Finger Movement Classification via Machine Learning using EMG Armband for 3D Printed Robotic Hand

THESIS

SUBMITTED TO THE FACULTY OF THE

UNIVERSITY OF MINNESOTA

BY

SHAYAN ALI BHATTI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

ADVISOR: PROFESSOR DESINENI SUBBARAM NAIDU, PHD

SEPTEMBER 2019

Dedication

To my parents who offered support, guidance, and encouragement.

Acknowledgements

As an Industrial Electronics undergraduate from Pakistan, it was an honor for me to get admission at the University of Minnesota Duluth, USA. I am grateful to the Electrical Engineering Department at the University of Minnesota Duluth (UMD) for providing me the opportunity and funding for my masters studies. As a graduate student, I had the opportunity to work as Teaching Assistant for many courses which gave me the valuable experience of being a teacher and mentor to many students.

I would like to thank my advisor, Dr. Naidu, for his patience and encouragement. Machine Learning is considered somewhat out of the scope of Electrical Engineering and required me to study via online courses and books which took a fair amount of time but Dr. Naidu was extremely encouraging of the idea of incorporating Machine Learning in my thesis to keep up-to-date with the latest developments in engineering. I consider it an honor to have a kind person as him as my thesis advisor.

I am also grateful to Dr. Eleazar Leal in Computer Science Department at UMD who was extremely kind and helpful with his expert advice on issues regarding Machine Learning.

I would like to thank Dr. Imran Hayee in Electrical Engineering Department at UMD who has been extremely helpful to me throughout my masters studies in USA.

Lastly, I would like to thank Natural Resources Research Institute (NRRI), Duluth, for their help regarding 3D printing of the robotic hand. Without NRRI's help with their state-of-the-art 3D printers, the 3D print of robotic hand would not have been possible.

Abstract

Millions of people lose their limbs due to accidents, infections and/or wars. While prosthetics are the best solution for amputees, designing autonomous prosthetic hand that can perform major operations is a complicated task and thus the prosthetic hands that are designed are very expensive and also a bit heavy.

The biggest challenge in designing a prosthetic hand is the classification of EMG signals generated by neurons in the arm to distinguish finger movements. These EMG signals vary in strength from person to person and from movement to movement.

This thesis proposes a computationally efficient way that uses Machine Learning to classify 5 and 12 finger movements from EMG signals captured by a device called “Myo Gesture Control Armband”. Further, an ergonomic design of robotic hand is also presented that is small, lightweight and cheap, designed using a 3D printer.

Table of Contents

List of Tables

List of Figures

Abbreviations

EMG - ElectroMyoGraphy

LDA - Linear Discriminant Analysis

KNN - K-Nearest Neighbors

HMM - Hidden Markov Models

ANFIS - Adaptive Neuro Fuzzy Inference Systems

SVM - Support Vector Machines

AI - Artificial Intelligence

ReLU - Rectified Linear Unit

CNN - Convolutional Neural Network

RNN - Recurrent Neural Network

ABS - Acrylonitrile Butadiene Styrene

CHAPTER 1

INTRODUCTION

1.1 Background

According to the information provided by "Amputee-coalition.org" for Limb Loss Statistics, which cites the article, "Estimating the Prevalence of Limb Loss in the United States: 2005 to 2050" [1], there are nearly 2 million people living with limb loss in the United States of America. Approximately 185,000 amputations occur each year in USA. The reasons for amputations include accidents, infections, diseases such as diabetes, trauma and cancer [1]. The wars in Iraq and Afghanistan substantially increased the number of amputees. About 80% of amputees use prosthetic devices [2] and around 30-50% of amputees are using Myoelectric controlled devices [3].

While prosthetic legs for amputees are common, building a prosthetic autonomous hand is a lot more complicated particularly because of the tasks we do using our fingers and hand are more complicated than the foot. For this purpose, classification of Electromyography (EMG) signals generated by neurons in arm is extremely important so that the prosthetic hand can do the desired task.

EMG measures response of a muscle or electrical activity in response to a nerve's stimulation of the muscle. There are a lot of ways to acquire EMG signals, such as by inserting a needle electrode directly into a muscle or by using surface EMG (sEMG) sensors. An example is shown in Figure 1.1 below:

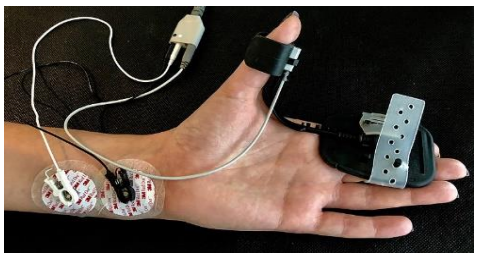

Figure 1.1: Surface EMG sensor. Picture by Paul Anthony Stewart published under Creative Commons Attribution-Share Alike 4.0 International license on Wikipedia Commons

These EMG values are very random, vary from person-to-person and need application of efficient algorithms for extraction of meaningful information from them. Hence, these signals must be processed correctly and then features are extracted from them.

1.2 Previous Work

EMG Classification has been a hot topic, many research groups have worked on it and are still doing research on it. Different approaches have been used in the past including different sensors, classification techniques to do EMG classification.

In [4], frequency domain analysis of EMG signals is discussed. The authors applied Fast Fourier Transform on signals coming from 3 sEMG sensors and fed into multi-layer neural network for identification of 4 movements namely, thumb and index finger's flexion and extension giving above 57% accuracy. In [5], 16 time-domain features were extracted from sEMG sensors and Adaptive Neuro-Fuzzy Inference System (ANFIS) was implemented for classification of 5 finger (thumb, index, middle, ring and pinky) extension and 5 flexion movements were classified with average 72% accuracy. A combination of time and frequency domain features was extracted in [6] to develop a wearing independent hand movement classification algorithm. The authors implemented a light-weight random forest for classifying 15 hand and finger movements with 91.47% accuracy. The authors in [7] used a state-of-the-art Bagnoli Desktop EMG system for EMG data acquisition and used time-domain features like mean, standard deviation and skewness. They applied Linear Discriminant Analysis (LDA) and K-Nearest Neighbors on EMG features to get a fast and high accuracy classification on 10 finger movements using LDA. Deep Learning involving a ConvNet has been also employed for this task achieving 98.31% accuracy for 6 gestures while 69.89% accuracy for 18 gestures over 10 participants [8]. Other machine learning algorithms have also been used by researchers for EMG classification such as KNN [9], SVM [10] and decision trees and Hidden Markov Models (HMM) [11].

All above mentioned methods have their distinct features but some disadvantages of them include expensive hardware as in [7], extensive training methods, over-expensive

computations using deep convolutional neural networks or lesser number of recognized movements.

In the last decade, the EMG data acquisition technology has also seen some advancements. In the past, there used to be a needle electrode based EMG acquisition method, that required a needle to be inserted in the hand of the subject. However, with advancements in this technology, we now have surface EMG (sEMG) sensors that do not require the subject to go through the pain of needle insertion. Instead, the EMG signals are measured with the help of sEMG sensors. One such product is the "Myo Gesture Control Armband", designed by a Canadian company "Thalmic Labs". This study used the "Myo Gesture Control Armband", more details of this armband are provided in the chapter 3.

It is also worthy to mention that this idea of real-time gesture classification using surface EMG Myo armband to control robotics hand has been an active research area. In [12], three hand movements were classified and performed custom designed robotic hand. Six hand movements were classified using the same armband and imitated by robotic hand in [13]. In [14], Reinforcement Learning was used to teach a prosthetic robotic arm. However, Johns Hopkins Applied Physics Laboratory made a breakthrough in their $120-million research project [15], in which they designed a modular prosthetic arm for amputees using 2 Myo armbands for signal processing. This prosthetic allowed an amputee to do most of the movements that a human arm can perform and required a surgical procedure to fit the arm.

1.3 Objectives

There are two main objectives of this study. First objective is to design a machine learning based algorithm that can process EMG signals and classify finger movements from it. The second objective is to design a realistic 3D printed robotic hand that can imitate those finger movements.

1.4 Proposed Methodology

In this study, a single hidden layer neural network based approach for EMG classification is proposed that is computationally efficient. EMG signals are acquired from Myo EMG Control Armband shown in Figure 1.2. We classify 12 vital finger movements via EMG signals and a signal is sent to Arduino via Bluetooth that actuates motors on a 3D printed robotic hand to imitate the classified finger movement.

Figure 1.2: Myo Gesture Control Armband

The robotic hand designed, takes inspiration from InMoov's open source design [16] of 3D printed robotic hand but instead of a large robotic hand with motors above the wrist, the proposed design uses micro servo motors that easily fit in the palm of robotic hand, making it a compact, realistic and light-weight robotic hand.

CHAPTER 2

Introduction to Machine Learning and Neural Networks

In this chapter, a brief introduction of Machine Learning and Neural networks is provided. Machine Learning and Deep Learning (based on neural networks) fall under the umbrella of Artificial Intelligence as depicted in Figure 2.1.

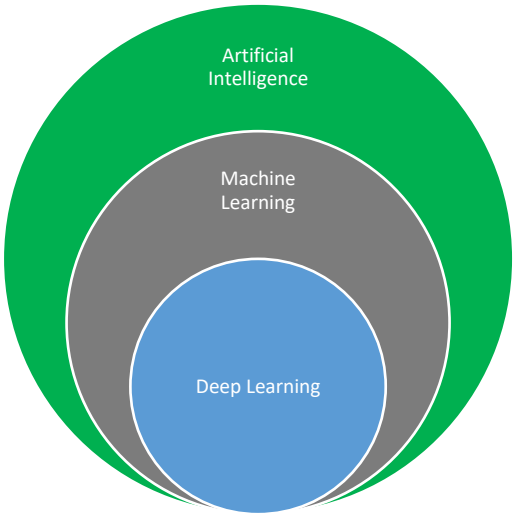

Figure 2.1 : Advent of Artificial Intelligence

2.1 Machine Learning

Arthur Samuel was the pioneer of Artificial Intelligence (AI) and coined the term machine learning. In the words of Tom M. Mitchell, an American computer scientist known for his contributions to the field of AI:

> "A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P* if its performance at tasks in *T*, as measured by *P*, improves with experience *E".[17]*

Simply put, in machine learning, machine/software is made to learn by making mistakes. There are 4 categories of machine learning tasks:

1) Supervised learning.
2) Unsupervised learning.
3) Self-supervised learning.
4) Reinforcement learning.

2.1.1 Supervised Learning

In supervised learning, the machine learning model learns from a set of data that contains inputs as well as the desired output. Supervised learning tasks usually consists of classification or regression [18]. For example, a machine learning model can be made to predict stock prices or train image classification model with images of dog, then the model classifies if it is a picture of "dog" or not. Hence input data is "**labeled**" in supervised learning.

Popular algorithms used for supervised learning are Support Vector Machines, Linear regression, Logistic regression, Naive Bayes, Linear Discriminant Analysis, Decision Trees, k-Nearest Neighbor and Neural Networks. Figure 2.2 shows a simple decision tree.

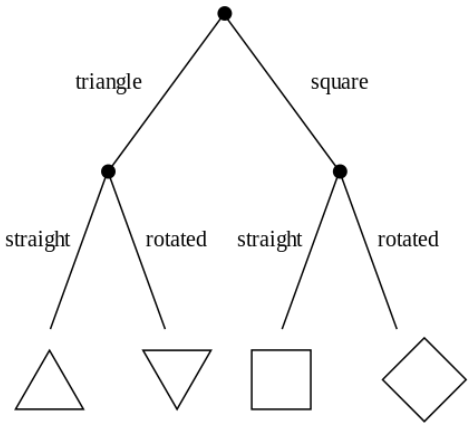

Figure 2.2 : Decision Tree. Photograph by Eviatar Bach under license Creative Commons CC0 1.0 on Wikipedia

2.1.2 Unsupervised Learning

In unsupervised learning, the model learns from "**unlabeled**" data i.e. the model does not know the output of the input it is being trained with. Unsupervised Learning is mostly used

for clustering data although it is not restricted to it. Unsupervised learning is also used for anomaly detection.

Commonly used unsupervised learning algorithms are k-means clustering, DBSCAN, Hierarchical clustering etc. Figure 2.3 shows an example of clustering.

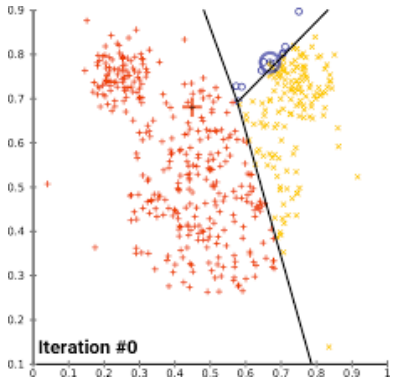

Figure 2.3: An example of data clustered via K-means Clustering. Photograph by Chire, distributed under a GNU Free Documentation License on Wikipedia

2.1.3 Self-supervised Learning

Self-supervised learning is a special type of supervised learning that learns without labels. Labels are still involved but they are learnt from the input data using a heuristic algorithm. Autoencoders are a type of self-supervised learning, in which the generated targets are the inputs, unmodified [19].

2.1.4 Reinforcement Learning

In reinforcement learning, a software agent is made to take actions in an environment with the target to maximize the cumulative reward [20]. Google DeepMind's designed algorithm AlphaGo beat the champion of Chinese game Go and is based on reinforcement learning. Google DeepMind's Go has also mastered Chess and Shogi [21].

2.2 Artificial Neural Networks

Artificial neural networks take their inspiration from biological neural networks in which the dendrites take input and send to cell body where its processed and output signal is sent via axon terminal [22]. A biological neuron is shown in Figure 2.4 on next page.

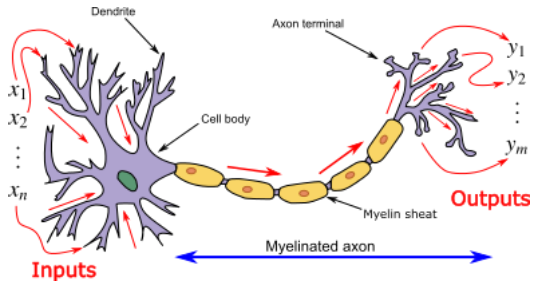

Figure 2.4 : A biological neuron. Picture by Prof. Loc Vu-Quoc distributed under Creative Commons Attribution-Share Alike 3.0 Unported on Wikipedia Commons

Similarly, a neuron in artificial neural network also takes inputs, processes them by applying an activation function and finally we get an output. A neuron is shown in Figure 2.5.

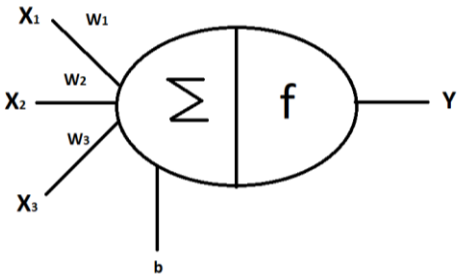

Figure 2.5 : A neuron in Artificial Neural Network (ANN)

In Figure 2.5 above X_1, X_2 and X_3 are the inputs, W_1, W_2 and W_3 are weights, b is bias and Y being the output of neuron. In a single neuron, first all weights are multiplied to their respective inputs and then added in Equation 2.1 as,

$$Z = W * X + b \tag{2.1}$$

where equation 3.1 shows a vectorized representation of a neuron with 'W' being weight vector, 'X' being input vector, 'b' is bias vector and 'Z' is the output [23].

After computing 'Z', an activation function also called non-linearity is applied to it, which is discussed next.

2.2.1 Activation Function

A neuron, as shown in equation 2.1 computes a linear function, but our inputs might not be linear. It means that in order to capture the non-linearity of input, we need to do something.

That is what an activation function / non-linearity does. There are several types of activation functions, four of them are discussed next.

A. Sigmoid or Logistic Activation Function

A sigmoid or logistic activation function [24] takes input and suppresses it to value between 0 and 1. It is mathematically shown in equation 2.2 as follows,

$$A = \frac{1}{1+e^{-Z}} \tag{2.2}$$

where 'A' is the output of activation function and 'Z' is given in Equation 3.1 above.

Sigmoid activation function is used to find probability of events as if the output of sigmoid is greater than 0.5 then event is likely to happen otherwise not. Figure 2.6 shows the output of a sigmoid function.

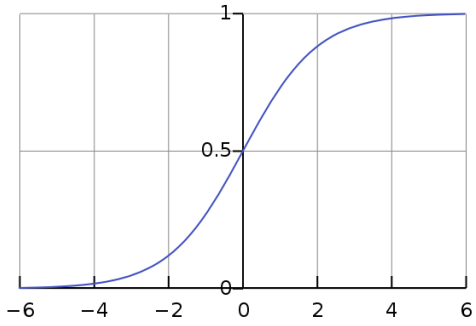

Figure 2.6: Output of sigmoid activation function. Picture taken by Qef made public on Wikipedia Commons

B. Tanh Activation Function

A tanh activation function [24] suppresses the input between -1 and 1. Compared with sigmoid, tanh also gives negative output. It was used extensively for a long time by researchers. Mathematically given in Equation 2.3 as,

$$A = \frac{e^{Z}-e^{-Z}}{e^{Z}+e^{-Z}} \tag{2.3}$$

where 'A' is the output of activation function and 'Z' is given in Equation 3.1 above.

Its output is graphically shown in Figure 2.7 on next page.

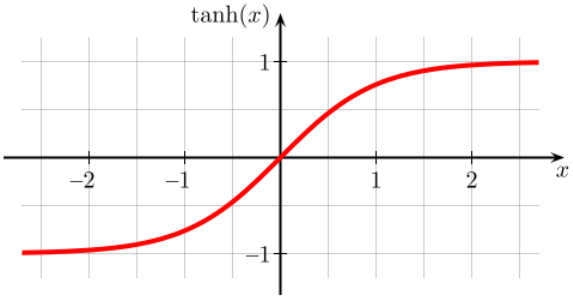

Figure 2.7: Tanh activation function output. Picture taken by Geek3 under Creative Commons Attribution-Share Alike 3.0 Unported license on Wikipedia Commons

C. ReLU Activation Function

ReLU is Rectified Linear Unit activation function [24]. It is the most commonly used activation function for deep learning applications as it has a constant slope from input (0,∞). One disadvantage of ReLU is that it suppresses the negative inputs to 0 output but gives positive input as it is. Mathematically shown in Equation 2.4 as,

$$F(x) = \begin{cases} x \; for \; x > 0 \\ 0 \; for \; x \leq 0 \end{cases} \tag{2.4}$$

where 'x' is the input value and F(x) is output of relu function.

The output of ReLU activation function is graphically shown in Figure 2.8 below:

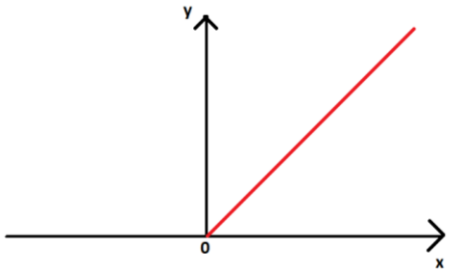

Figure 2.8: ReLU activation function output

The biggest reason of using ReLU for deep learning applications is that unlike sigmoid and tanh activation functions, ReLU does not have saturation region, hence ReLU always has a constant derivative which tanh and sigmoid don't, as their derivative is very small in saturation region.

D. Softmax Activation Function

A softmax activation function [24] is used in the output layer of neural network. It turns numbers into probabilities that sum to 1. It is mathematically given in Equation 2.5 as,

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}} \tag{2.5}$$

where y_i is the probability of an event and $S(y_i)$ is the output of softmax function.

There are other activation functions too like leakyReLU, sReLU, pReLU etc which are not that common but are used based on need.

A full neural network can be seen in Figure 2.9.

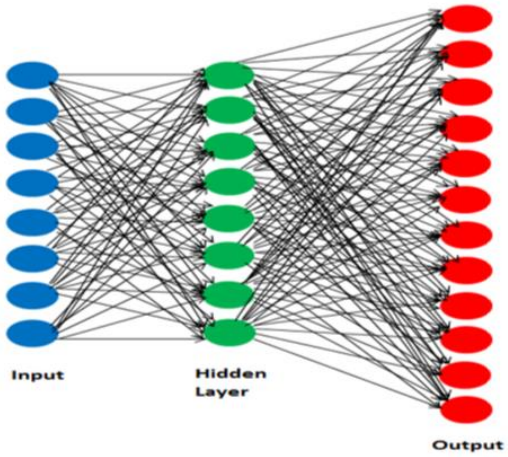

Figure 2.9: A single hidden layer neural network

As seen in Figure 2.9, a neural network has an input layer, and atleast one hidden layer with neurons and an output layer with one or more neurons.

2.3 Types of Neural Networks

There are different types of neural networks. Three are discussed below with applications as follows:

2.3.1 Convolutional Neural Networks (CNN)

In Convolutional Neural networks, we convolute our input with filter(s) to extract features from them. Depending upon application, we can have multiple hidden layers for convoluting our input with the filter. An illustration is shown in Figure 2.10 on next page.

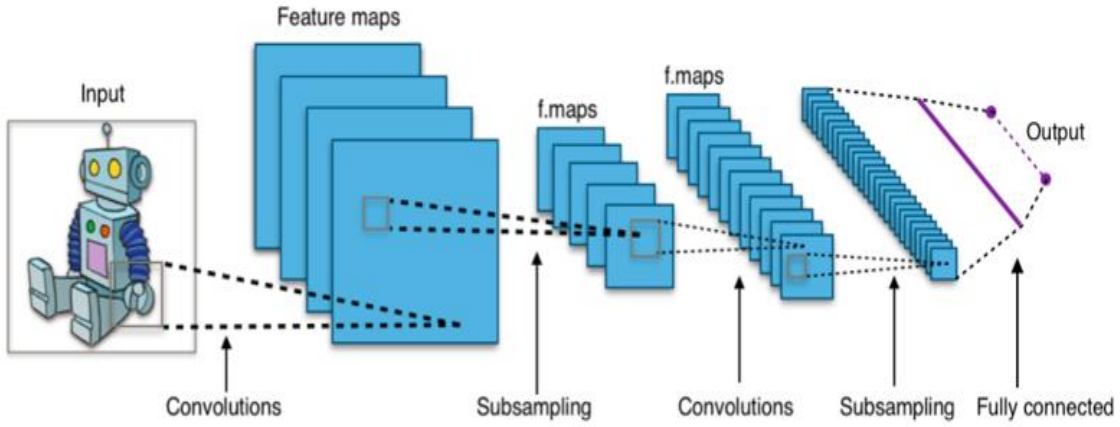

Figure 2.10: A Convolutional Neural Network (CNN). Picture taken by Aphex34 distributed under Creative Commons Attribution-Share Alike 4.0 International license on Wikipedia Commons

The applications of CNN include image [25] and video processing. With the help of CNNs, 96% classification accuracy was achieved over 1.2 million images comprising of 1000 classes. A popular object detection algorithm used for autonomous vehicles and other applications, also uses CNN.

2.3.2 Recurrent Neural Network (RNN)

In Recurrent Neural Networks, connections between nodes form a directed graph along the temporal sequence [26]. It is Illustrated in Figure 2.11 below:

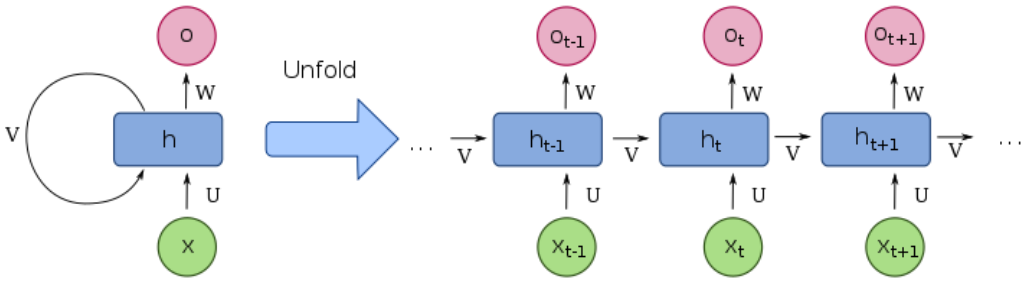

Figure 2.11: Recurrent Neural Network (RNN). Picture by François Deloche distributed under Creative Commons Attribution-Share Alike 4.0 International license on Wikipedia Commons

Recurrent neural networks are used for time-series, natural language processing and speech related tasks because they are good at capturing context of sentence.

2.3.3 AutoEncoders

AutoEncoders are an unsupervised machine learning algorithm. The input and output layers are the same. Autoencoder, tries to make the hidden layers learn the input to successfully recreate it [27]. An autoencoder is shown in Figure 2.12.

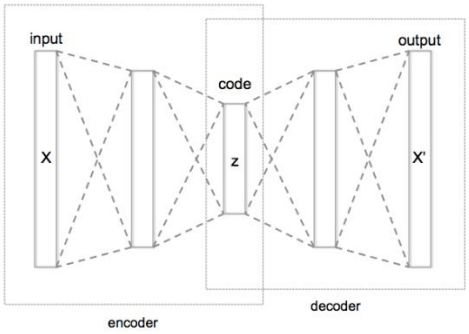

Figure 2.12: An Autoencoder. Picture by Chervinskii released under Creative Commons Attribution-Share Alike 4.0 International license on Wikipedia Commons

AutoEncoders are used for anomaly detection as well as compression purposes.

Besides the above mentioned types of neural networks, there are others such as Generative Adversarial Networks, Long Short Term Models etc.

2.4 Deep Learning

Deep Learning falls under a broad family of machine learning based on Artificial Neural Networks. It can be used for supervised as well as unsupervised learning. Deep Learning is famous for its high performance on image classification, speech processing and natural language processing tasks. The word ‘deep’ comes from depth of layers.

As deep learning is based on neural networks, there is no set formula on how many hidden layers should the neural network have, but if a network has atleast 3 hidden layers then it is considered deep neural network.

2.5 Example of Working of Neural Network

To further explain the mathematics behind a neural network, the example of a single neuron with binary output is explained below using Figure 2.13.

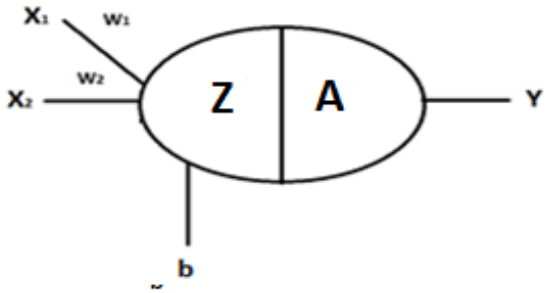

Figure 2.13: A single neuron of neural network

First, the product of weights and their respective inputs is accumulated with the bias as shown in Equation 2.6,

$$Z = W * X + b \tag{2.6}$$

where 'Z' is the accumulated output of weight vector 'W' (containing W_1,W_2) and input vector 'X' (containing X_1,X_2) and bias 'b'. After calculating 'Z' we feed it into activation function. For this example, it is sigmoid activation function. It is mathematically given in Equation 2.7 as,

$$Y_{pred} = A = \frac{1}{1+e^{-Z}} \tag{2.7}$$

where 'A' is sigmoid activation function's output. In this case, 'A' is also the prediction 'Y_{pred}' of our network, since it is a single neuron network. After getting prediction 'Y_{pred}', we calculate the loss value as shown in binary cross-entropy formula in Equation 2.8,

$$L = -[Y * \log Y_{pred} + (1 - Y) * \log(1 - Y_{pred}) \tag{2.8}$$

where 'Y' is the expected output, since we know output in supervised learning and 'L' is the loss value. Our target is to minimize this loss value so that we can get good predictions for our network. To minimize loss value, we tweak the weights and biases. Specifically,

we find out how the change in weight affects the loss value by using chain rule of derivative. It is shown in Equation 2.9 below,

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y_{pred}} * \frac{\partial Y_{pred}}{\partial Z} * \frac{\partial Z}{\partial W} \tag{2.9}$$

where '$\frac{\partial L}{\partial W}$' is rate of change of loss value with respect to weights, '$\frac{\partial L}{\partial Y_{pred}}$' is the rate of change of loss value with respect to predicted value, '$\frac{\partial Y_{pred}}{\partial Z}$' is the rate of change of predicted value 'Y_{pred}' with respect to 'Z' and finally '$\frac{\partial Z}{\partial W}$' is the rate of change of 'Z' with respect to weights. Hence, we find the derivative of the terms as follows,

$$\frac{\partial L}{\partial Y_{pred}} = \frac{Y_{pred} - Y}{Y_{pred}(1 - Y_{pred})} \tag{2.10}$$

$$\frac{\partial Y_{pred}}{\partial Z} = Y_{pred}(1 - Y_{pred}) \tag{2.11}$$

$$\frac{\partial Z}{\partial W} = X \tag{2.12}$$

substituting the above 3 equations in Equation 2.9 gives,

$$\frac{\partial L}{\partial W} = X * (Y_{pred} - Y) \tag{2.13}$$

where Equation 2.13 shows the rate of change of Loss function with respect to weights '$\frac{\partial L}{\partial W}$'. After finding it, we use the formula for gradient descent to update the weights and biases as shown in Equation 2.14 and Equation 2.15 below,

$$W(t) = W(t-1) - \alpha \frac{\partial L}{\partial W} \tag{2.14}$$

$$b(t) = b(t-1) - \alpha \frac{\partial L}{\partial b} \tag{2.15}$$

where 'W(t)', 'b(t)' are updated weights and biases, W(t-1), b(t-1) are previous weights and biases, $\frac{\partial L}{\partial W}$, $\frac{\partial L}{\partial b}$ are partial derivatives of Loss with respect to weight and bias and 'α' is the learning rate which is kept small to reach convergence. This whole process of updating weights and biases using chain rule of derivative is called backpropagation.

2.6 Common Issues Dealing with Machine Learning Algorithms

When training our model with data using machine learning algorithms, the following two issues are most common:

1) Underfitting
2) Overfitting

2.6.1 Underfitting

Underfitting means our model is failing to capture the trend of the input data. It shows that the model is not doing well in terms of training accuracy [29]. Mathematically speaking, it means that our model is exhibiting high bias. Figure 2.14(a) shows underfitting.

2.6.2 Overfitting

Overfitting means our model is trying to memorize the data. Mathematically speaking, it means that our data has high variance which our model is trying to capture by memorizing it. It also means that the model is doing great on training data but does not do well on unseen (test) data [29]. Figure 2.14(b) shows overfitting.

Our target in training machine learning models is to get the best fit, as shown in Figure 2.14(c) we can such that our model does not underfit or overfit. It is also called bias-variance tradeoff.

Figure 2.14: Underfitting, overfitting and good fit

2.7 Tackling Underfitting and Overfitting in Neural Networks

Underfitting is usually tackled in neural networks by increasing the hidden layers and/or neurons whereas overfitting is tackled using regularization, adding more data or dropout. Regularization and dropout are briefly explained below:

1. **Regularization:**

   Regularization is a technique to help machine learning model do well on unseen data (test set) [30]. Following are the 2 types of regularizations commonly used:

   a) L2 Regularization:

   In L2 regularization, weights of neural network are penalized using L2-norm [30] shown mathematically in Equation 2.16 as,

   $$J(w; X, y) = \frac{\lambda}{m} w^T w + J_{prev}(w; X, y) \tag{2.16}$$

   where J(w;X,y) is the loss function, 'w' is weight vector, 'λ' is the regularization parameter typically kept small, 'm' is the number of training examples '$J_{prev}(w; X, y)$' is the previous value of loss function.

   L2 regularization penalizes larger weights more.

   b) L1 Regularization:

   In L1 regularization, simply L1 norm of weight is taken [30]. Mathematically, given in Equation 2.17 as,

   $$J(w; X, y) = \lambda * ||w|| + J_{prev}(w; X, y) \tag{2.17}$$

   where 'J(w;X,y)' is the loss function, 'λ' the regularization parameter, $J_{prev}(w; X, y)$ is the previous value of loss function and 'w' being the weight vector.

2) Dropout:

In this technique, neurons are randomly disappeared in hidden layers during epochs to make sure that our model does not memorize the training data. This is a very effective technique to tackle overfitting [31]. It is shown in Figure 2.15 on next page,

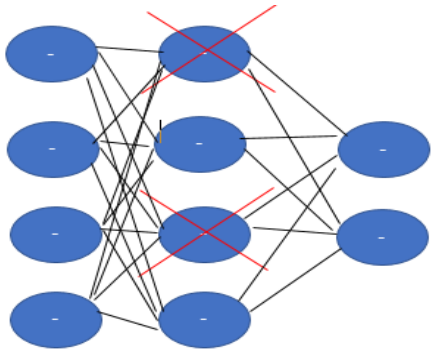

Figure 2.15: Dropout in neural network

CHAPTER 3

Feature Extraction and Data Processing

In this section EMG data processing and feature extraction is discussed. The data for EMG is collected from the Myo Gesture Control Armband shown in Figure 3.1.

Figure 3.1: Myo Gesture Control Armband

In the proposed scheme, 1000 samples of EMG data are collected for each finger movement. Since armband sends EMG data of 8 sensors at 200Hz, hence the training period of each finger lasts 5 seconds. Our dataset has 12 finger movements which are opening of thumb, index, middle, ring and pinky finger then two, three, four, five fingers open, then fist, grab and pick movement as shown in Figure 3.2 below.

Figure 3.2: 12 finger movements used for data classification

3.1 Data Extraction via Myo Gesture Control Armband

The EMG data for classification is collected from “Myo Gesture Control Armband”. The device is provided with 8 EMG electrodes along with 3-axes accelerometer, 3-axes gyroscope and 3-axes magnetometer. This armband is equipped with ARM Cortex M4 processor. It transmits data via Bluetooth at 200Hz frequency. This device requires user to wear it and synchronize with the hand movements before it can be used. This process can take a few minutes. By default, it can detect 5 signature movements as shown in Figure 3.3 taken from Myo armband’s website.

Figure 3.3: Myo Armband’s signature movements

3.2 Feature Extraction

The EMG data gathered from the armband is very random in nature and must be processed to extract important features from it which can be used to classify EMG data. These features are of 2 types:

a) Time-domain features.
b) Frequency-domain features.

The time-domain features include averaging, zero detection, root mean square, standard deviation etc. whereas frequency-domain features include taking power-spectral density, fast fourier transform and other techniques.

In this study, we analyzed time-domain based feature extraction in which we extract the absolute value of EMG data and take windowed average of the collected 1000 EMG data points from the 8 sensors. The window size used here is 50. It is discussed in detail below.

3.2.1 Absolute Value of EMG

EMG values coming from armband by default are positive as well as negative making it more difficult to recognize gesture. To illustrate this, EMG data for one sensor for index finger opening is shown in Figure 3.4.

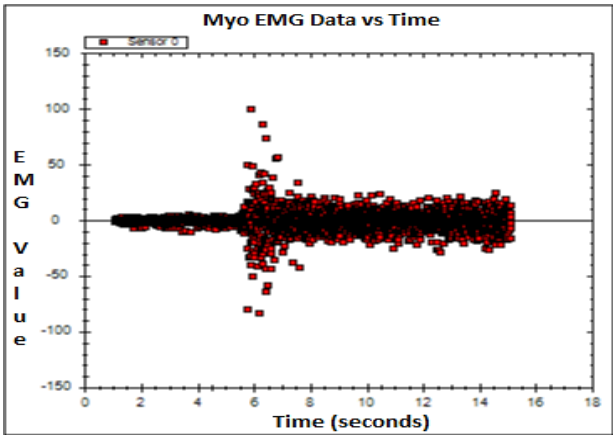

Figure 3.4: EMG value of one sensor for index finger open movement

Hence we take the absolute value of EMG values of each sensor, mathematically shown in (1).

$$EMG_{absolute} = |\text{EMG}| \tag{4.1}$$

3.2.2 Windowed Batch Average of EMG

Using the absolute value of EMG data, we use a different approach than original moving average of whole data which is given in (2),

$$E_{MA} = \frac{1}{W}\sum_{i=j*n}^{n-1+j*n} EMG_{absolute} \tag{4.2}$$

where 'W' is sample window size. Instead, we computed average of batch of EMG values for each of the 8 sensors.

Hence, for 1000 EMG samples collected for each of the 8 sensors, we take average of n=50 EMG samples per batch and j=0,1,..,19 i.e. average of 0 to 49, 50 to 99…. 950 to 999, giving us 20 data points per sensor, per finger movement, which are then fed into neural network for classification. So, for each finger movement that we want to classify, there will be 20*8 = 160 samples for 8 sensors.

By using the absolute value of EMG and taking its windowed average, the data is fed into neural network to get classification of input.

CHAPTER 4

Experiments and Proposed Method

In this chapter, data collection via EMG armband, and the various experiments done using that data are explained in detail. Basically, two approaches were used for data processing and classification:

a) Offline data processing

b) Real-time data processing

4.1 Offline Data Processing

In offline data processing, experiments were done on EMG data from a subject that was stored on computer from the EMG armband for different finger movements and then the stored data was sent to the neural network algorithm for classification. This approach was used primarily to get comfortable with neural networks and to verify if the desired results can be achieved or not. The details of an experiment that gave adequate results is discussed here.

Experiment:

Following are the features of the experiment:

1. Rstudio was used for writing the code with Keras being the library for neural network algorithm.
2. 10 second data for each of the 8 sensors for index, middle, ring and pinky finger's open and close position was taken at 200Hz and stored in a .txt file giving 2000 EMG values for each sensor.
3. 101 such files were created for each finger movement.
4. Absolute values for all the EMG values were taken.
5. Mean of each of the 101 files with 2000 EMG values for each of the 8 sensors were taken for each finger movement

6. Thus 101 data points are created and fed to a neural network with 2 hidden layers with 32 and 16 neurons of ReLU activation function respectively. Output layer had softmax activation function with 8 outputs, one for each finger movement.
7. Dataset was divided into 80:20 split to get training and validation set.

Observations:

The 2 hidden-layer neural network used adam optimizer with 0.01 learning rate and ran for 500 EPOCHS yielding 96.8% training accuracy and 93.5% validation accuracy as shown in Figure 4.1.

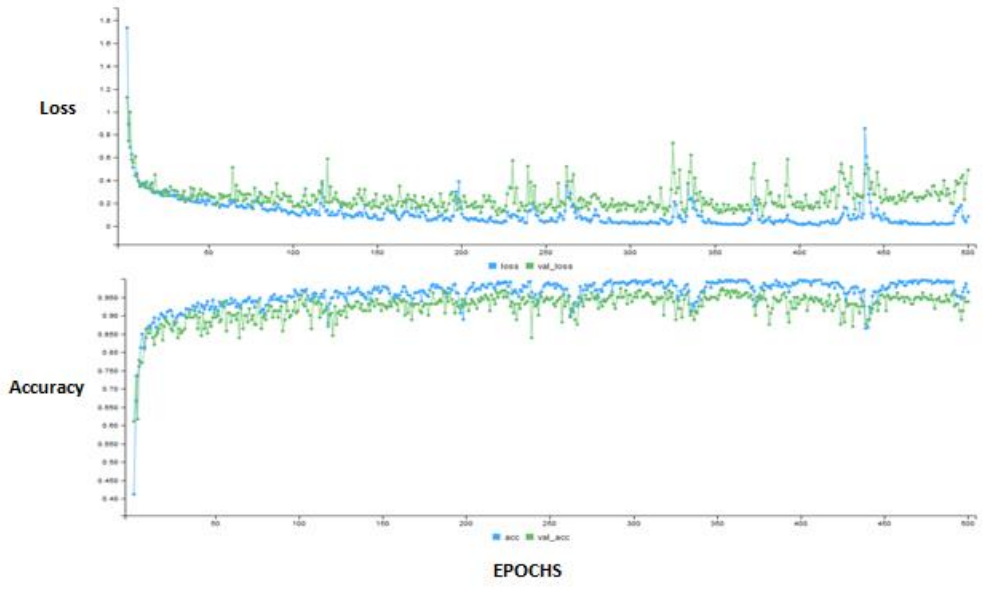

Figure 4.1: Training and test loss (above) and accuracy (below) for offline processing

Inference:

The main purpose of this experiment was just to get familiar with neural networks. It was seen that while the neural network trained well on the data, test accuracy was below 80% when tested on the EMG data of 2 different subjects. There are 2 obvious flaws in this approach:

1. The obvious flaw in this approach is that the EMG data needs to be classified in real-time and is to be sent to robotic hand, not in offline mode.

2. This approach required 101 samples of 10 second EMG data to train a network and we took absolute value of average of each of those samples, which is very time consuming and not good for implementing as there are temporal features in each sample, that need to be looked at, rather than to be processed as a whole by taking average.
3. It was also seen that RStudio did not provide real-time data acquisition support that was required for this project.

In spite of the above mentioned flaws, this experiment gave confidence to implement neural networks on EMG data which was important.

4.2 Real-Time Data Processing

EMG Data classification from armband requires quick real-time classification for tasks to be performed after classification. These tasks can be of wide variety such as light on/off, robot movement or in our case, sending classified finger movement to a robotic hand so that it can imitate that movement.

For this task, Python was chosen to be the programming language for developing the whole software for data acquisition via Bluetooth from the Myo Gesture Control Armband and Keras Deep Learning library was used for neural network algorithm development. The software utilizes an API called “Myo-Python” designed by Niklas Rosenstein [32] published on Github as open-source code. This API receives data from the EMG armband which we then utilize for feature extraction.

Figure 4.2 on next page explains the real-time data processing communication flow of the EMG data.

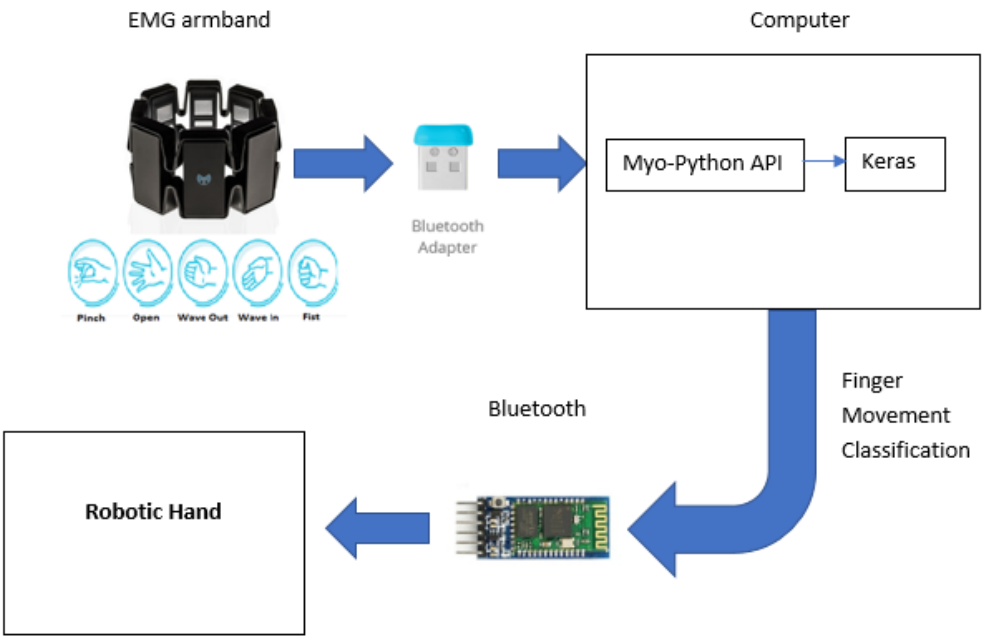

Figure 4.2: Communication flow between hardware components of project

After setting up the hardware and code in the computer, experiments were performed for real-time data processing of EMG data in 2 parts. First, we started with 5 finger open movements and then we worked on 12 finger open movement recognition.

4.2.1 Five Finger Movements Classification Experiments

In this series of experiments, 5 finger movement recognition was performed which were, thumb open, index finger open, middle finger open, ring finger open and pinky finger open. The EMG data for these finger movements is very random as shown in Figure 5.3(a) to 5.3(e) on the following pages:

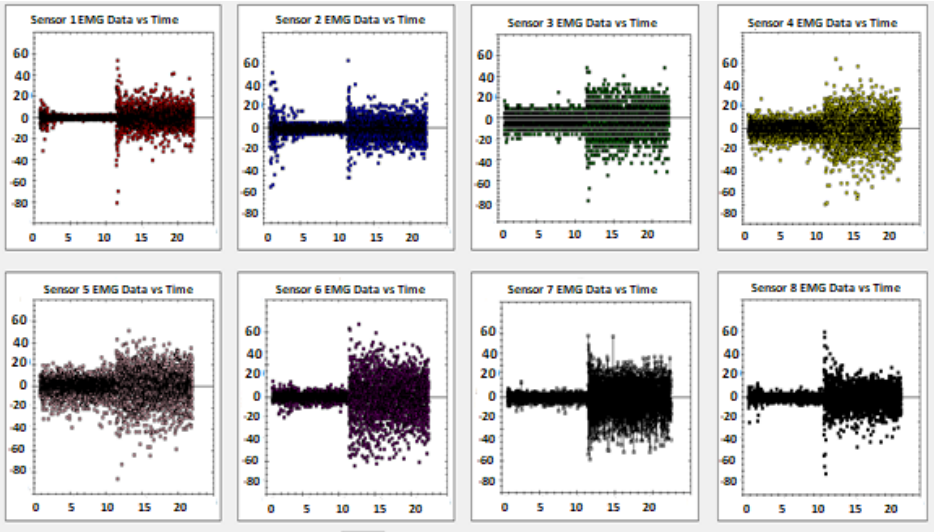

Figure 4.3(a): Thumb open EMG data (y-axis) vs Time (x-axis) for 8 sensors of EMG armband

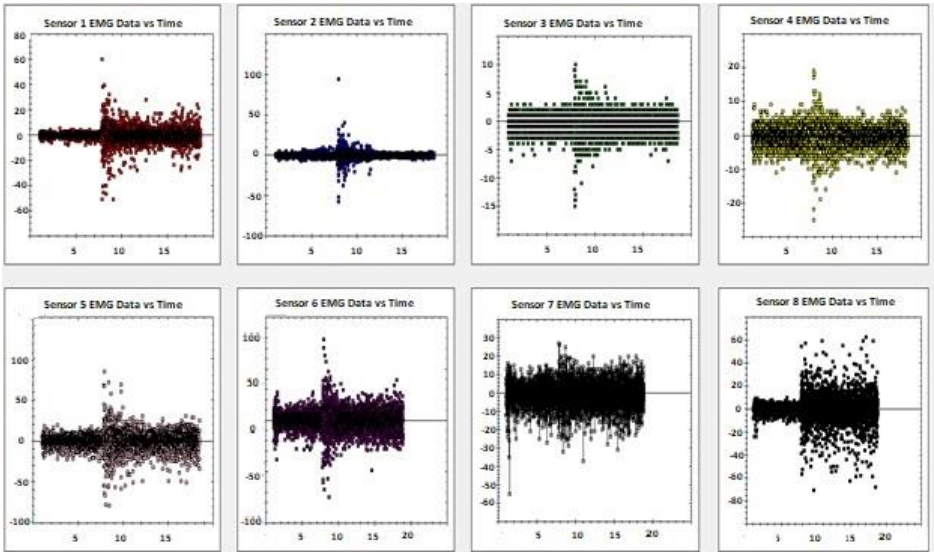

Figure 4.3(b): Index finger open EMG data (y-axis) vs Time (x-axis) for 8 sensors of EMG armband

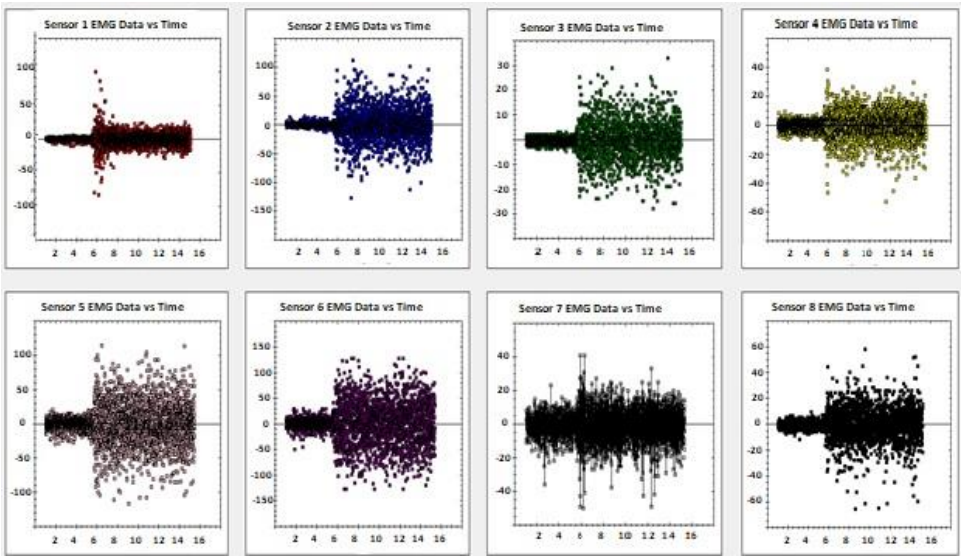

Figure 4.3(c): Middle finger open EMG data (y-axis) vs Time (x-axis) for 8 sensors of EMG armband

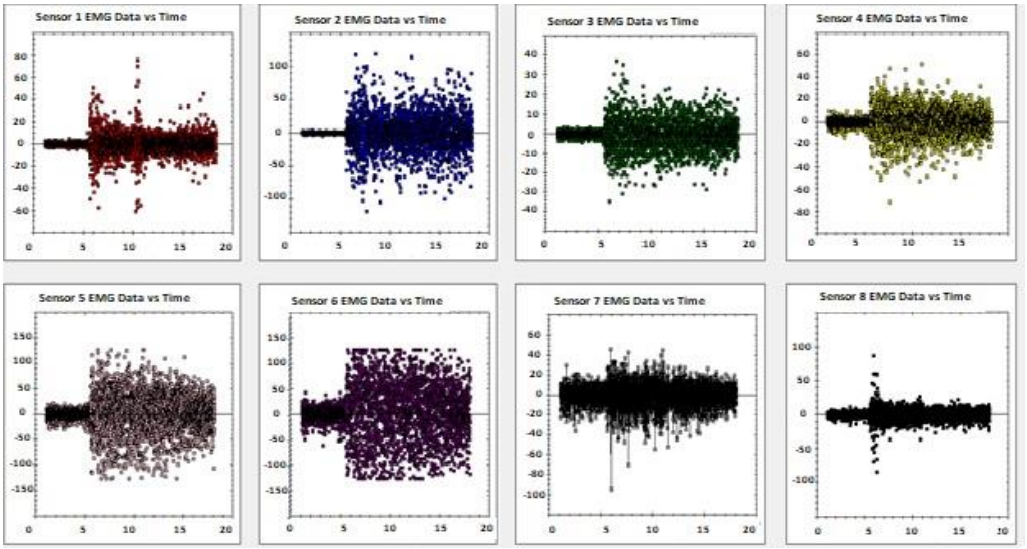

Figure 4.3(d): Ring finger open EMG data (y-axis) vs Time (x-axis) for 8 sensors of EMG armband

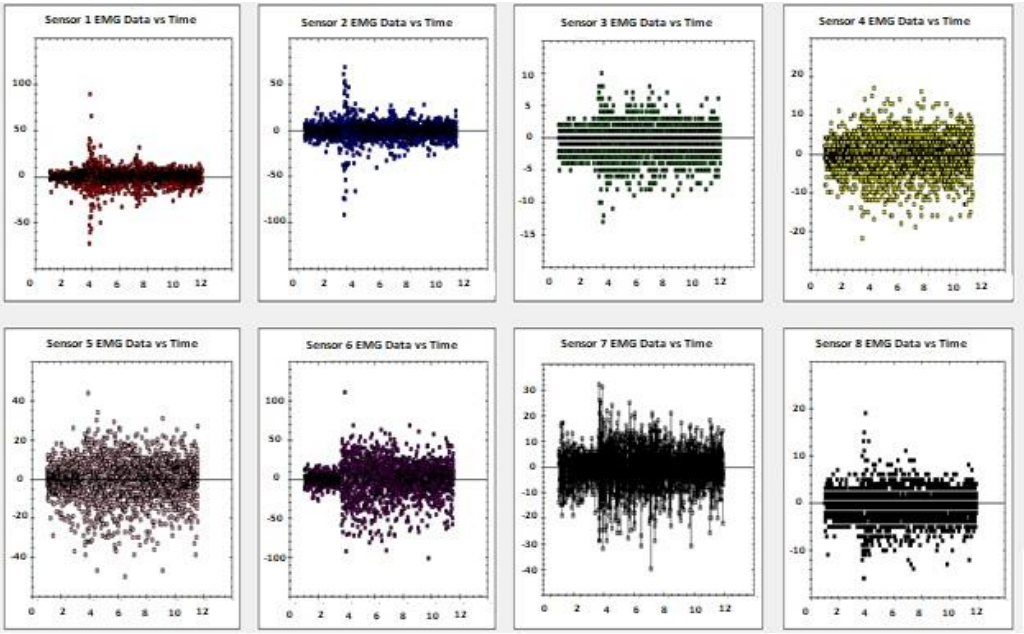

Figure 4.3(e): Pinky finger open EMG data (y-axis) vs Time (x-axis) for 8 sensors of EMG armband

In all parts of Figure 4.3, a spike in EMG is observed when the respective finger is opened and then EMG values get stable. For all the experiments performed, Following are the details of the experiments done to classify these movements.

Experiment 1 – Taking data without preprocessing

Initially, we started with feeding the raw EMG data as shown in Figure 5.3, without any preprocessing to the neural network. Following were the features of the neural network:

1) 1500 EMG samples for each finger movement were collected and split 80:20 for training and validation set.
2) 2 hidden layers each with 8 neurons of ReLU activation function were used.
3) Output layer had 5 neurons with softmax activation function.
4) Adam optimizer with 0.01 learning rate was used.
5) Model was trained for 500 EPOCHS (iterations).

Observations:

The training and validation accuracy and loss observed are shown in Figure 4.4 on next page.

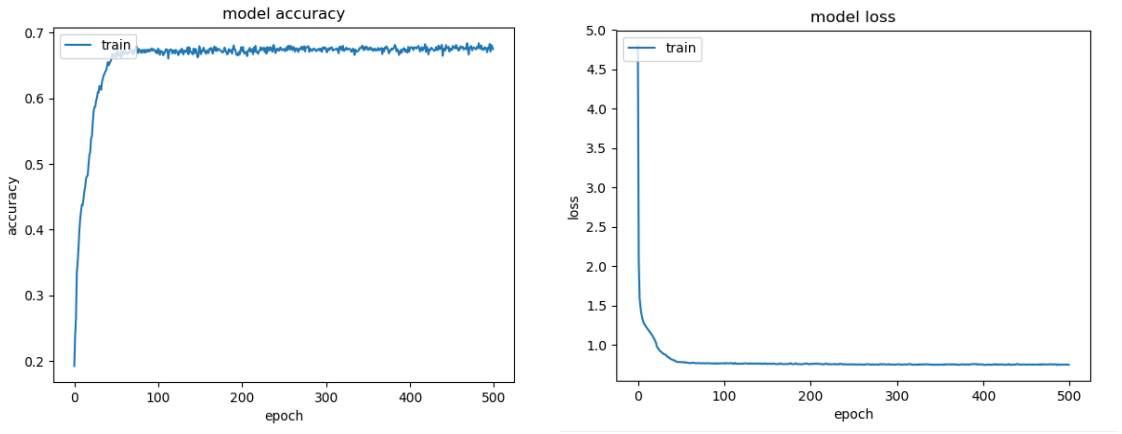

Figure 4.4: Training accuracy (left) and loss (right) curves for the model

The model achieved 66.5% training accuracy and 60.2% test accuracy. All movements were misclassified. Table I shows the results of further experiments with raw EMG data.

Table I: Experimental observations on raw EMG data

Experiment No.	Hidden layers	Hidden layer neurons	EPOCHS	Training accuracy (%)	Training loss	Validation accuracy (%)	Validation loss	Inference
1	2	16 (each)	500	78.1	0.5962	71.3	0.8613	Underfit, so we increase neurons in next experiment
2	2	50 (each)	500	97.17	0.1	74.2	2.33	Overfitting, so we add regularization in next experiment
3	2	50 (each)	500	76.9	0.75	69.6	0.82	Underfit

Inference:

It seemed obvious that the model is underfitting so for next experiments, the number of neurons in hidden layers was increased. It can also be seen in Figure 5.3 that EMG data had very high variance, so in order to train model for it, the model will be prone to overfitting.

Experiment 2 – Taking Absolute EMG Value

For this series of experiments, we took the absolute value of EMG data coming from armband because the raw EMG data had positive as well as negative values.

Observations:

Table II shows the observations of above experiment.

TABLE II: Experimental observations on absolute values of EMG data

Experiment No.	Hidden layers	Hidden layer neurons	EPOCHS	Training accuracy (%)	Training loss	Validation accuracy (%)	Validation loss	Inference
1	1	24	300	69.92	0.7553	66.3	0.84	Underfit, so we increase neurons in next experiment
2	1	50	300	72.3	0.7	68.3	0.83	Underfitting, so we add neurons in next experiment
3	1	300	300	85.9	0.65	69.6	0.82	Overfitting
4	2	50 (each)	300	84.3	0.34	63	2.19	Overfitting
5	2	100 (each)	300	90	0.2	72.2	1.72	Overfitting, we add L2 regularization
6	2	200	300	83.3	0.45	73.35	1.67	Underfitting

Inference:

Using absolute values of EMG data did not give good results.

Experiment 3 – Preprocessing using standardized absolute value of EMG data

Standardization is the process of making input data to have mean 0 and variance 1. It is a common approach to preprocess data and standardization is a widely used approach as it centers the data around 0 and neural networks are seen to optimize quickly on preprocessed data.

Observations:

Table III shows the experiments done using standardized EMG data.

Table III: Experimental observations on standardized absolute value of EMG data

Experiment No.	Hidden layers	Hidden layer neurons	EPOCHS	Training accuracy (%)	Training loss	Validation accuracy (%)	Validation loss	Inference
1	2	8 (each)	300	19.5	1.6	19.4	1.6	Underfit, so we increase neurons in next experiment
2	2	50 (each)	300	78.17	0.6	37.7	2.5	Overfitting, so we add regularization in next experiment

Inference:

It was quite surprising that the standardized EMG value gave poor results than the raw EMG data.

Experiment 4 – Using absolute EMG value with windowed average

In this method, EMG data for each of the 5 finger movements was collected for 7.5 seconds giving 1500 EMG data points for each movement. Then average of batches of 50 EMG samples were taken, leaving us with 30 data points per finger movement. Data was split into 80:20 for training and validation set.

Observations:

Following were the results of using this approach:

Table IV: Experiments using absolute EMG values with windowed average

Experiment No.	Hidden layers	Hidden layer neurons	EPOCHS	Training accuracy (%)	Training loss	Validation accuracy (%)	Validation loss	Inference
1	1	8	500	96	0.5274	88.67	0.68	Overfitting, so we add L2 regularization and batch normalization
2	1	8	500	98.33	0.3274	96.67	0.28	All finger movements were perfectly recognized in real-time

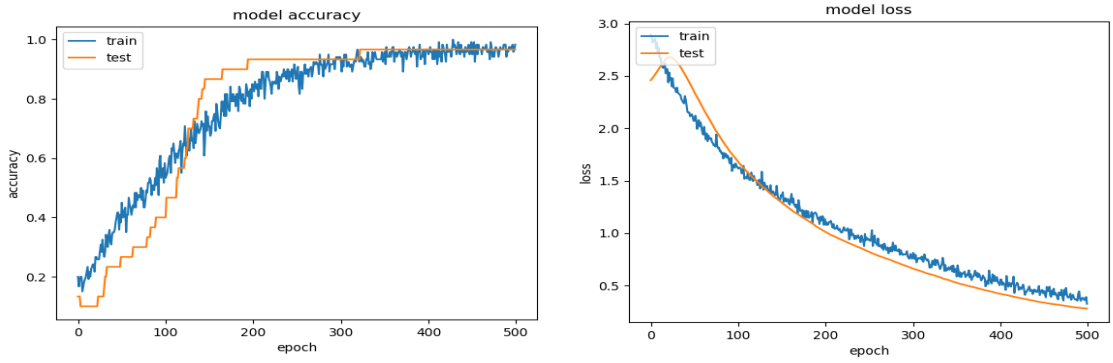

Figure 4.5: Training and validation accuracy (left) and loss (right) curves for 5 movements

Inference:

Figure 4.5 shows the results of 2nd experiment done using this approach. It can be seen that this approach gave the best results and gave confidence to work on 12 finger movements.

4.2.2 Twelve Finger Movements Classification Experiments

After getting desired results on 5 finger movement classification, experiments were performed for 12 finger movement classification.

Experiment:

For this task, 12 finger movements were chosen for classification. Those movements are:

1) Thumb open	2) Index open	3) Middle open	4) Ring open
5) Pinky open	6) Two fingers open	7) Three fingers open	8) Four fingers open
9) Five fingers open	10) All fingers closed	11) Grasp movement	12) Pick movement

A series of experiments were performed for these finger movements using absolute value of windowed average of EMG values with window size 50, starting from collecting EMG samples per finger movement for 7.5 seconds, to decreasing it to 5 seconds. Same single hidden layer 8 neuron network was used, learning rate and Epochs were varied.

Observations:

Figure 4.6 below shows the training and validation accuracy and loss curves for the Table V on next page shows the results of experiments done.

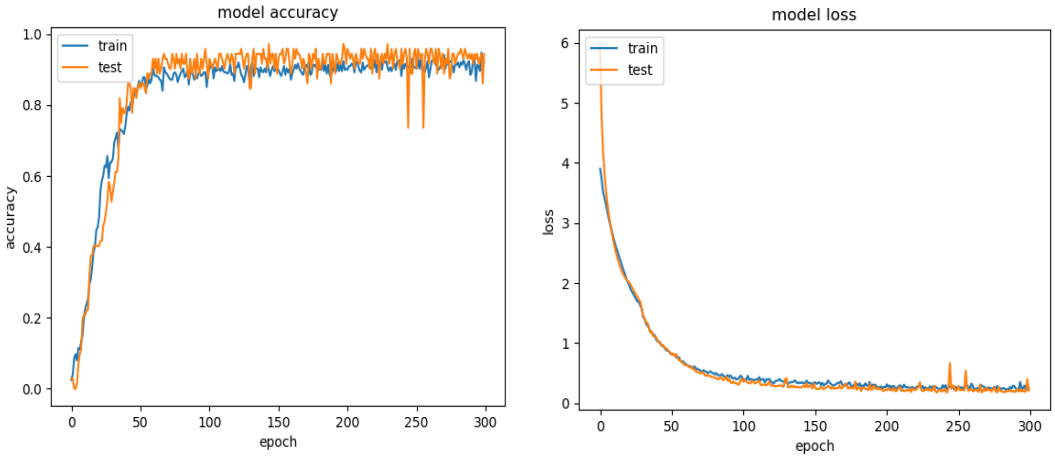

Figure 4.6: Training and validation accuracy (left) and loss (right) curves for 12 movements

Table V: Experiments for 12 finger movements classification

No.	Hidden layers	Hidden layer neurons	Learning Rate	EPOCHS	Training accuracy (%)	Training loss	Validation accuracy (%)	Validation loss	Inference
1	1	8	0.0001	500	78.82	0.8631	90.28	0.65	Underfitting, 5 out of 12 movements not recognized
2	1	8	0.0001	700	85.76	0.5585	88.89	0.4280	All finger movements except 5 finger open were perfectly recognized in real-time
3	1	8	0.001	300	92.01	0.2445	94.44	0.212	All movements recognized
4	1	4	0.001	300	0.81	0.79	75	0.9	Few movements not recognized
5	1	2	0.001	300	53.65	1.074	43.75	1.32	Underfitting, poor recognition

Inference:

It was observed that learning rate of 0.001 gave the best results as shown in Table V on previous page, experiment # 3, with its training and validation accuracy and loss graphically represented in Figure 5.6 above. All finger movements were correctly recognized. For finger movement recognition, 200 EMG samples were collected which takes 1 second as the armband works at 200Hz frequency. It was also observed that decreasing the neurons in hidden layer deteriorated the results.

4.3 Testing Algorithm on Subjects

The configuration of experiment # 3 in Table V was used for testing finger movement classification on other 3 subjects other than the one trained to use it. For these experiments, subjects were asked to do different finger movements in random order.

Observations:

Figure 4.7 on next page shows the observations for different experiments on 3 subjects. Figure 4.7(A) shows the results of the subjects on 5 finger movement recognition algorithm, it can be seen in the confusion matrix that no problems were seen for 5 finger recognition. Figure 4.7(B) shows the confusion matrix for 12 finger movement recognition algorithm. Figure 4.7(C) and Figure 4.7(D) shows the training and validation accuracy and loss for 5 finger recognition algorithm whereas Figure 4.7(E) and Figure 4.7(F) shows the training and validation accuracy and loss for 12 finger movement recognition algorithm. Both algorithms were different only in that the output layer for 5 finger movement recognition algorithm had 5 neurons whereas 12 neurons for 12 finger movement recognition. It was seen in different experiments that some finger movements had problems in recognition and the main reason for it was that subjects would sometimes do different movements for recognition than they used for training, either in different position or different intensity such as hold movement tighter or looser than they used for training. To prevent that, subjects were asked to put hand on an arm chair for finger movements, which improved recognition accuracy.

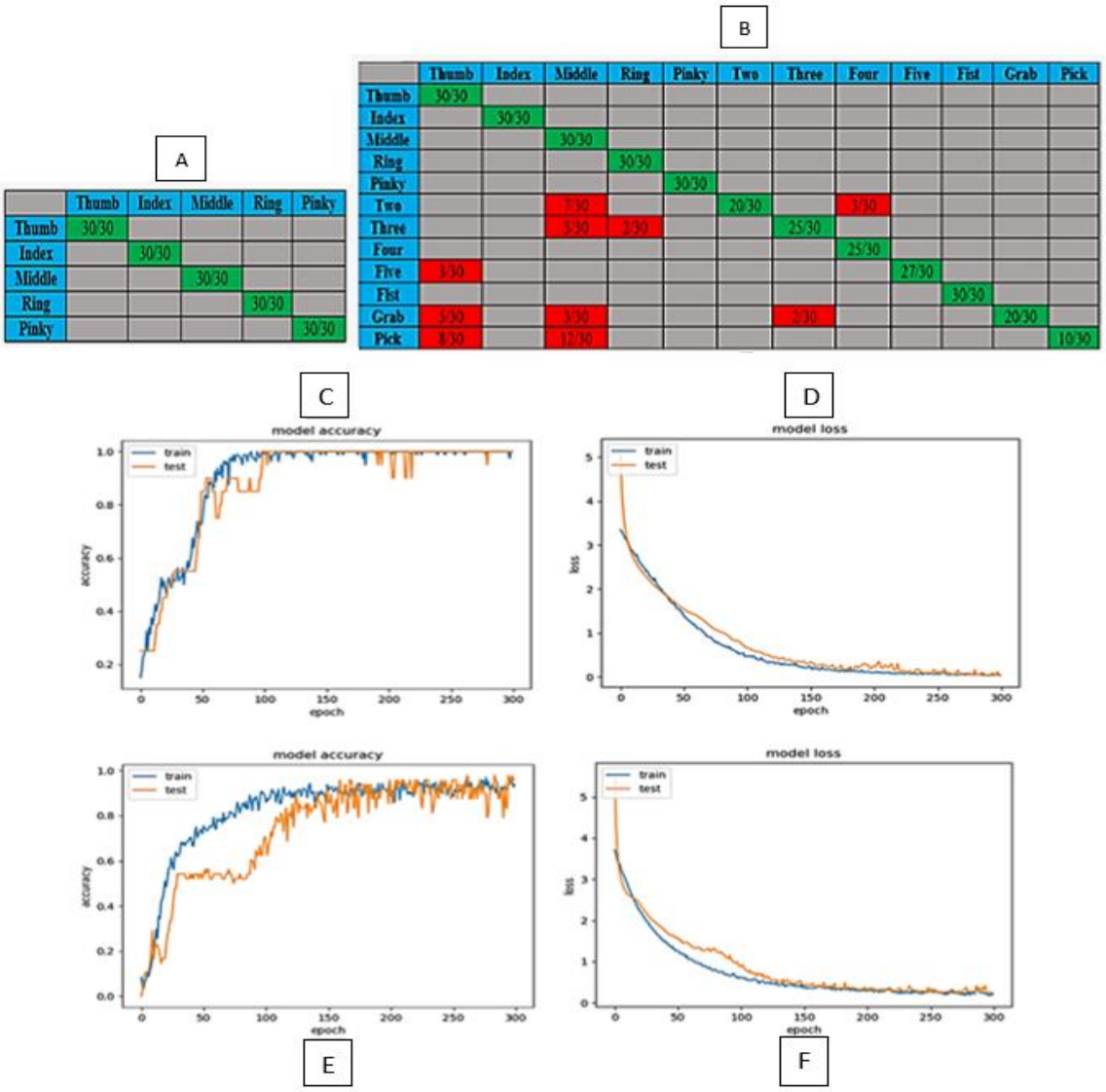

Figure 4.7: Performance of neural network. A & B show the confusion matrix of 5 and 12 finger movements results on human subjects, C & E show the training (blue line) and test accuracy (orange line) of model on 5 and 12 finger movements respectively and D & F show the training (blue line) and test loss (orange line) for 5 and 12 finger movements respectively

Inference:

1) From experiments with 3 subjects, It was seen that the finger recognition algorithm worked great taking 5 seconds for training each finger movement and 1 second for recognition.
2) Putting hand on armchair for training and verification helped achieve good recognition accuracy.

3) Even though we can train the finger recognition for a user, if the position of wearing armband changes or if the armband is removed and worn again, then finger movements will have to be re-trained.

CHAPTER 5

Design of Robotic Hand

For the scope of this project, it was also desired to design a robotic hand that is smaller, lighter and more realistic than the robotic hand designs found online so that recognized finger movement from machine learning algorithm can be imitated by the designed robotic hand. For this purpose a 3D design of robotic hand was designed from scratch, with motivation for design taken from the popular InMoov Robotic hand [16] which is an open source robotic hand design.

The software used for 3D design of robotic hand was RS Component's DesignSpark Mechanical which is an open source 3D design software and quite easy to use. For the robotic hand to be realistic, the dimensions of palm and fingers were based on a human hand.

The biggest advantage that this robotic hand has, compared to the robotic hand designs online is that it uses micro servo motors which are small enough to fit in the palm of robotic hand, thus greatly reducing the size of the robotic hand whereas common robotic hand designs fit regular servo motors which are large, heavy and cannot fit in the palm of robotic hand, thus fitting above the wrist, making the hand large and heavy.

5.1 Materials Used

Following materials were used for designing the robotic hand:

1. For pulling the fingers, 1mm braided fishing line was used.
2. ABS material was used for 3D printing the robotic hand and fingers.
3. FUTABA S3114 Micro Servo motors were used as actuators.
4. Arduino Uno is used for controlling motors.
5. HC-05 Bluetooth module is used for sending signals for finger recognition from computer to Arduino.
6. Servo motor driver for controlling the servo motors.

5.2 Design

For robotic hand design, the fingers were 3D printed first. Figure 6.1 shows the design of fingers and Figure 6.2 shows different views of the palm of robotic hand.

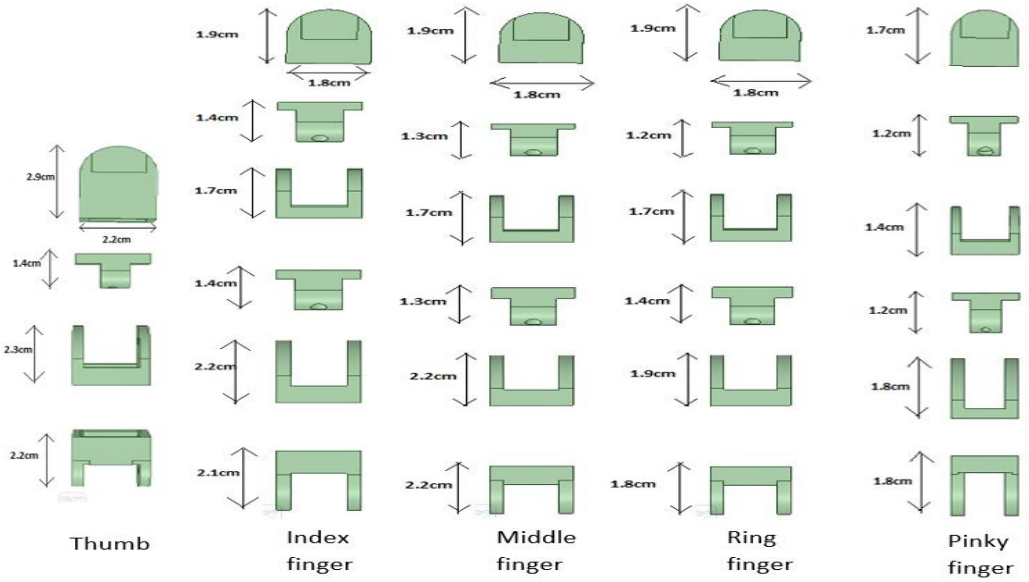

Figure 5.1: 3D designs of fingers of robotic hand

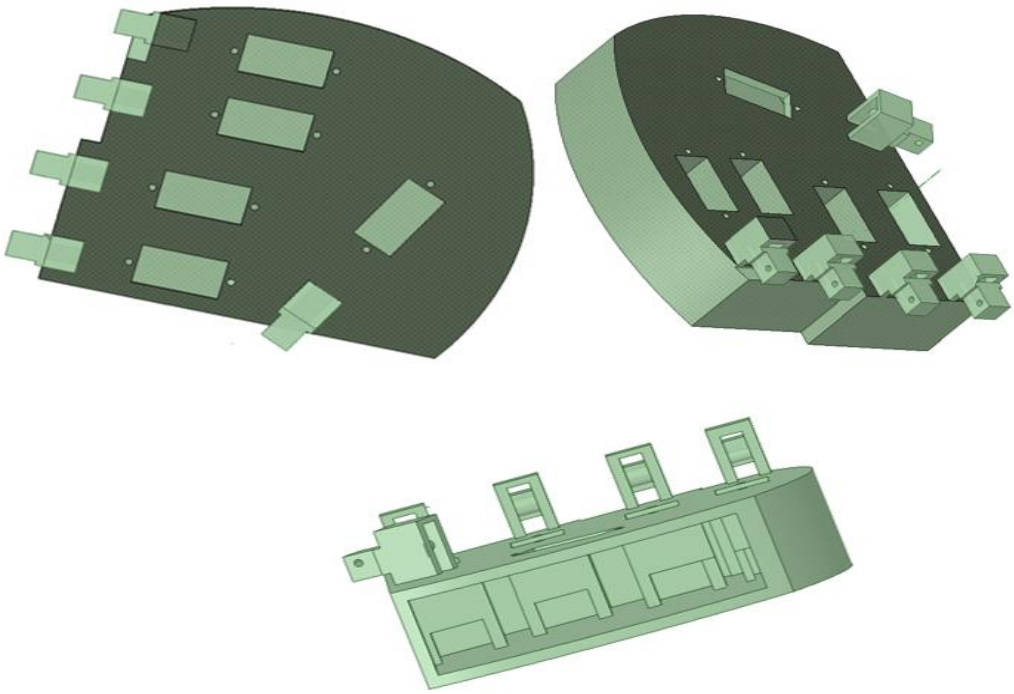

Figure 5.2: Different views of 3D design of robotic hand's palm

5.3 Hardware

The palm of robotic hand was 3D printed with assistance from Natural Resources Research Institute (NRRI), Duluth, whereas the fingers of robotic hand were printed in University of Minnesota Duluth's 3D printers. Figure 5.3 shows the complete 3D printed design of the robotic hand.

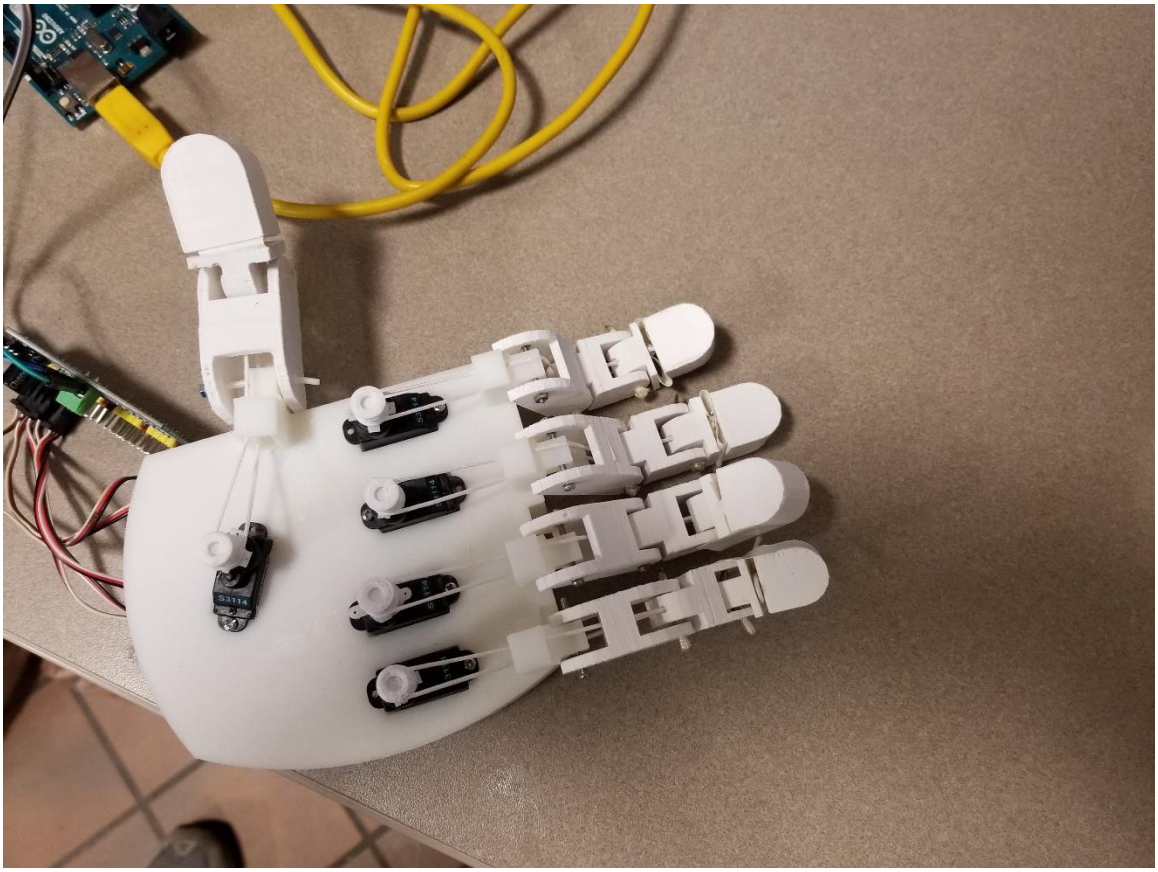

Figure 5.3: 3D printed robotic hand

It can be seen that the finger pulling mechanism of servo motors is based on micro servo motors and 1mm braided fishing line. Figure 5.4 on next page shows the complete hardware used for controlling the robotic hand.

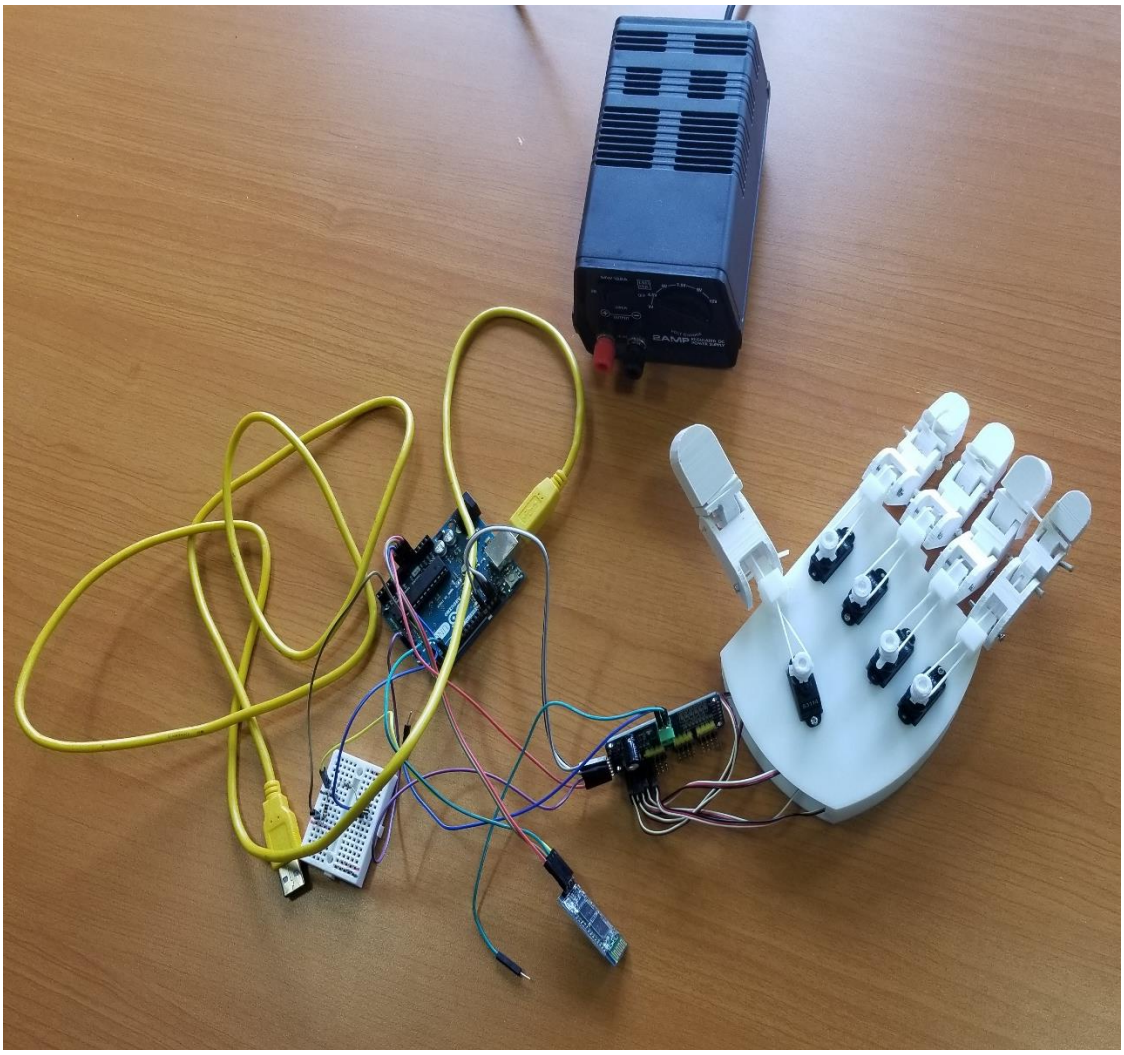

Figure 5.4: Complete hardware of robotic hand

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

In this study, a machine learning based approach to classify EMG signals using an inexpensive EMG armband is presented. This approach uses a single hidden layer neural network to train and recognize the finger movements using EMG data and has not been used in previous research works. The training for each finger movement takes just 5 seconds, while verification takes just a second. For feature extraction, a lot of approaches were tried and tested, but windowed averages of absolute values of EMG gave best results to feed into neural network for training and verification.

This approach when tested on 4 subjects gave perfect results for 5 finger movement recognition and above 80% accuracy on 12 finger movement recognition. The main reason for classification error was seen to be different gestures used for training and verification. Using an armchair for gesture recognition and verification helped.

In the second part of this study, a novel design for robotic hand is presented. The robotic hand is designed from scratch using 3D printing, is a very light, small and realistic version of a middle-aged human hand. Using micro-servo motors greatly decreases the size of robotic hand mechanism enabling us to fit the whole control mechanism inside the palm of robotic hand.

The single hidden layer neural network approach is easy to implement even on embedded systems such as microcontroller, in which we can hardcode the trained weights of neural network. Thus, without needing expensive computer always at disposal, the whole finger recognition algorithm can be implemented on inexpensive hardware keeping robotic hand small.

This machine learning based approach is good if a quick and easy task is to be achieved using finger movement recognition such as gesture based robot control, games or other applications. However, using it for amputees, will require a lot of tests to be done with the help of amputees. On the other hand, the 3D printed robotic hand design presented in this study can act as a novel robotic hand prototype for amputees if built with strong materials and sturdy finger pulling mechanism because it is small, realistic and lightweight.

6.2 Future Work

In this research, a single hidden layer neural network is used for training EMG data and absolute values of EMG and their windowed averages are used to capture the trend of the EMG data for finger movement classification. It is obvious that time is of essence for finger movement classification via EMG. There is a special type of neural network called Recurrent Neural Networks which is used specially for time-series based data. Using Recurrent Neural Networks for EMG Classification might provide better results in capturing the trends of varying EMG, based on different finger movements than the single hidden layer neural network approach provided in this study.

For this study, a laptop computer was used for training and verification of EMG signals for finger movements. However, taking advantage of simplicity of a single hidden layer neural network, this algorithm can be implemented on a microcontroller or a Raspberry PI which is a small, inexpensive credit card sized computer and also provides the support for deep learning frameworks like Keras and Tensorflow, thus getting rid of large computer dependence.

Bibliography

[1] K. Ziegler-Graham, E. J. MacKenzie, P. L. Ephraim, T. G. Travison, and R. Brookmeyer, “Estimating the Prevalence of Limb Loss in the United States: 2005 to 2050,” Archives of Physical Medicine and Rehabilitation, vol. 89, no. 3, pp. 422–429, Mar. 2008.

[2] Biddiss, E. A., & Chau, T. T. (2007). Upper limb prosthesis use and abandonment: A survey of the last 25 years. Prosthetics and Orthotics International, 31(3), 236–257. https://doi.org/10.1080/03093640600994581

[3] A. Fougner, Ø. Stavdahl, P. J. Kyberd, Y. G. Losier and P. A. Parker, "Control of Upper Limb Prostheses: Terminology and Proportional Myoelectric Control—A Review," in IEEE Transactions on Neural Systems and Rehabilitation Engineering, vol. 20, no. 5, pp. 663-677, Sept. 2012. doi: 10.1109/TNSRE.2012.2196711.

[4] A. Harada, T. Nakakuki, M. Hikita, and C. Ishii, “Robot finger design for Myoelectric prosthetic hand and recognition of finger motions via surface EMG,” in 2010 IEEE International Conference on Automation and Logistics, Shatin, Hong Kong, 2010, pp. 273–278.

[5] W. Caesarendra et al 2018 J. Phys.: Conf. Ser. 1007 012005

[6] Y. Zhang, Y. Chen, H. Yu, X. Yang, W. Lu, and H. Liu, “Wearing-independent hand gesture recognition method based on EMG armband,” Personal and Ubiquitous Computing, vol. 22, no. 3, pp. 511–524, Jun. 2018.

[7] S. A. Fandakli and O. Aydemir, “A fast and highly accurate EMG signal classification approach for multifunctional prosthetic fingers control,” in 2017 40th International Conference on Telecommunications and Signal Processing (TSP), Barcelona, Spain, 2017, pp. 395–398.

[8] U. Côté-Allard et al., “Deep Learning for ElectroMyographic Hand Gesture Signal Classification Using Transfer Learning,” arXiv:1801.07756 [cs, stat], Jan. 2018.

[9] M. E. Benalcazar, A. G. Jaramillo, Jonathan, A. Zea, A. Paez, and V. H. Andaluz, “Hand gesture recognition using machine learning and the Myo armband,” in 2017 25th European Signal Processing Conference (EUSIPCO), Kos, Greece, 2017, pp. 1040–1044.

[10] F. Kerber, M. Puhl, and A. Krüger, “User-independent real-time hand gesture recognition based on surface electroMyography,” in Proceedings of the 19th International Conference on Human-Computer Interaction with Mobile Devices and Services - MobileHCI ’17, Vienna, Austria, 2017, pp. 1–7.

[11] X. Zhang, X. Chen, Y. Li, V. Lantz, K. Wang, and J. Yang, “A Framework for Hand Gesture Recognition Based on Accelerometer and EMG Sensors,” IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans, vol. 41, no. 6, pp. 1064–1076, Nov. 2011.

[12] M. Cognolato et al., “Multifunction control and evaluation of a 3D printed hand prosthesis with the Myo armband by hand amputees,” Bioengineering, preprint, Oct. 2018.

[13] K. Akhmadeev, E. Rampone, T. Yu, Y. Aoustin, and É. Le Carpentier, “A real-time gesture classification using surface EMG to control a robotics hand,” in ENOC 2017, Budapest, Hungary, 2017, vol. 2017.

[14] G. Vasan, “Teaching a Powered Prosthetic Arm with an Intact Arm Using Reinforcement Learning,” p. 112.

[15] “Press Release.” [Online]. Available: https://www.jhuapl.edu/PressRelease/160112. [Accessed: 19-Aug-2019].

[16] “Hand and Forarm – InMoov.” Accessed August 18, 2019. http://inmoov.fr/hand-and-forarm/.

[17] Mitchell, T. (1997). Machine Learning. McGraw Hill. p. 2. ISBN 978-0-07-042807-2.

[18] A. Müller and S. Guido, “Introduction to Machine Learning with Python: A Guide for Data Scientists,” 2016.

[19] F. Chollet and J. J. Allaire, Deep Learning with R. Manning Publications Company, 2018.

[20] V. Francois-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An Introduction to Deep Reinforcement Learning," FNT in Machine Learning, vol. 11, no. 3–4, pp. 219–354, 2018.

[21] "Google DeepMind's Go AI Has Mastered Chess And Shogi." [Online]. Available: https://www.forbes.com/sites/samshead/2018/12/07/google-deepminds-go-ai-has-mastered-chess-and-shogi/#25a4cc1b4ca3. [Accessed: 18-Aug-2019].

[22] O. S. Eluyode and D. T. Akomolafe, "Comparative study of biological and artificial neural networks," 2013.

[23] "The matrix calculus you need for deep learning." [Online]. Available: https://explained.ai/matrix-calculus/index.html. [Accessed: 18-Aug-2019].

[24] B. Karlik and A. V. Olgac, "Performance Analysis of Various Activation Functions in Generalized MLP Architectures o," Journal of Artificial Intelligence and Expert Systems, pp. 111–122, 2010.

[25] S. Hijazi, R. Kumar, and C. Rowen, "Using Convolutional Neural Networks for Image Recognition," p. 12.

[26] A. Sherstinsky, "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network," arXiv:1808.03314 [cs, stat], Aug. 2018.

[27] R. Goroshin and Y. LeCun, "Saturating Auto-Encoders," arXiv:1301.3577 [cs], Jan. 2013.

[28] Y. Lecun, L. Bottou, G. B. Orr, and K.-R. Müller, Efficient BackProp. 1998.

[29] Allamy, Haider. (2014). Methods To Avoid Over-Fitting And Under-Fitting In Supervised Machine Learning (comparative study).

[30] S. Demyanov, "Regularization methods for neural networks and related models," 2015.

[31] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” p. 30.

[32] Niklas Rosenstein 2018, Myo-python v1.0.3, available from https://github.com/NiklasRosenstein/Myo-python

Appendices

A.1: Code for Training and verification of EMG data for 5 finger movements

Python Code:

```
from __future__ import print_function
from collections import deque
from threading import Lock, Thread
import matplotlib
matplotlib.use("TkAgg")
import matplotlib.pyplot as plt

import numpy as np
import tensorflow as tf
from tensorflow import keras
from keras import regularizers
from keras.models import load_model

from sklearn import preprocessing

import myo

import time
import sys
import psutil
import os
import serial

# This training set will contain 1000 samples of 8 sensor values
global training_set
global number_of_samples
global index_training_set, middle_training_set,thumb_training_set,verification_set
global data_array
number_of_samples = 1000
data_array=[]

Sensor1 = np.zeros((1,number_of_samples))
Sensor2 = np.zeros((1,number_of_samples))
Sensor3 = np.zeros((1,number_of_samples))
Sensor4 = np.zeros((1,number_of_samples))
Sensor5 = np.zeros((1,number_of_samples))
Sensor6 = np.zeros((1,number_of_samples))
Sensor7 = np.zeros((1,number_of_samples))
Sensor8 = np.zeros((1,number_of_samples))

index_open_training_set = np.zeros((8,number_of_samples))
middle_open_training_set = np.zeros((8,number_of_samples))
thumb_open_training_set = np.zeros((8,number_of_samples))
ring_open_training_set = np.zeros((8,number_of_samples))
pinky_open_training_set = np.zeros((8,number_of_samples))

verification_set = np.zeros((8,number_of_samples))
training_set = np.zeros((8,number_of_samples)
```

```
thumb_open_label = 0
index_open_label = 1
middle_open_label = 2
ring_open_label = 3
pinky_open_label = 4

name = input("Enter name of Subject")

def find_one_hot(labels,classes):
    # = tf.constant(C)
    output = tf.one_hot(labels,classes,axis=0)
    sess = tf.Session()
    out = sess.run(output)
    sess.close
    return out

# This process checks if Myo Connect.exe is running
def check_if_process_running():

    try:
        for proc in psutil.process_iter():
            if proc.name()=='Myo Connect.exe':
                return True

        return False

    except (psutil.NoSuchProcess,psutil.AccessDenied, psutil.ZombieProcess):
        print (PROCNAME, " not running")

# If the process Myo Connect.exe is not running then we restart that process
def restart_process():
    PROCNAME = "Myo Connect.exe"

    for proc in psutil.process_iter():
        # check whether the process name matches
        if proc.name() == PROCNAME:
            proc.kill()
            # Wait a second
            time.sleep(1)

    while(check_if_process_running()==False):
        path = 'C:\Program Files (x86)\Thalmic Labs\Myo Connect\Myo Connect.exe'
        os.startfile(path)
        time.sleep(1)
        #while(check_if_process_running()==False):
        #    pass

    print("Process started")
    return True

# This is Myo-python SDK's listener that listens to EMG signal
class Listener(myo.DeviceListener):
    global data_array

    def __init__(self, n):
        self.n =
```

```
        self.lock = Lock()
        self.emg_data_queue = deque(maxlen=n)

    def on_connected(self, event):
        print("Myo Connected")
        self.started = time.time()
        event.device.stream_emg(True)

    def get_emg_data(self):
        with self.lock:
            print("H")

    def on_emg(self, event):
        with self.lock:
            self.emg_data_queue.append((event.emg))

            if len(list(self.emg_data_queue))>=number_of_samples:
                data_array.append(list(self.emg_data_queue))
                self.emg_data_queue.clear()
                return False

# This method is responsible for training EMG data
def Train(conc_array):
    global training_set
    global index_open_training_set, middle_open_training_set, thumb_open_training_set,
ring_open_training_set, pinky_open_training_set, verification_set
    global number_of_samples
    verification_set = np.zeros((8,number_of_samples))
    print (number_of_samples)
    labels = []
    print(conc_array,conc_array.shape)

# This division is to make the iterator for making labels run 30 times in inner loop and 10 times in outer
loop running total 300 times for 5 finger movements
    samples = conc_array.shape[0]/5
    # Now we append all data in training label
    # We iterate to make 5 finger movement labels.
    for i in range(0,5):
        for j in range(0,int(samples)):
            labels.append(i)
    labels = np.asarray(labels)
    print(labels, len(labels),type(labels))
    print(conc_array.shape[0])
    permutation_function = np.random.permutation(conc_array.shape[0])

    total_samples = conc_array.shape[0]
    all_shuffled_data,all_shuffled_labels = np.zeros((total_samples,8)),np.zeros((total_samples,8))

    all_shuffled_data,all_shuffled_labels = conc_array[permutation_function],labels[permutation_function]
    print(all_shuffled_data.shape)
    print(all_shuffled_labels.shape)
    number_of_training_samples = np.int(np.floor(0.8*total_samples))
    train_data = np.zeros((number_of_training_samples,8))
    train_labels = np.zeros((number_of_training_samples,8))
    print("TS ", number_of_training_samples, " S " , number_of_samples)
    number_of_validation_samples = np.int(total_samples-number_of_training_samples)
```

```
    train_data = all_shuffled_data[0:number_of_training_samples,:]
    train_labels = all_shuffled_labels[0:number_of_training_samples,]
    print("Length of train data is ", train_data.shape)
    validation_data = all_shuffled_data[number_of_training_samples:total_samples,:]
    validation_labels = all_shuffled_labels[number_of_training_samples:total_samples,]
    print("Length of validation data is ", validation_data.shape , " validation labels is " ,
validation_labels.shape)
    print(train_data,train_labels)

    model = keras.Sequential([
    # Input dimensions means input columns. Here we have 8 columns, one for each sensor
    keras.layers.Dense(8, activation=tf.nn.relu,input_dim=8,kernel_regularizer=regularizers.l2(0.1)),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(5, activation=tf.nn.softmax)])

    adam_optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None,
decay=0.0, amsgrad=False)
    model.compile(optimizer=adam_optimizer,
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

    history = model.fit(train_data, train_labels,
epochs=300,validation_data=(validation_data,validation_labels),batch_size=16)
    model.save('C:/Users/shaya/Desktop/'+name+'_five_finger_model.h5')
    plt.plot(history.history['acc'])
    plt.plot(history.history['val_acc'])
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()
    # summarize history for loss
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()

    averages = number_of_samples/50
    # Initializing array for verification_averages
    verification_averages = np.zeros((int(averages),8))


    while True:
        while True:
            try:
                input("Hold a finger movement and press enter to get its classification")
                hub = myo.Hub()
                number_of_samples=200
                listener = Listener(number_of_samples)
                hub.run(listener.on_event,20000)

            # Here we send the received number of samples making them a list of 1000 rows 8 columns
                verification_set = np.array((data_array[0]))
```

```
            data_array.clear()
            break
        except:
            while(restart_process()!=True):
                pass
            # Wait for 3 seconds until Myo Connect.exe starts
            time.sleep(3)

    verification_set = np.absolute(verification_set)

    div = 50
    # We add one because iterator below starts from 1
    batches = int(number_of_samples/div) + 1
    for i in range(1,batches):
        verification_averages[i-1,:] = np.mean(verification_set[(i-1)*div:i*div,:],axis=0)

    verification_data = verification_averages
    print("Verification matrix shape is " , verification_data.shape)

    predictions = model.predict(verification_data,batch_size=16)
    predicted_value = np.argmax(predictions[0])
    print(predictions[0])
    print(predicted_value)
    if predicted_value == 0:
        print("Thumb open")
    elif predicted_value == 1:
        print("Index finger open")
    elif predicted_value == 2:
        print("Middle finger open")
    elif predicted_value == 3:
        print("Ring finger open")
    elif predicted_value == 4:
        print("Pinky finger open")
    else:
        pass

    ## Here i send the predicted value to Arduino via Bluetooth so that it can open appropriate fingers ##

    # While 1 is used because sometimes bluetooth port throws exception in opening the COM Port
    # So i keep trying until the data is sent and confirmation received.
    while(1):
        try:
            # Bluetooth at COM6
            serialPort =
serial.Serial(port="COM6",baudrate=9600,bytesize=8,timeout=2,stopbits=serial.STOPBITS_ONE)
            value_to_bluetooth = str(predicted_value).encode()
            serialPort.write(value_to_bluetooth)
            time.sleep(1)
            if serialPort.in_waiting>0:
                serialString = serialPort.readline()
                print(serialString)
                # If we receive what we sent from Arduino bluetooth then all OK else bad value
                if serialString == value_to_bluetooth:
                    print("Received")
                else:
                    print("Bad value")
```

```
            serialPort.close()
            break
        except serial.SerialException as e:
            #There is no new data from serial port
            print (str(e))
        except TypeError as e:
            print (str(e))
            ser.port.close()

def main():
    global data_array
    index_open_training_set = np.zeros((8,number_of_samples))
    middle_open_training_set = np.zeros((8,number_of_samples))
    thumb_open_training_set = np.zeros((8,number_of_samples))
    ring_open_training_set = np.zeros((8,number_of_samples))
    pinky_open_training_set = np.zeros((8,number_of_samples))

    verification_set = np.zeros((8,number_of_samples))

    training_set = np.zeros((8,number_of_samples))

    # This function kills Myo Connect.exe and restarts it to make sure it is running
    # Because sometimes the application does not run and crash even when Myo Connect process is running
    # So i think its a good idea to just kill if its not running and restart it

    while(restart_process()!=True):
        pass
    # Wait for 3 seconds until Myo Connect.exe starts
    time.sleep(3)

    # Initialize the SDK of Myo Armband
    myo.init('C:\\Users\\shaya\\AppData\\Local\\Programs\\Python\\Python36\\myo64.dll')
    hub = myo.Hub()
    listener = Listener(number_of_samples)

    legend = ['Sensor 1','Sensor 2','Sensor 3','Sensor 4','Sensor 5','Sensor 6','Sensor 7','Sensor 8']

    ######### HERE WE GET TRAINING DATA FOR THUMB FINGER OPEN ########
    while True:
        try:
            hub = myo.Hub()
            listener = Listener(number_of_samples)
            input("Open THUMB ")
            start_time = time.time()
            hub.run(listener.on_event,20000)
            thumb_open_training_set = np.array((data_array[0]))
            print(thumb_open_training_set.shape)
            data_array.clear()
            break
        except:
            while(restart_process()!=True):
                pass
            # Wait for 3 seconds until Myo Connect.exe starts
            time.sleep(3)
```

```
    # Here we send the received number of samples making them a list of 1000 rows 8 columns just how we
need to feed to tensorflow

    ######### HERE WE GET TRAINING DATA FOR INDEX FINGER OPEN ########
    while True:
        try:
            input("Open index finger")
            start_time = time.time()
            hub = myo.Hub()
            listener = Listener(number_of_samples)

            hub.run(listener.on_event,20000)
            # Here we send the received number of samples making them a list of 1000 rows 8 columns
            index_open_training_set = np.array((data_array[0]))

            data_array.clear()
            break
        except:
            while(restart_process()!=True):
                pass
            # Wait for 3 seconds until Myo Connect.exe starts
            time.sleep(3)

    ###### HERE WE GET TRAINING DATA FOR MIDDLE FINGER OPEN #######
    while True:
        try:
            input("Open MIDDLE finger")
            start_time = time.time()
            hub = myo.Hub()
            listener = Listener(number_of_samples)
            hub.run(listener.on_event,20000)
            middle_open_training_set = np.array((data_array[0]))
            data_array.clear()
            break
        except:
            while(restart_process()!=True):
                pass
            # Wait for 3 seconds until Myo Connect.exe starts
            time.sleep(3)

    # Here we send the received number of samples making them a list of 1000 rows 8 columns

    ######## HERE WE GET TRAINING DATA FOR RING FINGER OPEN ########
    while True:
        try:
            input("Open Ring finger")
            start_time = time.time()
            hub = myo.Hub()
            listener = Listener(number_of_samples)
            hub.run(listener.on_event,20000)
            ring_open_training_set = np.array((data_array[0]))
            data_array.clear()
            break
        except:
            while(restart_process()!=True):
                pass
```

```
            # Wait for 3 seconds until Myo Connect.exe starts
            time.sleep(3)

    ######### HERE WE GET TRAINING DATA FOR PINKY FINGER OPEN #########
    while True:
        try:
            input("Open Pinky finger")
            start_time = time.time()
            hub = myo.Hub()
            listener = Listener(number_of_samples)
            hub.run(listener.on_event,20000)
            pinky_open_training_set = np.array((data_array[0]))
            data_array.clear()
            break
        except:
            while(restart_process()!=True):
                pass
            # Wait for 3 seconds until Myo Connect.exe starts
            time.sleep(3)

    # Absolute of finger open data
    thumb_open_training_set = np.absolute(thumb_open_training_set)
    index_open_training_set = np.absolute(index_open_training_set)
    middle_open_training_set = np.absolute(middle_open_training_set)
    ring_open_training_set = np.absolute(ring_open_training_set)
    pinky_open_training_set = np.absolute(pinky_open_training_set)

    div = 50
    averages = int(number_of_samples/div)
    thumb_open_averages = np.zeros((int(averages),8))
    index_open_averages = np.zeros((int(averages),8))
    middle_open_averages = np.zeros((int(averages),8))
    ring_open_averages = np.zeros((int(averages),8))
    pinky_open_averages = np.zeros((int(averages),8))

    # Here we are calculating the mean values of all finger open data set and storing them as n/50 samples
because 50 batches of n samples is equal to n/50 averages
    for i in range(1,averages+1):
        thumb_open_averages[i-1,:] = np.mean(thumb_open_training_set[(i-1)*div:i*div,:],axis=0)
        index_open_averages[i-1,:] = np.mean(index_open_training_set[(i-1)*div:i*div,:],axis=0)
        middle_open_averages[i-1,:] = np.mean(middle_open_training_set[(i-1)*div:i*div,:],axis=0)
        ring_open_averages[i-1,:] = np.mean(ring_open_training_set[(i-1)*div:i*div,:],axis=0)
        pinky_open_averages[i-1,:] = np.mean(pinky_open_training_set[(i-1)*div:i*div,:],axis=0)

    # Here we stack all the data row wise
    conc_array = np.concatenate([thumb_open_averages,index_open_averages,middle_open_averages,
ring_open_averages, pinky_open_averages],axis=0)
    print(conc_array.shape)
    np.savetxt('C:/Users/shaya/Desktop/'+name+'_five_movements.txt', conc_array, fmt='%i')
    # In this method the EMG data gets trained and verified
    Train(conc_array)

if __name__ == '__main__':
    main()
```

A.2: Code for Training and verification of EMG data for 12 finger movements

Python Code:

```
from __future__ import print_function
from collections import deque
from threading import Lock, Thread
import matplotlib
matplotlib.use("TkAgg")
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow import keras
from keras import regularizers
from keras.models import load_model
from sklearn import preprocessing
import myo
import time
import sys
import psutil
import os

# Used for bluetooth signal sending and receiving
import serial

# This training set will contain 1000 samples of 8 sensor values
global training_set
global number_of_samples
global index_training_set, middle_training_set,thumb_training_set,verification_set
global data_array
number_of_samples = 1000
data_array=[]

Sensor1 = np.zeros((1,number_of_samples))
Sensor2 = np.zeros((1,number_of_samples))
Sensor3 = np.zeros((1,number_of_samples))
Sensor4 = np.zeros((1,number_of_samples))
Sensor5 = np.zeros((1,number_of_samples))
Sensor6 = np.zeros((1,number_of_samples))
Sensor7 = np.zeros((1,number_of_samples))
Sensor8 = np.zeros((1,number_of_samples))

unrecognized_training_set = np.zeros((8,number_of_samples))
index_open_training_set = np.zeros((8,number_of_samples))
middle_open_training_set = np.zeros((8,number_of_samples))
thumb_open_training_set = np.zeros((8,number_of_samples))
ring_open_training_set = np.zeros((8,number_of_samples))
pinky_open_training_set = np.zeros((8,number_of_samples))
two_open_training_set = np.zeros((8,number_of_samples))
three_open_training_set = np.zeros((8,number_of_samples))
four_open_training_set = np.zeros((8,number_of_samples))
five_open_training_set = np.zeros((8,number_of_samples))
all_fingers_closed_training_set = np.zeros((8,number_of_samples))
grasp_training_set = np.zeros((8,number_of_samples))
```

```
pick_training_set = np.zeros((8,number_of_samples))
verification_set = np.zeros((8,number_of_samples))
training_set = np.zeros((8,number_of_samples))

thumb_open_label = 0
index_open_label = 1
middle_open_label = 2
ring_open_label = 3
pinky_open_label = 4
two_open_label = 5
three_open_label = 6
four_open_label = 7
five_open_label = 8
all_fingers_closed_label = 9
grasp_label = 10
pick_label = 11

name = input("Enter name of Subject")

def find_one_hot(labels,classes):
    output = tf.one_hot(labels,classes,axis=0)
    sess = tf.Session()
    out = sess.run(output)
    sess.close
    return out

# Check if Myo Connect.exe process is running
def check_if_process_running():
    try:
        for proc in psutil.process_iter():
            if proc.name()=='Myo Connect.exe':
                return True
        return False
    except (psutil.NoSuchProcess,psutil.AccessDenied, psutil.ZombieProcess):
        print (PROCNAME, " not running")

# Restart myo connect.exe process
def restart_process():
    PROCNAME = "Myo Connect.exe"
    for proc in psutil.process_iter():
        # check whether the process name matches
        if proc.name() == PROCNAME:
            proc.kill()
            # Wait a second
            time.sleep(1)

    while(check_if_process_running()==False):
        path = 'C:\Program Files (x86)\Thalmic Labs\Myo Connect\Myo Connect.exe'
        os.startfile(path)
        time.sleep(1)

    print("Process started")
    return True

# This class from Myo-python SDK listens to EMG signals from armband
class Listener(myo.DeviceListener):
```

```
global data_array

    def __init__(self, n):
        self.n = n
        self.lock = Lock()
        self.emg_data_queue = deque(maxlen=n)

    def on_connected(self, event):
        print("Myo Connected")
        self.started = time.time()
        event.device.stream_emg(True)

    def get_emg_data(self):
        with self.lock:
            print("H")

    def on_emg(self, event):
        with self.lock:
            self.emg_data_queue.append((event.emg))

            if len(list(self.emg_data_queue))>=number_of_samples:
                data_array.append(list(self.emg_data_queue))
                self.emg_data_queue.clear()
                return False

# This method is responsible for training EMG data
def Train(conc_array):
    global training_set
    global index_open_training_set, middle_open_training_set, thumb_open_training_set,
ring_open_training_set, pinky_open_training_set, verification_set
    global two_open_training_set, three_open_training_set,
four_open_training_set,five_open_training_set,all_fingers_closed_training_set,grasp_training_set,pick_trai
ning_set
    global number_of_samples
    verification_set = np.zeros((8,number_of_samples))
    print (number_of_samples)
    labels = []
    print(conc_array,conc_array.shape)

    # This division is to make the iterator for making labels run 30 times in inner loop and 10 times in outer
loop running total 300 times for 10 finger movements
    samples = conc_array.shape[0]/12
    # Now we append all data in training label
    # We iterate to make 12 finger movement labels.
    for i in range(0,12):
        for j in range(0,int(samples)):
            labels.append(i)
    labels = np.asarray(labels)
    print(labels, len(labels),type(labels))
    print(conc_array.shape[0])
    permutation_function = np.random.permutation(conc_array.shape[0])

    total_samples = conc_array.shape[0]
    all_shuffled_data,all_shuffled_labels = np.zeros((total_samples,8)),np.zeros((total_samples,8))

    all_shuffled_data,all_shuffled_labels = conc_array[permutation_function],labels[permutation_function]
```

```
print(all_shuffled_data.shape)
    print(all_shuffled_labels.shape)

    number_of_training_samples = np.int(np.floor(0.8*total_samples))
    train_data = np.zeros((number_of_training_samples,8))
    train_labels = np.zeros((number_of_training_samples,8))
    print("TS ", number_of_training_samples, " S " , number_of_samples)
    number_of_validation_samples = np.int(total_samples-number_of_training_samples)
    train_data = all_shuffled_data[0:number_of_training_samples,:]
    train_labels = all_shuffled_labels[0:number_of_training_samples,]
    print("Length of train data is ", train_data.shape)
    validation_data = all_shuffled_data[number_of_training_samples:total_samples,:]
    validation_labels = all_shuffled_labels[number_of_training_samples:total_samples,]
    print("Length of validation data is ", validation_data.shape , " validation labels is " ,
validation_labels.shape)
    print(train_data,train_labels)

    model = keras.Sequential([
    # Input dimensions means input columns. Here we have 8 columns, one for each sensor
    keras.layers.Dense(8, activation=tf.nn.relu,input_dim=8,      kernel_regularizer=regularizers.l2(0.1)),
keras.layers.BatchNormalization(),
    keras.layers.Dense(12, activation=tf.nn.softmax)])

    adam_optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None,
decay=0.0, amsgrad=False)
    model.compile(optimizer=adam_optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    history = model.fit(train_data, train_labels, epochs=300,
validation_data=(validation_data,validation_labels), batch_size=16)
    model.save('C:/Users/shaya/Desktop/'+name+'_realistic_model.h5')
    # Here we display the training and test loss for model
    plt.plot(history.history['acc'])
    plt.plot(history.history['val_acc'])
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()
    # summarize history for loss
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.show()

    averages = number_of_samples/50
    # Initializing array for verification_averages
    verification_averages = np.zeros((int(averages),8))


    while True:
        while True:
            try:
                input("Hold a finger movement and press enter to get its classification")
```

```
            hub = myo.Hub()
            number_of_samples=200
            listener = Listener(number_of_samples)
            hub.run(listener.on_event,20000)

        # Here we send the received number of samples making them a list of 1000 rows 8 columns
            verification_set = np.array((data_array[0]))
            data_array.clear()
            break
        except:
            while(restart_process()!=True):
                pass
    # Wait for 3 seconds until Myo Connect.exe starts
            time.sleep(3)
    verification_set = np.absolute(verification_set)

    div = 50
    # We add one because iterator below starts from 1
    batches = int(number_of_samples/div) + 1
    for i in range(1,batches):
        verification_averages[i-1,:] = np.mean(verification_set[(i-1)*div:i*div,:],axis=0)

    verification_data = verification_averages
    print("Verification matrix shape is " , verification_data.shape)
    predictions = model.predict(verification_data,batch_size=16)
    predicted_value = np.argmax(predictions[0])
    print(predictions[0])
    print(predicted_value)
    if predicted_value == 0:
        print("Thumb open")
    elif predicted_value == 1:
        print("Index finger open")
    elif predicted_value == 2:
        print("Middle finger open")
    elif predicted_value == 3:
        print("Ring finger open")
    elif predicted_value == 4:
        print("Pinky finger open")
    elif predicted_value == 5:
        print("Two fingers open")
    elif predicted_value == 6:
        print("Three fingers open")
    elif predicted_value == 7:
        print("Four fingers open")
    elif predicted_value == 8:
        print("Five fingers open")
    elif predicted_value == 9:
        print("All fingers closed")
    elif predicted_value == 10:
        print("Grasp movement")
    elif predicted_value == 11:
        print("Pick movement")
    else:
        pass
    #### Here i send the predicted value to Arduino via Bluetooth so that it can open appropriate fingers
####
```

```
        # While 1 is used because sometimes bluetooth port throws exception in opening the COM Port
        # So i keep trying until the data is sent and confirmation received.
        while(1):
            try:
                # Bluetooth at COM6
                serialPort =
serial.Serial(port="COM6",baudrate=9600,bytesize=8,timeout=2,stopbits=serial.STOPBITS_ONE)
                value_to_bluetooth = str(predicted_value).encode()
                if predicted_value == 10:
                    value_to_bluetooth = 'a'.encode()
                if predicted_value == 11:
                    value_to_bluetooth = 'b'.encode()
                serialPort.write(value_to_bluetooth)
                time.sleep(1)
                if serialPort.in_waiting>0:
                    serialString = serialPort.readline()
                    print(serialString)
                    # If we receive what we sent from Arduino bluetooth then all OK else bad value
                    if serialString == value_to_bluetooth:
                        print("Received")
                    else:
                        print("Bad value")
                serialPort.close()
                break
            except serial.SerialException as e:
                #There is no new data from serial port
                print (str(e))
            except TypeError as e:
                print (str(e))
                ser.port.close()

def main():
    global data_array
    unrecognized_training_set = np.zeros((8,number_of_samples))
    index_open_training_set = np.zeros((8,number_of_samples))
    middle_open_training_set = np.zeros((8,number_of_samples))
    thumb_open_training_set = np.zeros((8,number_of_samples))
    ring_open_training_set = np.zeros((8,number_of_samples))
    pinky_open_training_set = np.zeros((8,number_of_samples))
    two_open_training_set = np.zeros((8,number_of_samples))
    three_open_training_set = np.zeros((8,number_of_samples))
    four_open_training_set = np.zeros((8,number_of_samples))
    five_open_training_set = np.zeros((8,number_of_samples))
    all_fingers_closed_training_set = np.zeros((8,number_of_samples))
    grasp_training_set = np.zeros((8,number_of_samples))
    pick_training_set = np.zeros((8,number_of_samples))

    verification_set = np.zeros((8,number_of_samples))

    training_set = np.zeros((8,number_of_samples))
    # This function kills Myo Connect.exe and restarts it to make sure it is running
    # Because sometimes the application does not run even when Myo Connect process is running
    # So i think its a good idea to just kill if its not running and restart it
```

```
while(restart_process()!=True):
    pass
# Wait for 3 seconds until Myo Connect.exe starts
time.sleep(3)

# Initialize the SDK of Myo Armband
myo.init('C:\\Users\\shaya\\AppData\\Local\\Programs\\Python\\Python36\\myo64.dll')
hub = myo.Hub()
listener = Listener(number_of_samples)

legend = ['Sensor 1','Sensor 2','Sensor 3','Sensor 4','Sensor 5','Sensor 6','Sensor 7','Sensor 8']

########## HERE WE GET TRAINING DATA FOR THUMB FINGER OPEN ########
while True:
    try:
        hub = myo.Hub()
        listener = Listener(number_of_samples)
        input("Open THUMB ")
        start_time = time.time()
        hub.run(listener.on_event,20000)
        thumb_open_training_set = np.array((data_array[0]))
        print(thumb_open_training_set.shape)
        data_array.clear()
        break
    except:
        while(restart_process()!=True):
            pass
        # Wait for 3 seconds until Myo Connect.exe starts
        time.sleep(3)

# Here we send the received number of samples making them a list of 1000 rows 8 columns just how we
need to feed to tensorflow

################## HERE WE GET TRAINING DATA FOR INDEX FINGER OPEN ########
while True:
    try:
        input("Open index finger")
        start_time = time.time()
        hub = myo.Hub()
        listener = Listener(number_of_samples)

        hub.run(listener.on_event,20000)
        # Here we send the received number of samples making them a list of 1000 rows 8 columns
        index_open_training_set = np.array((data_array[0]))

        data_array.clear()
        break
    except:
        while(restart_process()!=True):
            pass
        # Wait for 3 seconds until Myo Connect.exe starts
        time.sleep(3)

######### HERE WE GET TRAINING DATA FOR MIDDLE FINGER OPEN ############
while True:
    try:
```

```
        input("Open MIDDLE finger")
        start_time = time.time()
        hub = myo.Hub()
        listener = Listener(number_of_samples)
        hub.run(listener.on_event,20000)
        middle_open_training_set = np.array((data_array[0]))
        data_array.clear()
        break
    except:
        while(restart_process()!=True):
            pass
        # Wait for 3 seconds until Myo Connect.exe starts
        time.sleep(3)

################## HERE WE GET TRAINING DATA FOR RING FINGER OPEN ##########
while True:
    try:
        input("Open Ring finger")
        start_time = time.time()
        hub = myo.Hub()
        listener = Listener(number_of_samples)
        hub.run(listener.on_event,20000)
        ring_open_training_set = np.array((data_array[0]))
        data_array.clear()
        break
    except:
        while(restart_process()!=True):
            pass
        # Wait for 3 seconds until Myo Connect.exe starts
        time.sleep(3)

############ HERE WE GET TRAINING DATA FOR PINKY FINGER OPEN ############
while True:
    try:
        input("Open Pinky finger")
        start_time = time.time()
        hub = myo.Hub()
        listener = Listener(number_of_samples)
        hub.run(listener.on_event,20000)
        pinky_open_training_set = np.array((data_array[0]))
        data_array.clear()
        break
    except:
        while(restart_process()!=True):
            pass
        # Wait for 3 seconds until Myo Connect.exe starts
        time.sleep(3)

############ HERE WE GET TRAINING DATA FOR TWO FINGER OPEN #############
while True:
    try:
        input("Open Two fingers")
        start_time = time.time()
        hub = myo.Hub()
        listener = Listener(number_of_samples)
        hub.run(listener.on_event,20000)
```

```
        two_open_training_set = np.array((data_array[0]))
        data_array.clear()
        break
    except:
        while(restart_process()!=True):
            pass
        # Wait for 3 seconds until Myo Connect.exe starts
        time.sleep(3)

######### HERE WE GET TRAINING DATA FOR THREE FINGER OPEN ##########
while True:
    try:
        input("Open Three fingers")
        start_time = time.time()
        hub = myo.Hub()
        listener = Listener(number_of_samples)
        hub.run(listener.on_event,20000)
        three_open_training_set = np.array((data_array[0]))
        data_array.clear()
        break
    except:
        while(restart_process()!=True):
            pass
        # Wait for 3 seconds until Myo Connect.exe starts
        time.sleep(3)

########## HERE WE GET TRAINING DATA FOR THREE FINGER OPEN ##########
while True:
    try:
        input("Open Four fingers")
        start_time = time.time()
        hub = myo.Hub()
        listener = Listener(number_of_samples)
        hub.run(listener.on_event,20000)
        four_open_training_set = np.array((data_array[0]))
        data_array.clear()
        break
    except:
        while(restart_process()!=True):
            pass
        # Wait for 3 seconds until Myo Connect.exe starts
        time.sleep(3)

########## HERE WE GET TRAINING DATA FOR FIVE FINGER OPEN ###########
while True:
    try:
        input("Open Five fingers")
        start_time = time.time()
        hub = myo.Hub()
        listener = Listener(number_of_samples)
        hub.run(listener.on_event,20000)
        five_open_training_set = np.array((data_array[0]))
        data_array.clear()
        break
    except:
        while(restart_process()!=True):
```

```
            pass
        # Wait for 3 seconds until Myo Connect.exe starts
        time.sleep(3)

######### HERE WE GET TRAINING DATA FOR ALL FINGERS CLOSED ###########
while True:
    try:
        input("Make all fingers closed")
        start_time = time.time()
        hub = myo.Hub()
        listener = Listener(number_of_samples)
        hub.run(listener.on_event,20000)
        all_fingers_closed_training_set = np.array((data_array[0]))
        data_array.clear()
        break
    except:
        while(restart_process()!=True):
            pass
        # Wait for 3 seconds until Myo Connect.exe starts
        time.sleep(3)

######### HERE WE GET TRAINING DATA FOR GRASP MOVEMENT ###########
while True:
    try:
        input("Make Grasp movement")
        start_time = time.time()
        hub = myo.Hub()
        listener = Listener(number_of_samples)
        hub.run(listener.on_event,20000)
        grasp_training_set = np.array((data_array[0]))
        data_array.clear()
        break
    except:
        while(restart_process()!=True):
            pass
        # Wait for 3 seconds until Myo Connect.exe starts
        time.sleep(3)

######### HERE WE GET TRAINING DATA FOR PICK MOVEMENT ##########
while True:
    try:
        input("Make Pick movement")
        start_time = time.time()
        hub = myo.Hub()
        listener = Listener(number_of_samples)
        hub.run(listener.on_event,20000)
        pick_training_set = np.array((data_array[0]))
        data_array.clear()
        break
    except:
        while(restart_process()!=True):
            pass
        # Wait for 3 seconds until Myo Connect.exe starts
        time.sleep(3)

# Absolute of finger open data
```

```
    thumb_open_training_set = np.absolute(thumb_open_training_set)
    index_open_training_set = np.absolute(index_open_training_set)
    middle_open_training_set = np.absolute(middle_open_training_set)
    ring_open_training_set = np.absolute(ring_open_training_set)
    pinky_open_training_set = np.absolute(pinky_open_training_set)
    two_open_training_set = np.absolute(two_open_training_set)
    three_open_training_set = np.absolute(three_open_training_set)
    four_open_training_set = np.absolute(four_open_training_set)
    five_open_training_set = np.absolute(five_open_training_set)
    all_fingers_closed_training_set = np.absolute(all_fingers_closed_training_set)
    grasp_training_set = np.absolute(grasp_training_set)
    pick_training_set = np.absolute(pick_training_set)

    div = 50
    averages = int(number_of_samples/div)
    thumb_open_averages = np.zeros((int(averages),8))
    index_open_averages = np.zeros((int(averages),8))
    middle_open_averages = np.zeros((int(averages),8))
    ring_open_averages = np.zeros((int(averages),8))
    pinky_open_averages = np.zeros((int(averages),8))
    two_open_averages = np.zeros((int(averages),8))
    three_open_averages = np.zeros((int(averages),8))
    four_open_averages = np.zeros((int(averages),8))
    five_open_averages = np.zeros((int(averages),8))
    all_fingers_closed_averages = np.zeros((int(averages),8))
    grasp_averages = np.zeros((int(averages),8))
    pick_averages = np.zeros((int(averages),8))

    # Here we are calculating the mean values of all finger open data set and storing them as n/50 samples
because 50 batches of n samples is equal to n/50 averages
    for i in range(1,averages+1):
        thumb_open_averages[i-1,:] = np.mean(thumb_open_training_set[(i-1)*div:i*div,:],axis=0)
        index_open_averages[i-1,:] = np.mean(index_open_training_set[(i-1)*div:i*div,:],axis=0)
        middle_open_averages[i-1,:] = np.mean(middle_open_training_set[(i-1)*div:i*div,:],axis=0)
        ring_open_averages[i-1,:] = np.mean(ring_open_training_set[(i-1)*div:i*div,:],axis=0)
        pinky_open_averages[i-1,:] = np.mean(pinky_open_training_set[(i-1)*div:i*div,:],axis=0)
        two_open_averages[i-1,:] = np.mean(two_open_training_set[(i-1)*div:i*div,:],axis=0)
        three_open_averages[i-1,:] = np.mean(three_open_training_set[(i-1)*div:i*div,:],axis=0)
        four_open_averages[i-1,:] = np.mean(four_open_training_set[(i-1)*div:i*div,:],axis=0)
        five_open_averages[i-1,:] = np.mean(five_open_training_set[(i-1)*div:i*div,:],axis=0)
        all_fingers_closed_averages[i-1,:] = np.mean(all_fingers_closed_training_set[(i-
1)*div:i*div,:],axis=0)
        grasp_averages[i-1,:] = np.mean(grasp_training_set[(i-1)*div:i*div,:],axis=0)
        pick_averages[i-1,:] = np.mean(pick_training_set[(i-1)*div:i*div,:],axis=0)

    # Here we stack all the data row wise
    conc_array = np.concatenate([thumb_open_averages,index_open_averages,middle_open_averages,
ring_open_averages,pinky_open_averages,two_open_averages,three_open_averages,four_open_averages,
five_open_averages,all_fingers_closed_averages,grasp_averages,pick_averages],axis=0)
    print(conc_array.shape)
    np.savetxt('C:/Users/shaya/Desktop/'+name+'.txt', conc_array, fmt='%i')
    # In this method the EMG data gets trained and verified
    Train(conc_array)

if __name__ == '__main__':
    main()
```

A.3: Code for receiving data from bluetooth to Robotic hand and sending acknowledgment to PC

Arduino Code:

```
#include <SoftwareSerial.h>
#include <Wire.h>
#include <Adafruit_PWMServoDriver.h>

SoftwareSerial mySerial(4, 2); // RX, TX
int ledpin=13; // led on D13 will show blink on / off
int BluetoothData; // the data given from Computer

// called this way, it uses the default address 0x40
Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver();

// Depending on your servo make, the pulse width min and max may vary, you
// want these to be as small/large as possible without hitting the hard stop
// for max range. You'll have to tweak them as necessary to match the servos you
// have!
#define SERVOMIN  400 // this is the 'minimum' pulse length count (out of 4096)
#define SERVOMAX  750 // this is the 'maximum' pulse length count (out of 4096)

// our servo # counter
uint8_t servonum = 0;
// Here we store previous finger's number, initially an unrealistic number is assigned
uint8_t previous_finger_servo_number = 50;

void close_all_fingers()
{
  // Initially we begin by closing all fingers one by one
  for (int i=0;i<5;i++)
  {
    for (uint16_t pulselen = SERVOMIN; pulselen < SERVOMAX; pulselen++)
    {
      pwm.setPWM(i, 0, pulselen);
    }

    // Stop all motors after they are closed so they dont vibrate.
    pwm.setPWM(i, 0, 4096);

    delay(500);
  }
}
// First we initialize UART and close all fingers of robotic hand
void setup()
{
  // put your setup code here, to run once:
  mySerial.begin(9600);
  pinMode(ledpin,OUTPUT);
  pwm.begin();
  pwm.setPWMFreq(60);  // Analog servos run at ~60 Hz updates
  delay(10);
  close_all_fingers();
```

```
}

// you can use this function if you'd like to set the pulse length in seconds
// e.g. setServoPulse(0, 0.001) is a ~1 millisecond pulse width. its not precise!
void setServoPulse(uint8_t n, double pulse)
{
  double pulselength;

  pulselength = 1000000;   // 1,000,000 us per second
  pulselength /= 60;   // 60 Hz
//  Serial.print(pulselength); Serial.println(" us per period");
  pulselength /= 4096;  // 12 bits of resolution
//  Serial.print(pulselength); Serial.println(" us per bit");
  pulse *= 1000000;  // convert to us
  pulse /= pulselength;
//  Serial.println(pulse);
  pwm.setPWM(n, 0, pulse);
}

// This function opens the designated servo motor
void finger_open(uint8_t servonum)
{
  // Open the classified finger
  for (uint16_t pulselen = SERVOMIN; pulselen < SERVOMAX; pulselen--)
  {
    pwm.setPWM(servonum, 0, pulselen);
  }
  // This line stops the motor so that it doesnt keep "buzzing"
  pwm.setPWM(servonum,0,4096);
  delay(200);
}

// This function closes the designated servo motor
void finger_close(uint8_t servonum)
{
  // Close the previous finger
  for (uint16_t pulselen = SERVOMIN; pulselen < SERVOMAX; pulselen++)
  {
    pwm.setPWM(servonum, 0, pulselen);
  }
  // This line stops the motor so that it doesnt keep "buzzing"
  pwm.setPWM(servonum,0,4096);
  delay(200);
}
// Runs forever
void loop()
{
// Only run if something is received from serial port
  if (mySerial.available()>0)
  {
    // We read the bluetooth data. In this case the finger to be moved by servo motor
    BluetoothData=mySerial.read();
    if (BluetoothData == '0')
    {
      servonum = 0;
    }
```

```
  else if (BluetoothData == '1')
  {
   servonum = 1;
  }
  else if (BluetoothData == '2')
  {
   servonum = 2;
  }
  else if (BluetoothData == '3')
  {
   servonum = 3;
  }
  else if (BluetoothData == '4')
  {
   servonum = 4;
  }
  else if (BluetoothData == '5')
  {
   servonum = 5;
  }
  else if (BluetoothData == '6')
  {
   servonum = 6;
  }
  else if (BluetoothData == '7')
  {
   servonum = 7;
  }
  else if (BluetoothData == '8')
  {
   servonum = 8;
  }
  else if (BluetoothData == '9')
  {
   servonum = 9;
  }
  else if (BluetoothData == 'a')
  {
   servonum = 10;
  }
  else if (BluetoothData == 'b')
  {
   servonum = 11;
  }
  else
  {
  }
  mySerial.write(BluetoothData);
  digitalWrite(ledpin,!digitalRead(ledpin));
  // If previous finger was different than the received finger servo number then open received servo
number otherwise leave as it is
  if (previous_finger_servo_number != servonum)
  {
    // Open thumb finger, close others
    if (servonum==0)
    {
```

```
  finger_open(0);
  finger_close(1);
  finger_close(2);
  finger_close(3);
  finger_close(4);
}
// Open index finger, close others
else if (servonum==1)
{
  finger_open(1);
  finger_close(0);
  finger_close(2);
  finger_close(3);
  finger_close(4);
}
// Open middle finger, close others
else if (servonum==2)
{
  finger_open(2);
  finger_close(0);
  finger_close(1);
  finger_close(3);
  finger_close(4);
}
// Open ring finger, close others
else if (servonum==3)
{
  finger_open(3);
  finger_close(0);
  finger_close(1);
  finger_close(2);
  finger_close(4);
}
// Open pinky finger, close others
else if (servonum==4)
{
  finger_open(4);
  finger_close(0);
  finger_close(1);
  finger_close(2);
  finger_close(3);
}
// Two fingers open i.e. index and middle
else if (servonum == 5)
{
  finger_close(0);
  finger_close(3);
  finger_close(4);
  finger_open(1);
  finger_open(2);
}

// Three fingers open i.e. index, middle and ring
else if (servonum == 6)
{
  finger_close(0);
```

```
        finger_close(4);
        finger_open(1);
        finger_open(2);
        finger_open(3);
      }
      // Four fingers open i.e. index, middle, ring and pinky
      else if (servonum == 7)
      {
        finger_close(0);
        finger_open(1);
        finger_open(2);
        finger_open(3);
        finger_open(4);
      }
      // Five fingers open i.e. index, middle and ring
      else if (servonum == 8)
      {
        finger_open(0);
        finger_open(1);
        finger_open(2);
        finger_open(3);
        finger_open(4);
      }
      // All fingers closed (fist)
      else if (servonum == 9 || servonum == 10)
      {
        finger_close(0);
        finger_close(1);
        finger_close(2);
        finger_close(3);
        finger_close(4);
      }
      // Pick movement. Thumb, index middle close rest open
      else if (servonum == 11)
      {
        finger_close(0);
        finger_close(1);
        finger_close(2);
        finger_open(3);
        finger_open(4);
      }
      previous_finger_servo_number = servonum;
    }
    delay(100);
  }
}
```