figure 8, was required to perform classification, without which the app would crash upon completion of training. The crash would be caused when the app attempts to send a gesture prediction to a Hackberry Arm which, if absent, would

be a null object. This was easily solved by inserting logic in the Classifier class that would check whether an instance of the Hackberry Arm exists before sending the prediction.

When performing tests on the performance of mMyoHMI, it would be ignorant to use live EMG data to train the classifier. The reason for this being that, if an unexpected error were to occur, we would not know whether it is because of a bug in the app, or the integrity of the EMG data itself. It is good practice in any experiment to isolate one variable while keeping all others constant to gain a better understanding of the system. For this reason, we will transmit from a prerecorded set of EMG data rather than from live EMG data sampled by the microcontroller's ADC.

To achieve this, we first build the dataset of EMG data by reading the ADC values as normally. Instead of transmitting this live data via the BLE module, the data is sent to and stored in a UART terminal log which we save as a .csv file. For each of the 6 gestures we are testing, we recorded 4000 samples to ensure we have a sufficient batch size for training and testing the Classifier. Each of these samples are stored in a 4xN Array of signed bytes on the Arduino. However, storing this much data in only 2kB of cache memory (where program variables are stored) would be impossible unless only a few number of samples per gesture (~75) are stored. For this reason, we use the special identifier PROGMEM to instead store the set of prerecorded in the Arduino's Flash memory which has a much greater size at 32kB. This will allow as many as ~1000 samples per gesture to be stored in the Arduino. The samples stored in the flash memory will be sent to MyoHMI in lieu of samples from the ADC module. In this situation, we can use buttons to simulate performing different gestures.

We would expect classification to perform similarly to how previous versions of the app would with the Myo Armband. Having reduced the number of channels by half, it wouldn't come as a surprise if the accuracy of the classifier would reduce as well since the classifier will have less information to infer a model off of. The most vulnerable to error would be the gestures that utilize similar muscle groups such as the Fist and Thumbs Up gestures. Nevertheless, 4 channels of EMG data should still be sufficient to differentiate between 6 different gestures. We would also expect training time to be shorter than when the Myo Armband is used since the sampling rate of EMG signals has been increased to 300Hz.

Implementation of the Deep Learning Method

For years, Google's open-source deep learning library known as TensorFlow has been used in countless situations, often for applications involving object recognition, natural language processing, and sentiment analysis. deep learning models developed using TensorFlow can also be used for sEMG pattern recognition in a manner similar to image recognition wherein sEMG readings are used instead of pixels. However, until recently, deep learning was used almost exclusively on computers because mobile devices simply lack to resources needed to train models quickly. This can be due to their lack of a strong Graphics Processing Unit (GPU) which would otherwise perform all necessary matrix multiplications in parallel. But now, with the newly released *TensorFlow Lite* library, mobile devices can take advantage of deep learning as well. TensorFlow Lite is a set of packages from the TensorFlow geared specifically towards embedded systems and mobile devices. Rather than train a deep learning model from scratch on a mobile device, TensorFlow Lite accepts and can make inferences on models already trained on a computer. This process is shown in Figure 9.

Figure 9. Finetuning Process

Rather than train the model from scratch, the Android system will "finetune" a preexisting model, requiring less computation.

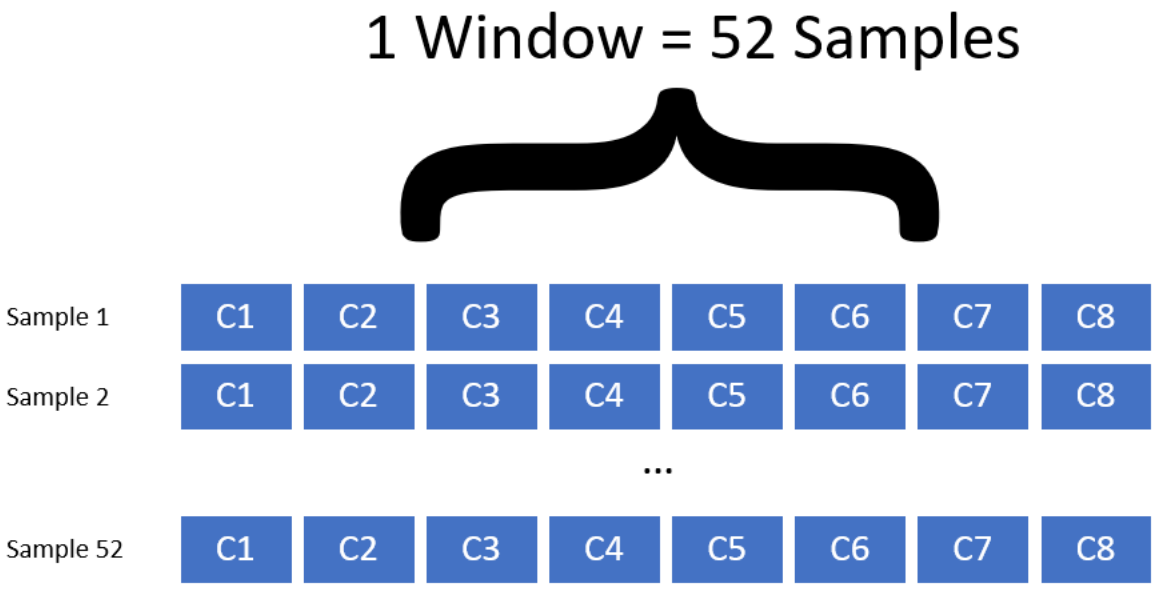
Model Design

To use a model on a mobile device, one must first design that model on a computer running TensorFlow and convert it to the .tflite format. For this design, we will use Python as it is the most commonly used language for TensorFlow applications.

The main difference in training a deep learning model is that we will no longer need to explicitly extract features from the raw sEMG data. The reason for this is that neural networks will automatically learn features from the data provided as it passes through each layer in the network. This will prove advantageous to us as features will no longer need to be extracted by the mobile device, reducing computational load and latency. With this in mind, the input to the neural network will be a 2-dimensional array where the number of rows is represented by the

number of channels (8) whilst the number of columns is represented by window length (52). The structure of the input layer is shown in Figure 10.

Figure 10. Input Layer Structure



A 2D sEMG image will be fed into the input layer of the network. The number of channels being collected, and the window size will define the size of each dimension which, in this case, will be 8x52.

The window length will represent how many samples will be represented in each image. Because each sample will be taken from a different instance in time, this will allow the network to establish a temporal understanding of the sEMG data being fed. This is crucial for sEMG-based gesture recognition as a gesture cannot be identified from one sample in time, but rather from a collection of samples taken one after another. A larger window size will allow for greater temporal feature extraction but will increase computational complexity and latency. Through trial-and-error, we concluded that a window size of 52 is optimal for our application.

When designing the architecture of our neural network, several attributes of the network, known as parameters, can be adjusted to fit the needs of the application it is used in. While it is

not necessary for the developer using the network to have knowledge of them, parameters will often have a major effect on the accuracy of the neural network and the speed at which inferences are made. Oftentimes, these two will vary inversely with each other, so a tradeoff must be made depending on the application. For our purposes, we will optimize each parameter to favor inference speed over accuracy, as our application will be used in real-time and on a mobile device. A list of some important parameters and their optimized settings are as follows:

- Number of Convolutional Hidden Layers = 2: These layers are responsible for extracting the features from the sEMG data. A greater number of layers will more effectively extract stable features from sEMG data which will be more easily learned by the network. However, a greater number of layers will increase both the training time and the inference latency of the network, the latter of which must be kept to a minimum to ensure the mobile device can make real-time predictions. Therefore, we will limit the number of layers to 2.
- Number of Filters per Convolutional Layer = 32,64: The convolutional filters are responsible for actually filtering out features in each hidden layer. Increasing the number of filters will allow the network to derive more patterns in the data that it is fed. The reason for doubling this number in the second layer is because the second layer will be fed with data already containing features derived from the first, making it more likely to successfully identify patterns. On the other hand, the raw data being fed to the first layer will be more ambiguous, so not as many abstractions can be made.

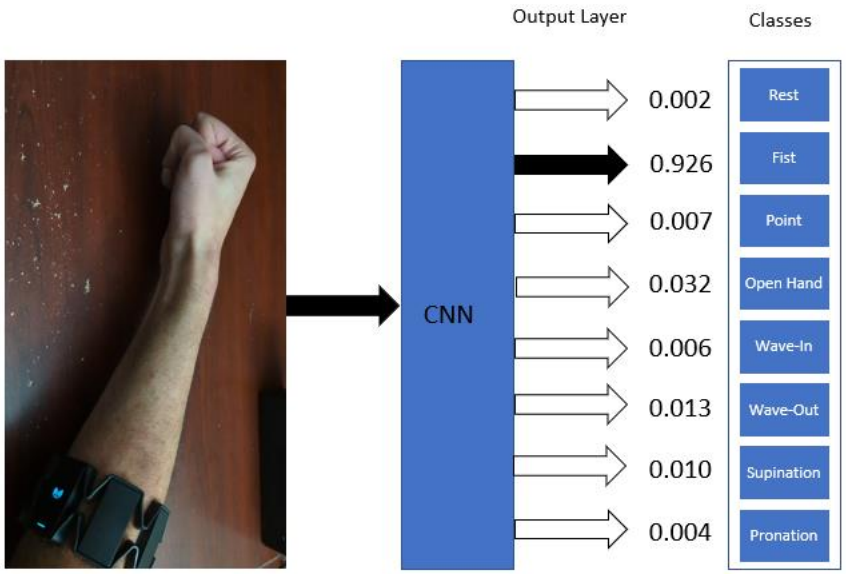
- Learning Rate = 0.001: This will affect how much the network will be affected by each window of training data. A lower number means that the network will not be severely damaged by outliers which may confuse the network. This is important when training a network on sEMG data as there is often a fair amount of noise involved when collecting EMG data from the skin. However, this means that the network must be trained for more cycles, or epochs, in order to fully train the network.
- Epochs = 1000: The number of cycles we will train the neural network with the entire training set, known as epochs. If the number of epochs is too low, the model is undertrained and hasn't fully recognized a pattern in the data yet. On the other hand, if the number of epochs is too large, the model is considered overtrained and the model will be unable to successfully classify any data other than the data it was trained on. The value we should assign to this parameter is dependent upon some of the other parameters including learning rate and the size of the training set. This parameter does not affect the inference latency, so we adjusted it to achieve maximum testing accuracy based on trial-and-error.

Although the mobile device will be able to access and make inferences on the model if it were converted at this point, it would be unable to update the model with its own data collected by the user. Model personalization is especially crucial for our purpose as the model will need to learn the user's specific sensor setup which can be changed from session to session. We will assign "signatures" to certain functions which will allow the mobile device to update the model. These signatures act as handles which run python functions meaning that TensorFlow functions

can be called directly from a mobile environment. This also means that user data can be passed as a parameter into the python function and used to finetune the model to learn the user's setup.

The final layer will consist of eight neural units, each pertaining to one class. This means that the neural model will be able to classify eight gestures where the class with the highest weight will be treated by the app as the prediction. An example of this is shown in Figure 11 where a trained neural network is fed sEMG data associated with a fist and, as a result, predicts that a fist is being made with 92.6% certainty.

Figure 11. Output Layer Structure



The output layer generates a list of weights, each one representing the network's confidence in the associated gesture. The gesture with the highest confidence will be treated as the network's prediction which, in the above example, would be the fist.

Once the model has been trained, it can be converted into the .tflite format for usage by TensorFlow Lite. While in this format, the model will retain many of the same core features as it would have in TensorFlow while being considerably smaller in size. This means that TensorFlow

Lite models will perform significantly faster than their desktop counterparts, albeit with lower accuracy.

Model Usage

After a .tflite file is created, it shall be transferred to the mobile device's internal storage for use in the application. Here, the app can access the model saved on the file and update it by calling the signatures described above.

While the model can be used for inference immediately, the model will be more likely to generate correct predictions if it is first finetuned with data collected from the user. This can be done in a manner similar to the training phase of the other machine learning algorithms in which the user must hold each gesture for a set amount of time while the app collects data. Fortunately, we won't have to extract features as we did with the standard algorithms since the neural network will accept raw sEMG data and automatically extract features on its own.

The biggest drawback to using TensorFlow Lite is that there is no way to modify the structure of the neural network. That means that certain elements of the network, such as the size of the input layer, are fixed upon creation. Therefore, the model used for classification shall have a fixed number of channels and window length.

While the structure of the model cannot be changed by the app, we can interpret the output of the model in such a way that will allow the user to classify a variable number of gestures using the same model. While the softmax function of the model will always produce 8 outputs, we can disregard a number of these outputs if the user is only interested in distinguishing between a lesser number of gestures. Instead of simply choosing the class with the highest probability, we would only regard gestures selected by the user and select the highest

probability from that pool of gestures. Note, that in order to select more than 8 gestures, a new model must be trained that can accommodate those extra gestures.

Model Training & Fine-Tuning

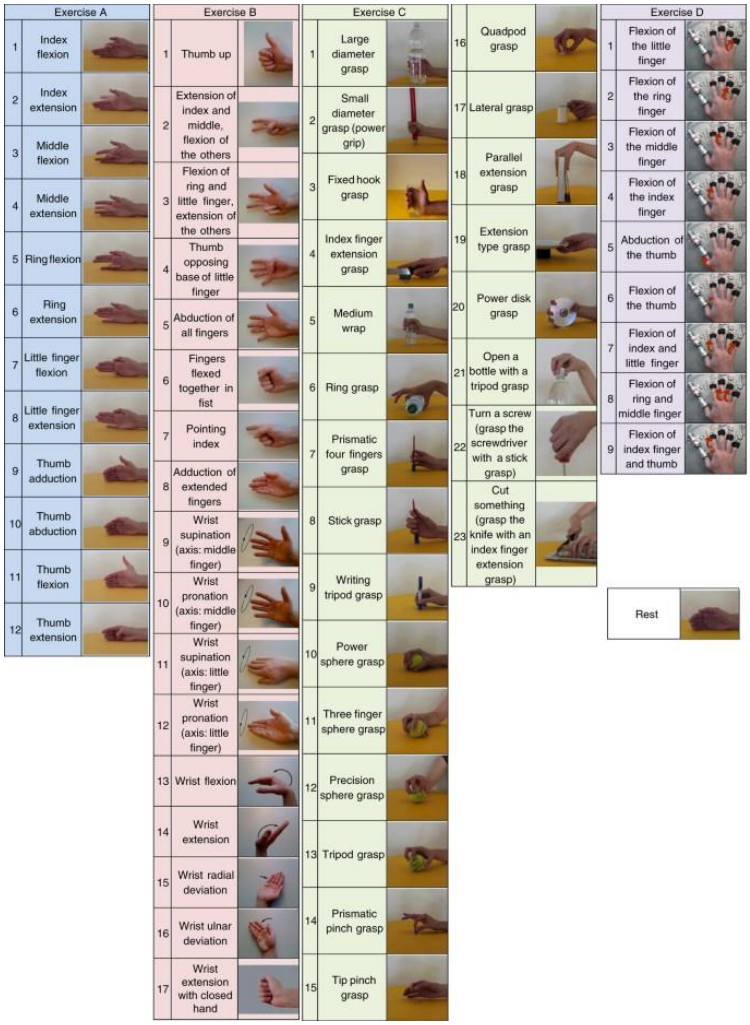
The key advantage that TensorFlow Lite will offer us in our project is the ability to train our Deep Learning model prior to loading it onto the mobile application. This step is completely optional, so a fresh model can still be trained from scratch on the mobile device as it would be done in a standard TensorFlow project. However, such a model may be unable to achieve high accuracy as Deep Learning typically reach their full potential only when trained with massive amounts of data. For this reason, we will first train the model on an sEMG dataset. After loading this trained model onto the mobile device, the model will only need to be finetuned with the user's sEMG data.

We will train the model using the Ninapro DB5 dataset, presented by Atzori et al., before saving it on the mobile device [14]. This dataset has several characteristics which make it desirable to use in mMyoHMI:

- This dataset contains sEMG data for 10 subjects, each of varying height, weight, and gender, so the neural network will have a wide breadth of data to learn from.
- The dataset is publicly available which is essential if we plan to make mMyoHMI open to any user.
- Data for 52 gestures are recorded in the dataset, giving us the freedom to select which gestures we wish to classify. All gestures are shown in Figure 12.
- Each subject was required to wear two Myo Armbands on the right arm, with one placed closer to the elbow. Because users of mMyoHMI may wear the armband on

different parts of the forearm, this makes it critical for the model to learn all possible placements.

Figure 12. Ninapro Dataset



DB5 from the Ninapro Dataset contains sEMG data for 52 gestures excluding Rest; Image Credit @ Atzori et al. [14]

Results

Gesture Recognition Using Custom Sensor

We were able to read EMG data from 4 different channels at 300Hz, as suggested in Table 1. If desired, we would even be able to increase the sampling rate to ~355Hz. Regardless of how many channels are transmitted via the BLE module, the number of total bytes transmitted per second cannot exceed ~1.5kHz.

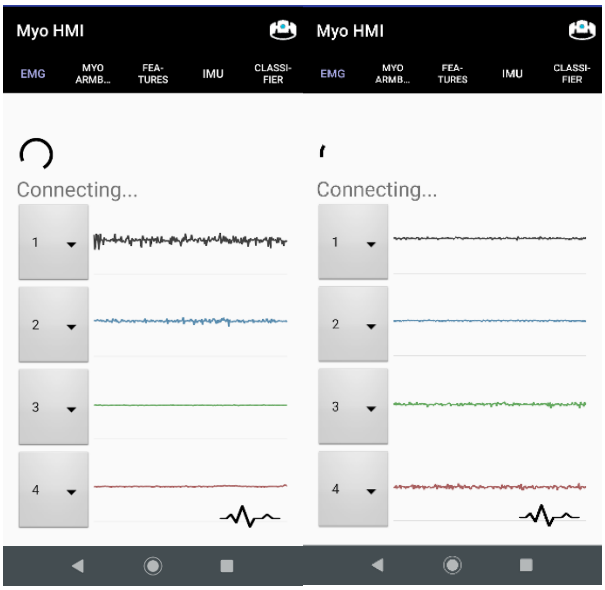
Table 1. Maximum Sampling Rate at which data loss does not occur

# of Channels	Maximum Sampling Rate (Hz)
1	1487.2
2	716.7
3	474.4
4	355.4

As more channels are used, the sampling rate of each channel must be reduced accordingly to avoid surpassing device bandwidth.

As seen in Figure 13, the EMG signals displayed by the app are as expected. When performing a Wave-Out, it would be appropriate for two of the muscle groups to be completely at rest while the others are engaged. On the contrary if a Wave-In were to be performed, we can see that the muscle groups at rest for a Wave-Out are now going to be engaged. This is perfectly reasonable as the two gestures are opposite.

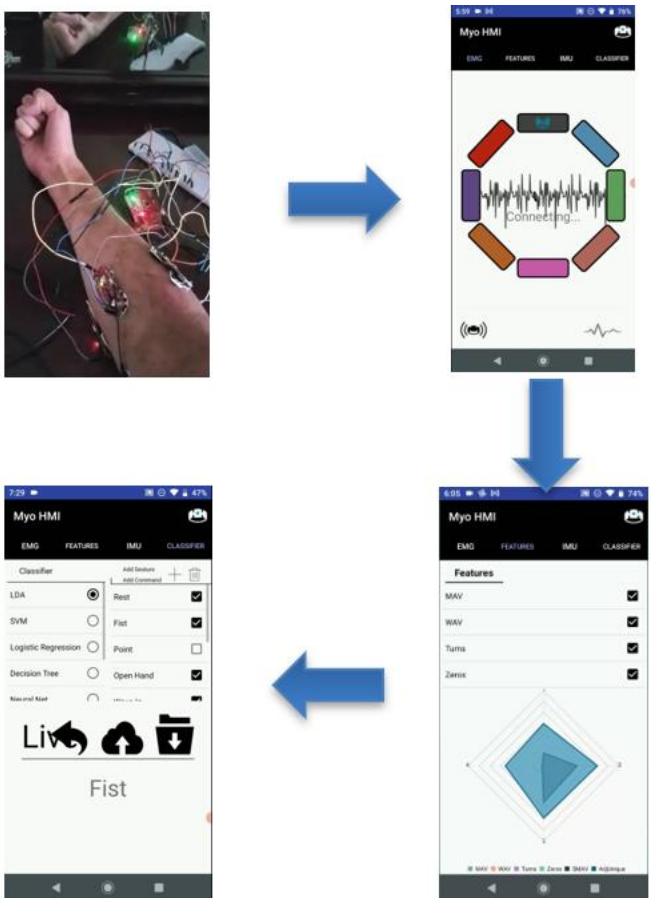
Figure 13. EMG Data for a Wave-Out and Wave-In



Gestures are differentiable based on their unique sEMG behavior such as the Wave-Out (Left) and the Wave-In (Right).

After verifying the integrity of the EMG signals being received, we then proceeded to test mMyoHMI's classifier with only 4 channels of EMG data. Figure 14 shows that gestures can be predicted successfully even with the 4-channel setup. With the simple LDA classifier, the classifier predicted gestures from a set of 6 with a fair degree of confidence. Only during transitions between gestures, would the classifier switch back and forth between different gestures.

Figure 14. Prediction Flow



Gestures can be predicted by collecting raw sEMG data, extracting its features, and feeding them into a classifier.

Gesture Recognition Using CNN

To determine the performance of the deep learning model, we recorded quantifiable metrics of the model as it makes predictions on test sEMG data. These metrics include the models's testing accuracy, loss, and training latency. Experimentation was performed on two gestures, Fist and Rest, for 10 trials. The results of this experiment are shown in Table 2.

Table 2. Deep Learning Model Metrics

Trial #	Accuracy (%)	Loss	Latency (s)
1	95.5	1.7395839	1.949
2	96.0	1.4582085	1.893
3	96.5	1.6551415	2.019
4	95.0	2.5115694	1.928
5	94.5	3.0840187	1.902
6	95.0	1.9385860	1.921
7	97.5	1.4048695	2.148
8	97.0	1.5835839	1.930
9	98.5	1.5937592	1.987
10	97.0	1.8494085	1.941
Average	96.25	1.88187291	1.9618

Metrics for binary gesture recognition suggest deep learning can be used in real-time setting

The testing accuracy is defined by the number of correct predictions the model makes on a set of test data, where 100% accuracy represents the ideal scenario in which all predictions match the intended gesture. As shown above, the average testing accuracy for binary classification was 96.25% while all trials exhibited accuracies above 90%. This supports the fact that a fine-tuned deep learning model can be used to accurately predict gestures, though its accuracy when predicting many gestures depends heavily on the training data.

The loss defines how poorly the model performs with a set of weights after each training iteration. As the model completes each iteration, or epoch, the weights will be optimized to

reduce the model's loss. Therefore, a low loss is desirable as it indicates that the model has been sufficiently tuned to classify the data it was trained on. Note that caution should be taken to avoid *overfitting* the model. In such a case, the loss would be minimal, though the model would not be able to generalize enough to make correct predictions on anything except for the data it was trained on. Table 2 demonstrates that losses above 2 generally exhibited slightly lower accuracies, inidcating that these models were slightly underfitted. Note that, while high accuracies tend to be associated with lower loss values, losses extremely close to zero would likely indicate overfitting. Nevertheless, we did not encounter overfitting during our trials as our number of epochs was not excessive.

The training latency represent the amount of time it takes for the model to be finetuned after the app has collected all training data from the user. It is crucial to minimize the training latency of the model especially since the model will be used in a real-time setting. While training a network would typically range from hours to days, simply finetuning an already-trained model will require far less time. Out of the metrics we assessed, latency is the most significant as users must be able to view predictions in the same session that the training data is collected, otherwise the experimental apparatus would change and a new model would need to be finetuned. Fortunately, our models only took around two seconds to finetune, meaning that predictions can be made almost immediately after finetuning the model.

Discussion

While many of the computationally expensive operations such as training a network are done offline on a computer, the mobile device may still exhibit some latency when inferring on complex neural networks with several layers. Fortunately, we can perform these computations in a cloud server rather than performing them locally. Unlike our local device, a cloud-server can have theoretically infinite computing power, meaning that classification can occur rapidly. Even then, classification may still exhibit latency due to the increased latency of transmitting the data to a global server as opposed to a local device.

Another benefit of expanding the app's infrastructure to the cloud is that neural networks may be created on-demand by the user rather than having to train a separate model prior to using the app. This would allow the user to decide on the structure of the neural network including input size, number of layers, number of neural units per layer, output size, among many other parameters.

We were able to confirm the compatibility of mMyoHMI with four channels of EMG data and when using a CNN classifier. However, the lack of hardware we currently possess limits us from performing rigorous testing on the flexibility of the mMyoHMI app. To truly conclude that the app is compatible with ANY number of channels, we would have to test a peripheral device with more channels of data, such as an EMG grid. We would also be able to demonstrate the true effectiveness of the CNN classifier when used on HD data.

Conclusion

Throughout this project, we have focused our efforts on the development of the mMyoHMI app which we have proven to work with half the number of channels that was used in previous works. We have also built a peripheral device whose attributes can be configured in various ways. For instance, we can change the number of channels that the peripheral can sample, the rate at which data is sampled and transmitted, and whether the data is sampled online or offline. The mMyoHMI app, previously only compatible with the 8-Channel Myo Armband, is now confirmed to fully support a 4-channel device as well, with a varying sample rate whose threshold is dependent upon the number of channels being sent. Furthermore, the app still supports the Myo Armband, so the number of channels being sent to the app is no longer assumed to be a fixed value. We have even successfully implemented and used an additional deep learning classifier which has potential to classify HD sEMG with high accuracy. With more peripheral devices to test the app with, we would be able to conclude that the app can support a wide range of devices as per our ultimate goal.

References

[1] I. Donovan, K. Valenzuela, A. Ortiz, S. Dusheyko, H. Jiang, K. Okada, X. Zhang, "MyoHMI: A low-cost and flexible platform for developing real-time human machine interface for myoelectric controlled applications," 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2016, pp. 004495- 004500, doi: 10.1109/SMC.2016.7844940.

[2] J. E. Muñoz, T. Paulino, H. Vasanth and K. Baras, "PhysioVR: A novel mobile virtual reality framework for physiological computing," 2016 IEEE 18th International Conference on e-Health Networking, Applications and Services (Healthcom), 2016, pp. 1-6, doi: 10.1109/HealthCom.2016.7749512.

[3] S. Bhavani, K. L. Krishna, R. B. Y. Reddy and T. Geethika, "A Low Cost Bionic Arm Based on Electromyography Sensor," 2020 7th International Conference on Signal Processing and Integrated Networks (SPIN), 2020, pp. 896-901, doi: 10.1109/SPIN48934.2020.9070936.

[4] A. Naber, Y. Karayiannidis and M. Ortiz-Catalan, "Universal, Open Source, Myoelectric Interface for Assistive Devices," 2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV), 2018, pp. 1585-1589, doi: 10.1109/ICARCV.2018.8581344.

[5] E. Ceolini, G. Taverni, L. Khacef, M. Payvand and E. Donati, "Sensor fusion using EMG and vision for hand gesture classification in mobile applications," 2019 IEEE Biomedical

Circuits and Systems Conference (BioCAS), 2019, pp. 1-4, doi: 10.1109/BIOCAS.2019.8919210.

[6] E. Altan, K. Pehlivan and E. Kaplanoğlu, "Comparison of EMG Based Finger Motion Classification Algorithms," 2019 27th Signal Processing and Communications Applications Conference (SIU), 2019, pp. 1-4, doi: 10.1109/SIU.2019.8806331.

[7] Ortiz-Catalan, M., Brånemark, R. & Håkansson, B. BioPatRec: A modular research platform for the control of artificial limbs based on pattern recognition algorithms. Source Code Biol Med 8, 11 (2013).

[8] E. Mastinu, B. Håkansson and M. Ortiz-Catalan, "Low-cost, open source bioelectric signal acquisition system," 2017 IEEE 14th International Conference on Wearable and Implantable Body Sensor Networks (BSN), 2017, pp. 19-22, doi: 10.1109/BSN.2017.7935997.

[9] G. Bhat, R. Deb and U. Y. Ogras, "OpenHealth: Open-Source Platform for Wearable Health Monitoring," in IEEE Design & Test, vol. 36, no. 5, pp. 27-34, Oct. 2019, doi: 10.1109/MDAT.2019.2906110.

[10] Y. Omama et al., "Surface EMG Classification of Basic Hand Movement," 2019 Fifth International Conference on Advances in Biomedical Engineering (ICABME), 2019, pp. 1-4, doi: 10.1109/ICABME47164.2019.8940352.

[11] P. Zhu, H. Zhou, S. Cao, P. Yang and S. Xue, "Control with Gestures: A Hand Gesture Recognition System Using Off-the-Shelf Smartwatch," 2018 4th International Conference on Big Data Computing and Communications (BIGCOM), 2018, pp. 72-77, doi: 10.1109/BIGCOM.2018.00018.

[12] Deng J., Niu J., Wang K., Xie L., Yang G. (2018) Discriminant Analysis Based EMG Pattern Recognition for Hand Function Rehabilitation. In: Perego P., Rahmani A., TaheriNejad N. (eds) Wireless Mobile Communication and Healthcare. MobiHealth 2017. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 247. Springer, Cham.

[13] J. Zea, M. E. Benalcázar, L. I. Barona Lôpez and Á. L. Valdivieso Caraguay, "An Open-Source Data Acquisition and Manual Segmentation System for Hand Gesture Recognition based on EMG," 2021 IEEE Fifth Ecuador Technical Chapters Meeting (ETCM), 2021, pp. 1-6, doi: 10.1109/ETCM53643.2021.9590811.

[14] M. Atzori et al., "Building the Ninapro database: A resource for the biorobotics community," 2012 4th IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics (BioRob), 2012, pp. 1258-1265, doi: 10.1109/BioRob.2012.6290287.

[15] W. Geng, Y. Du, W. Jin, W. Wei, Y. Hu, and J. Li, “Gesture recognition by instantaneous surface emg images,” Scientific reports, vol. 6, no. 1, pp. 1–8, 2016.

[16] F. Quivira, T. Koike-Akino, Y. Wang, and D. Erdogmus, “Translating semg signals to continuous hand poses using recurrent neural networks,” in 2018 IEEE EMBS International Conference on Biomedical & Health Informatics (BHI). IEEE, 2018, pp. 166–169.

[17] U. Coˆte´-Allard, C. L. Fall, A. Campeau-Lecours, C. Gosselin, F. Lavio- lette, and B. Gosselin, “Transfer learning for semg hand gestures recog- nition using convolutional

neural networks,” in 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC). IEEE, 2017, pp. 1663–1668.

[18] Shen, S., Gu, K., Chen, XR. et al. Gesture Recognition Through sEMG with Wearable Device Based on Deep Learning. Mob. Networks Appl., vol. 25, pp. 2447–2458, 2020.

[19] Nastuta, Andrei Vasile, and Catalin Agheorghiesei. "Monitoring Hand Gesture and Effort Using a Low-Cost Open-Source Microcontroller System Coupled with Force Sensitive Resistors and Electromyography Sensors." International Conference on Global Research and Education. Springer, Cham, 2017.

[20] Fang, Chaoming, et al. "EMG-centered multisensory based technologies for pattern recognition in rehabilitation: state of the art and challenges." Biosensors 10.8 (2020): 85.

[21] Nabian, Mohsen, et al. "An open-source feature extraction tool for the analysis of peripheral physiological data." IEEE Journal of Translational Engineering in Health and Medicine 6 (2018): 1-11.

[22] Sgambato, Bruno G., and Gabriela Castellano. "Development of a Haptic Feedback Device Based on Electromyography Signals and an Arduino."

[23] Morón, J., DiProva, T., Cochrane, J. R., Ahn, I. S., & Lu, Y. (2018, August). EMG-based hand gesture control system for robotics. In 2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS) (pp. 664-667). IEEE.

[24] Akhmadeev, K., Rampone, E., Yu, T., Aoustin, Y., & Le Carpentier, E. (2017). A testing system for a real-time gesture classification using surface EMG. IFAC-PapersOnLine, 50(1), 11498-11503.

[25] Vasylkiv, Y., Neshati, A., Sakamoto, Y., Gomez, R., Nakamura, K., & Irani, P. (2019, January). Smart Home Interactions for People with Reduced Hand Mobility Using Subtle EMG-Signal Gestures. In ITCH (pp. 436-443).

[26] J. S. Artal-Sevil, A. Acón, J. L. Montañés and J. A. Domínguez, "Design of a Low-Cost Robotic Arm controlled by Surface EMG Sensors," 2018 XIII Technologies Applied to Electronics Teaching Conference (TAEE), 2018, pp. 1-8, doi: 10.1109/TAEE.2018.8476126.

[27] Pizzolato et al., Comparison of Six Electromyography Acquisition Setups on Hand Movement Classification Tasks, Plos One 2017

[28] Atzori M, Gijsberts A, Castellini C, Caputo B, Hager AGM, Elsig S, et al. Electromyography data for non-invasive naturally-controlled robotic hand prostheses. Scientific Data. 2014;1. pmid:25977804

[29] X. Zhang, H. Huang and Q. Yang, "Implementing an FPGA system for real-time intent recognition for prosthetic legs," DAC Design Automation Conference 2012, 2012, pp. 169-175.

[30] Kang K, Shin H-C. EMG Based Gesture Recognition Using the Unbiased Difference Power. Applied Sciences. 2021; 11(4):1526.

[31] Kutafina, E.; Laukamp, D.; Bettermann, R.; Schroeder, U.; Jonas, S.M. Wearable Sensors for eLearning of Manual Tasks: Using Forearm EMG in Hand Hygiene Training. Sensors 2016, 16, 1221.

Appendix/Appendices

Appendix A: Arduino Code for Sampling from ADC

```
void TimerInit() {
  cli();//stop interrupts

  //set timer1 interrupt at 300Hz
  TCCR1A = 0;// set entire TCCR1A register to 0
  TCCR1B = 0;// same for TCCR1B
  TCNT1  = 0;//initialize counter value to 0
  // set compare match register for 300hz increments
  OCR1A = 51;//300Hz
  // turn on CTC mode
  TCCR1B |= (1 << WGM12);
  // Set CS12 and CS10 bits for 1024 prescaler
  TCCR1B |= (1 << CS12) | (1 << CS10);
  // enable timer compare interrupt
  TIMSK1 |= (1 << OCIE1A);

  sei();//allow interrupts

}

// the setup routine runs once when you press reset:
void setup() {
  TimerInit();
  // initialize serial communication at 115200 bits per second:
  Serial.begin(115200);
  //pinMode(13, OUTPUT);    // sets the digital pin 13 as output for logic analyzing
}

ISR(TIMER1_COMPA_vect){//timer1 interrupt at 300Hz
  //int8_t sensorValue = (analogRead(A0) >> 2) - 125;   //No Amplification
  //int8_t sensorValue = (analogRead(A0) >> 1) - 250; //x2 Amplification
  int8_t sensorValue0 = analogRead(A0)-500;         //x4 Amplification
  int8_t sensorValue1 = analogRead(A1)-500;         //x4 Amplification
  int8_t sensorValue2 = analogRead(A2)-500;         //x4 Amplification
  int8_t sensorValue3 = analogRead(A3)-500;         //x4 Amplification
  //digitalWrite(13, !digitalRead(13));
  Serial.write(sensorValue0);
  Serial.write(sensorValue1);
  Serial.write(sensorValue2);
  Serial.write(sensorValue3);
}


// the loop routine runs over and over again forever:
void loop() {

}
```

Appendix B: Arduino Code for Sampling from Flash Memory

```
const int8_t rest[][4] PROGMEM = {{0,-2,2,-1}};                         //All samples not shown
const int8_t fist[][4]  PROGMEM = {{17,-2,5,2}};                        //All samples not shown
const int8_t open_hand[][4] PROGMEM = {{7,-4,-2,-2}};  //All samples not shown
const int8_t wave_in[][4] PROGMEM = {{-33,8,3,3}};      //All samples not shown
const int8_t wave_out[][4] PROGMEM = {{7,-3,-1,5}};     //All samples not shown
const int8_t thumbs_up[][4] PROGMEM = {{-1,-3,1,-2}};  //All samples not shown

int sample[] = {0,0,0,0,0,0};

void TimerInit() {
  cli();//stop interrupts

  //set timer1 interrupt at 300Hz
  TCCR1A = 0;// set entire TCCR1A register to 0
  TCCR1B = 0;// same for TCCR1B
  TCNT1  = 0;//initialize counter value to 0
  // set compare match register for 300hz increments
  OCR1A = 51;//300Hz
  // turn on CTC mode
  TCCR1B |= (1 << WGM12);
  // Set CS12 and CS10 bits for 1024 prescaler
  TCCR1B |= (1 << CS12) | (1 << CS10);
  // enable timer compare interrupt
  TIMSK1 |= (1 << OCIE1A);

  sei();//allow interrupts

}

// the setup routine runs once when you press reset:
void setup() {
  TimerInit();
  // initialize serial communication at 115200 bits per second:
  Serial.begin(115200);
  //pinMode(13, OUTPUT);    // sets the digital pin 13 as output for logic analyzing
  pinMode(2, INPUT_PULLUP); // Pulls Floating pin up to VDD
  pinMode(3, INPUT_PULLUP); // Pulls Floating pin up to VDD
  pinMode(4, INPUT_PULLUP); // Pulls Floating pin up to VDD
}

ISR(TIMER1_COMPA_vect){//timer1 interrupt at 300Hz
  // Rest
  if(digitalRead(2) && digitalRead(3) && digitalRead(4)) {
    Serial.write(pgm_read_byte(&(rest[sample[0]][0])));
    Serial.write(pgm_read_byte(&(rest[sample[0]][1])));
    Serial.write(pgm_read_byte(&(rest[sample[0]][2])));
    Serial.write(pgm_read_byte(&(rest[sample[0]][3])));

    if(sample[0] >= (sizeof(rest)/sizeof(rest[0]))) {
      sample[0] = 0;
    } else {
      sample[0]++;
    }
  }
  // Fist
```

```
else if(!digitalRead(2) && digitalRead(3) && digitalRead(4)) {
  Serial.write(pgm_read_byte(&(fist[sample[1]][0])));
  Serial.write(pgm_read_byte(&(fist[sample[1]][1])));
  Serial.write(pgm_read_byte(&(fist[sample[1]][2])));
  Serial.write(pgm_read_byte(&(fist[sample[1]][3])));

  if(sample[1] >= (sizeof(fist)/sizeof(fist[0]))) {
    sample[1] = 0;
  } else {
    sample[1]++;
  }
}
// Wave-In
else if(digitalRead(2) && !digitalRead(3) && digitalRead(4)) {
  Serial.write(pgm_read_byte(&(wave_in[sample[2]][0])));
  Serial.write(pgm_read_byte(&(wave_in[sample[2]][1])));
  Serial.write(pgm_read_byte(&(wave_in[sample[2]][2])));
  Serial.write(pgm_read_byte(&(wave_in[sample[2]][3])));

  if(sample[2] >= (sizeof(wave_in)/sizeof(wave_in[0]))) {
    sample[2] = 0;
  } else {
    sample[2]++;
  }
}
// Wave-Out
else if(digitalRead(2) && digitalRead(3) && !digitalRead(4)) {
  Serial.write(pgm_read_byte(&(wave_out[sample[3]][0])));
  Serial.write(pgm_read_byte(&(wave_out[sample[3]][1])));
  Serial.write(pgm_read_byte(&(wave_out[sample[3]][2])));
  Serial.write(pgm_read_byte(&(wave_out[sample[3]][3])));

  if(sample[3] >= (sizeof(wave_out)/sizeof(wave_out[0]))) {
    sample[3] = 0;
  } else {
    sample[3]++;
  }
}
// Open Hand
else if(!digitalRead(2) && !digitalRead(3) && digitalRead(4)) {
  Serial.write(pgm_read_byte(&(open_hand[sample[4]][0])));
  Serial.write(pgm_read_byte(&(open_hand[sample[4]][1])));
  Serial.write(pgm_read_byte(&(open_hand[sample[4]][2])));
  Serial.write(pgm_read_byte(&(open_hand[sample[4]][3])));

  if(sample[4] >= (sizeof(open_hand)/sizeof(open_hand[0]))) {
    sample[4] = 0;
  } else {
    sample[4]++;
  }
}
// Thumbs Up
else if(digitalRead(2) && !digitalRead(3) && !digitalRead(4)) {
  Serial.write(pgm_read_byte(&(thumbs_up[sample[5]][0])));
  Serial.write(pgm_read_byte(&(thumbs_up[sample[5]][1])));
  Serial.write(pgm_read_byte(&(thumbs_up[sample[5]][2])));
  Serial.write(pgm_read_byte(&(thumbs_up[sample[5]][3])));
```

```
    if(sample[5] >= (sizeof(thumbs_up)/sizeof(thumbs_up[0]))) {
      sample[5] = 0;
    } else {
      sample[5]++;
    }
  }
}


// the loop routine runs over and over again forever:
void loop() {

}
```