DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Robotics, Artificial Intelligence and Real-time Systems

Comparison of Solutions for Efficient Implementation of Sensor based Hand Prosthesis Control on a Microcontroller using Regression based Machine Learning and Neural Networks

Shreyas Vijaykumar Waichal

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Robotics, Artificial Intelligence and Real-time Systems

Comparison of Solutions for Efficient Implementation of Sensor based Hand Prosthesis Control on a Microcontroller using Regression based Machine Learning and Neural Networks

Vergleich von Lösungen zur effizienten Implementierung einer sensorgestützten Handprothesensteuerung auf einem Mikrocontroller unter Verwendung von regressionsbasiertem maschinellem Lernen und neuronalen Netzen

Author:	Shreyas Vijaykumar Waichal
Supervisor:	Prof. Dr.-Ing. habil. Alois Christian Knoll
Advisor:	Mathilde Connan, Bernhard Vodermayer
Submission Date:	6th April 2020

I confirm that this master's thesis in robotics, artificial intelligence and real-time systems is my own work and I have documented all sources and material used. I am aware that only this digital version is valid and will be graded. Further digital versions or printed versions will not be considered for grading.

SWaichal

Munich, 6th April 2020 Shreyas Vijaykumar Waichal

Acknowledgments

I would like to express my gratitude to Mathilde Connan (DLR), Bernhard Vodermayer (DLR), Markus Nowak (DLR) and Andrea Gigli (DLR) for their continued support, guidance, and encouragement during my thesis. I also thank Maximilian Denninger (DLR) for his valuable suggestions.

I sincerely appreciate the "Adaptive Bionics Interfaces" group at DLR for providing excellent research facilities and the necessary apparatus required for the completion of my thesis.

Abstract

Hands are an essential part of daily human life. Without them, one cannot perform even the simplest of activities like brushing, opening doors. However, many people around the world suffer from upper limb amputations due to a variety of reasons. Prosthetic limbs have been on the market for many years. They seem to be a viable solution to boost the confidence and morale of amputees. Despite this fact, such artificial limbs are still limited in terms of their functionality, efficiency, and affordability. Many different control strategies recognizing different hand gestures have been investigated by the scientific community. But it is also important to be able to deploy these algorithms on compact and economical processing hardware. Hence, in this thesis, two control algorithms, namely, regression and neural networks are analyzed for identification of hand movements and their implementation roadmap on a cost and energy-efficient Cortex M microcontroller has been discussed. This solution could help the prosthesis to reach out to the common man.

Contents

1. Introduction

This thesis is written with the Chair of Robotics, Artificial Intelligence and Real-time Systems at TUM, in cooperation with the Institute of Robotics and Mechatronics at the German Aerospace Center (DLR). I started my work as part of my internship in the Adaptive Bionics Group at DLR, Oberpfaffenhofen and the research topic was to find an efficient implementation of regression and neural network control algorithms for an artificial limb on a resource-constrained embedded platform.

1.1. Motivation

Hands play an essential part in the daily life of humans. Without them, we would not be able to perform any sort of movement or activity. Because of their excellent mobility and flexibility, humans can complete multiple complex actions with their hands. The thumb provides humans with a higher level of adroitness which enables humans to achieve tasks such as holding, lifting, grasping and picking more efficiently than other living beings. Hands also give us a perception of touch as they are the primary means for obtaining information like pain, tactile and temperature. Moreover, it is worth noting that hands are used to express sign language which helps us to communicate with others.

The World Health Organization's report on disability shows that about ten percent of the total world's population consists of individuals with disabilities. Additionally, the World Health Survey shows that around 785 million (15.6%) persons, 15 years and older, suffer impairments and about 110 million people (2.2%) have very significant difficulties in functioning, while the Global Burden of Disease estimates a figure of around 975 million (19.4%) specially-abled persons of which 190 million (3.8%) have severe ailments [1]. Out of these disabilities, some people suffer from upper limb amputations. Considering the crucial functions of hands, the everyday life of such amputees is complicated. It inevitably takes more time for them to execute something that is typically considered natural. For example, it is arduous for them to apply toothpaste onto a toothbrush, cook, tie shoes, or even open a bottle. Furthermore, the person missing two hands normally cannot lead a comfortable life. They cannot even eat by themselves easily and mostly must be cared by their loved ones. Moreover, impaired people have a tougher time being employed. Most jobs require people to have hands, which limits the domain of work for such handicapped people. They cannot do assembly-line jobs where heavy manual labor is needed. More than that, the physically challenged are sometimes discriminated just because they are lacking hands. Along with such physical limitations they also have to bear emotional and psychological challenges.

1.2. Past Research

Artificial hands seem to be a possibility that can help amputees to be more flexible in performing their daily activities and provide them a better quality of life. Such artificial limbs exist since the ancient Egyptian empire [2]. In its initial development phase, they acted as dummy body parts and provided only basic functionality. However, the advent of microprocessors and significant research and development in the area of artificial intelligence have paved the way for modern rehabilitation devices. The modern prosthesis is lightweight and provides more agility through machine learning-based control driven by body signals.

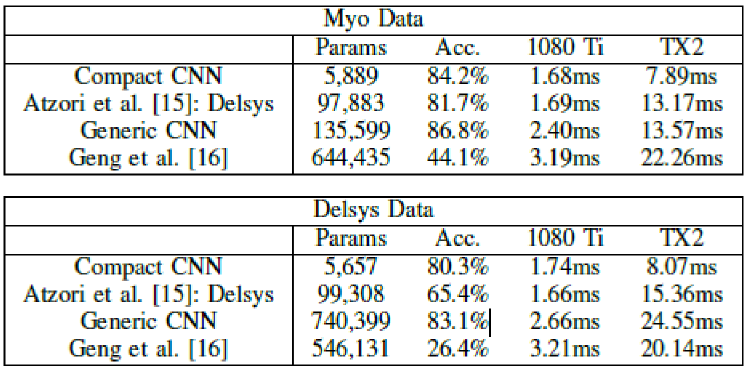
For biosignal based gesture recognition it is important to consider which type of sensor to use, for example, Electromyography (EMG), Force myography (FMG) or Electroencephalography (EEG) sensor. Which machine learning method to use and which set of actions to identify should also be noted. Additionally, when developing the control algorithm, it is important to think upon the feature sets to use e.g. zero crossing, mean absolute value, etc. Finally, the implementation path of this model on small and energy-efficient hardware for portability is of importance.

EMG-based gesture recognition algorithms can be summarized as classifier-based algorithms [3]– [4], artificial neural network (ANN)-based algorithms [5], [6], fuzzy logic-based algorithms [7], and probabilistic model-based algorithms [8].

Primarily, for these learning algorithms to work correctly, stable and good quality signals must be generated. This depends on the proper design of the sensor including the material used for the sensor electrodes, inter-electrode distances and finally preparation of the skin, appropriate sensor placement and its orientation on the muscle of interest. However, there is a wide discrepancy in the procedure followed among different groups of users. Surface Electromyography for the Non-Invasive Assessment of Muscles (SENIAM) is a European group with know-how about sEMG sensors and sensor placement properties as well as practical guidelines for the proper use of sEMG [9]. From the study of about 144 publications, SENIAM has defined a standardized procedure for conducting EMG based research on a more comparable level.

In recent years, many developments have occurred in the design, control and implementation strategies of hand prosthetics. For instance, S.M. Mane et al. proposed to use a single channel surface EMG (sEMG) over multichannel sEMG signals to reduce the complexity of the system. Due to the nonlinear processing capabilities of Artificial Neural Networks, it was used as the learning algorithm along with the wavelet transform. NI ELVIS was the sensor used for data collection and three actions namely open palm, closed palm and wrist extension were classified with an average accuracy of 93.25% [10]. P. Chrapka employed a multiclass classification using "one against one" SVMs to classify two gestures only: open and closed hand, with 80% accuracy [11]. However, no discussion was done on the embedded application of the algorithms. In contrast to this, A. Hartwell et. al [12] analyzed an embedded implementation of a modified SqueezeNet based CNN along with Myo and Delsys electrodes for 15 gestures classification. This gave accuracies about (84.6 ± 6.0) % with Myo and (80.3 ± 7.0) %

with Delsys (Figure 1.1). The spatial reduction strategy used helped to reduce the number of parameters. The trained model (using NVIDIA 1080Ti GPU) was implemented on an NVIDIA Jetson TX2 GPU and benchmarked against SVM (Fig. 1.1) showing ~15% performance gain.



Myo Data				
	Params	Acc.	1080 Ti	TX2
Compact CNN	5,889	84.2%	1.68ms	7.89ms
Atzori et al. [15]: Delsys	97,883	81.7%	1.69ms	13.17ms
Generic CNN	135,599	86.8%	2.40ms	13.57ms
Geng et al. [16]	644,435	44.1%	3.19ms	22.26ms

Delsys Data				
	Params	Acc.	1080 Ti	TX2
Compact CNN	5,657	80.3%	1.74ms	8.07ms
Atzori et al. [15]: Delsys	99,308	65.4%	1.66ms	15.36ms
Generic CNN	740,399	83.1%	2.66ms	24.55ms
Geng et al. [16]	546,131	26.4%	3.21ms	20.14ms

Figure 1.1.: Comparison of accuracy and run-times of deep CNN used in hand movement classification [12]

Another attempt was made by M. Esponda and T. Howard [14] who inspected a vision based adaptive grasp control incorporating sEMG signals with Myo, visual information, and a multi-modal interface with touchscreen and speech inputs to control and interactively teach a prosthesis for 14 gestures recognition. The system was realized on a Raspberry Pi and depicted difficulty in the parsing of voice samples. A novel work is the KIT prosthetic hand with similar capabilities [15]. Such systems [12], [14], [15] seem bulky, expensive, limited and can be uncomfortable for routine use. They are not applicable in every situation, e.g. driving a car would require the person to keep his hand on the steering wheel thus blocking the camera view. As part of the Hand of Hope project and pursuit to develop low-cost robotic prostheses, a feedforward ANN was used by C. Cordova and colleagues for classifying relaxed hand, cylindrical grip, pinch grip, thumb adduction and index finger extension considering ANN's high success rate (~95%) [16] but did not mention about a real-time embedded implementation. N. Rashid et.al [17] developed a two-stage logistic regression classifier using Electroencephalogram (EEG) signals and tested for 3 movements: thumb, finger, and fist. This model was implemented on an Arduino Uno with an SD card to save data but lacked real-time recognition. Also, the small spatial distance between index and middle fingers made the classification difficult and the overall accuracy was 70%. T. Teban et. al [18], concludes that a nonlinear autoregressive network with exogenous inputs recurrent NN outperforms a linear recurrent NN. Anyhow, this study puts the real-time application as future work. J. Hahne et. al used an Atmega32 microcontroller for data acquisition but the regression training was performed on the PC [19].

All in all, many novel machine learning architectures for hand movement recognition have been studied by the scientific community nonetheless, there is a deviation from the embedded implementation of these algorithms. Though a few attempts have been made in [12], [14], [15], their feasibility for daily use is still a question. There is a necessity to

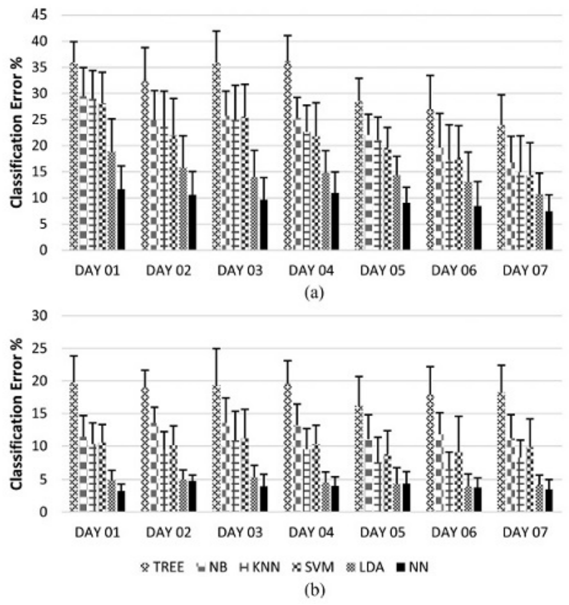


Figure 1.2.: Mean classification error averaged across (a) amputees and (b) able-bodied subjects for all classifiers (Decision Tree, Naive Bayes, K-Nearest Neighbour, Support Vector Machine, Linear Discriminant Analysis, Artificial Neural Network) within a day [13]

concentrate on implementing prosthetic control algorithms on compact, low power and affordable processing platforms. This is where the work of my thesis plays a major role in discussing the implementation workflow for regression and neural networks on high performance and low-cost Cortex M3 microcontroller.

From an algorithmic perspective, Artificial Neural Networks [20]– [21], perform better [22] than popular recognition algorithms like Linear Discriminant Analysis [23], fuzzy classifier, Support Vector Machine, classifiers based on Principal Component Analysis [24] and are the current hot topics in sEMG based prosthesis control. Also, the figure 1.2, shows that neural networks are a better choice for hand pattern recognition in both able-bodied subjects and amputees. Moreover, the scrutiny of [10], [16], [22] and [25], suggests that neural networks are a better alternative due to their capacity to learn complex features and stability over other hand recognition methods. Hence, in this thesis the use of neural networks, deployable on an embedded platform using suitable libraries, will also be demonstrated in parallel to regression.

1.3. Problem Statement

A technology, like a prosthetic hand, which can replace missing hands could help thousands of disabled people improve their lives, enhance their self-confidence, and participate in social activities. Though such prosthetic hands have been in research for decades, they still are limited in their functionality and need to be smarter, faster and energy-efficient. This thesis will investigate two machine learning algorithms namely regression and neural networks for better hand gesture recognition. Generally, machine learning algorithms are resource hungry and are implemented on expensive and energy-consuming hardware. Therefore the main goal of this research would be to thoroughly describe the process to implement the two learning algorithms on a resource-constrained low power embedded system. Furthermore, the required static analysis for the correct choice of the computing platform, usage of computationally efficient libraries, cross-compilation techniques, solving memory constraints and other functional properties will be elaborated. In the end, it is desired to have a working real-time embedded hand gesture recognition system.

1.4. Structure

The structure of this thesis is defined as per the chronological order in which the research was conducted. Chapter 2 talks about the factors affecting the quality of the generated biosignals. Chapter 3 enlightens us on the Electromyography (EMG) and Force myography (FMG) sensors used during this thesis, their advantages, and disadvantages. Chapter 4 introduces the processing setup, that is, the embedded platform, the programming environment, libraries used and their limitations. Chapter 5 discusses the regression and neural network algorithms investigated during the research and the problems of overfitting and underfitting. Chapters 6 and 7 extensively explain the process followed for porting the regression and neural network algorithms on the Cortex M microcontroller respectively. Chapter 8 is reserved for discussing the results and observations. Finally, chapter 9 provides a detailed explanation for the results and rates the examined methods based on its ease of implementation and then we wrap up with the conclusion and future scope.

2. Related Theory

In this chapter we will talk about the two important factors that influence the recognition accuracy of hand gestures: structure of muscles and the limb position effect.

2.1. Muscular Structure

Electromyography is the measure of electrical activity across the muscles of interest and Force myography records the deformations of the forearm muscles. Muscles generate the force required in all types of activities including sports and exercise. They enable us to jump, lift objects, cycle, throw, hit, run and kick. This force is generated by the contraction of skeletal muscles. Interestingly, this contraction is caused by the transmission of tiny electrical impulses, along the motor nerves, from the brain to the muscles.

The forearm muscles that control hand motions can be generally grouped into flexors in the anterior surface, and extensors in the posterior surface surrounding the radius and ulna bones [26]. The forearm enables us to perform four sets of actions: flexion, extension, pronation, and supination. The intrinsic and extrinsic muscles of the hand allow us to perform wrist, hand, and finger movements. The extrinsic muscles of the hand originate from the forearm whereas the palm is the origin of the intrinsic muscles. The anatomy of the forearm can be seen in figure 2.1 and figure 2.2. The anterior muscles of the forearm (Fig.2.2) facilitate wrist and fingers flexion, and pronation. The Flexor Carpi Ulnaris performs wrist flexion and adduction, Palmaris Longus does wrist flexion, Flexor Carpi Radialis helps for wrist flexion and abduction and Pronator Teres are responsible for pronation. The posterior muscles a.k.a extensors (Fig.2.1) contain Brachioradialis for elbow flexion, Extensor Carpi Radialis Longus and Brevis for wrist extension and abduction, Extensor Digitorum for finger movements, Extensor Digiti Minimi for little finger extension, Extensor Carpi Ulnaris for wrist extension and adduction and Anconeus for extension and stabilisation of forearm.

Every individual performs different kinds of activities in daily life and the muscle sizes and strength also varies from person to person. Handgrip strength (HGS) is used as a measure for determining skeletal muscle functions [29]- [30] and disability [31]. Figure 2.3 depicts the strength of muscles across distinct categories of people depending on their age, sex, hand size, arm circumference, etc and thus implies variations in skeletal muscles structures.

As the hand gesture recognition system relies on muscle signals, bracelet position and electrode distribution play a critical role in high accuracy detection. During this thesis, both EMG and FMG sensors are placed at the thickest part of the forearm just below the elbow,

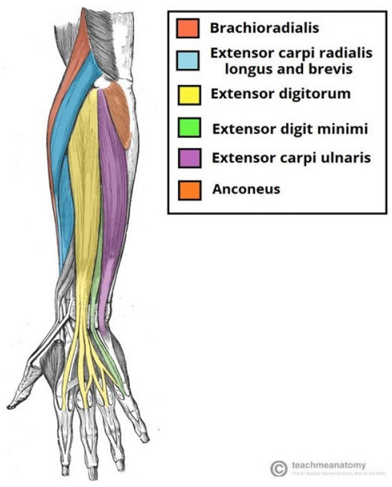


Figure 2.1.: Posterior Muscles of the Forearm [27]

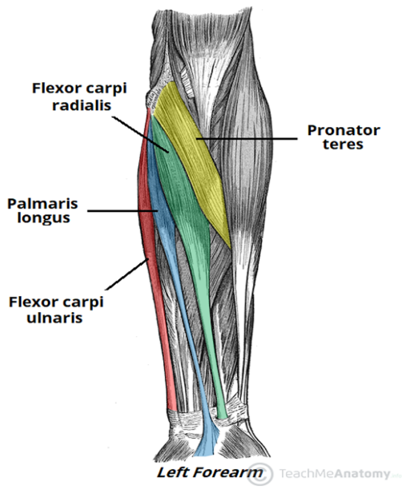
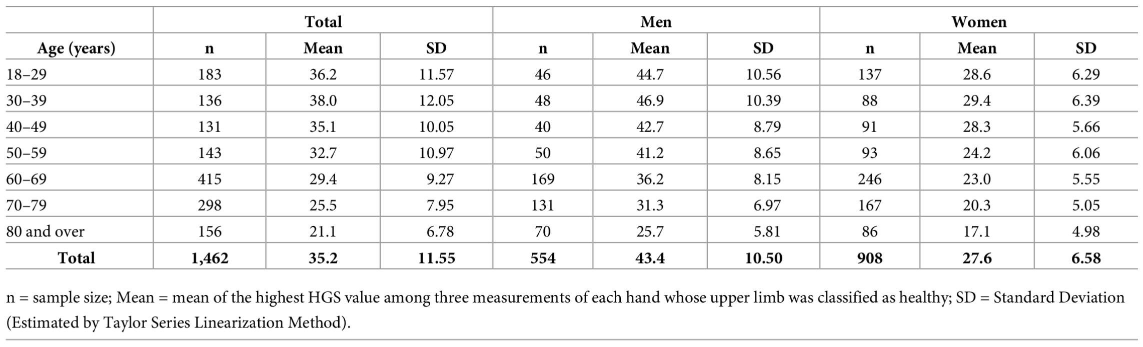


Figure 2.2.: Anterior Muscles of the Forearm [28]



	Total			Men			Women		
Age (years)	n	Mean	SD	n	Mean	SD	n	Mean	SD
18–29	183	36.2	11.57	46	44.7	10.56	137	28.6	6.29
30–39	136	38.0	12.05	48	46.9	10.39	88	29.4	6.39
40–49	131	35.1	10.05	40	42.7	8.79	91	28.3	5.66
50–59	143	32.7	10.97	50	41.2	8.65	93	24.2	6.06
60–69	415	29.4	9.27	169	36.2	8.15	246	23.0	5.55
70–79	298	25.5	7.95	131	31.3	6.97	167	20.3	5.05
80 and over	156	21.1	6.78	70	25.7	5.81	86	17.1	4.98
Total	**1,462**	**35.2**	**11.55**	**554**	**43.4**	**10.50**	**908**	**27.6**	**6.58**

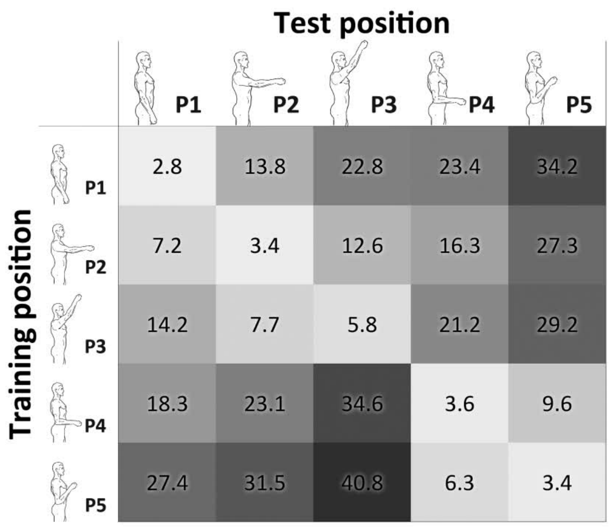
n = sample size; Mean = mean of the highest HGS value among three measurements of each hand whose upper limb was classified as healthy; SD = Standard Deviation (Estimated by Taylor Series Linearization Method).

Figure 2.3.: Mean and standard deviation of maximum Handgrip Strength (HGS) according to sex and age [32]

with the main electrode directly facing up, as this section contains the maximum muscles (suggested by Myo Armband manufacturer).

2.2. Limb Position Effect

Most pilot experiments in this context are performed by keeping a fixed hand position. But in real-life scenarios, this seems impractical. For instance, to drink water one must lift the forearm to reach the face. Thus the hand position changes and affects the muscle length and shape thereby affecting the EMG signals generated. The dependence of the muscular signals on the limb position is known as the limb position effect. EMG classification error is strongly dependent on the limb position [33]. As a result, there exists a gap between the research findings and commercial implementation of the prosthesis control.



Training position \ Test position	P1	P2	P3	P4	P5
P1	2.8	13.8	22.8	23.4	34.2
P2	7.2	3.4	12.6	16.3	27.3
P3	14.2	7.7	5.8	21.2	29.2
P4	18.3	23.1	34.6	3.6	9.6
P5	27.4	31.5	40.8	6.3	3.4

Figure 2.4.: Classification error (in %) for different limb positions [34]

To make the pattern recognition system more robust, one solution can be to get the arm position from accelerometers and use along with EMG data [34]. But more data needs to be collected and additional hardware is required. Another approach is to use an aggregate classifier obtained from training in different limb positions or by integrating some knowledge about the limb position to weigh the individual classifiers appropriately [35]. In this thesis, this would not be under focus but it is worth mentioning this effect for better analysis of the final results.

3. Sensors

This study was conducted with the help of two types of sensors provided by the Adaptive Bionics Group at DLR. The first one is an electromyographic sensor, called the Myo band, from Thalmic Labs (now North Inc.). The second sensor bracelet is based on high-density force myography (FMG) called the Tactile Bracelet from Bielefeld University. The following sections discuss the particulars of each sensor array.

3.1. The Myo Armband

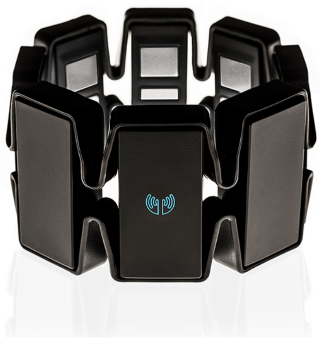
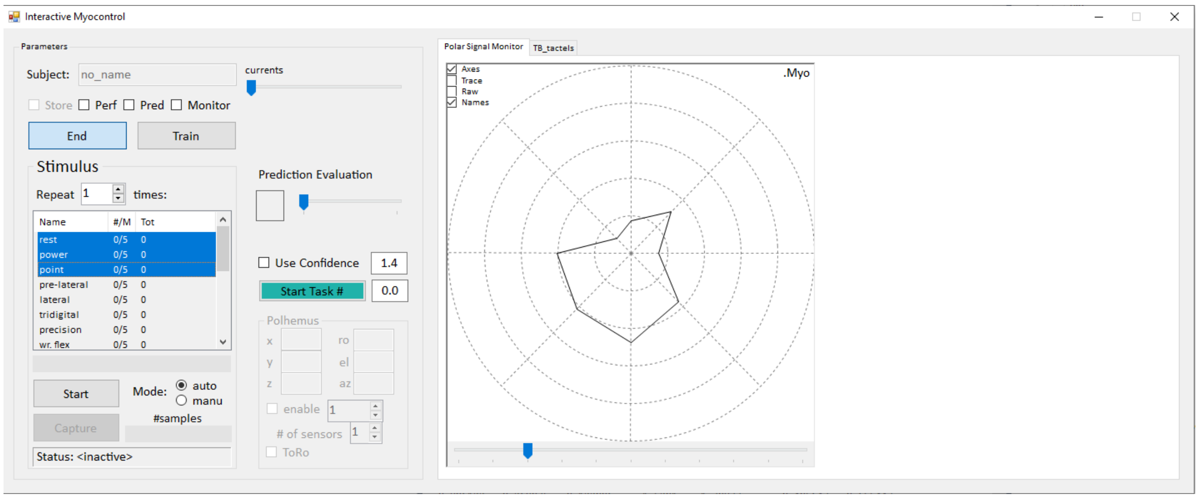


Figure 3.1.: Myo Armband [36]

The non-invasive EMG based Myo bracelet with 8 electrodes is depicted in figure 3.1. It is a wearable device equipped with a 9-axes inertial measurement unit and a transmission module. It is battery powered and uses Bluetooth Low Energy technology for transmission of data. The Myo Connect software is primarily needed to interface the band [37]. Following are the technical details of the Myo:

- Kinetis ARM Cortex M4 120Mhz MK22FN1M MCU
- Inven-sense MPU-9150 at 9 axes IMU
- BLE NRF51822 chip & vibration motor
- 2 x lithium battery 3.7V - 260mAh

As mentioned in Chapter 2, the Myo band will be placed on the bulky muscle just below the elbow with the central electrode facing upwards. For data acquisition, the "Interactive Myocontrol" software, developed by the Adaptive Bionics group was used (Figure 3.2). The sampling rate of the Myo is 200 Hz and for the pilot study, the stimulus capture time is set as 2 seconds and defined in the config file for interactive myocontrol (i.e a total of 400



Subject: no_name

Repeat 1 times:

Name	#/M	Tot
rest	0/5	0
power	0/5	0
point	0/5	0
pre-lateral	0/5	0
lateral	0/5	0
tridigital	0/5	0
precision	0/5	0
wr. flex	0/5	0

Use Confidence 1.4

Start Task # 0.0

Status: <inactive>

Figure 3.2.: Interactive Myo Control software interface

samples/action/repetition are acquired). Care is taken that only the stable action is recorded and not when moving into or out of the action.

The Myo Armband results in a mean classification error of $(9.86 \pm 8.05)\,\%$ compared to $(5.82 \pm 3.63)\,\%$ for conventional EMG system [38]. There are signal variations due to the different muscle contractions, dynamic arm movements and outer interfering forces [39].

Advantages:

1. Low cost compared to others
2. Simple to use and portable
3. No wires
4. One fit for all

Disadvantages:

1. Variations of EMG patterns due to electrode-skin impedance changes, cross talk, sweating, electrode donning-doffing, short circuits or loss of electrode contact to the skin [40]
2. Low sampling rate
3. Lower resolution due to the lower number of electrodes
4. Unstable signals
5. Recharging required at regular intervals
6. Finger movements difficult to identify [41]

In a nutshell, the Myo band is an easy-to-use sensor and widely used for sEMG analysis but at the same time we need to consider its limitations in terms of instability of signals, class separability [42] and interference from various noise sources.

3.2. Tactile Bracelet

Most of the scientific community is researching using surface EMG for hand gesture recognition systems. However, a new technology based on force sensing is making its way through the limitations of sEMG. This methodology is called force myography. Furthermore, high density force myography [43] is called tactile myography and the associated sensor is called the Tactile bracelet (TB).

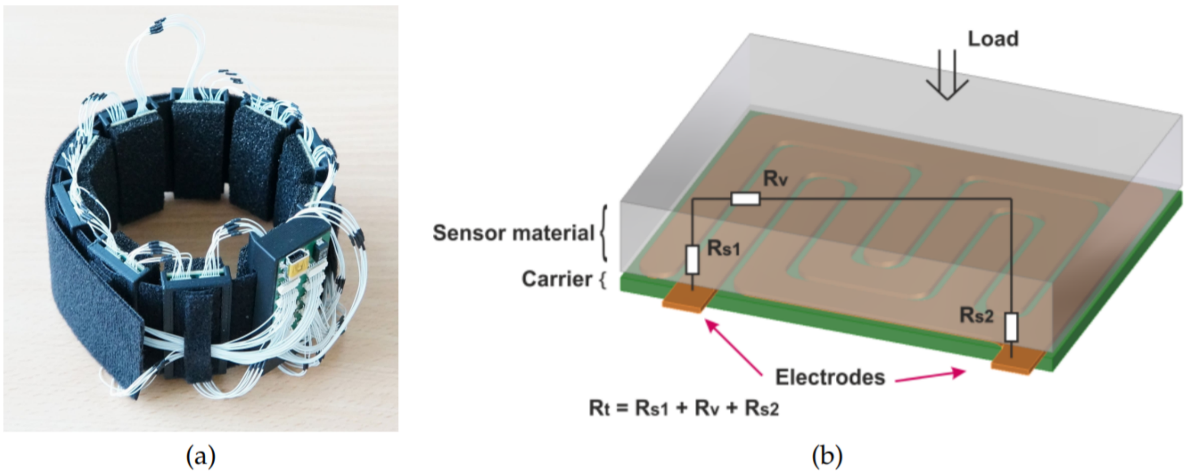
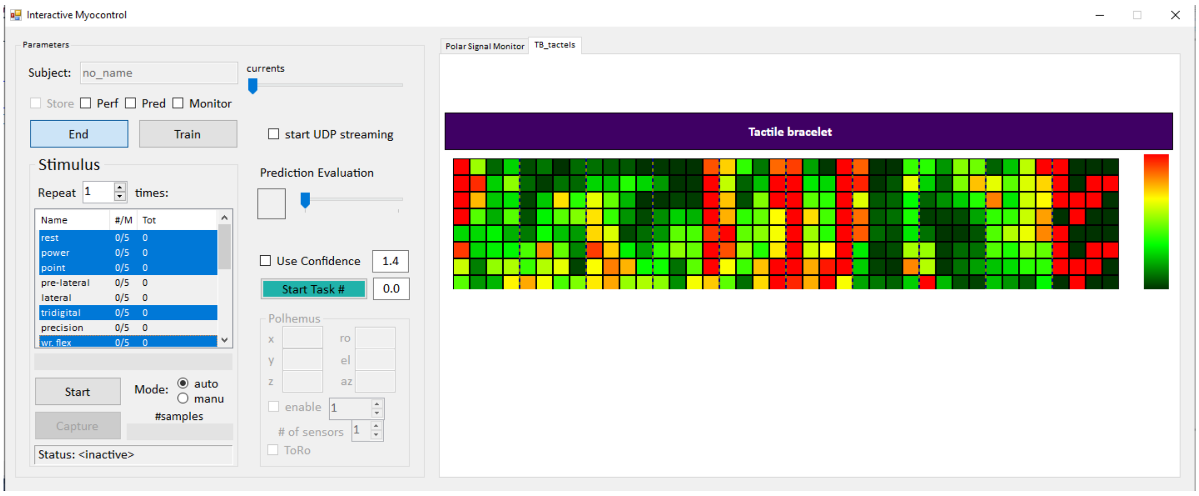


Figure 3.3.: (a) The tactile bracelet prototype with 10 tactile sensor modules and soft conductive foam (b) Working principle of the bracelet: The resistance changes according to the load applied to the foam [43]

The central concept is that flexing a finger causes bulging of the corresponding muscle, which in turn produces increased pressure on the surface of the skin above the location of the muscle [44]. This non-invasive shape conformable sensory device is capable of capturing pressure data, with the help of tactile sensors, from around the full circumference of the arm. The interface resistivity between two electrodes changes according to the applied load and a conductive elastomer foam is used as the sensor material [45]. The foam is very sensitive to low forces due to its hyperbolic characteristics [45]. The tactile sensor cell resistance, R_t, is the aggregate of the variable surface interface resistance, consisting of $R_{s1} + R_{s2}$, and a constant sensor material volume resistance R_v. A voltage divider is used to convert the variable sensor cell resistance into a voltage change and then digitized using a 12bit ADC (range 0 to 4095). There are 10 modules (only 9 used in this study) on the bracelet with each module having 32 electrodes (taxels) in a 4x8 cell arrangement i.e. total of 320 taxels [46]. This is illustrated in figure 3.3. Here, each sensor output represents the intensity of a pixel (Fig 3.4) and hence this pressure map can be used in image processing algorithms, like Convolutional Neural Networks, for further exploitation.



Subject: no_name

Repeat 1 times:

Name	#/M	Tot
rest	0/5	0
power	0/5	0
point	0/5	0
pre-lateral	0/5	0
lateral	0/5	0
tridigital	0/5	0
precision	0/5	0
wr. flex	0/5	0

Use Confidence 1.4

Start Task # 0.0

enable 1

# of sensors 1

Status: <inactive>

Figure 3.4.: Interactive Myo Control software interface showing 288 Taxels from 9 modules

Another point to note is that FMG, unlike EMG, is not affected by the conductivity of the skin or by fatigue and has the potential to produce more stable signals. Having said that, there are concerns that the force sensors stay in place and contact with the skin. Also, force sensing enables the detection of contact forces, accelerations, and orientations, as well as the deformations in the subject's body, which are due to skeletal structures other than muscles (e.g., tendons). These factors can influence the capturing of muscle activations.

Even though FMGs have higher stability in time and the signals are better separated in the input space [47], the classification accuracies tend to decrease with the inclusion of finger flexion along with wrist actions [43].

Advantages:

1. Shape conformable
2. High density of sensors (320 taxels)
3. High sensitivity
4. Stable signals compared to Myo

Disadvantages:

1. Wires are prone to breaking and frequent disconnections
2. Low sampling rate (100Hz)

3. Foam is too sensitive at the edges
4. Modules need to be removed to fit a thin person (low effective resolution)
5. Settling time required for the foam (around 15 minutes)
6. USB wire can become uncomfortable while testing
7. Help is needed to fit the bracelet firmly unlike the Myo

To sum up, a question arises whether tactile myography is a valid replacement or a companion to sEMG. Therefore, in this thesis, the focus would be on exploiting this pressure map generated by the tactile bracelet for better classification and implementation on an embedded platform.

4. Processing setup

The computing device is the heart of a real-time gesture recognition system. Therefore, the embedded hardware used for the employment of regression and neural network algorithms will be deliberated within the next sections.

4.1. The Microcontroller

The primary hardware platform used in this thesis is the Keil MCB 1800 evaluation board. It is populated with the NXP LPC1857 chip which belongs to the family of ARM Cortex M3 processors. A top view of the board can be seen in figure 4.1.

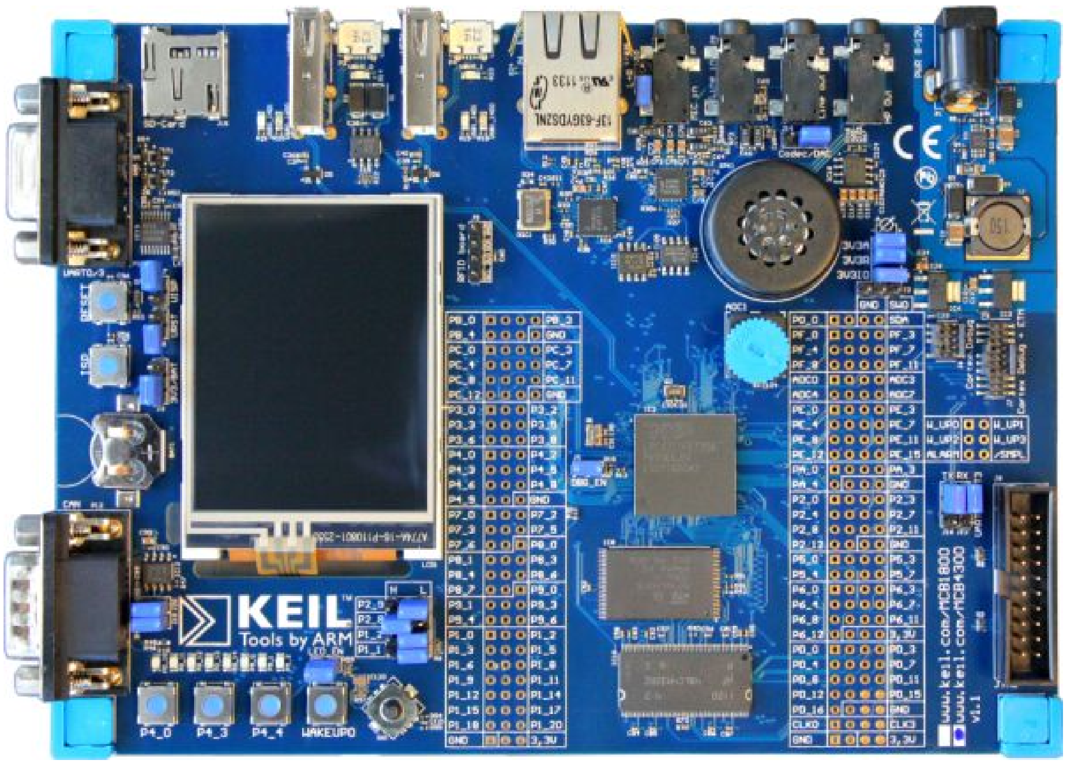


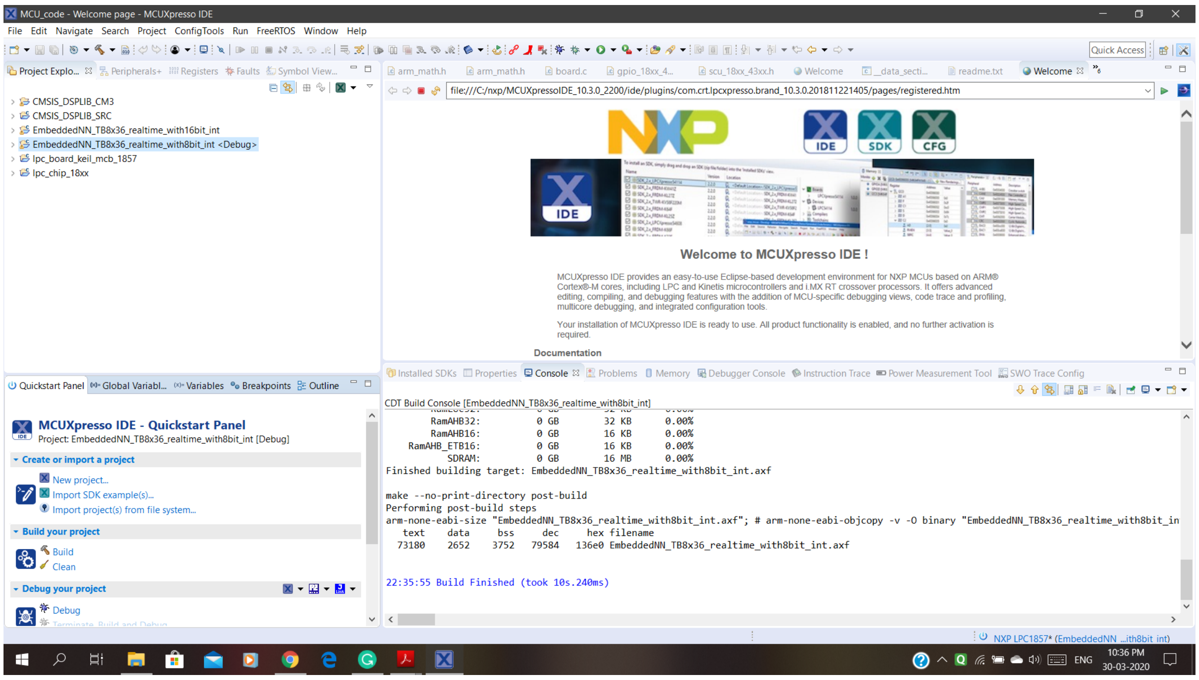
Figure 4.1.: Keil MCB 1800 board [48]

The specifications of the board [48] are as follows:

- 180MHz ARM Cortex-M3 processor-based MCU in LBGA256

- 136KB On Chip SRAM
- 1MB dual bank On Chip Flash memory
- On-Board Memory: 16MB NOR Flash, 4MB Quad-SPI Flash, 16 MB SDRAM & 16KB EEPROM (I2C)
- Color QVGA TFT LCD with touchscreen
- 10/100 Ethernet Port
- High-speed USB 2.0 Host/Device/OTG interface (USB host + Micro USB Device/OTG connectors)
- Full-speed USB 2.0 Host/Device interface (USB host + micro USB Device connectors)
- CAN interfaces
- Serial/UART Port
- MicroSD Card Interface
- 4 user push-buttons + reset
- Digital Temperature Sensor (I2C)
- Analog Voltage Control for ACD Input
- Audio CODEC with Line-In/Out and Microphone/headphone connector + Speaker
- Debug Interface Connectors:
  - 20-pin JTAG (0.1 inch)
  - 10-pin Cortex debug (0.05 inch)
  - 20-pin Cortex debug + ETM Trace (0.05 inch)

This board supports the Keil μVision IDE as well as MCUXpresso IDE by NXP. The programming environment chosen here is the MCUXpresso IDE v10.3.0 to overcome the code size limitations of μVision IDE. A screenshot of the IDE can be seen in figure 4.2. Example projects are available from both Keil and NXP. A collection of free software libraries i.e drivers, middleware and example programs is provided by LPCOpen (NXP) for getting started with the LPC microcontrollers. All programs are developed and tested on a Windows 10 host machine and LPCOpen v3.02 was used for this thesis.



Welcome to MCUXpresso IDE !

MCUXpresso IDE provides an easy-to-use Eclipse-based development environment for NXP MCUs based on ARM® Cortex®-M cores, including LPC and Kinetis microcontrollers and i.MX RT crossover processors. It offers advanced editing, compiling, and debugging features with the addition of MCU-specific debugging views, code trace and profiling, multicore debugging, and integrated configuration tools.

Your installation of MCUXpresso IDE is ready to use. All product functionality is enabled, and no further activation is required.

CDT Build Console [EmbeddedNN_TB8x36_realtime_with8bit_int]

```
        RamAHB32:        0 GB        32 KB      0.00%
        RamAHB16:        0 GB        16 KB      0.00%
    RamAHB_ETB16:        0 GB        16 KB      0.00%
           SDRAM:        0 GB        16 MB      0.00%
Finished building target: EmbeddedNN_TB8x36_realtime_with8bit_int.axf

make --no-print-directory post-build
Performing post-build steps
arm-none-eabi-size "EmbeddedNN_TB8x36_realtime_with8bit_int.axf"; # arm-none-eabi-objcopy -v -O binary "EmbeddedNN_TB8x36_realtime_with8bit_in
   text    data     bss     dec     hex filename
  73180    2652    3752   79584   136e0 EmbeddedNN_TB8x36_realtime_with8bit_int.axf


22:35:55 Build Finished (took 10s.240ms)
```

Figure 4.2.: A screenshot of MCUXpresso IDE

Cortex Microcontroller Software Interface Standard (CMSIS)

This board also supports the Cortex Microcontroller Software Interface Standard which provides a vendor-independent software framework for embedded applications that run on Cortex-M based microcontrollers and Cortex-A based processors. Interfaces to the processor and peripherals, real-time operating systems, and middleware components are provided by CMSIS. Within the scope of this thesis, the CMSIS-DSP that allows the use of digital signal processing functions and CMSIS-NN that provides efficient neural network functions are employed. Following are the categories included in the CMSIS-DSP [49] library:

- Basic math functions
- Fast math functions
- Complex math functions
- Filters
- Matrix functions
- Transform functions
- Motor control functions
- Statistical functions

- Support functions
- Interpolation functions

This library has separate functions for operating on 8-bit integers, 16-bit integers, 32-bit integer, and 32-bit floating-point values.

A variety of efficient neural network kernels are supported by the CMSIS-NN library to maximize the performance and minimize the memory footprint of neural networks on Cortex-M processor cores [50]. Functions supported by the CMSIS-NN [51] library are:

- Neural Network Convolution Functions
- Neural Network Activation Functions
- Fully-connected Layer Functions
- Neural Network Pooling Functions
- Softmax Functions
- Neural Network Support Functions

This library has separate functions for operating on different weight and activation data types including 8-bit integers (q7_t) and 16-bit integers (q15_t). Figure 4.3 gives an overview of this library.

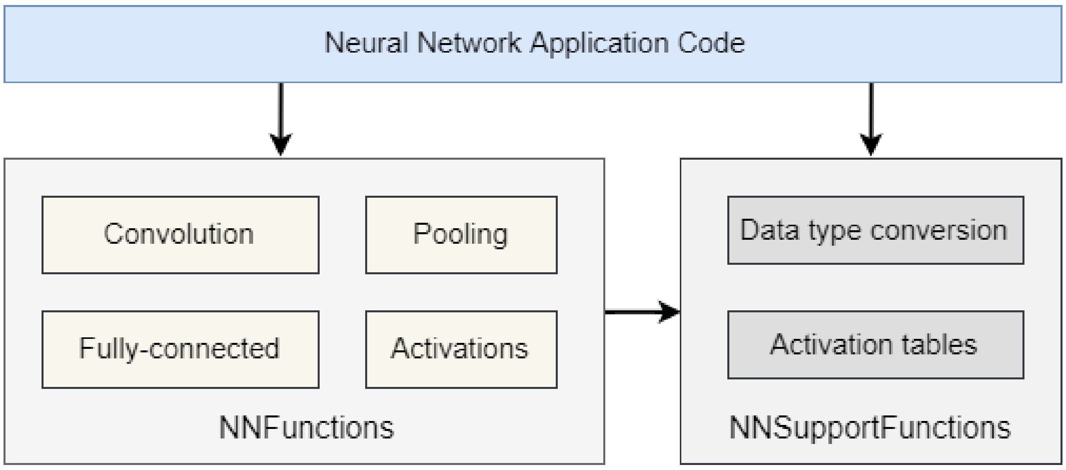


Figure 4.3.: CMSIS NN Overview [50]

It is important to note that currently CMSIS NN supports only the convolutional, pooling, rectified linear activation and fully connected layers and also the Cortex M3 board does not have a floating-point unit (FPU). Therefore, the neural network architecture is chosen likewise in this research study. Further implementation details will be touched in chapter 7.

4.2. Intel Neural Compute Stick 2

Another popular hardware used for AI Inferencing is the Intel Movidius Stick 2. It is a simple Plug and Play USB device based on Intel Movidius Myriad X Vision Processing Unit (VPU). The VPU comprises of 16 powerful SHAVE processing cores and a dedicated hardware accelerator for high performance deep neural network inferences with low power requirements. An out of the box view of the Neural Compute Stick 2 (NCS2) is depicted in figure 4.4.

Figure 4.4.: Intel Neural Compute Stick 2 [52]

The basic architecture of the Vision Processing Unit can be visualised in figure 4.5.

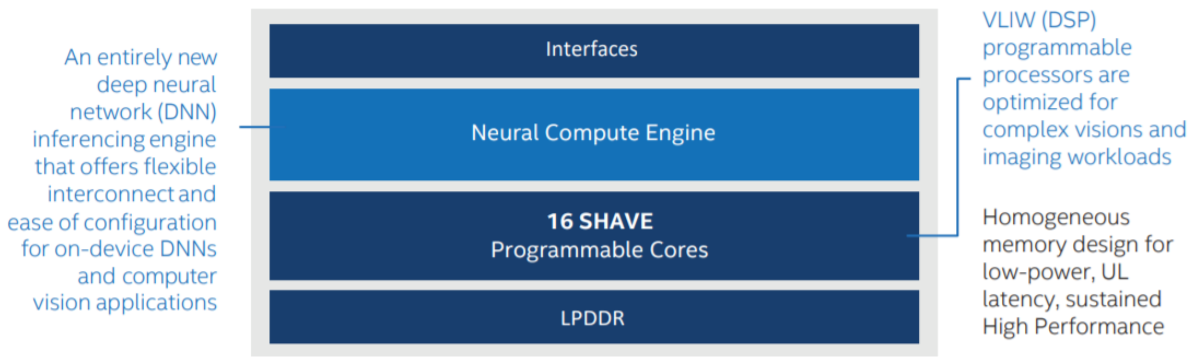

An entirely new deep neural network (DNN) inferencing engine that offers flexible interconnect and ease of configuration for on-device DNNs and computer vision applications
Interfaces
Neural Compute Engine
16 SHAVE Programmable Cores
LPDDR
VLIW (DSP) programmable processors are optimized for complex visions and imaging workloads
Homogeneous memory design for low-power, UL latency, sustained High Performance

Figure 4.5.: Vision Processing Unit Architecture [53]

The Intel Distribution of OpenVINO toolkit is the default software development kit to optimize performance, integrate deep learning inference, and run deep neural networks (DNN) on the NCS2. It enables easy usage of Computer Vision functions and preoptimized kernels. Moreover, it supports popular frameworks like TensorFlow, Caffe, Apache MXNet, Open Neural Network Exchange (ONNX), PyTorch, and PaddlePaddle via an ONNX conversion. It also provides access to numerous pretrained models (like SqueezeNet, MobileNet, ResNet, etc) and inference examples on Github.

With this toolkit pretrained deep learning models can be deployed through a high-level C++ or Python inference engine API onto the Movidius stick. The workflow for deploying a trained deep learning model on the NCS2 is shown in figure 4.6.

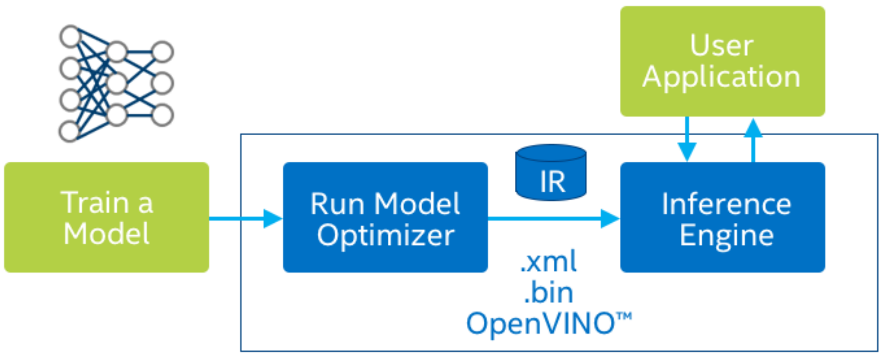


Figure 4.6.: Flowchart for inferencing with NCS2 [54]

The major components of this toolkit [54] are:

1. **Model Optimizer (MO):** This Python-based command line tool imports trained models from popular deep learning frameworks mentioned previously. It performs analysis and adjustments of the trained models for optimal execution on endpoint target devices and thereafter serializes and adjusts the model into an intermediate representation (IR) format. Floating point 16 (FP16) quantization format is also supported in addition to low precision INT8. The IR consists of a pair of files describing the model:
   - .xml - Describes the network topology
   - .bin - Contains the weights and biases binary data.
2. **Inference Engine (IE):** This engine works on the IR files and uses a common API to deliver inference solutions on platforms like CPU, GPU, VPU, or FPGA. Additionally, it allows different layers to run on different targets i.e implement custom layers on a CPU while running the remaining topology on a GPU without the need to rewrite the custom layers. The IE also performs computational graph analysis, scheduling, and model compression for target hardware with an embedded-friendly scoring solution thus allowing asynchronous execution to improve frame-rate performance while limiting wasted cycles.

5. Algorithms

The basic definition of machine learning, problems of overfitting and underfitting will be mentioned in this chapter. In addition to this, the details of the types of regression and classification algorithms examined within this thesis will be unfolded in the following sections.

5.1. Machine Learning

Machine Learning (ML) was defined by Arthur Samuel, in 1959, as a *"Field of study that gives computers the ability to learn without being explicitly programmed"* [55]. In 1997, Tom Mitchell defined it as: *"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*" [56].

Generally, machine learning is classified broadly into two categories:

1. Supervised learning
   - Ground truth is available
   - Examples are: Regression and Classification algorithms
2. Unsupervised learning
   - Ground truth is unavailable
   - Examples are: Clustering algorithms

A good machine learning model is one that can generalize correctly on new unseen input data from the problem domain. Thus the model is capable of making rightful predictions on future inputs. To judge whether the model is well generalized or not, machine learning curves play a significant role. These curves help us to infer if the algorithm is overfitting, underfitting or is a good fit.

1. **Underfitting:** A learning model is underfitting when it is unable to learn the trend in the variability of data [57]. In one sense this means that the model is over-generalizing the trend in the sample data. Consequently, the final classifier will not be able to predict correctly on unseen inputs. Underfitting indicates that the model is too simple to fit the data or the dataset is small. This case is also called "High Bias". Figure 5.1 shows a typical case of underfitting. Underfitting can be solved by adding more data or by using a higher degree polynomial to fit the data i.e increase the model complexity. Figure 5.2 narrates that when training and test error are both high, it is High Bias.

2. **Overfitting:** This implies that the model is memorizing the behavior of the sample dataset and generating a very specific model [58]. Also, the model is too complex for the given dataset and is learning too many details including the noise in the data. Figure 5.1 shows a typical case of overfitting a.k.a "High Variance". A high variance occurs when the training error is decreasing but test error is increasing (see Figure 5.2). The solution here is to reduce the number of features thus reducing the complexity of the model or by using regularization. Regularization introduces a penalty to the cost function so that less relevant features make less impact on the learning. λ is called as the regularization parameter. The two major types of regularization are:
   - **L1 regularization (Lasso regularization):** An absolute value of magnitude of coefficient is added as penalty to the loss function by the Least Absolute Shrinkage and Selection Operator (LASSO). The loss equation with L1 regularization [59] is:

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

   - **L2 regularization (Ridge regularization):** This adds a squared magnitude of coefficient as penalty to the loss function. The loss equation with L2 regularization [59] is:

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} {w_i}^2$$

   where:

   y: is the true label
   $\hat{y}$: is the predicted label
   $|w_i|$: is the norm of the i^{th} weight in an N dimensional weight vector w

The value of λ must be chosen wisely as a high value can cause underfitting. Regularization also addresses the problems of non-invertibility of matrices when the number of examples in the dataset is less than or equal to the number of features.

A good fit model is one that has low training and test losses. Thus it will be able to perform well on unseen data. The optimal capacity in figure 5.2 is the point of a good fit.

5.2. Regression

Regression is a type of supervised learning algorithm that predicts a continuous-valued output based on the given input. For instance, the prediction of the price of a house based on its size/area is a regression problem. Regression analysis gives us a mapping between the dependent variable and the independent variable. Linear regression, logistic regression, and polynomial regression are common types of regression algorithms.

Within the Adaptive Bionics Group, two selected pattern recognition algorithms are used for hand prosthesis control. The first one is a ridge regression algorithm used together with

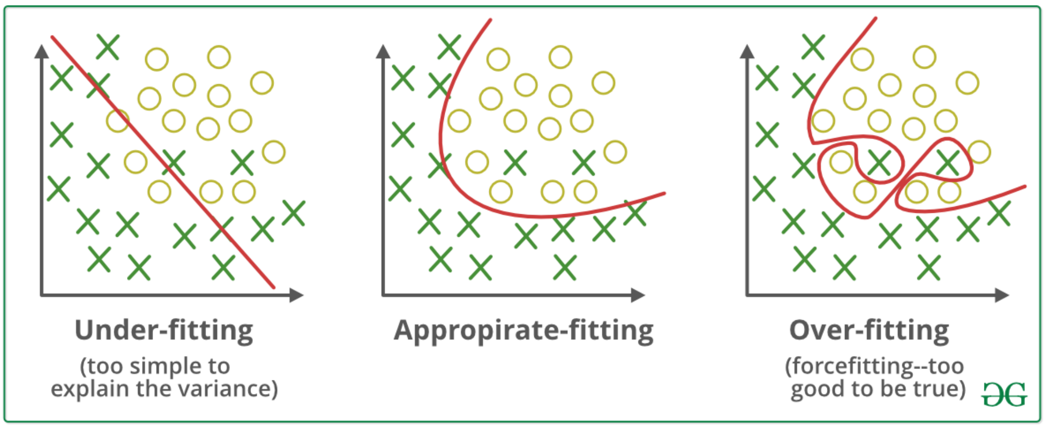

Under-fitting
(too simple to explain the variance)
Appropirate-fitting
Over-fitting
(forcefitting--too good to be true)

Figure 5.1.: Underfitting, Good fit and Overfitting in machine learning [60]

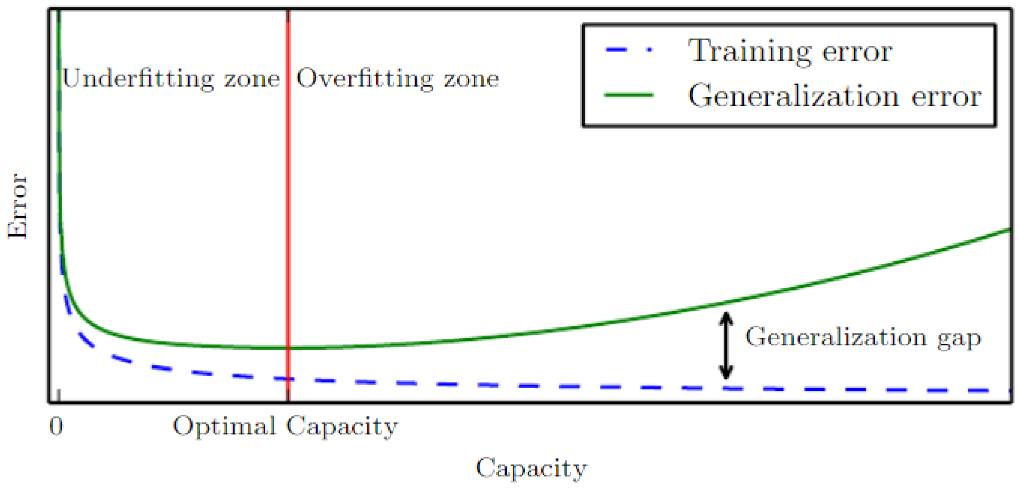


Figure 5.2.: Error vs Model capacity(or complexity). The red point marks a good fit where training and test error are both small [61]

.

input data from the Tactile bracelet and the other one is an extension of this approach with random Fourier features used along with EMG input from the Myo. This is illustrated in the following subsections.

5.2.1. Linear Regression

The starting point for the ridge regression is a multivariable linear regression model. This multivariate model studies the effect of multiple variables on the dependent variable , say y_i. For a sample set $\{y_i, x_{i1}, ..., x_{in}\}$, this effect can then be formulated as a linear combination of the independent variables, $\underline{x} = \{x_{i1}, ..., x_{in}\}$, as follows [62], [63]:

$$y_i = w_1 x_{i1} + ... + w_n x_{in} = \underline{x_i}^T \underline{w} \tag{5.1}$$

A compact representation of the output is given as:

$$\underline{y} = X\underline{w} \tag{5.2}$$

with:

$\underline{y}$: a m-dimensional vector consisting the output values for the bionic hand.
m: the number of degrees of freedom of the hand prosthesis
$\underline{w}$: a n-dimensional weight vector
n: the number of sensors electrodes. For this thesis, n=288 for TB and n=8 for Myoband
X: training matrix of size (m $\times$ n)

$$X = \begin{bmatrix} \underline{x_1}^T \\ \underline{x_2}^T \\ ... \\ \underline{x_m}^T \end{bmatrix}$$

Note: For the prosthesis at DLR, m=9, due to its 9 degrees of freedom.

5.2.2. Ridge Regression

The concept of L2 regularisation is already introduced in previous sections. In general, the ridge regression (RR) is a regularised linear regression with regularisation parameter λ.

Usually, the X matrix has more number of samples (m) than variables (n). To solve this over-determined system for the weight vector $\underline{w}$, the classical approach used is the "least squares" method to minimize the error in the predicted value [64], [65]. The least squares solution would then be:

$$\underline{w} = (X^T X)^{-1} X^T \underline{y} \tag{5.3}$$

However, a common problem faced in multiple linear regression is that the inverse of $X^T X$ may not exist. Therefore to find a particular solution with desirable properties, a penalty term Γ, called the Tikhonov matrix, is included in equation 5.3. In most cases, $\Gamma = \lambda I$ is chosen as it gives preference to solutions with smaller norms. This transforms the optimization problem as [66]:

$$\underline{w} = \arg\min_{\underline{w}} \sum_i (y_i - \underline{x_i}^T \underline{w_i})^2 + \lambda \sum_j w_j{}^2 \tag{5.4}$$

This final solution can be rewritten in terms of the training matrix (X) and labels ($\underline{y}$) as:

$$\underline{w} = (X^T X + \lambda I)^{-1} X^T \underline{y} \tag{5.5}$$

One of the motivations of using the Ridge Regression is that it allows updating the model incrementally without the need to store the training samples [66].

5.2.3. Ridge Regression with Random Fourier Features

The Ridge Regression is limited due to its linearity. To overcome this limitation, a "kernel trick" [66] is applied which can turn any linear model into a non-linear model by replacing its features by a kernel function and can operate in a potentially infinite-dimensional space. However, this advantage comes at the price of additional computational and storage requirements which scale poorly with the increasing size of the dataset [67], [68]. This makes kernel ridge regression unsuitable for real-time operation.

A solution to this issue is to approximate the mapping of the kernel function to a finite-dimensional feature [66]. As a result, the training process is accelerated while still guaranteeing an increased classifier capacity due to the utilization of higher dimensions. Consequently, the memory and timing constraints are relaxed. Ali Rahimi and Ben Recht [69] proposed to sample a finite number of random Fourier features from shift-invariant kernel functions (e.g. the Gaussian kernel) [67]. The mapping function for Random Fourier features (RRRFF) is:

$$\phi(\underline{x}) = \sqrt{2} cos(\underline{\Omega}^T \underline{x} + b) \tag{5.6}$$

where:

$\underline{\Omega}$ is drawn from a Gaussian distribution
b is drawn from a uniform distribution from 0 to 2π

Then, the D-dimensional weight vector $\underline{\widehat{w}}$ can then be estimated using the equation:

$$\underline{\widehat{w}} = (\Phi^T \Phi + \lambda I)^{-1} \Phi^T \underline{y} = A^{-1}\beta \tag{5.7}$$

with:

A: a $(D \times D)$ matrix $\Phi^T\Phi + \lambda I$
β: a D-dimensional vector $\beta = \Phi^T \underline{y}$
n: the number of EMG sensors
Φ: $\Phi = \phi(X)$, a $(m \times D)$ matrix
X: a (m $\times$ n) training matrix with sensor input values
$\underline{y}$: a m-dimensional vector consisting the output values for the bionic hand

m: the number of degrees of freedom of the hand prosthesis

Overall, the accuracy of the resulting algorithm is directly proportional to the number of Fourier features D. However, choosing larger D demands higher computational capacity.

5.3. Neural Networks

The human brain is a masterpiece of complex computing systems. It can store and process tons of information daily. But how is it able to achieve this complex task? The human brain is made up of billions of neurons, arranged together in a network-like structure, that transmit and process the information that we perceive through our senses. Tiny electrical impulses are passed from one neuron to the other.

The same concept forms the basis of the widely used supervised learning method of neural networks (NN) wherein the neurons are artificially created and complex architecture of interconnected neurons is developed. The working of this network is equivalent to neurons in our brains. In figure 5.3, $x_1, x_2, ..., x_n$ are the inputs, $w_1, w_2, ..., w_n$ are the weights, β is the bias, $f[.]$ is the activation function and y_k is the output. The relation between input and output is:

$$y_k = f(\sum_{i=1}^{N} x_i * w_i + \beta) \tag{5.8}$$

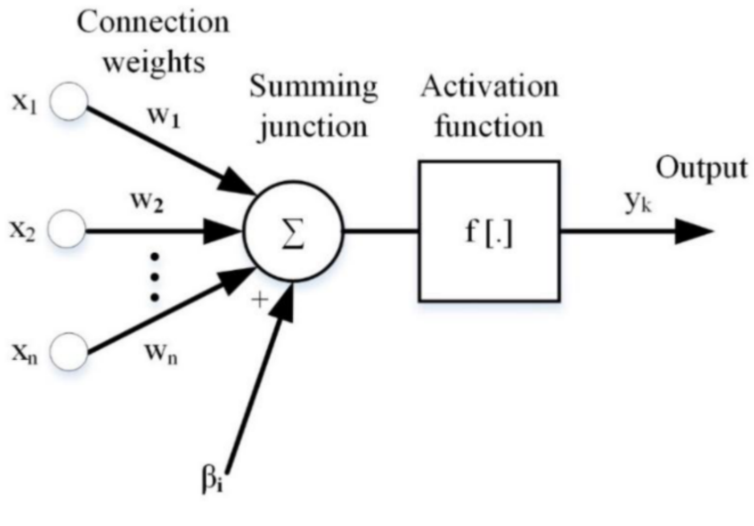


Figure 5.3.: An artificial neuron [70]

Neural Networks give discrete output and are used for classification tasks where the input belongs to either of n probable classes. The classification of an email as spam or not spam is a binary classification problem. In multi-class classification, the input instance can belong to any of n possible outputs e.g. classify a fruit like an apple, banana, orange, etc. An artificial neural network learns to perform tasks without being explicitly programmed by extracting complex hidden features. Pattern recognition and prediction are two prominent applications of neural nets [70].

Neural networks have distributive associative memory meaning that the knowledge is distributed across all neurons in the network. Thus, when the trained network is provided with an incomplete input the network will choose the closest match to that input and generate the output accordingly. Neural computing systems are also tolerant of noisy input. The network performance suffers from increasing noise in the input signal but the system does not fail completely. In pattern recognition, large amounts of information are processed simultaneously to generate a binary or categorical output. It is therefore required that the network provides a reasonable response to noisy or incomplete inputs. Neural networks have proven to be very good pattern recognizers with the ability to build unique structures for a specific problem [71]. Furthermore, the studies [10], [16], [22] and [25], suggests that neural networks are a better choice than prevailing learning algorithms like Linear Discriminant Analysis, fuzzy classifiers, Support Vector Machines due to their capacity to model complex non-linearities and hence are well suited for hand gesture recognition systems. The two types of neural network architectures used in this study are multilayer perceptrons to process vectorized inputs from the Myo as well as TB and convolutional neural networks to benefit from the pressure image produced by the TB. Other popular types of neural networks like Recurrent Neural networks were not chosen due to the limited layers and network architectures supported by CMSIS NN library as intimated in chapter 3.

5.3.1. Multilayer Perceptrons

A neural network consisting of one input layer, one hidden layer, and an output layer is known as a multilayer perceptron (MLP) or a feed-forward network (see figure 5.4). This means, there are no loops in the network, unlike recurrent neural networks, and information is always fed in the forward direction. Structures with more than one hidden layer are called deep networks. In supervised training, both the inputs and the ground truths or labels are provided. In the first step, the network processes the inputs and compares the predicted outputs to the desired outputs and in the second step, the computed errors are then propagated back through the system (a.k.a backpropagation), causing the system to adjust the weights which control the network. These steps occur iteratively and the weights are tweaked till the output error is minimized. The network is trained on a training set and tested on a validation set to keep track of its performance. Popular optimizers such as Gradient Descent, Adam, RMSProp, etc are used as optimization functions to adjust the weights according to the error and minimize the cost function or loss. An activation function introduces non-linear properties into our network so that the mappings between the inputs and outputs of a node make sense. The extensively used activation functions are rectified linear unit (relu), sigmoid (for binary classification), softmax (for multi-class classification), etc.

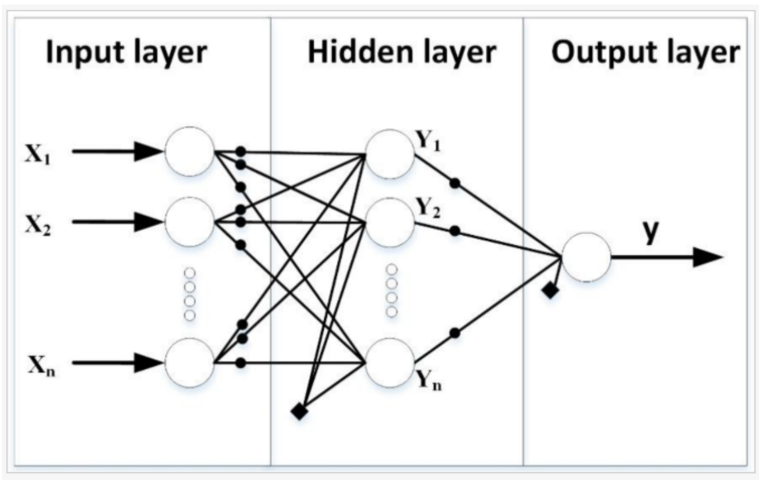


Figure 5.4.: Multilayer Perceptron [70]

Back propagation algorithm

The backpropagation algorithm is used to update the weights of the network to minimize the cost function or loss function. Mean Squared Error (MSE), binary crossentropy or categorical crossentropy are generally used as the cost function. MSE is used for regression problems wheres the other two for classification.

The algorithm is described as [72]:

1. Initialize the weights randomly.
2. Choose an input vector $\underline{x}^u$
3. Feed the network with the input and calculate output as per equation 5.8
4. Compute ${\delta_i}^L$ in the output layer L

$$ {\delta_i}^L = f^{'}({h_i}^L)[{d_i}^L - {y_i}^L] $$

where ${\delta_i}^L$ is the error of node i in layer L, ${h_i}^L$ represents the net input to the i^{th} neuron in the L^{th} layer, d is the expected output value, y is the predicted value and $f^{'}$ is the derivative of activation function f

5. Compute the deltas for preceding layers by propagating the error backwards

$$ {\delta_i}^l = f^{'}({h_i}^l)\sum_j {w_{ij}}^{l+1} - {\delta_j}^{l+1} $$

for $l = (L-1), ..., 1$

6. Update weights using
$$\Delta w_{ji}{}^{l} = \eta \delta_i{}^{l} y_j{}^{l-1}$$
where η is the learning rate between 0 and 1

7. Repeat from step 2 for next input until the error is minimum or the max number of iterations is reached

5.3.2. Convolutional Neural Networks

Convolutional Neural Networks (CNN) deal with inputs that are images which permits us to integrate certain aspects into the architecture. This facilitates a more efficient implementation and reduction in the number of parameters of the network [73]. CNNs have gained excellent performance in many computer vision and machine learning problems due to their ability to capture spatial and temporal dependencies in images. Applications of CNN include image classification, semantic segmentation, object detection, etc.

CNNs are made up of a sequence of layers, mainly, convolutional layers, pooling layers, rectified linear units (ReLU) and fully connected layers. Each layer extracts high level abstract features known as feature maps which preserves the most important plus unique information. A representation of a CNN is shown in figure 5.5.

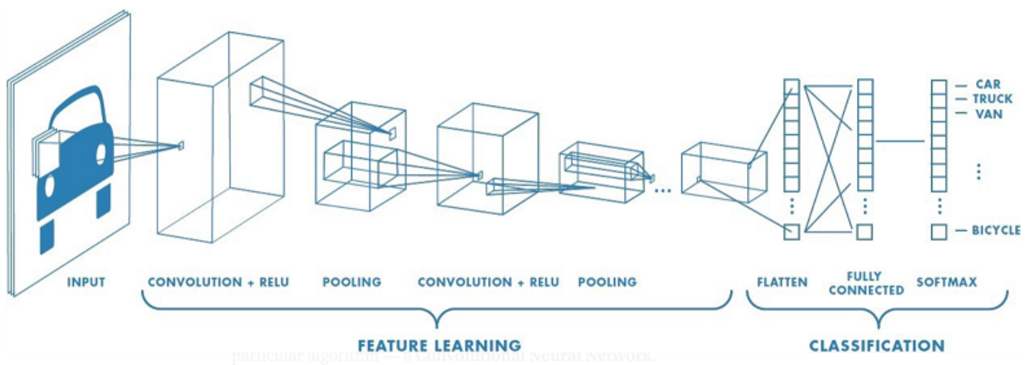


Figure 5.5.: A CNN architecture [74]

Convolutional Layer

This is the core layer in the network and is the most computationally expensive. It consists of learnable filters or kernels. The input image is fed as a 3 dimensional tensor in the form of height × width × channel (e.g. 32 × 32 × 3) matrix. The filter then slides and convolves over the input image. The filter values are multiplied with pixel values of the image and all multiplications are then summed up to a scalar value. This process repeats for every location on the input size and finally provides a feature map. It provides an opportunity to detect and recognize features, like edges, regardless of their positions in the image [73]. A CNN can have multiple such convolutional layers and each layer has its own filter and therefore extracts different features from the input.

Pooling Layer

The pooling layer performs down-sampling of the input. It reduces the spatial size of the convolved feature map while retaining significant information. This dimensionality reduction decreases the computational power required. There are two types of pooling: Max Pooling and Average Pooling. Max Pooling returns the maximum value from the portion overlapped by the filter whereas Average Pooling returns the average of all the values. Max pooling helps in better generalization by eliminating noisy activations and hence performs better than average pooling [75].

ReLU Activation

ReLU is a non-linear activation function and includes a rectifier. It is applied element-wise per pixel and all negative values in the feature map are replaced by zero. The function is: $f(x) = max(0, x)$, where x is the neuron input. This layer is applied to saturate the output or limit the generated output. ReLU is popular due to its simplicity and helps to speed up the processing since its gradient is a constant for a positive input [73]- [76].

Fully connected layer

In this layer, every neuron is connected to each node in the previous layer as observed in feedforward neural networks. It produces non-linear combinations of high-level features from previous layers. One major disadvantage of a fully-connected layer is its high computational effort due to a large number of parameters [73].

Softmax layer

The fully connected layer is followed by a Softmax activation function that converts the output into probabilities and its sum is 1 [75]. The class with the highest probability is chosen as the predicted output. Softmax layer is mostly used for multiclass classification problems.

5.3.3. Limitations of Artificial Neural Networks

The downsides of ANN are:

- The output may be unpredictable
- Black box Nature: Difficult to understand how ANN solve problems [77]
- Greater computational burden and lack of standard software
- Prone to over fitting [78]

6. Regression Workflow

The previous chapters have laid the foundation of the processing setup and algorithms examined during this thesis. In this section, we would discuss thoroughly the various steps followed from the analysis of the Ridge Regression (RR) and Ridge Regression with Random Fourier Features (RRRFF) methods, choice of hardware to the final implementation on the Cortex M board.

6.0.1. Complexity analysis of regression algorithms

The goal of this study is to go towards an embedded solution for the control of a prosthetic hand. The setup that already included a complete C# based software called the Interactive Myocontrol (Fig. 3.2), which also incorporates the two regression-based learning methods and multiple sensor drivers was provided by the Adaptive Bionics group at DLR. This was a convenient reference point for commencing the work. However, the myocontrol software runs on a computer. To be able to translate and fit the program on the microcontroller, it was essential to do a static analysis of the computational complexity of the training functions in both methods.

Visual Studio 2015 running on a Windows 7 host machine was the primary IDE used for the analysis. It offers a powerful tool, called Profiler, for measuring application performance by analyzing the CPU usage. The following steps are followed for the diagnosis of the training function:

1. Open the Intercative Myocontrol solution in Visual Studio and set a breakpoint at the beginnning of the training function and another at the end of the function.

2. Run the program and check the timings in the Profiler sidebar.

The myocontrol software contains training functions for both RR and RRRFF which can be selected by modifying the config file. A set of 17 actions with 5 repetitions of each is available for both methods. The available actions are: rest, power, point, pre-lateral, lateral, tridigital, precision, wrist flex, wrist extension, wrist pronation, wrist supination, rotate thumb, flex thumb, flex index, flex middle, flex ring and flex little. The initial timing tests were conducted with the Myoband only for different combinations of the number of actions, repetitions, ridge regression and ridge regression with random Fourier features.

The training function for ridge regression is:

$$W = (X.Transpose() \times X + lambda \times I).Inverse() \times (X.Transpose() \times Y)$$

and for ridge regression with random fourier features is:

$$W = (phi(X).Transpose() \times phi(X) + lambda \times I_D).Inverse() \times (phi(X).Transpose() \times Y)$$

where:
- X: An $m \times n$ input matrix
- Y: label matrix
- W: weight matrix
- $phi(X)$: mapping function as described in equation 5.6
- $IandI_D$: identity matrices
- $lambda$: regularization parameter (= 1)

The processor performance depends on: clock cycle, clock cycles per instruction, and instruction count [79]. Hence to calculate the number of instructions executed during training the following formula is applied:

$$T_{app} = N_{instr} \times CPI \times T_{clk} \tag{6.1}$$

with:
- T_{app}: Time of the train function
- N_{instr}: Number of instructions in the program
- CPI: Clock cycles per instruction
- T_{clk}: One clock period

The time complexities for both Ridge Regression (RR) as well as Ridge Regression Random Fourier Features (RRRFF) came out to be as follows:

- RRRFF : $O(dDm) + O(D^2m)$
- RR: $O(d^2m)$

where:
- d: Number of channels (8 for Myo and 288 for TB)
- D: Number of random Fourier features
- m: Number of samples (at 200 Hz) & for 2 seconds. Hence, total 400 samples

Observations from complexity graphs

Based on the time values observed and the time complexities calculated, correlation graphs are plotted using MATLAB. The graphs contain both normalized as well as non-normalized figures for both RR and RRRFF and all combinations.

- *Ridge Regression:*
  1. From the graphs of Ridge Regression, it can be observed that the complexity calculations show a linear behavior because $'d'$ (number of channels) is constant and $'m'$ (number of samples) increases linearly with the number of actions e.g. for 1 action, 8 channels, 5 repetitions the time complexity is: $O(d^2m) = 8 \times 8 \times 400 \times 5 \times 1$

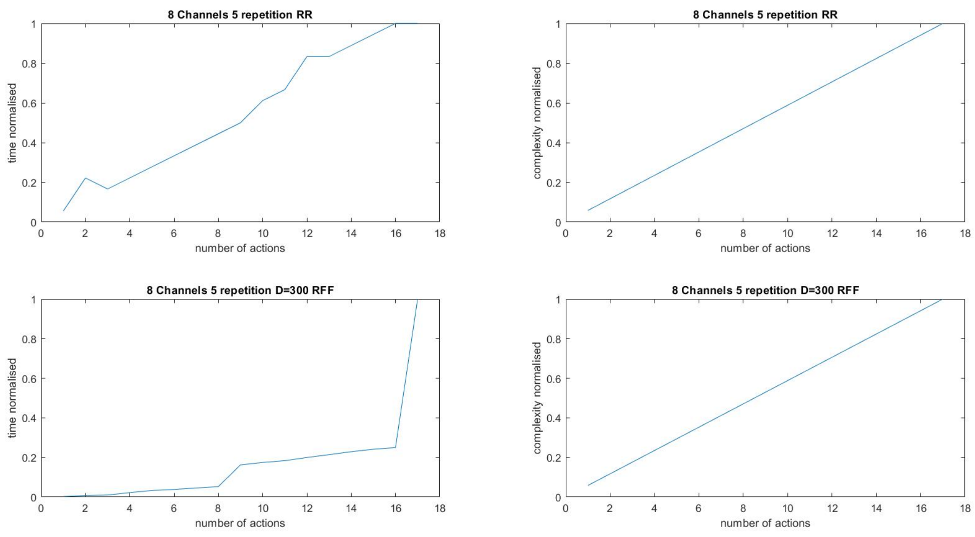


Figure 6.1.: Complexity comparison of RR and RRRFF

2. From the timings obtained from the Profiler the same linear behavior is seen (see figure 6.1). Hence the calculations are verified.

- *Ridge Regression Random Fourier Features:*
  1. The graphs of RRRFF show that the complexity calculations have a linear behavior because $'d'$ (number of channels), as well as $'D = 300'$ (number of Fourier features), is constant and $'m'$ (number of samples) increases linearly with the number of actions e.g. for 1 action, 8 channels, 5 repetitions we have: RFF: $O(dDm) + O(D^2m) = 8 \times 300 \times 400 \times 5 \times 1 + 300 \times 300 \times 400 \times 5 \times 1$
  2. But from the values obtained from Profiler, we see a sudden rise at the end of each graph (see figure 6.1).
     a) To analyze this, the value of RRRFF for 8 channels, 5 repetitions, 17 actions for the different number of Fourier features, that is, D=100 and D=500 were calculated and the same jump as in case of D=300 was seen (see figure 6.2).
     b) To analyze further, the value of RRRFF for 8 channels, 5 repetitions, 17 actions this time with D=8 and without the $phi(X)$ function, that is, the mapping function from the code (see figure 6.2) was computed. Thus, now it simply multiplies and calculates the inverse of the matrix. So now it is expected to have the same time values as obtained for a similar configuration in case of RR only. Interestingly, the same timings were obtained in both cases and thus the conclusion is that the mapping function takes more time as the number of actions goes on increasing in the case of RRRFF, which justifies the sudden jump in the graph.

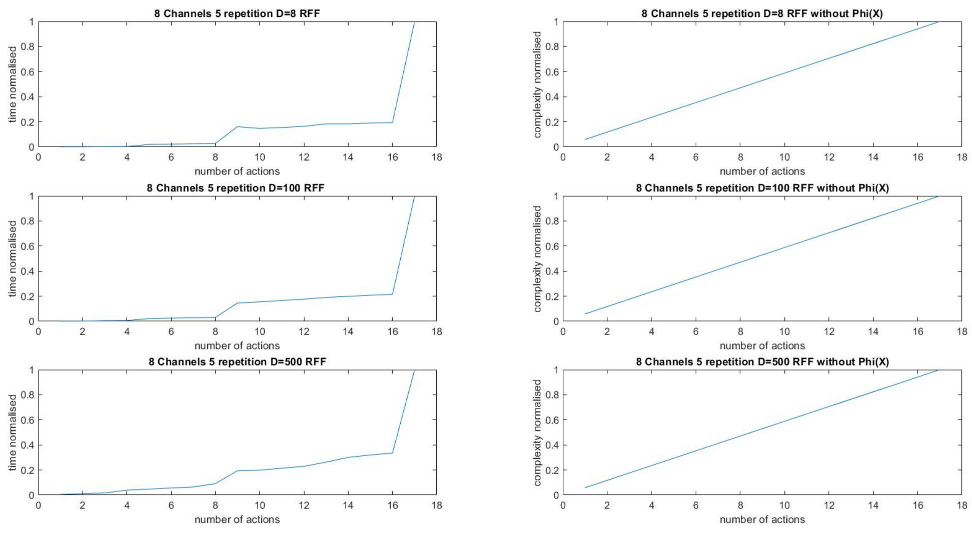


Figure 6.2.: RRRFF complexity comparison with and without mapping function $phi(X)$

6.0.2. Choice of microcontroller

In this section, the Million Instructions Per Second (MIPS) of the host computer will be calculated. The laptop used encompasses an Intel i7 7700HQ processor with 4 cores and running at a max frequency of 3.80GHz/core.

As per the Export Compliance Metrics provided by Intel [80] and the formula for GFLOPS as follows:

$$GFLOPS = CPU_{clk}(\, in\, GHz) \times \#cores \times IPC \times \#CPU_{sockets}$$

where:

IPC: Instructions per clock cycle
CPI: Clock cycles per instruction

Substituting the values,

$$179.2 = 3.8 \times 4 \times IPC \times 1$$

$$IPC = CPI^{-1} = 11.789 \implies CPI = 0.0848$$

$$CPI(per\ processor) = 0.0848 \times 4 = 0.36$$

Therefore, effective $MIPS = 11.789 \times 3.8GHz = 44798.2\, MIPS\, @3.8\, GHz$

Calculations of N_{instr} for RR and RRRFF

Using the above value for MIPS, the number of instructions for the training operation in various configurations were calculated:

1. RR with Myo Band:
    a) 8 channels, 3 repetitions, 7 actions
    $4ms \times 44798.2MIPS = 179.1928\, M\, instructions \quad (M = million)$
    b) 8 channels, 3 repetitions, 17 actions
    $10ms \times 44798.2MIPS = 447.98\, M\, instructions$
    c) 8 channels, 5 repetitions, 7 actions
    $7ms \times 44798.2MIPS = 313.5874\, M\, instructions$
    d) 8 channels, 5 repetitions, 17 actions
    $18ms \times 44798.2MIPS = 806.367\, M\, instructions$
2. RRRFF with Myo Band:
    a) 8 channels, 3 repetitions, D=300, 7 actions
    $10175ms \times 44798.2MIPS = 4,55,821.685\, M\, instructions$
    b) 8 channels, 3 repetitions, D=300, 17 actions
    $63462ms \times 44798.2MIPS = 28,42,983.368\, M\, instructions$
    c) 8 channels, 5 repetitions, D=300, 7 actions
    $16996ms \times 44798.2MIPS = 7,61,390.2072\, M\, instructions$
    d) 8 channels, 5 repetitions, D=300, 17 actions
    $371762ms \times 44798.2MIPS = 1,66,54,268\, M\, instructions$
    e) 8 channels, 3 repetitions, D=100, 7 actions
    $8182ms \times 44798.2MIPS = 3,66,538.8724\, M\, instructions$
    f) 8 channels, 3 repetitions, D=100, 17 actions
    $291079ms \times 44798.2MIPS = 1,30,39,815.26\, M\, instructions$
3. TB (from dummy signal) with RR:
    a) 320 channels, 3 repetitions, 7 actions
    $2388ms \times 44798.2MIPS = 1,06,978.10\, M\, instructions$
    b) 320 channels, 3 repetitions, 17 actions
    $6502ms \times 44798.2MIPS = 2,91,277.8964\, M\, instructions$
    c) 320 channels, 5 repetitions, 7 actions
    $4181ms \times 44798.2MIPS = 1,87,301.2742\, M\, instructions$
    d) 320 channels, 5 repetitions, 17 actions
    $10375ms \times 44798.2MIPS = 4,64,781.325\, M\, instructions$

Based on these N_{instr} we can determine the approximate training time required for popular embedded boards.

1. With ST μC @800 MIPS:
    - RR RFF with Myo Band

a) 8 channels, 3 repetitions, D=300, 7 actions
$4, 55, 821.685\, M\, instructions / 800\, MIPS = 9.49\, minutes$

b) 8 channels, 3 repetitions, D=300, 17 actions
$28, 42, 983.368\, M\, instructions / 800\, MIPS = 59\, minutes$

c) 8 channels, 5 repetitions, D=300, 7 actions
$7, 61, 390.2072\, M\, instructions / 800\, MIPS = 15.86\, minutes$

d) 8 channels, 5 repetitions, D=300, 17 actions
$1, 66, 54, 268\, M\, instructions / 800\, MIPS = 5.78\, hours$

e) 8 channels, 3 repetitions, D=100, 7 actions
$3, 66, 538.8724\, M\, instructions / 800\, MIPS = 7\, minutes$

f) 8 channels, 3 repetitions, D=100, 17 actions
$1, 30, 39, 815.26\, M\, instructions / 800\, MIPS = 4.5\, hours$

- TB with RR:

a) 320 channels, 3 repetitions, 7 actions
$1, 06, 978.10\, M\, instructions / 800\, MIPS = 2.228\, minutes$

b) 320 channels, 3 repetitions, 17 actions
$2, 91, 277.8964\, M\, instructions / 800\, MIPS = 6\, minutes$

c) 320 channels, 5 repetitions, 7 actions
$1, 87, 301.2742\, M\, instructions / 800\, MIPS = 3.9\, minutes$

d) 320 channels, 5 repetitions, 17 actions
$4, 64, 781.325\, M\, instructions / 800\, MIPS = 9.68\, minutes$

2. With ESP32 @600 MIPS:

- RR RFF with Myo Band

a) 8 channels, 3 repetitions, D=300, 7 actions
$4, 55, 821.685\, M\, instructions / 600\, MIPS = 12.66\, minutes$

b) 8 channels, 3 repetitions, D=300, 17 actions
$28, 42, 983.368\, M\, instructions / 600\, MIPS = 78\, minutes$

c) 8 channels, 5 repetitions, D=300, 7 actions
$7, 61, 390.2072\, M\, instructions / 600\, MIPS = 21.14\, minutes$

d) 8 channels, 5 repetitions, D=300, 17 actions
$1, 66, 54, 268\, M\, instructions / 600\, MIPS = 7.71\, hours$

e) 8 channels, 3 repetitions, D=100, 7 actions
$3, 66, 538.8724\, M\, instructions / 600\, MIPS = 10.18\, minutes$

f) 8 channels, 3 repetitions, D=100, 17 actions
$1, 30, 39, 815.26\, M\, instructions / 600\, MIPS = 6.03\, hours$

- TB with RR:

a) 320 channels, 3 repetitions, 7 actions
$1,06,978.10\,M\,instructions/600\,MIPS = 2.97\,minutes$

b) 320 channels, 3 repetitions, 17 actions
$2,91,277.8964\,M\,instructions/600\,MIPS = 8.09\,minutes$

c) 320 channels, 5 repetitions, 7 actions
$1,87,301.2742\,M\,instructions/600\,MIPS = 5.20\,minutes$

d) 320 channels, 5 repetitions, 17 actions
$4,64,781.325\,M\,instructions/600\,MIPS = 12.9\,minutes$

3. With RPi3 @2441 MIPS:

- RR RFF with Myo Band

a) 8 channels, 3 repetitions, D=300, 7 actions
$4,55,821.685\,M\,instructions/2441\,MIPS = 3.11\,minutes$

b) 8 channels, 3 repetitions, D=300, 17 actions
$28,42,983.368\,M\,instructions/2441\,MIPS = 19.41\,minutes$

c) 8 channels, 5 repetitions, D=300, 7 actions
$7,61,390.2072\,M\,instructions/2441\,MIPS = 5.19\,minutes$

d) 8 channels, 5 repetitions, D=300, 17 actions
$1,66,54,268\,M\,instructions/2441\,MIPS = 1.89\,hours$

e) 8 channels, 3 repetitions, D=100, 7 actions
$3,66,538.8724\,M\,instructions/2441\,MIPS = 2.50\,minutes$

f) 8 channels, 3 repetitions, D=100, 17 actions
$1,30,39,815.26\,M\,instructions/2441\,MIPS = 1.48\,hours$

- TB with RR:

a) 320 channels, 3 repetitions, 7 actions
$1,06,978.10\,M\,instructions/2441\,MIPS = 0.73\,minutes$

b) 320 channels, 3 repetitions, 17 actions
$2,91,277.8964\,M\,instructions/2441\,MIPS = 1.988\,minutes$

c) 320 channels, 5 repetitions, 7 actions
$1,87,301.2742\,M\,instructions/2441\,MIPS = 1.27\,minutes$

d) 320 channels, 5 repetitions, 17 actions
$4,64,781.325\,M\,instructions/2441\,MIPS = 3.17\,minutes$

Note that all these values are related to the training function written in C#. Due to the computational overhead of C#, it was decided to shift to the fast and efficient low-level C language. Also, most embedded platforms work with C rather than C#.

6.0.3. Translating C# program into C

The C programming language is adopted for system development because it produces code that runs almost as fast as the code written in assembly language. C outperforms other popular languages (e.g C#, python, etc) in terms of both speed and memory usage but at the cost of more code length [81]. Most microcontrollers on the market are programmed in C due to its efficient low-level implementation capability. Hence, the training function from Interactive Myocontrol was rewritten in C using the freely available GNU Scientific Library (GSL) [82]. It is a numerical library with Basic Linear Algebra Subprograms (BLAS) support as well as a variety of other functions. The X matrix (20352×8) or sensor values from Myo and Y matrix (20352×9) or label matrix obtained for RRRFF with 17 actions and 3 repetitions of each and D=100 at first is stored in a text file on the PC and then read using GSL file operations. The results for training using GSL v2.5 are displayed in table 6.1.

Programming language	Time	RAM usage
C#	5 minutes	650 MB
C	25 seconds	54 MB

Table 6.1.: Execution time required for 17 actions, 8 channels and 3 repetitions with RRRFF (D=100) and Myo on desktop (Intel Xeon)

The reason for this speedup is because C# has a lot of overhead (e.g garbage collection [83], etc.) and libraries included before compilation whereas C is lightweight. Using this fact it was decided to choose a low-performance controller in the thesis and hence the first board bought was LPCXpresso 4367, a Cortex M4/M0 based microcontroller with 154kB SRAM, 1MB dual flash bank, clock frequency of 204 Mhz and dedicated floating-point unit (FPU) (see figure B.1).

6.0.4. Cross compilation of GSL library

The GSL library works on systems with an operating system but the microcontroller uses bare-metal programming i.e without OS. Therefore, to port the code onto the LPC4367 (ARMv7 architecture) controller, it was necessary to cross-compile the library for $arm - none - eabi - gcc$ toolchain. This specifies that the target is ARM architecture, has no vendor, no operating system and complies with ARM embedded application binary interface. The host and target systems are both set as $arm - none - eabi$ and optimization level, which tells the compiler to generate faster codes, is set as O3. Detailed steps for cross-compilation are explained in the Appendix A.1.

Initially, the GSL library was used at O0 optimization level, X and Y matrices were not specified in the RRRFF code and D was 300. This led to an overflow, during compilation, of the flash bank as 100.27 % of 512 kB was used up. Storing the X and Y matrices on an SD card was proposed but for real-time use, this would not be feasible. So the next case was to try with smaller dimensions of RFF with D=20, the number of sensor channels d = 11, X and Y matrices of size 11 ×11 both were specified in the code. The overall optimization was set at Os (optimize for size). This required 71.15% memory usage. During runtime, a Hardfault was detected which was due to memory constraints of the LPC4367 board. GSL uses dynamic memory allocation and hence a new board, Keil MCB1800 (see figure 4.1), with external SDRAM of 16 MB was purchased. Now a final test version of the GSL code for 2 actions, 2 repetitions, and D=300 was implemented on the new board. X and Y matrices of size 1604 × 8 and 1604 × 9 were stored in the FlashB, consuming 41.61% of 512kB while the code used up to 71.12% of Flash A (512 kB). The optimization level of code and library was at O3. 16MB of external SDRAM was reserved for heap and 14.37% of SRAM(40kB) was exhausted. The weight matrix W (300 × 9) was produced with an execution time of 6.5 minutes. The table 6.2 summarizes these points.

Matrix	D	Library	Flash	SRAM	SDRAM	Remarks
X: 20352× 8 Y: 20352×9	300	GSL (O0)	100.27%	10.14%	-	Without X, Y on LPC4367
X: 11×11 Y: 11×11	20	GSL (Os)	71.12%	9.29%	-	X, Y on LPC4367
X: 1604×8 Y: 1604×9	300	GSL (O3)	A: 71.15% B: 41.61%	14.37%	100%	X, Y on MCB1800
X: 11×11 Y: 11×11	20	DSP (O3)	16.55%	8.10%	-	X, Y on LPC4367, gaussian & uniform distribution of GSL
X: 11×11 Y: 11×11	20	DSP (O3)	9.75%	6.51%	-	X, Y on LPC4367, custom gaussian & uniform distribution
X: 1596×8 Y: 1596×9	20	DSP (O3)	25.55%	2.71%	-	X, Y on LPC4367, custom gaussian & uniform distribution
X: 1604×8 Y: 1604×9	100	DSP (O3)	30.76%	6.48%	24.75%	X, Y on MCB1800, custom gaussian & uniform distribution

Table 6.2.: Different RRRFF configurations trials on LPC4367 and Keil MCB1800

6.0.5. Jump from GSL to CMSIS-DSP

Due to excessive memory usage of the GSL library, the ARM specific CMSIS-DSP library (chapter 4) was examined further. It is an optimized library built at optimization level O2 and supports many mathematical functions. The first version of the RRRFF code developed using CMSIS DSP library was with the number of RFF D=20, the number of sensors d=11, and O3 optimization level. The Gaussian distribution and uniform distribution functions, required for the mapping function in RRRFF (see equation 5.6), were not readily available from CMSIS and hence these functions were imported from the GSL library. Here, 16.55% flash memory was used and SRAM usage was 8.10%. However, using custom Gaussian and uniform distribution functions for RRRFF led to improvement with only 9.75% flash usage and 6.51% SRAM usage. This again justifies the memory hunger of GSL. The second version tried was with 2 actions 2 repetitions, D=20, d=8 and O3 optimization where 25.55% flash and 2.71% SRAM was used. Finally, 2 actions 2 repetitions, D=100, d=8 utilized 30.76% of flash, 24.75% SDRAM for storing matrices and 6.48% SRAM (refer table 6.2). Here a bug was found in the CMSIS-DSP library which allows only smaller matrices to be multiplied. To overcome this limitation, custom matrix multiplication functions were used in place of the ones from CMSIS DSP. Two approaches were compared, one with only custom multiplication function and other functions e.g. inverse of a matrix from the DSP library and the second one was using all custom written functions i.e without any DSP functionality. A comparison of execution times can be seen in figure 6.3 and in table 6.3. The SDRAM usage varied between 12% - 86% & Flash usage between 30% - 56%. Note that the MCB1800 does not have an FPU and therefore the performance of CMSIS-DSP libraries is not optimum.

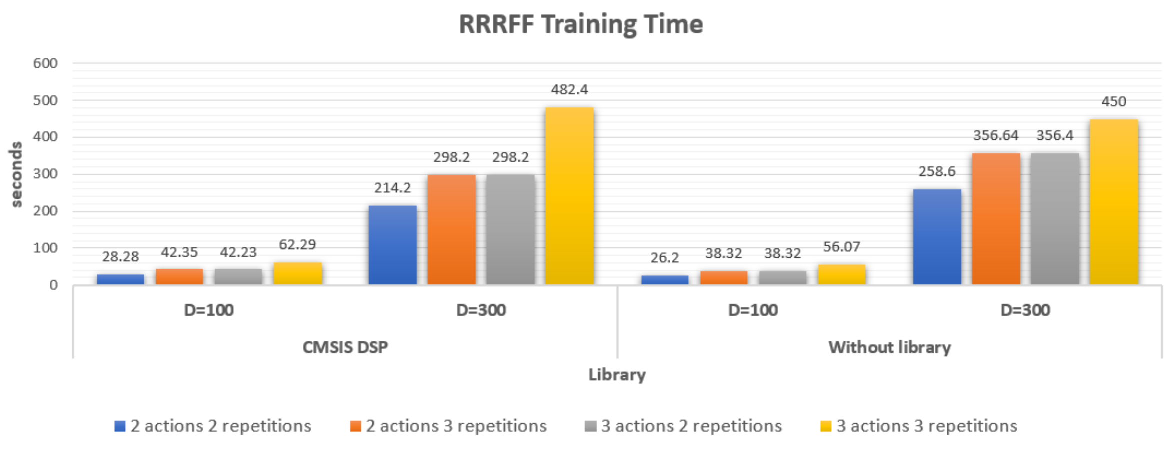


Figure 6.3.: Graphical view of execution times of different RRRFF configurations for rest, power and point on Keil MCB1800

6.0.6. The final implementation

Due to the previously mentioned problem of matrix multiplication in CMSIS-DSP, the final version of the code includes implementation of RR as well as RRRFF without the DSP library. Optimization level is kept at O3 and all values are stored as floats instead of doubles to lower memory requirements. All the matrices required to store intermediate results are globally declared and stored as zero-initialized data using the *__BSS* macro, by NXP, in the external RAM of 16 MB. This code can train on 3 actions, 3 repetitions (rest, power, point) with the number of Fourier features D=300 at maximum for RRRFF before overflowing the memory. The X matrix for Myo (8 channels), preprocessed with a first order Butterworth filter, and TB (288 channels) is transferred from the Myocontrol software with the help of a special

Configuration	X Matrix	Y Matrix	CMSIS DSP		Without library	
			D=100	D=300	D=100	D=300
2 actions 2 repetitions	1606× 8	1606× 9	28.28 sec	3.57 min	26.20 sec	4.31 min
2 actions 3 repetitions	2405× 8	2405× 9	42.35 sec	4.97 min	38.32 sec	5.944 min
3 actions 2 repetitions	2405× 8	2405× 9	42.23 sec	4.97 min	38.32 sec	5.94 min
3 actions 3 repetitions	3603× 8	3603× 9	62.29 sec	8.04 min	56.07 sec	7.5 min

Table 6.3.: Execution times of different RRRFF configurations for rest, power and point on Keil MCB1800

device driver written in C# which uses the USB protocol. Initially, the UART protocol was experimented but to transfer a matrix of 400 $\times$ 8 i.e one repetition took about 1.5 minutes due to lower bandwidth. Hence a switch from UART to UART-over-USB protocol was made which facilitated sending this data volume in 450-500 milliseconds with the help of handshaking. This was possible because USB 2.0 High Speed supports bandwidths up to 480 Mbps. Data from each repetition for each action is received into the microcontroller and stored in a dynamically allocated memory of the corresponding size. Each float value is converted into a four-byte array on the PC, transmitted over USB and converted back into float value in the MCU. Around 9% of the SDRAM is reserved for the heap to store these matrices. The remaining memory is used for the intermediate buffers.

Switch	Operation performed
Joystick UP	Record data for 1 repetition of REST
Joystick DOWN	Record data for 1 repetition of POWER
Joystick LEFT	Record data for 1 repetition of POINT
Joystick PRESS	Train for selected actions
Push Button 1	Start predicting
Push Button 2	Stop predicting

Table 6.4.: MCB 1800 joystick configurations

A menu is shown on the console interface and the joystick along with push buttons are used to select actions, train the algorithm and then predict in real time (see table 6.4). LEDs are used to indicate the status of training. The overall details of final version of RR and RRRFF can be seen in the table 6.5.

The predicted values from the MCU are sent back over USB to the interactive Myocontrol software and further sent to a virtual 3D hand model (Fig.6.4), developed in python and blender, via the UDP protocol.

Matrix size	D	Optimization	Flash	SRAM	SDRAM	Train Time
X: 3600× 8 Y: 3600×9	100	O3	12.17%	13.83%	96.77%	56.07 sec
X: 3600× 8 Y: 3600×9	300	O3	12.17%	13.83%	96.77%	7.5 min
X: 1800× 288 Y: 1800×9	RR only	O3	9.88%	13.80%	45.71%	3.81 min

Table 6.5.: RR and RRRFF final implementation details on Keil MCB1800. First two entries are with Myoband and RRRFF learning while the last entry is with Tactile Bracelet and RR only.

6.0.7. Dataset and MCU Training Procedure

This is a gesture recognition system that can ease the daily life of amputees. The fact that the amputee would be utilizing the prosthetic hand only for himself/herself makes it sensible to record data from the amputee itself rather than collecting a generalized dataset. Hence, in this study, for testing the different regression implementations, the actions were recorded and trained for the respective subject only. The Myo or TB is worn on the right arm by the subject and just on the muscle below the elbow as described in chapter 3.

An overall sketch of the training procedure is as follows:

1. Wear the bracelet firmly on your right arm.
2. Run the RR or RRRFF code on the microcontroller.
3. Start the Interactive myocontrol software on the PC.
4. Use the joystick (refer 6.4) to record data from each repetition for Rest, Power, Point actions.
   a) If max repetitions (= 3), due to MCU memory constraints, is reached, then the MCU will display a "Max repetitions reached" message on the console.
   b) If not, go to step 4.
5. Then press the Joystick to train on these actions.
   a) If data is not recorded previously: "First record the data" message is displayed.
   b) Otherwise select the actions to train on with the help of the joystick again.
6. Once the training is complete, start predicting on the 3D hand model by pressing push button 1 or stop with the help of push button 2.

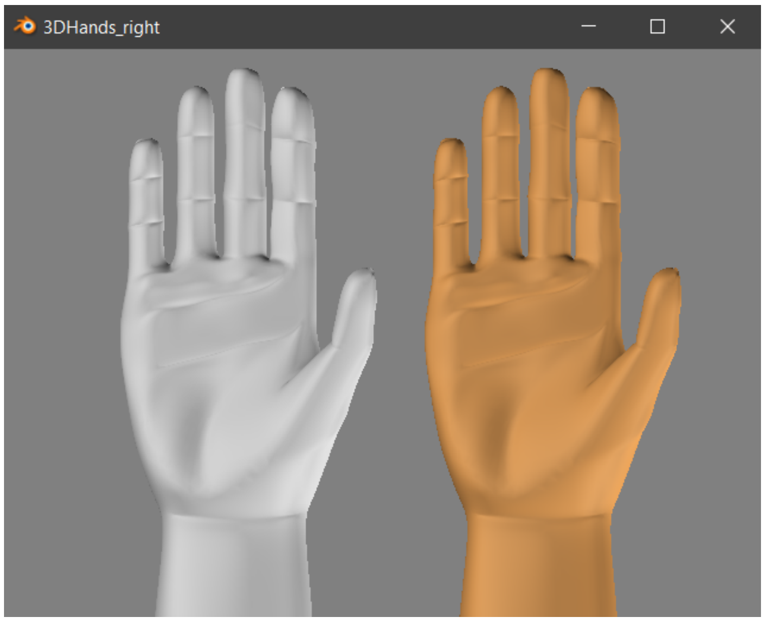


Figure 6.4.: 3D hand model for prediction. The white hand is the stimulus to follow when recording training data whereas orange hand is used for real-time prediction.

7. Neural Network Workflow

Neural networks can model complex problems and are a better choice for pattern recognition applications [70], [71]. From the literature survey, it follows that neural networks show commendable performance for classification tasks and therefore can also be applied for hand gesture identification. This chapter demonstrates the use of ANNs in combination with muscular force sensors, or in other words, the tactile bracelet for more robust classification. The embedded implementation of vivid NN models on the low-performance Cortex M3 board along with suitable and efficient NN libraries is also presented here.

7.1. Multilayer Perceptron

The theory of MLPs is already documented in chapter 5. The dataset, similar to regression, was recorded from an individual subject on which the algorithm has to be trained and tested. In the MLPs, the input data is fed as a vector. Two models were developed, one for the Myo Armband and the other for the Tactile Bracelet. Although the Myo is not in focus for this study, a quick overview of different models tested with it is shown.

All the neural networks are defined in Python using the Keras library, pandas and Scikit-Learn modules. For all Python and Keras models, unless mentioned otherwise, the data (each sample vector) is sent from Interactive Myocontrol software via UDP port to the python program for prediction and then the predicted output is directly transmitted to the 3D hand model (refer figure 6.4) again over UDP. The arm is kept in a stable and fixed position for data acquisition and prediction phase to minimize the limb position effect and improve classification accuracy.

7.1.1. Case 1: MyoBand

The data is recorded with the Myo, placed on the right arm, for 6 gestures, namely, rest, power, point, tridigital, wrist flex and wrist extension with 5 repetitions of each. The dataset is a matrix of $\approx (12000 \times 8)$ dimension. Each sample is a 1×8 vector which is fed as input to the MLP. The dataset is shuffled randomly and then split into three subsets: Training set, Validation set, and Test set. The Training and Test sets are divided into the ratio of (70:30) % respectively while the validation set is 25% of the training set. The training set is used to fit the model, validation set to evaluate the trained model and optimize the model parameters accordingly. Validation is carried out after every epoch in Keras. Finally, the trained model is evaluated against the test set which the model has never seen before. (Note that in all learning curves shown here the naming "test" is used for "validation".)

The dataset is already preprocessed inside the interactive myocontrol by a 1st order Butterworth filter with a cut off frequency of 1.5Hz [84] to remove high-frequency noise. No separate preprocessing was done in the python program.

Recording the correct data is of utmost importance in machine learning. Two datasets were recorded, one is with some hinges attached to the Myo for a firm grip on a skinny arm whereas the other without them. The Principal Component analysis tool was used to visualize the datasets. Dataset 1 (figure 7.1) is without hinges and Dataset 2 (figure 7.2) is with hinges. Both PCA explain 92% variance with 3 components. It is clear that the data without the hinges is not very well separated like Dataset 2.

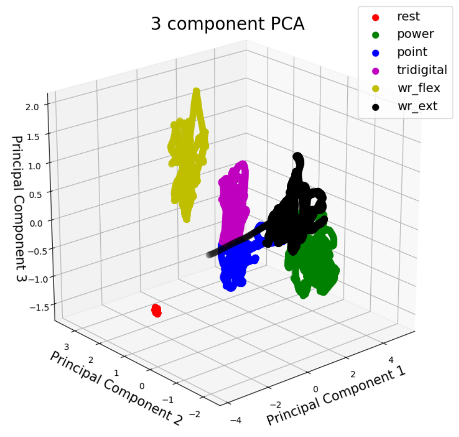


Figure 7.1.: Myo dataset 1

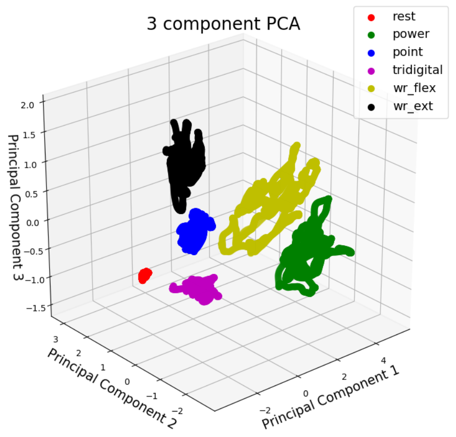


Figure 7.2.: Myo dataset 2

Without Preprocessing

The first model defined with dataset 1 has one input layer with 8 neurons and relu activation, one hidden layer with 20 neurons and relu activation, and an output layer with 6 neurons and softmax activation. Stochastic Gradient Descent (SGD) is used as the optimizer with a learning rate of 0.0001, categorical crossentropy loss, Earlystopping mechanism [85] monitoring the validation accuracy, 5 epochs and batch size of 4. The training data is shuffled randomly before each epoch. The test accuracy is 98.35% and loss is 26.53%. The learning curves are shown in figures 7.3 and 7.4. Its Confusion matrix is shown in appendix (fig. B.2). This model was unable to predict correctly despite its high accuracy as the learning was not sufficient and not converged.

The second model defined with dataset 2 is similar in structure to the first except that the learning rate of SGD is 0.00003, no Earlystopping to allow convergence of loss, 15 epochs and a batch size of 2. The test accuracy is 100% and loss is 8.426%. A good loss decay was

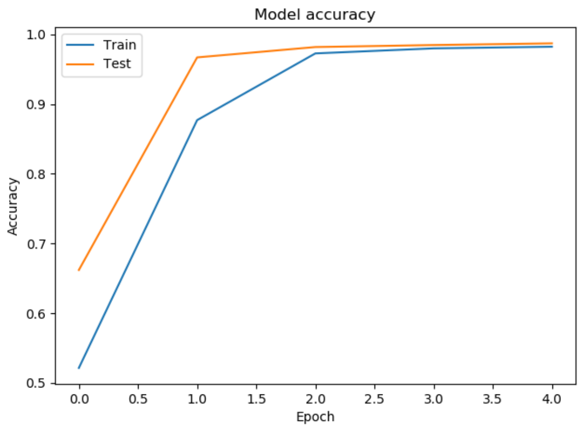


Figure 7.3.: Myo accuracy for dataset 1

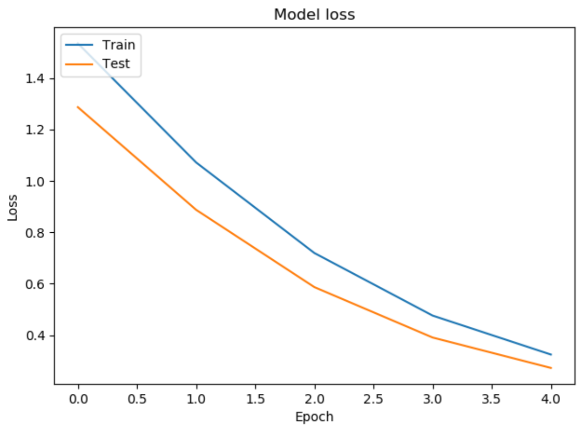


Figure 7.4.: Myo loss for dataset 1

obtained and the curves are shown in figures 7.5 and 7.6. Its Confusion matrix is shown in appendix (fig. B.3).

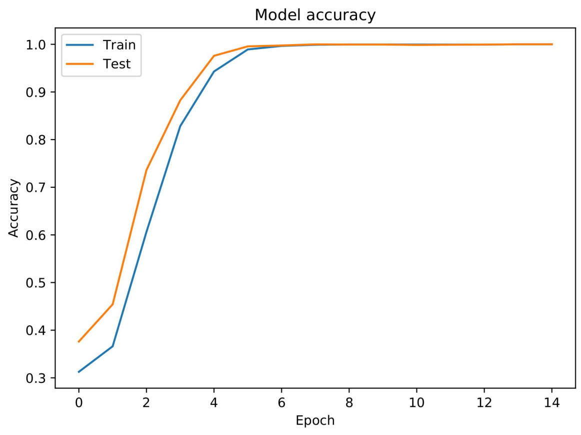


Figure 7.5.: Myo accuracy for dataset 2

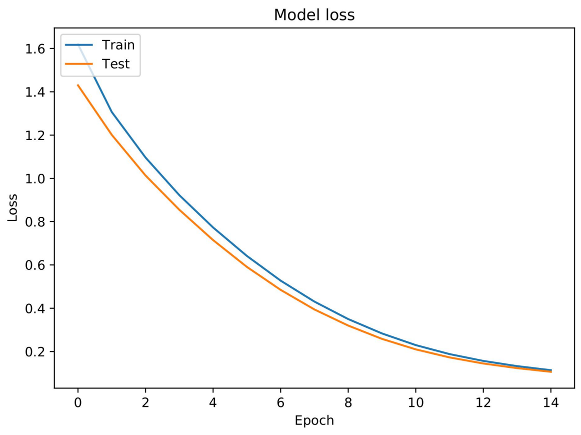


Figure 7.6.: Myo loss for dataset 2

With Standard Scaler Preprocessing

Most machine learning algorithms perform badly if the input features do not follow a Gaussian distribution i.e. zero mean and unit variance [85]. StandardScaler is a preprocessing technique offered by scikit-learn to standardize features by removing the mean and scaling to unit variance. A sample x is standardized as:

$$z = (x - u)/s$$

where u is the mean and s is the standard deviation of the training samples.

Considering this fact, the next model developed with the Myo dataset 2 used a two-level preprocessing. The first preprocessing occurs due to the Butterworth filter in the interactive myocontrol software and the second step with the StandardScaler in python. This 1×8 preprocessed input is then fed to the feed-forward structure defined previously with one input layer, one hidden layer, and an output softmax layer. Adam optimizer is chosen in this case with a learning rate of 0.0001, 6 epochs and batch size of 2. Adam is an improved version of the traditional stochastic gradient descent [86], [85] and is more practical to use [87]. The overall test accuracy and loss values are 100% and 2.123% respectively. The corresponding graphs can be seen in figures 7.7, 7.8 and B.4.

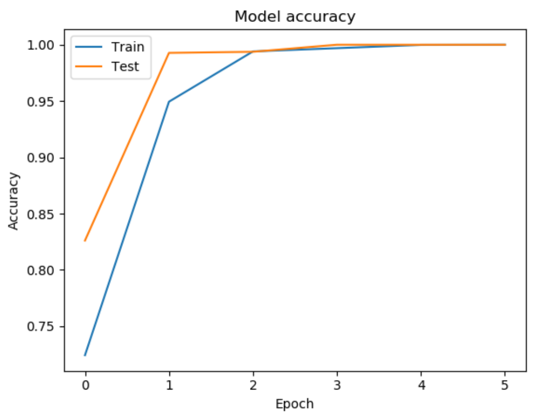


Figure 7.7.: Myo acc with preprocessing

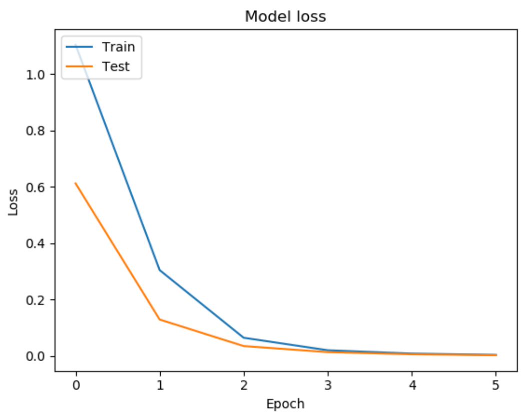


Figure 7.8.: Myo loss with preprocessing

As perceived, with StandardScaler, the learning converges much faster than previous models and the predictions are also better.

7.1.2. Case 2: Tactile Bracelet

Data is recorded and split as per the Myo version. The dataset is $\approx (5500 \times 288)$ matrix for 6 actions and 5 repetitions of each. The PCA of the dataset with 3 components describing 81.28% variance is shown in figure 7.9. The tactile bracelet has 9 modules with 32 force sensors each: total 288 values. This 1×288 vector serves as the input to the NN model. The input samples are preprocessed with the Standard Scaler and then given to the MLP with one hidden layer of 30 neurons. The Adam optimizer with a learning rate of 0.00002, 20 epochs and batch size of 2 is enforced. The test accuracy is around 98% and loss is 11.27%. The learning curves are depicted in figures 7.10, 7.11 and B.5.

Another modification of this model is to use StandardScaler followed by feature reduction using PCA corresponding to 98% variance i.e with 59 components as input. The architecture is the same as the previous TB model and Adam optimizer with a learning rate of 0.0001, 6 epochs and batch size of 2 is utilized. The overall accuracy is 99.94% and loss is 2.52%. The related curves are 7.12, 7.13 and B.6.

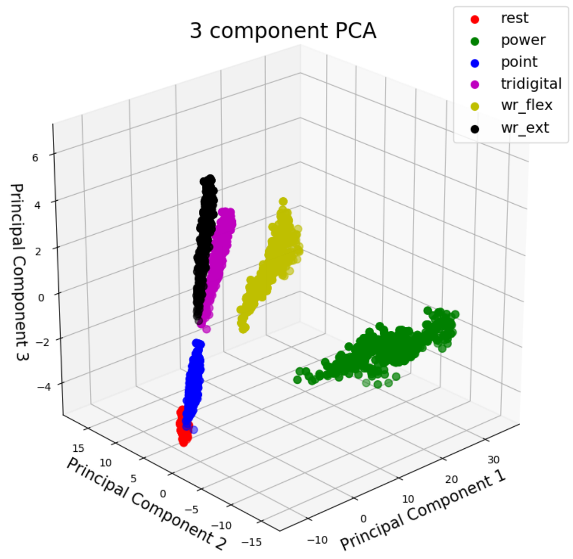
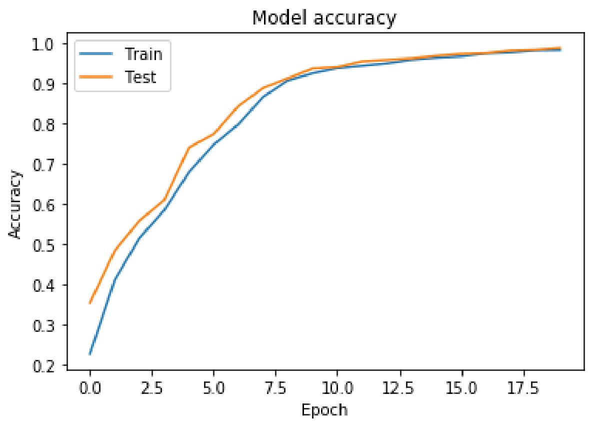


Figure 7.9.: Tactile Bracelet's dataset for vectorized input



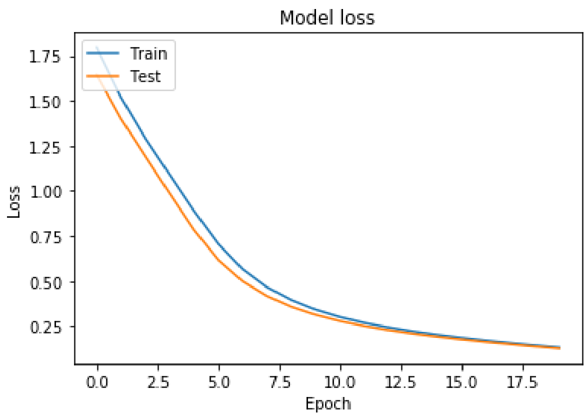


Figure 7.10.: TB accuracy with preprocessing

Figure 7.11.: TB loss with preprocessing

In general, the real-time predictions on the virtual hand model seem more stable with the TB compared to the Myo. Thanks to TB's higher resolution.

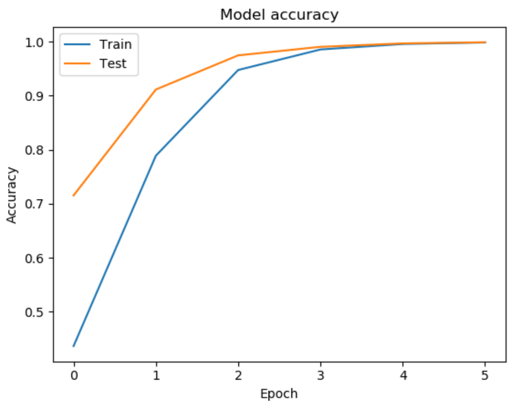


Figure 7.12.: TB accuracy with PCA and StandardScaler

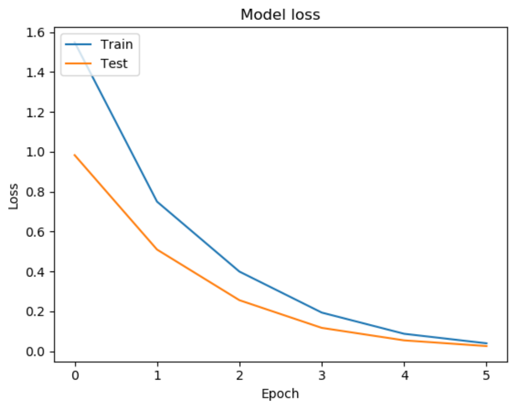


Figure 7.13.: TB loss with PCA and StandardScaler

7.2. Convolutional Neural Network with Tactile Bracelet

The tactile bracelet gives intensities of different pressure points on the forearm thus creating a pressure/force map. This image can be visualized in figure 7.14. In the previous MLP models, this property of the tactile bracelet was not exploited since the data was fed as a flattened array. As convolutional neural networks deal with inputs that are images, they allow for a more efficient implementation of the network with reduced parameters [73]. Thus the relationship among the pixels can be preserved which gives CNNs the potential for becoming better hand gesture classifiers.

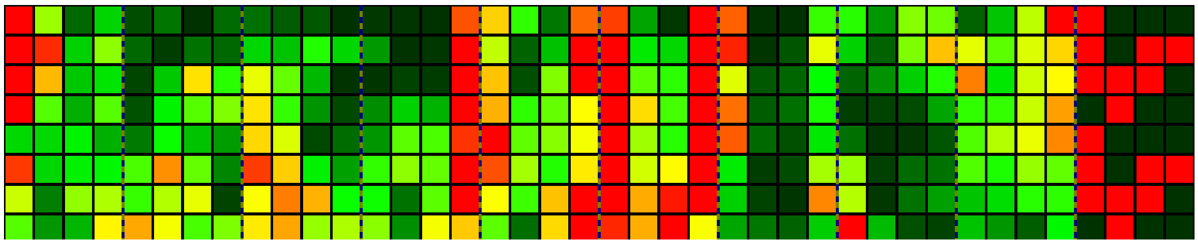


Figure 7.14.: Tactile bracelet showing different pixel intensities during Power action. Red signifies highest intensity and dark green the lowest. Only 9 modules of 8 × 4 pixels are connected. The 10th module to the extreme right is disconnected.

The final milestone of this research is to implement the CNN on the Cortex M3 microcontroller with the help of CMSIS NN libraries. This library currently supports finite layers (refer chapter 4) and hence the CNN architecture is also kept simple. The architecture of the CNN used here is depicted in figure 7.15.

7.2.1. Python Model on the PC

One of the example datasets, preprocessed with StandardScaler, and used for the CNN can be envisioned in figure 7.16. The data from 6 actions with 5 repetitions of each is split in the same sets and same ratios as the previous MLP models. The filtered dataset from the interactive Myocontrol is first converted from a 1×288 vector to 8×36 matrix using the following function:

```
static double[,] FromVectorToMatrix(double[] arr, int row, int col)
    {
        double[,] result = new double[row, col];
        for (int i = 0; i < 288; i++)
        {
            result[i % row, i / row] = arr[i];
        }
        return result;
    }
```

Listing 7.1: Vector to Matrix example

where $row = 8$ and $col = 36$.

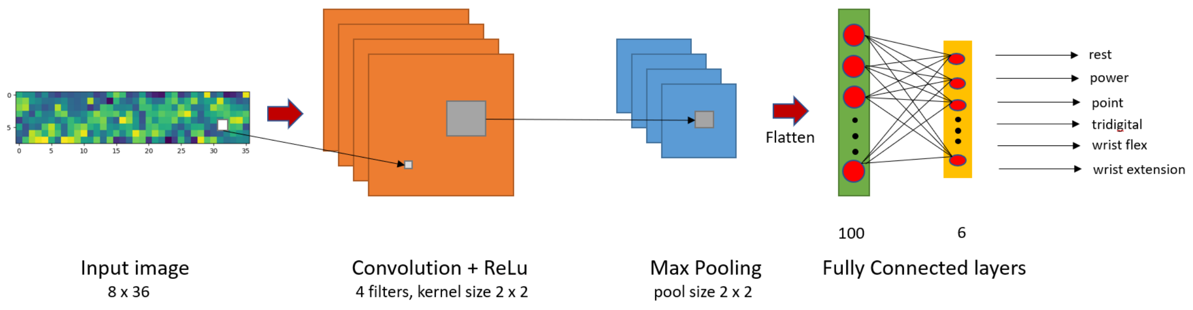


Figure 7.15.: The CNN Architecture used in this study

Then each image matrix in the dataset is standardized to a normal distribution and fed to the CNN. This process is also followed during the real-time prediction.

The CNN model in figure 7.15, defined in Keras, consists of a 2D convolutional layer with 4 filters and a kernel size of 2×2, the activation is rectified linear unit and input shape is $(8, 36, 1)$. The next layer is a Maxpooling layer with a pool size of 2×2 followed by a fully connected layer with 100 neurons and Relu activation. The CNN concludes with a softmax layer for converting the predictions into probabilities. The Adam optimizer is used with a learning rate of 0.0001, categorical crossentropy as the loss function, 20 epochs and batch size

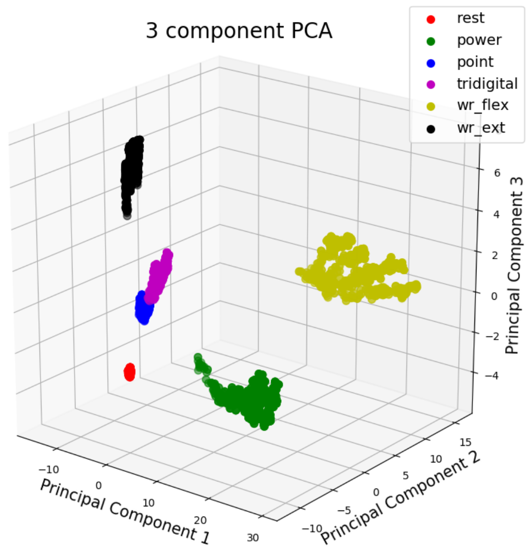


Figure 7.16.: PCA visualisation of the CNN dataset with 90.2% variance

of 100. The overall model test accuracy is 100% and the test loss is 2.91%. The related learning curves are pictured in figures 7.17, 7.18 and the confusion matrix is represented in 7.19.

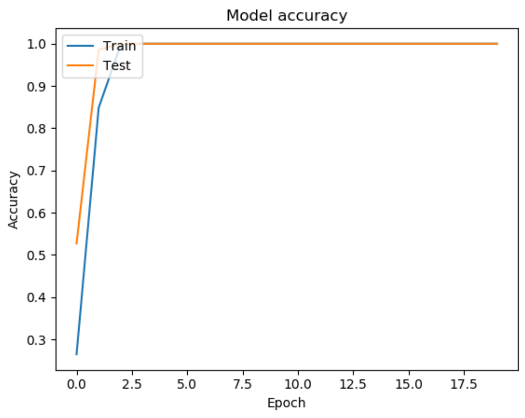


Figure 7.17.: CNN accuracy on PC

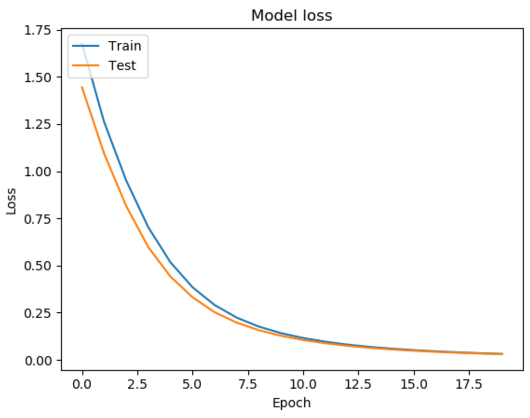


Figure 7.18.: CNN loss on PC

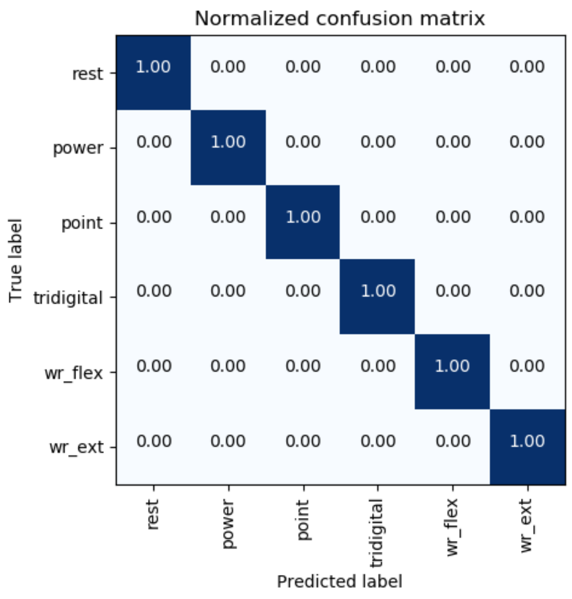


Figure 7.19.: CNN confusion Matrix

7.2.2. Caffe model

Caffe is a deep learning framework [88] developed at Berkeley AI Research by Yangqing Jia. It focuses on easy model descriptions, speed, and modularity. Models and optimizers can be defined as text files i.e. without any hard-coding. Therefore most neural networks for embedded implementations are defined in Caffe.

For porting the CNN onto the ARM Cortex M3 processor, the dataset is preprocessed using StandardScaler and stored in HDF5 format as required by Caffe. The Keras CNN model, described in the previous section, is rewritten in Caffe and stored as *mynet.prototxt* file and the solver/optimizer is defined in a *solver.prototxt* file. The solver parameters are as follows: Assuming a total of 5500 data points and the train-test split is 80:20 % then, training samples are 4400 and test samples are 1100. With a batch size of 100 to iterate over 4400 training samples, 44 iterations are required. Note that this is just one epoch of Keras. And for 20 epochs the max iterations in the Caffe solver file is set as 880. Validation must be carried out after each epoch i.e after every 4400 samples and so 11 test iterations are needed for traversing 1100 test samples @100 samples/iteration. The final model accuracy and loss values are reported after 880 iterations. The optimizer chosen is Adam with a learning rate of 0.0001. The solver mode is set as CPU to run the program on the CPU only. The sample model and solver definitions are shown in Appendix A.2.

A python script is written that automatically reads the dataset generated from Interactive Myocontrol, preprocesses it, converts it to HDF5 format, then defines the model and the solver in Caffe, and finally generates the trained model file with a *.caffemodel* extension. The CNN graph connectivity in Caffe is shown in figure 7.20.

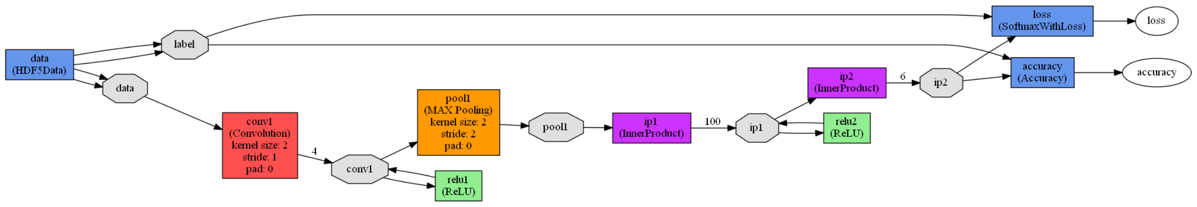


Figure 7.20.: Caffe CNN graph

7.2.3. Implementation on the MCU

The MCB1800 board uses bare-metal programming and therefore it is essential to translate the Caffe model into an equivalent C code to run on the microcontroller. Also, the Caffe model ($\approx$ 117KB) which contains the weights, biases, and activations will not fit directly into the microcontroller memory. Fixed-point quantization helps to avoid the costly floating-point computations and reduces the memory footprint for storing both weights and activations, which is critical for resource-constrained platforms [50]. Therefore quantization of input data, weights, biases, and activations from float 32 bit to integer 8 bit (or Q7 format) is done. The dataset in our problem is quantized to Q2.5 format meaning that 2 bits are used for the integer part and 5 bits for the fractional part.

To obtain the C program with CMSIS NN function calls from a trained Caffe model the following steps are performed in Python [89]:

1. Identify the network graph connectivity by parsing the Caffe model prototxt file
2. Quantize weights and activations to integer 8-bit
3. Generate optimized C code using CMSIS-NN Functions

ARM has readily provided two Python scripts: "*nn_quantizer.py*" for quantizing the model and "*code_gen.py*" for generating C code.

The quantizer code needs *mynet.prototxt* file which contains the Caffe model definition with valid paths to the HDF5 dataset and the trained model file (*.caffemodel*). The network graph connectivity is parsed and the Caffe model is quantized to 8-bit weights/activations layer-by-layer incrementally with minimal loss in accuracy on the test dataset [89]. Then the network graph connectivity and quantization parameters are stored into a pickle file (*.pkl*).

The code generator program reads the quantization parameters and network graph connectivity from the pickle file and generates the C code consisting of NN function calls. Currently, only convolution, innerproduct, pooling (max/average) and relu layers are supported [89]. The original ARM python script had some limitations regarding the order of layers to parse, so the program was modified accordingly to be compatible with the CNN Caffe model definition tested in this thesis. This script produces three files: weights.h, parameters.h (consisting of quantization ranges) and main.cpp containing the network code.

These 3 files can then be imported into the MCUXpresso IDE. Inside the interactive myocontrol, a separate function is defined to preprocess the input sample image as per the StandardScaler formula and quantize it to integer 8-bit (Q2.5) to reduce the overhead on the MCU side. Additionally, an integer 16-bit quantization function is also available. The quantized data samples (8×36) from Interactive myocontrol software are sent directly to the MCU via USB protocol for NN inference and the predicted values are communicated back to the C# program and further to the 3D hand model.

The CMSIS NN library was acquired from the demo programs in the eIQ library supplied with NXP's i.MX series processors. Although only square-shaped images were supported by the library originally, some modifications were made to the library functions to make it work with non-squared images. By pressing the Push Button 1 and 2 on the MCU board, prediction can be started and stopped respectively. The MCU model was then able to classify rest, tridigital, wrist flex, and wrist extension. Point and power were misclassified as tridigital.

7.2.4. Implementation on the Neural Compute Stick 2

As discussed in Chapter 4, the OpenVINO toolkit is the default software that provides different functions to get started with the Intel NCS2 on the Windows PC. The following procedure is obeyed to perform inference with the NCS2:

1. Connect the stick to the PC and set up OPENVINO environment variables
2. Preprocess the dataset and train the CNN using Keras APIs in Python and save the model as a single .h5 file.
3. Load the .h5 file and freeze the graph to a single TensorFlow .pb file.
4. Run the OpenVINO *mo_tf.py* script, which is a TensorFlow model optimizer, to convert the .pb file into an intermediate graph readable by the NCS2. The input shape is set as $[1, 8, 36, 1]$ and the datatype is set to floating-point 16 (FP16) to gain extra speed up when inferencing. Two files are generated: .xml and .bin files.
5. Load the model XML and BIN files with OpenVINO inference engine and make a prediction.

The same code snippet used to interface interactive myocontrol with the CNN python program on the PC to send input samples via UDP and prediction back to the 3D hand model is applied with the NCS2 program for real-time predictions.

7.3. A Generalised Neural Network

As a test case, it was tried to develop a general CNN model for the hand gesture recognition application with the tactile bracelet. At first, the foam on the TB was renewed and then the data was recorded from 3 female and 3 male subjects for a first dataset containing: rest, power, point, tridigital, wrist flex, and wrist extension and the second one with rest, power, tridigital, wrist flex, wrist extension, wrist pronation, wrist supination and five repetitions of each. The two datasets can be seen in figures 7.21 and 7.22 with ≈75% variance.

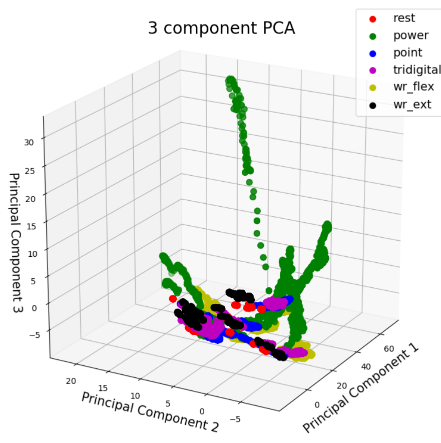


Figure 7.21.: Generalised CNN Dataset 1

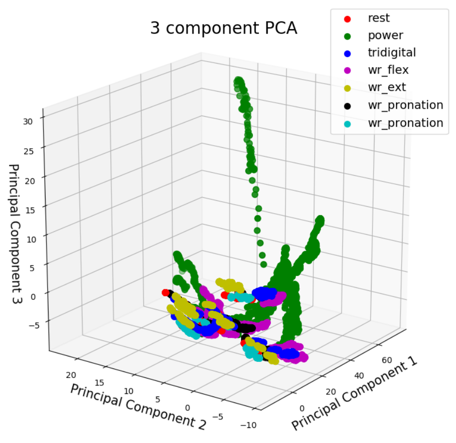


Figure 7.22.: Generalised CNN Dataset 2

Both datasets are trained with the same previous CNN architecture. The trained model was then used for prediction on a subject whose data was unseen by the model. The model accuracy with the first dataset without pronation and supination was 99.43% and loss was about 6%. For a new subject, it was observed that power, wrist flex, and extension were detected by this model with more stability than rest, point and tridigital. The corresponding accuracy and loss curves are shown in figures 7.23, 7.24 and B.7. With the second dataset, the model performed poorly due to close resemblances among the classes except for power.

The misclassifications with both datasets could be mainly due to variations in muscular sizes and activations among different people. Some may also lack a particular muscle of interest. However, this methodology was not examined in depth due to time constraints.

An overview of all neural network models analyzed are listed in table 7.1 and their accuracies and losses are compared in figure 7.25. In general, from the table, it can be reported that standardizing the dataset to a Gaussian distribution results in improved loss values. For the combination of MLP with the Myo and SGD, it is noted that high accuracy values are attained but still, the loss is not converged enough thereby suggesting to train the model for more epochs. This can be because the input dataset consists of small decimal values and the gradients change very fast due to the smaller batch size and therefore the learning rate must be kept low. But with an Adam optimizer, faster convergence of loss curve was observed in all cases. Also, due to the smaller batch size, the model learns from

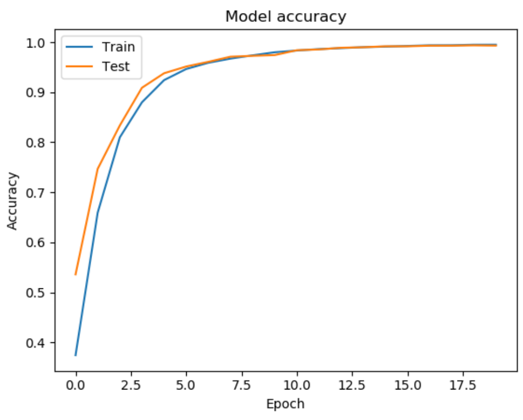


Figure 7.23.: Generalised CNN Accuracy

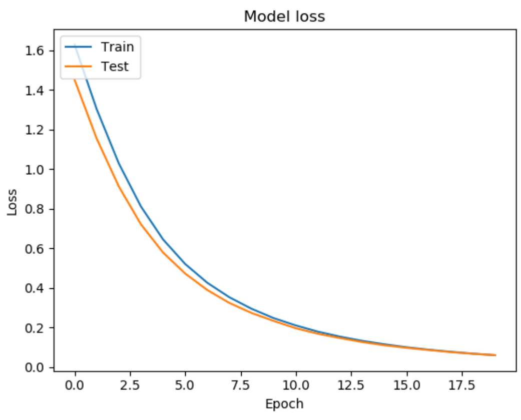


Figure 7.24.: Generalised CNN Loss

specific examples before updating the weights and hence cannot track the variable behavior of the dataset properly. Moreover, the instability of sEMG signals must also be considered and the smaller resolution of the Myo. Moving to the tactile bracelet, the MLP with reduced components using PCA plus StandardScaler shows significant improvement in loss values even with less number of epochs. CNN with TB uses a higher batch size thus making it capable of learning the variations in data in an improved manner which is not possible for the MLPs. Thus high accuracy values can be achieved while keeping the generalization error low. Due to the high density of the TB more training epochs are needed.

NN style	**Sensor**	**Optimizer**	**Batch Size**	**Epochs**	**Learning Rate**	**Accuracy**	**Loss**	**Remarks/ Preprocessing**
MLP Dataset 1(7.1)	Myo	SGD	4	5	0.0001	98.35%	26.53%	Early Stopping
MLP Dataset 2(7.2)	Myo	SGD	2	15	0.00003	100%	8.42%	-
MLP Dataset 2(7.2)	Myo	Adam	2	6	0.0001	100%	2.12%	Standard Scaler
MLP	TB	Adam	2	20	0.00002	98%	11.27%	Standard Scaler
MLP	TB	Adam	2	6	0.0001	99.94%	2.52%	PCA + Standard Scaler
CNN	TB	Adam	100	20	0.0001	100%	2.91%	Standard Scaler
General CNN	TB	Adam	100	20	0.0001	99.43%	6%	Standard Scaler

Table 7.1.: An overview of different NN implementations

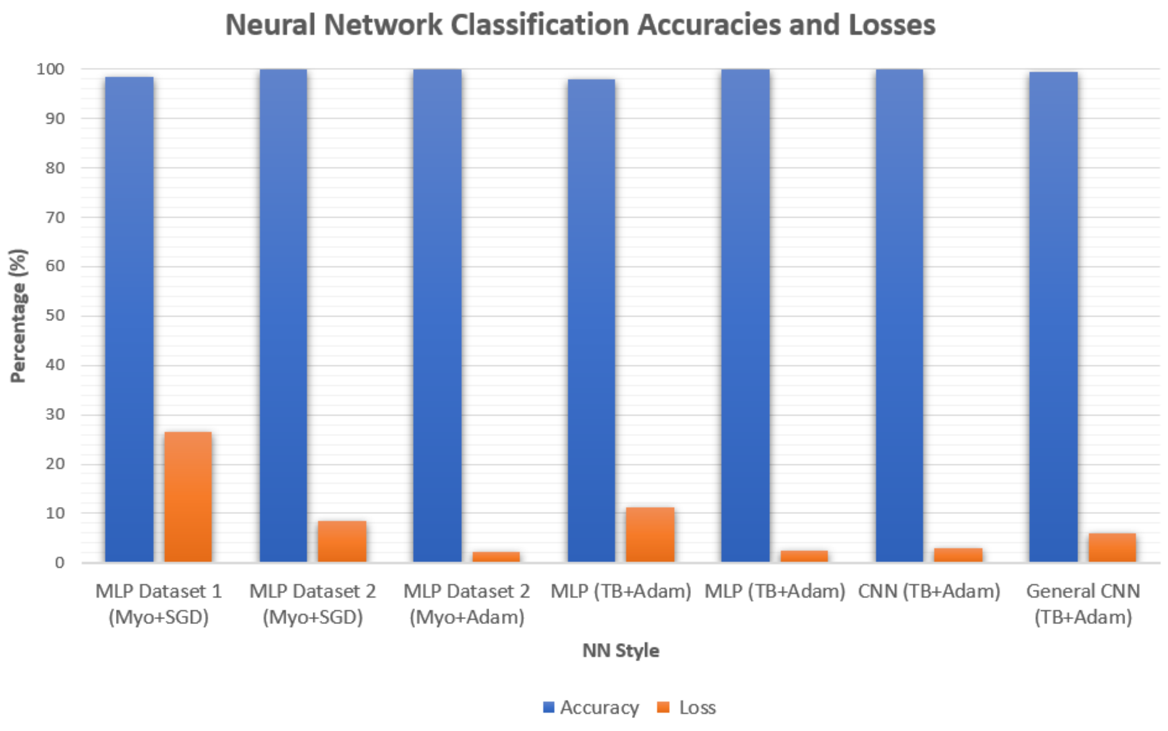


Figure 7.25.: Accuracies and losses for different neural networks

8. Results and Observations

This chapter presents the results of regression and neural network models examined in this research. They provide a comparative overview for Ridge Regression and Convolutional Neural Networks and their implementation on embedded hardware like the Cortex M3 microcontroller and the Intel Neural Compute Stick 2.

The analysis of memory footprint, training and inference speeds of the Ridge Regression algorithm on the microcontroller are listed in table 8.1. For the MCB 1800, Flash A and B are 512KB each, SRAM is 40KB and SDRAM is 16MB. Data were recorded for different configurations of (rest, power, point) on the author's right arm with the help of Tactile Bracelet. The 3D hand model is used for predictions. A similar dataset was created for the Ridge Regression with Random Fourier Features using the Myo Armband and implemented on the MCB1800 board which yields the results as per table 8.2. As expected from the static analysis of the regression methods in chapter 6, the RRRFF take more time to train as the number of Fourier features increase due to the high dimensional feature mapping. The memory requirements for both RR and RRRFF are similar except for the higher SDRAM usage in case of RFF because of the computations required for the mapping function. No significant improvement regarding prediction quality was observed with D=100 and D=300 concerning rest, power and point gestures. Inference speeds remain the same for RR as well as RFF.

Configuration	**Memory**				**Training Time**	**(180×288) X Matrix Transfer Time**	**(1×288) Vector Transfer Time**	**Inference Time**
	Flash A	Flash B	SRAM	SDRAM				
2 actions 2 repetitions	9.88%	0%	13.80%	45.71%	2.19 min	5.6 sec	50 ms	20-30 ms
2 actions 3 repetitions					2.84 min			
3 actions 2 repetitions					2.82 min			
3 actions 3 repetitions					3.81 min			

Table 8.1.: Memory requirements and execution times of different RR configurations for rest, power and point actions on Keil MCB1800 with the Tactile Bracelet

Configuration	**Memory**				**Training Time**		**(400×8) X Matrix Transfer Time**	**(1×8) Vector Transfer Time**	**Inference Time**
	Flash A	Flash B	SRAM	SDRAM	D=100	D=300			
2 actions 2 repetitions	12.17%	0%	13.83%	96.77%	26.20 sec	4.31 min	450 ms	30 ms	20-30 ms
2 actions 3 repetitions					38.32 sec	5.94 min			
3 actions 2 repetitions					38.27 sec	5.93 min			
3 actions 3 repetitions					56.07 sec	7.5 min			

Table 8.2.: Memory requirements and execution times of different RRRFF configurations for rest, power and point actions on Keil MCB1800 with the Myo Band

Implementation of the CNN architecture, defined in chapter 7, on the Cortex M3 board and Intel Neural Compute Stick 2 for the dataset with (rest, power, point, tridigital, wrist flex, and extension) actions, which were recorded from the author's right arm with the TB using integer 8-bit and 16-bit quantization (for MCB1800), and floating-point 16-bit quantization (for NCS2), are put together in tables 8.3 and 8.4 respectively. The frames per second (fps) value is calculated as reciprocal of the inference time. Memory requirements are compared concerning the size of the saved model and weight, biases, activations in Keras, Tensorflow and OpenVINO IR graph: XML and BIN files.

Configuration	**Memory**				**Training Time**	**X Matrix Transfer Time**	**(1×288) Vector Transfer Time**	**Inference Time**
	Flash A	Flash B	SRAM	SDRAM				
6 actions 5 repetitions (8 bit)	14.46%	0%	15.63%	0%	N/A	N/A	50ms	30ms
6 actions 5 repetitions (16 bit)	20.07%	0%	20.85%	0%	N/A	N/A	50ms	30ms

Table 8.3.: Memory requirements and execution times of different CNN configurations for rest, power, point, tridigital, wrist flex and extension actions on Keil MCB1800 with the Tactile Bracelet

Configuration	**Memory**				**Training Time**	**Inference Time**
	Keras	Tensorflow	XML	BIN		
6 actions 5 repetitions (FP16)	276 KB	86 KB	7 KB	42 KB	N/A	22.3ms (fps=44.729)

Table 8.4.: Memory requirements and inference time of CNN model for rest, power, point, tridigital, wrist flex and extension actions on Intel Neural Compute Stick 2 with the Tactile Bracelet

The table 8.5 depicts the output of the CNN running on a Windows 10 host CPU in Python and using Keras library. Model and weights are saved as .json and .h5 respectively using standard Keras functions.

Configuration	**Memory**		**Training Time**	**Inference Time**
	Keras model	Keras weights		
6 actions 5 repetitions (FP32)	2 KB	100 KB	16.02 sec	10.6ms (fps=94.178)

Table 8.5.: Memory requirements and inference time of CNN model for rest, power, point, tridigital, wrist flex and extension actions on Intel i7 dual core CPU with the Tactile Bracelet

Finally, to evaluate the ridge regression and CNN methods on the PC, a quick data recording session of 30 minutes each was performed on 3 male and 2 female subjects. The details of the procedure can be found in the appendix A.3. All of them wore the TB on their right arm and recorded data for 6 actions and 5 repetitions of (rest, power, point, wrist flex, and extension). Both the algorithms were compared on the PC by calculating the Normalised

Root Mean Squared Errors (nRMSE) for all the subjects (refer table 8.6). Figure 8.1 shows a better visualization of the comparisons. Though a direct comparison between regression and classification is not possible, to understand the performance of both methods, the CNN's output was converted to represent regression using a linear activation instead of softmax in the last layer. The NRMSE for the RR was calculated considering only a 5-DOF vector to remove redundancy from the original 9-DOF vector. In all subjects, although the NRMSE for the convolutional network is less than RR the fact that regression predictions track the hand in a more realistic sense cannot be overlooked.

Subject	**Gender**	**Age**	**Weight (kgs)**	**(Height (m)**	**Hand Dominance**	**Forearm circumference at Rest (cm)**	**Forearm circumference at Power (cm)**	**Hand activities**	NRMSE	
									RR	CNN
1	F	21	63	1.57	right	25.5	26	Daily	0.1147	0.0842
2	F	27	61	1.61	right	24.8	25.3	Daily	0.2544	0.1917
3	M	25	70	1.7	right	26	26.8	Daily	0.2421	0.1342
4	M	27	90	1.8	right	32	32.5	Gym	0.1058	0.1004
5	M	25	54	1.6	right	22	22.5	Daily	0.1124	0.0652

Table 8.6.: Comparison of normalised root mean square errors for various subjects in Ridge Regression (RR) and CNN running on the PC with the Tactile Bracelet

Method	**Memory hunger**	**Inference Speed**	**Training**	**#actions supported**	**Coding**	**Predictions robustness**	**QUALITY INDEX**
RR on MCU	3	2	1	3	1	2	2.00
CNN on MCU	1	2	5	1	4	3	2.67
CNN on Intel NCS2	2	2	5	1	2	1	2.17
CNN on PC	4	1	1	1	2	1	1.67
RR on PC	5	1	1	1	3	1	2.00
MLP on PC	2	1	1	1	2	4	1.83

Table 8.7.: Performance Index of different methods with Tactile Bracelet

Furthermore, to evaluate the ease of the process of defining the individual models to their implementation on the embedded platform, every method was giving a score from 1-5 for each criterion (using "lower the better" policy) listed in table 8.7. The Quality Index is then useful to determine how well each method performed. This index is calculated as an average of the scores obtained by each method in every criterion. Figure 8.2 gives a graphical view of the table 8.7. The CNN on PC ranks first due to easy access to ML libraries, smoother set up on the PC, a large volume of available memory and robust predictions. On similar grounds, the MLP on PC ranks seconds for its poor prediction stability. Both RR on PC and RR on MCU methods rank third due to high memory usage in Visual Studio during training in the former and a limited number of actions supported with intermediate memory consumption in the latter. CNN on Intel NCS2 shows comparable performance to its PC equivalent method but no training possibility brings it down to number four. CNN on MCU requires less memory footprint than other methods yet it gets the last position due to two reasons: firstly, because of the inefficiency in detecting point and power actions and secondly due to the tedious implementation process from training the model in Caffe, using python scripts for quantization plus C code generation and finally importing it in the MCU.

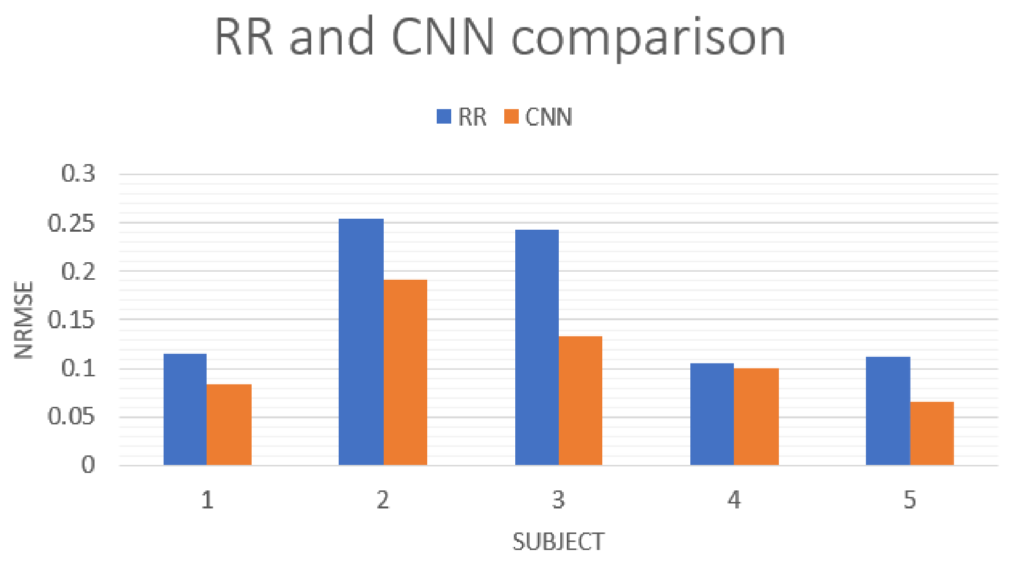


Figure 8.1.: Normalised Root mean square errors (NRMSE) in Ridge Regression (RR) and Convolution neural network (CNN) for 5 subjects

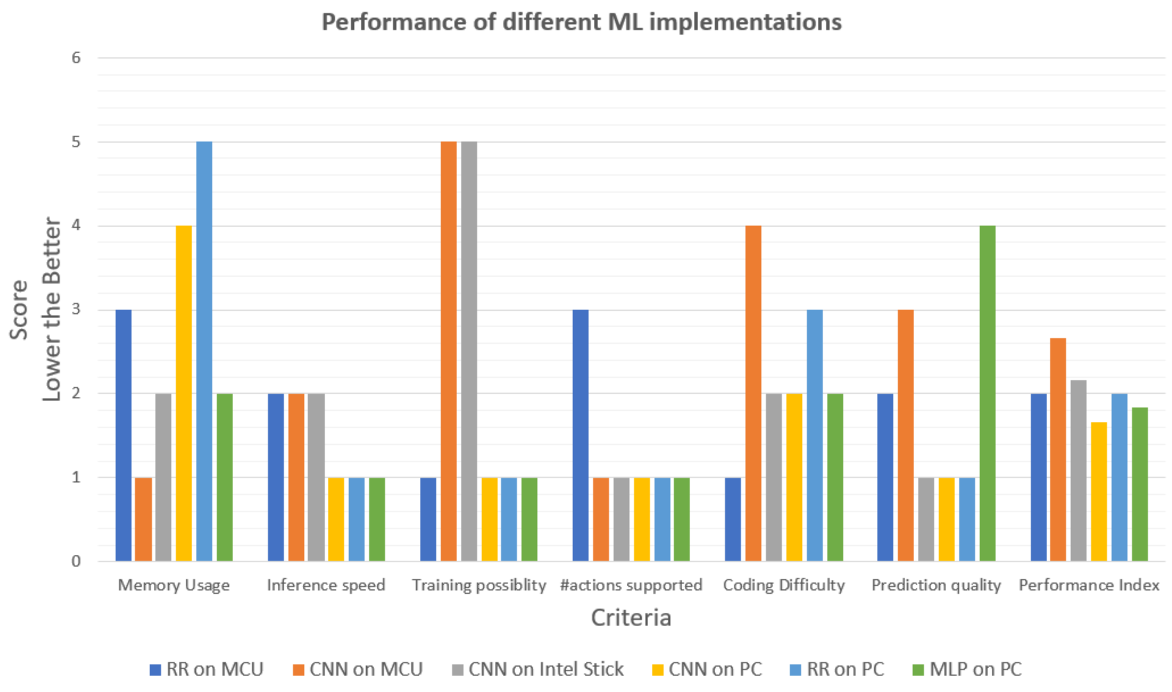


Figure 8.2.: Performance Index of different ML algorithms investigated with Tactile Bracelet

9. Discussion

This research proposes a further investigation into the practical usage of Force myography for a multi DOF prosthesis control on a low cost and efficient embedded platform. It is essential to jot down a couple of points.

Myocontrol based prosthetic devices can be a potential replacement for upper limb amputations. It uses electrical signals of the muscles to control the prostheses. The sensors used to measure these signals must be able to extract distinct and repeatable patterns reliably. Traditional methods depend on surface electromyography for signal recording. However, the disadvantages of sEMG are well known to the scientific community. Apart from the problems related to the electrode placements and lift off, the quality of EMG signals is also degraded due to sweating, crosstalk, fatigue and electrode skin impedance changes [90], [40], [42]. Force myography seems to be a viable alternative for this. FMG sensors detect the deformations of the residual limb via pressure and the signals are not affected by conductivity of the skin or by fatigue. But still, the problems related to electrode placement, good contact with the skin and position of the arm (limb position effect) persist. Every person has a different muscular structure and may lack a particular muscle of interest. These deformations in the skeletal muscles can be problematic for the detection of activation patterns. Having said that, the advantages of FMG uphold its value. The signals of FMG have less oscillatory behavior and the signals are better separated in the input space [47]. Many other studies also show that the use of FMG allows for the classification of a relatively high number of hand and wrist movements with remarkable accuracy.

Regression and Artificial Neural Networks are the two ML algorithms worked upon in the scope of this thesis. It indicates that the ridge regression with the tactile bracelet works well rather than using the computationally expensive ridge regression with random Fourier features with the Myo band. From the results, it is evident that RR with Tactile bracelet requires less memory footprint compared to the RFF version with Myoband. The training time increases linearly with an increasing number of actions and repetitions in case of RR whereas there is a steep jump in training time for RFF as the number of Fourier features is increased. This was also inferred from the complexity analysis of RFF in chapter 6. Also, the SDRAM is completely used up for RFF to store intermediate matrix multiplication results and any further functionality seems difficult to add. To the best of my knowledge, most regression-based hand prostheses control have been trained on a PC or a smartphone and only the inference is done with the help of a microcontroller [19]. However, within this thesis, a working microcontroller-based regression training has been implemented for recognition of 3 actions with 3 repetitions of each yielding recognition accuracies around 99-100% and within 20-30 milliseconds. Also, no windowing technique was used [43] and therefore there is a negligible delay between data acquisition and prediction.

A vital role is played by small efficient libraries for the microcontroller. The GSL library, in C, gave a speedup by a factor of 12 for RFF training with 17 actions and 3 repetitions of each on the PC. This was mainly due to the optimized C kernels. But on the microcontroller, GSL seemed to be a bad choice due to its dynamic memory allocation requirements. Keil's very own CMSIS-DSP libraries are designed efficiently for ARM architectures but have some bugs. Hence, using the custom matrix multiplication and inversion functions seemed relevant for our application and surprisingly have performed well. Even though floats were used instead of doubles there is almost no degradation in the prediction quality.

From past research [22], [10], [16], [22], [25], it can be seen that NN tend to perform better than most ML algorithms due to their ability to learn complex features on their own. NNs have outperformed other classical approaches not only in prosthetic control strategies but also in other fields like autonomous driving [91], [92], etc. Therefore, the first multiplayer perceptron was examined with the Myoband. Due to the small fractional numbers in the dataset, the network was kept simple with minimum neurons without degrading the performance. Moreover, the final aim was to implement it on the microcontroller so it is essential to keep the network small so that it fits inside the controller's memory. As a pilot study, the MLP was evaluated with 2 datasets, one where the Myo firmly fits the author's arm and the other when it is slightly loose due to thin arm. It was observed that the classifications with a firm fit were better than the other case. This implies that the dataset must be recorded in an appropriate manner and by keeping the arm in a stable position during experimentation. And also we must not forget about the instability of Myo signals. Extracting the right features is equally important for good learning in NNs. Hence, appropriate preprocessing tools must be used before actually showing data to the model. Here the dataset was brought into a normal distribution with some preprocessing as most NNs assume the input data to be normally distributed [85]. And as expected the loss converged must faster thereby giving more stable predictions.

But our main interest lies with the Tactile Bracelet. The TB with 288 FSRs generates a 1×288 row vector which is fed as an input to a feedforward neural network with one hidden layer. This small architecture makes sense since the output from the TB are doubles values (e.g. 0.113525391) and also the gradient takes tiny steps to reach the optimum. Hyper-tuning helped to adjust the parameters and fix the layers for the models defined in chapter 7. The learning rate was kept small. The standard scalar preprocessing was used to standardize the dataset. From the PCA analysis, it was seen that power and wrist movements have better separability than rest, tridigital and point. The pointing gesture decreases the prediction performance. Using the reduced features by PCA analysis in addition to standard scaling also gave comparable results to using standard scaling alone. Besides, the high resolution of TB gave superior stability during predictions in contrast to the Myo.

The way to unleash the potential of the TB is to exploit the high-density FMG images clicked by the 288 pressure sensing cells (in our case) further with the power of CNNs. A simple convolutional network was chosen after a few trials. The Adam optimizer seemed to converge better than stochastic gradient descent. The relationship between adjacent pixels helps to unveil important properties of the muscular signals which helps to gain superior performance than the MLPs. Again the recording of a correct dataset is an important task.

The PC version of the CNN was tested with 5 subjects and 6 actions. This network is capable of achieving accuracies up to 100% with models and weights weighing only about 100KBs. The pointing gesture was detected with better stability than previous vectorized versions. For the implementation of this model on the MCU, a lot of pre-setup was needed: Caffe installation, defining the model in Caffe and conversion to C using the python scripts by ARM. This is an additional effort compared to the regression implementation on the MCU. Also, there seems to be no NN training possibility at the moment on the embedded hardware. The CMSIS-NN functions help to do efficient computations. However, it was observed that the quantization of data, weights, activations and biases from float 32 bit to integer 8 bit (Q7 format) didn't perform well for this particular application. The analysis of datasets both before and after quantization showed no difference in the PCA suggesting that the problem is during the quantization of weights and activations. Due to the smaller scale available to represent each class i.e. (-128, +127) compared to the float 32 range and saturation of results, precision is lost. Though the quantization should affect the accuracy of the model by a negligible amount in image classification tasks as per ARM, it should be noted that the TB image is not actually grayscale in integer range (0, 255) and thus the loss in accuracy is not acceptable for this particular case. With a few trials on the MCU, it was still manageable to detect rest, tridigital, wrist flex and extension but the performance is poor compared to its PC or Caffe version where all actions including point are detected with good stability. The 16-bit quantization model performed even worse than the 8 bit model on the MCU. However, this experiment was done only on the author and was involved in operating the PC after recording the data to implement the code on the MCU for testing. Thus it can be due to hand movements that the signals were disturbed and hence the poor performance. Regarding the speed, it can be said that the prediction time (30 ms) on the MCU was almost equivalent to the experiment on the high-performance Nvidia TX2 even with a simpler CNN [12]. Also, the power consumption will be lower for the Cortex M processors as opposed to the TX2. One reason for this is that no OS and only bare-metal programming is used and no energy-hungry component like the GPU is needed.

Using a dedicated neural accelerator like the Intel neural compute stick is a pioneering approach for hand movement recognition applications. The CNN model was implemented in a much easier and cleaner way into the NCS2 with a floating-point 16 quantization. It yielded equivalent performance to the PC based model and outperformed the MCU version (without an FPU) as well. However, the inference time on the CPU was faster than the stick because only a simple model was being tested and the full potential of the SHAVE cores was not utilized. With a more complex CNN, the performance benchmarks stated by Intel imply that the optimized model on the stick will perform much better and efficiently than on the PC. Also, the models stored on the stick require only about 50KBs which is less than the PC counterpart.

Keeping a threshold of 20 predictions (i.e. if a particular action is predicted for 20 consecutive counts then only show it on the virtual hand model) helped to remove intermediate noise caused by fast finger movements or misclassifications thus giving stable gestures. However, it must be said that classification accuracies tend to decrease with the inclusion of finger movements along with wrist actions [43].

The TB consists of a foam between the sensor cell and the skin. As the bracelet is shape conformable, a settling time of 15 mins needs to be kept so that the foam settles on the hand. But after recording data for another 15 minutes, the foam takes time to adjust itself to the shape of the muscle activations and thus the prediction may take about 4-5 seconds to detect a gesture correctly which is a drawback for the TB (for instance, doing power and coming back to rest would not happen instantly but would need about 4-5 seconds due to foam deformity). Also, the tightness of the bracelet on the arm affected its signals unlike the "one size for all" Myo. In the case of TB, wires are prone to breaking and frequent disconnections were observed, the foam was too sensitive at the edges of the sensor modules thus creating an offset. Some modules would have to be discarded for a thin person resulting in reduced resolution. The USB cable caused discomfort for subjects when testing, unlike the Myo which uses Bluetooth protocol for communication with the PC.

The final experiments were recorded with 2 female and 3 male subjects belonging to the same age group (25-27 years) and the same dominant hand. All the experiments were carried out with RR and CNN running on the PC. The normalized RMSE values were calculated in the case of each subject and for both learning strategies to examine their performance. A direct comparison between regression and classification is however not possible. In one sense the lower errors in CNN indicate that CNNs are less susceptible to noise than regression which could be because the integrity of the pressure pixels is maintained.

The real-time hand gesture predictions on the MCU or Neural stick could not be performed for other subjects except the author due to time restrictions considering other critical situations at the time. However, an embedded solution with a good compromise between prediction accuracy, energy, and cost efficiency is possible with the demonstrated methods. To rate the different methods, a score was assigned based on different criteria for each method. The PC based CNN is the best shot to develop a CNN from scratch compared to others due to the use of high-level languages, high-end processor and better library support. The RR on MCU is as easy to implement as on the PC. CNN on the Intel stick is at an intermediate level while the CNN on MCU has a lower index overall because of the extra overhead to retrain the algorithm in Caffe, manually run quantization scripts and convert the model and weights to C and finally import the codes into the MCUXpresso IDE and run it.

A generalized CNN was also examined wherein the data recorded for 6 subjects can be used for prediction with a new person although every individual has different muscular activations. Surprisingly the model was able to predict wrist flex, wrist extension and power mostly for all unknown volunteers. But detecting rest, tridigital and point actions were quite challenging.

As far I know, this is the first attempt to train and predict hand gestures on low-performance and energy-efficient microcontrollers with tactile myography or high-density force myography (HD-FMG). Overall accuracies in most cases were between 99-100% and losses between 2-5%. All in all, HD-FMG serves to be a viable replacement or a companion to sEMG.

10. Conclusion and Future Work

In this research, a real-time hand gesture recognition model based on Tactile Bracelet has been presented. FMG signals, which are more stable than EMG, are extracted with the help of TB. The recorded data is normalized to a Gaussian distribution which showed better performance. A regression-based machine learning algorithm and a Convolutional Neural Network is implemented on a low performance embedded processor. For prediction by the CNN, a threshold was kept so that when a recognized label reaches the threshold of activation times then only the prediction can be seen on the 3D hand model. This helps for debouncing and improves prediction stability. The inferences of CNN and RR model are expected within 20-30 ms which is acceptable for real-time predictions. Recognition accuracy in most cases reaches 100%. The pros and cons of different methodologies and the sensors have also been listed. So far the TB seems to be a good solution for performing superior gesture recognition than the Myo.

More complex CNN architectures can be investigated in the future and could outperform the inference speeds on the PC. Also, a better processor with an onboard floating-point unit can help to significantly speed up the computations. The structural design of the TB can be improved further and more pressure sensing cells can be installed to get a more dense picture of the muscle activations thereby improving the system's ability to recognize finger movements.

In a nutshell, it is possible to develop low cost and efficient prosthetic devices with the use of low-performance processors along with efficient and optimized ML libraries. Hopefully, the device-specific libraries (like CMSIS) would also add new functionalities and support more layers in due course of time allowing us to implement complex preprocessing techniques and ML algorithms on the embedded platform. Perhaps, with additional memory on MCU, even more gestures can be classified. Point gesture could be detected with higher accuracy using the CNN approach. Other ML libraries like TensorFlow-Lite also could be an alternative. It is expected that ARM develops a better portation process for the Keras and TensorFlow models to the microcontrollers.

Neural network training on the MCU might also become a reality with rapid developments in the semiconductor industry and it cannot be denied that a generalized machine learning algorithm with suitable feature extraction can be developed for use by any person without the need to retrain again and again. These techniques can be applied not only in the medical industry but also for other domains like space, automobiles and many more where power, area, and cost are critical properties.

A. General Addenda

The following sections contain extra details for further reference.

A.1. GSL Cross Compilation procedure

1. Download following files to Linux desktop:
   a) Download gsl-2.5 file from `ftp://ftp.gnu.org/gnu/gsl/`
   b) Download gcc-arm-none-eabi-8-2018-q4-major-linux.tar.bz2 file from `https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads`

   Type the following commands in the Linux terminal:
2. *apt-get update*
3. *apt-get upgrade*
4. *sudo apt remove binutils-arm-none-eabi gcc-arm-none-eabi libnewlib-arm-none-eabi*
5. *sudo apt-get install make*
6. *cd Desktop*
7. *cd gsl-2.5*
8. *make clean*
9. *tar –xjvf gcc-arm-none-eabi-8-2018-q4-major-linux.tar.bz2*
10. *export PATH=$PATH:/home/username/Desktop/gcc-arm-none-eabi-8-2018-q4-major/bin/*
11. *CFLAGS="-Wall -O0 -g3"*
12. *CFLAGS+=" -mcpu=cortex-m4 -mfloat-abi=soft"*
13. *CFLAGS+=" –specs=nosys.specs"*
14. *export CFLAGS*
15. *./configure –host=arm-none-eabi –target=arm-none-eabi*
16. *make*

17. *make check* (will fail but continue with next step)
18. *make install*

GSL files must be in */usr/local/include* & *pkgconfig* + *libgsl.a* + *libgslcblas.a* in */usr/local/lib*

Important: Whenever user changes from/to root privileges need to repeat step 10

A.2. Caffe network and solver definitions

Model Definition (*mynet.prototxt*)

```
layer {
  name: "data"
  type: "HDF5Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  hdf5_data_param {
    source: "C:\\caffe_myexamples\\cmsis_nn_data_1\\train_myo.txt"
    batch_size: 100
  }
}
layer {
  name: "data"
  type: "HDF5Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  hdf5_data_param {
    source: "C:\\caffe_myexamples\\cmsis_nn_data_1\\test_myo.txt"
    batch_size: 100
  }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  convolution_param {
    num_output: 4
    kernel_size: 2
    weight_filler {
      type: "xavier"
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2
  }
}
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool1"
  top: "ip1"
  inner_product_param {
    num_output: 100
    weight_filler {
      type: "xavier"
    }
  }
}
layer {
  name: "relu2"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
```

```
layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  inner_product_param {
    num_output: 6
    weight_filler {
      type: "xavier"
    }
  }
}
layer {
  name: "accuracy"
  type: "Accuracy"
  bottom: "ip2"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip2"
  bottom: "label"
  top: "loss"
}
```

Solver Definition (*solver.prototxt*)

```
train_net: "C:\\caffe_myexamples\\cmsis_nn_data_1\\mynet_auto_train.prototxt"
test_net: "C:\\caffe_myexamples\\cmsis_nn_data_1\\mynet_auto_test.prototxt"
test_iter: 11
test_interval: 44
base_lr: 9.999999747378752e-05
display: 44
max_iter: 880
lr_policy: "fixed"
momentum: 0.8999999761581421
snapshot: 880
snapshot_prefix: "C:\\caffe_myexamples\\cmsis_nn_data_1\\snapsot\\"
```

```
solver_mode: CPU
momentum2: 0.9990000128746033
type: "Adam"
```

A.3. Data recording procedure

1. Have the subject read and sign the information sheet and consent form
2. Explain the subject about the different gestures to be recorded
3. Put on the bracelet on the participant's arm in a firm position
4. Record 5 repetitions of 6 actions each

B. Figures

B.1. LPCXpresso 4367 from NXP

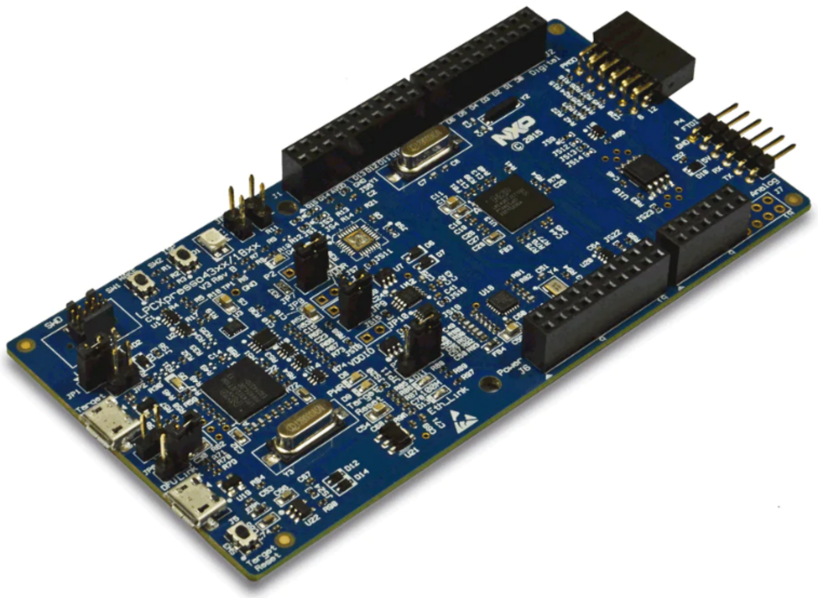


Figure B.1.: LPCXpresso 4367 from NXP [93]

This Cortex M4 based board was initially used as a starting point for implementing the regression model on the MCU. But after discovering its memory limitations, the Keil MCB 1800 board was bought.

B.2. Confusion Matrices

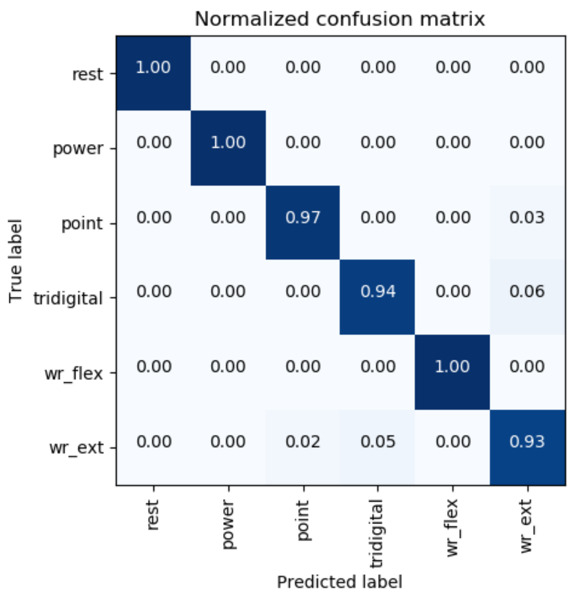


Figure B.2.: Confusion matrix for Myo dataset 1

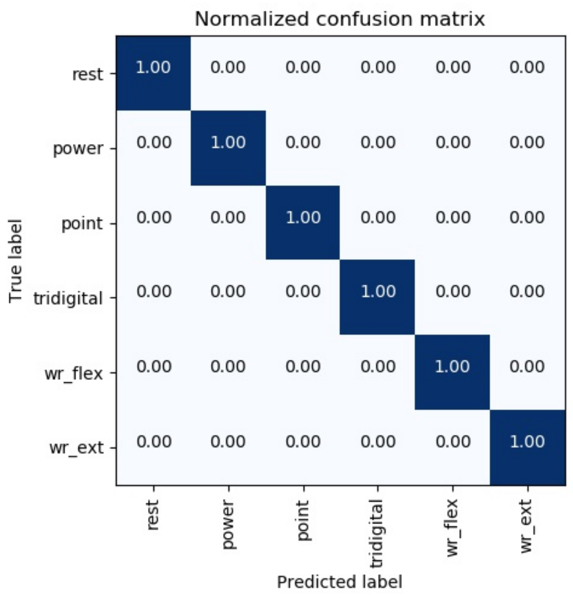


Figure B.3.: Confusion matrix for Myo dataset 2

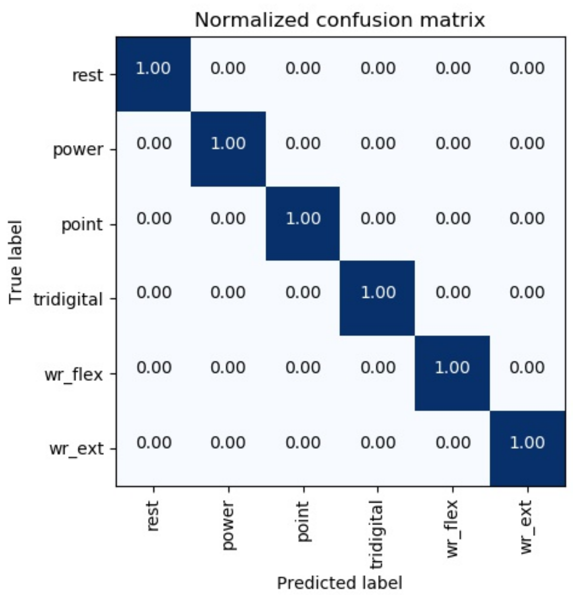


Figure B.4.: Confusion matrix for Myo dataset 2 with preprocessing

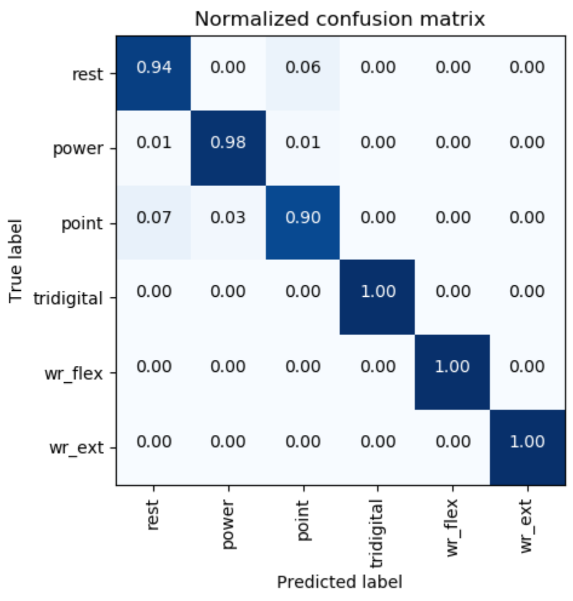


Figure B.5.: Confusion matrix for TB vectorized dataset with preprocessing

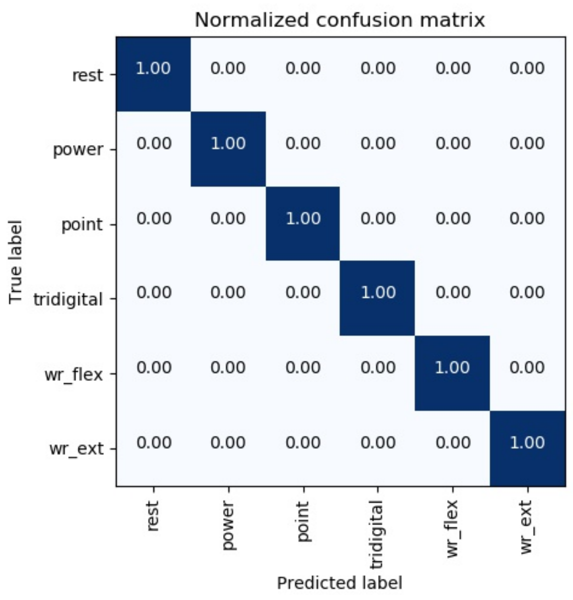


Figure B.6.: Confusion matrix for TB vectorized dataset with PCA and StandardScaler

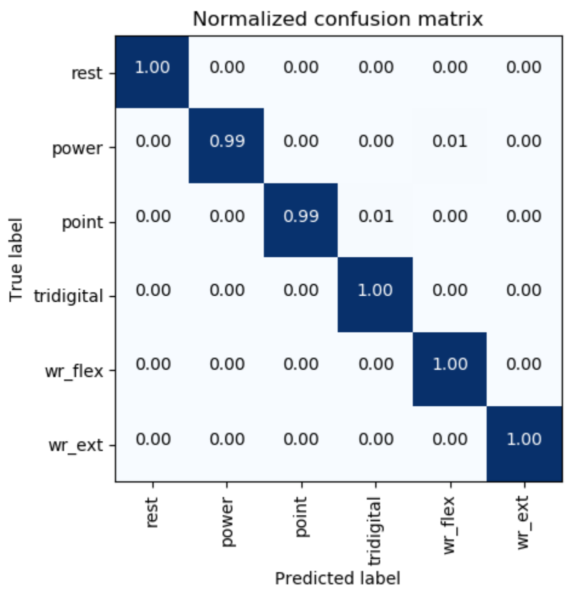


Figure B.7.: Generalised CNN Confusion Matrix with dataset 1

List of Figures

List of Tables

Bibliography

[1] W. H. Organization et al. *World report on disability 2011*. World Health Organization, 2011.

[2] K. Norton. “A brief history of prosthetics”. In: *InMotion* 17.7 (2007), pp. 11–3.

[3] X. Chen and Z. J. Wang. “Pattern recognition of number gestures based on a wireless surface EMG system”. In: *Biomedical Signal Processing and Control* 8.2 (2013), pp. 184–192.

[4] C. Savur and F. Sahin. “American Sign Language Recognition system by using surface EMG signal”. In: *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE. 2016, pp. 002872–002877.

[5] G. Kondo, R. Kato, H. Yokoi, and T. Arai. “Classification of individual finger motions hybridizing electromyogram in transient and converged states”. In: *2010 IEEE International Conference on Robotics and Automation*. IEEE. 2010, pp. 2909–2915.

[6] M. M. Hasan, A. Rahaman, M. F. Shuvo, M. A. S. Ovi, and M. M. Rahman. “Human hand gesture detection based on EMG signal using ANN”. In: *2014 International Conference on Informatics, Electronics & Vision (ICIEV)*. IEEE. 2014, pp. 1–5.

[7] A. B. Ajiboye and R. F. Weir. “A heuristic fuzzy logic approach to EMG pattern recognition for multifunctional prosthesis control”. In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 13.3 (2005), pp. 280–291.

[8] Y. Huang, K. B. Englehart, B. Hudgins, and A. D. Chan. “A Gaussian mixture model based classification scheme for myoelectric control of powered upper limb prostheses”. In: *IEEE Transactions on Biomedical Engineering* 52.11 (2005), pp. 1801–1811.

[9] H. J. Hermens, B. Freriks, C. Disselhorst-Klug, and G. Rau. “Development of recommendations for SEMG sensors and sensor placement procedures”. In: *Journal of Electromyography and Kinesiology* 10.5 (2000), pp. 361–374. ISSN: 1050-6411. DOI: `https://doi.org/10.1016/S1050-6411(00)00027-4`. URL: `http://www.sciencedirect.com/science/article/pii/S1050641100000274`.

[10] S. Mane, R. Kambli, F. Kazi, and N. M. Singh. “Hand Motion Recognition from Single Channel Surface EMG Using Wavelet & Artificial Neural Network”. In: 2015.

[11] P. Chrapka. “EMG controlled hand prosthesis: EMG classification system”. In: (2010).

[12] A. Hartwell, V. Kadirkamanathan, and S. R. Anderson. “Compact Deep Neural Networks for Computationally Efficient Gesture Classification From Electromyography Signals”. In: *2018 7th IEEE International Conference on Biomedical Robotics and Biomechatronics (Biorob)* (2018), pp. 891–896.

[13] A. Waris, I. K. Niazi, M. Jamil, K. Englehart, W. Jensen, and E. N. Kamavuako. "Multiday evaluation of techniques for EMG-based classification of hand motions". In: *IEEE journal of biomedical and health informatics* 23.4 (2018), pp. 1526–1534.

[14] M. Esponda and T. M. Howard. "Adaptive Grasp Control through Multi-Modal Interactions for Assistive Prosthetic Devices". In: *ArXiv* abs/1810.07899 (2018).

[15] P. Weiner, J. Starke, F. Hundhausen, J. Beil, and T. Asfour. "The KIT prosthetic hand: design and control". In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2018, pp. 3328–3334.

[16] C. Calderon-Cordova, C. Ramírez, V. Barros, P. A. Quezada-Sarmiento, and L. Barba-Guamán. "EMG signal patterns recognition based on feedforward Artificial Neural Network applied to robotic prosthesis myoelectric control". In: *2016 Future Technologies Conference (FTC)*. Dec. 2016, pp. 868–875. DOI: `10.1109/FTC.2016.7821705`.

[17] N. Rashid, J. Iqbal, A. Javed, M. I. Tiwana, and U. S. Khan. "Design of Embedded System for Multivariate Classification of Finger and Thumb Movements Using EEG Signals for Control of Upper Limb Prosthesis". In: *BioMed research international*. 2018.

[18] T. Teban, R. Precup, E. Voisan, T. E. A. de Oliveira, and E. M. Petriu. "Recurrent dynamic neural network model for myoelectric-based control of a prosthetic hand". In: *2016 Annual IEEE Systems Conference (SysCon)*. Apr. 2016, pp. 1–6. DOI: `10.1109/SYSCON.2016.7490531`.

[19] J. M. Hahne, M. A. Schweisfurth, M. Koppe, and D. Farina. "Simultaneous control of multiple functions of bionic hand prostheses: Performance and robustness in end users". In: *Science Robotics* 3.19 (2018), eaat3630.

[20] M. Arvetti, G. Gini, and M. Folgheraiter. "Classification of EMG signals through wavelet analysis and neural networks for controlling an active hand prosthesis". English. In: *2007 IEEE 10th International Conference on Rehabilitation Robotics, ICORR'07*. 2007, pp. 531–536. ISBN: 1424413206. DOI: `10.1109/ICORR.2007.4428476`.

[21] M. Gandolla, S. Ferrante, G. Ferrigno, D. Baldassini, F. Molteni, E. Guanziroli, M. Cotti Cottini, C. Seneci, and A. Pedrocchi. "Artificial neural network EMG classifier for functional hand grasp movements prediction". In: *Journal of International Medical Research* 45.6 (2017), pp. 1831–1847.

[22] X. Liu, J. Sacks, M. Zhang, A. G. Richardson, T. H. Lucas, and J. V. der Spiegel. "The Virtual Trackpad: An Electromyography-Based, Wireless, Real-Time, Low-Power, Embedded Hand-Gesture-Recognition System Using an Event-Driven Artificial Neural Network". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 64 (2017), pp. 1257–1261.

[23] S. A. Raurale. "Acquisition and processing real-time EMG signals for prosthesis active hand movements". In: *2014 International Conference on Green Computing Communication and Electrical Engineering (ICGCCEE)*. IEEE. 2014, pp. 1–6.

[24] S. D. Gupta, S. Kundu, R. Pandey, R. Ghosh, R. Bag, and A. Mallik. “Hand gesture recognition and classification by discriminant and principal component analysis using machine learning techniques”. In: *Hand* 1.9 (2012).

[25] P. Geethanjali. “Myoelectric control of prosthetic hands: state-of-the-art review”. In: *Medical Devices (Auckland, NZ)* 9 (2016), p. 247.

[26] Y. Wu, D. Jiang, J. Duan, X. Liu, R. Bayford, and A. Demosthenous. “Towards a high accuracy wearable hand gesture recognition system using EIT”. In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2018, pp. 1–4.

[27] Teach Me Anatomy. *Muscles in the Posterior Compartment of the Forearm*. `https://teachmeanatomy.info/upper-limb/muscles/posterior-forearm/`, Last accessed on 2020-3-15. 2020.

[28] Teach Me Anatomy. *Muscles in the Anterior Compartment of the Forearm*. `https://teachmeanatomy.info/upper-limb/muscles/anterior-forearm/`, Last accessed on 2020-3-15. 2020.

[29] R. W. Bohannon. “Reference values for extremity muscle strength obtained by hand-held dynamometry from adults aged 20 to 79 years”. In: *Archives of physical medicine and rehabilitation* 78.1 (1997), pp. 26–32.

[30] M. M. Schluessel, L. d. Anjos, and G. Kac. “Hand grip strength test and its use in nutritional assessment”. In: *Rev Nutr* 21.2 (2008), pp. 223–35.

[31] T. B. Costa, A. L. Neri, et al. “Indicators Of Physical Activity And Frailty In The Elderly: Data From The Fibra Study In Campinas, São Paulo State, Brazil [medidas De Atividade Fisica E Fragilidade Em Idosos: Dados Do Fibra Campinas, São Paulo, Brasil]”. In: *Cadernos de saude publica* (2011).

[32] C. A. Amaral, T. L. M. Amaral, G. T. R. Monteiro, M. T. L. Vasconcellos, and M. C. Portela. “Hand grip strength: Reference values for adults and elderly people of Rio Branco, Acre, Brazil”. In: *PloS one* 14.1 (2019), e0211452.

[33] E. Scheme, A. Fougner, Ø. Stavdahl, A. D. Chan, and K. Englehart. “Examining the adverse effects of limb position on pattern recognition based myoelectric control”. In: *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology*. IEEE. 2010, pp. 6337–6340.

[34] A. Fougner, E. Scheme, A. D. Chan, K. Englehart, and Ø. Stavdahl. “Resolving the limb position effect in myoelectric pattern recognition”. In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 19.6 (2011), pp. 644–651.

[35] M. R. Masters, R. J. Smith, A. B. Soares, and N. V. Thakor. “Towards better understanding and reducing the effect of limb position on myoelectric upper-limb prostheses”. In: *2014 36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. IEEE. 2014, pp. 2577–2580.

[36] Medium. *Unveiling the Final Design of the Myo Armband*. `https://medium.com/thalmic/unveiling-the-final-design-of-the-myo-armband-10576c0ae95b`, Last accessed on 2020-3-15. 2020.

[37] P. Visconti, F. Gaetani, G. Zappatore, and P. Primiceri. “Technical features and functionalities of Myo armband: An overview on related literature and advanced applications of myoelectric armbands mainly focused on arm prostheses”. In: *International Journal on Smart Sensing and Intelligent Systems* 11.1 (2018), pp. 1–25.

[38] I. Mendez, B. W. Hansen, C. M. Grabow, E. J. L. Smedegaard, N. B. Skogberg, X. J. Uth, A. Bruhn, B. Geng, and E. N. Kamavuako. “Evaluation of the Myo armband for the classification of hand motions”. In: *2017 International Conference on Rehabilitation Robotics (ICORR)*. IEEE. 2017, pp. 1211–1214.

[39] D. Yang, W. Yang, Q. Huang, and H. Liu. “Classification of multiple finger motions during dynamic upper limb movements”. In: *IEEE journal of biomedical and health informatics* 21.1 (2015), pp. 134–141.

[40] M.-C. V. Marina, H.-J. Hwang, S. Amsüss, J. M. Hahne, D. Farina, and K.-R. Müller. “Improving the Robustness of Myoelectric Pattern Recognition for Upper Limb Prostheses by Covariate Shift Adaptation”. In: ().

[41] J. G. Abreu, J. M. Teixeira, L. S. Figueiredo, and V. Teichrieb. “Evaluating sign language recognition using the myo armband”. In: *2016 XVIII Symposium on Virtual and Augmented Reality (SVR)*. IEEE. 2016, pp. 64–70.

[42] M. Sathiyanarayanan and S. Rajan. “MYO Armband for physiotherapy healthcare: A case study using gesture recognition application”. In: *2016 8th International Conference on Communication Systems and Networks (COMSNETS)*. IEEE. 2016, pp. 1–6.

[43] C. Castellini, R. Kõiva, C. Pasluosta, C. Viegas, and B. M. Eskofier. “Tactile myography: an off-line assessment of able-bodied subjects and one upper-limb amputee”. In: *Technologies* 6.2 (2018), p. 38.

[44] C. Castellini and R. Koiva. “Using a high spatial resolution tactile sensor for intention detection”. In: *2013 IEEE 13th International Conference on Rehabilitation Robotics (ICORR)*. IEEE. 2013, pp. 1–7.

[45] K. Weiß and H. Worn. “The working principle of resistive tactile sensor cells”. In: *IEEE International Conference Mechatronics and Automation, 2005*. Vol. 1. IEEE. 2005, pp. 471–476.

[46] R. Koiva, E. Riedenklau, C. Viegas, and C. Castellini. “Shape conformable high spatial resolution tactile bracelet for detecting hand and wrist activity”. In: *2015 IEEE International Conference on Rehabilitation Robotics (ICORR)*. IEEE. 2015, pp. 157–162.

[47] M. Connan, E. Ruiz Ramirez, B. Vodermayer, and C. Castellini. “Assessment of a wearable force-and electromyography device and comparison of the related signals for myocontrol”. In: *Frontiers in neurorobotics* 10 (2016), p. 17.

[48] Keil. *MCB1800*. `http://www.keil.com/mcb1800/`, Last accessed on 2020-3-18. 2019.

[49] Keil. *CMSIS DSP*. `http://www.keil.com/pack/doc/CMSIS/DSP/html/index.html`, Last accessed on 2020-3-18. 2019.

[50] L. Lai, N. Suda, and V. Chandra. "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus". In: *arXiv preprint arXiv:1801.06601* (2018).

[51] Keil. *CMSIS NN*. `http://www.keil.com/pack/doc/CMSIS_Dev/NN/html/index.html`, Last accessed on 2020-3-18. 2019.

[52] Intel. *Intel Neural Compute Stick 2*. `https://software.intel.com/en-us/articles/intel-neural-compute-stick-2-and-open-source-openvino-toolkit`, Last accessed on 2020-3-19. 2019.

[53] Intel. *Intel Neural Compute Stick 2 Product Brief*. `https://www.intel.com/content/dam/support/us/en/documents/boardsandkits/neural-compute-sticks/NCS2_Product-Brief-English.pdf`, Last accessed on 2020-3-19. 2019.

[54] Intel. *Deep Learning Inference*. `https://software.intel.com/en-us/openvino-toolkit/deep-learning-inference`, Last accessed on 2020-3-19. 2020.

[55] A. L. Samuel. "Some studies in machine learning using the game of checkers". In: *IBM Journal of research and development* 3.3 (1959), pp. 210–229.

[56] T. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997. ISBN: 9780071154673. URL: `https://books.google.de/books?id=EoYBngEACAAJ`.

[57] H. Jabbar and R. Z. Khan. "Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study)". In: *Computer Science, Communication and Instrumentation Devices* (2015).

[58] W. M. Van der Aalst, V. Rubin, H. Verbeek, B. F. van Dongen, E. Kindler, and C. W. Günther. "Process mining: a two-step approach to balance between underfitting and overfitting". In: *Software & Systems Modeling* 9.1 (2010), p. 87.

[59] Medium. *Intuitions on L1 and L2 Regularisation*. `https://towardsdatascience.com/intuitions-on-l1-and-l2-regularisation-235f2db4c261`, Last accessed on 2020-3-19. 2018.

[60] GeeksforGeeks. *Underfitting and Overfitting in Machine Learning*. `https://www.geeksforgeeks.org/underfitting-and-overfitting-in-machine-learning/`, Last accessed on 2020-3-19. 2020.

[61] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[62] A. Schneider, G. Hommel, and M. Blettner. "Linear regression analysis: part 14 of a series on evaluation of scientific publications". In: *Deutsches "A rzteblatt International* 107.44 (2010), p. 776.

[63] G. K. Uyanık and N. Güler. "A study on multiple linear regression analysis". In: *Procedia-Social and Behavioral Sciences* 106.1 (2013), pp. 234–240.

[64] P. Geladi and B. R. Kowalski. “Partial least-squares regression: a tutorial”. In: *Analytica chimica acta* 185 (1986), pp. 1–17.

[65] A. E. Hoerl and R. W. Kennard. “Ridge regression: Biased estimation for nonorthogonal problems”. In: *Technometrics* 12.1 (1970), pp. 55–67.

[66] A. Gijsberts, R. Bohra, D. Sierra González, A. Werner, M. Nowak, B. Caputo, M. A. Roa, and C. Castellini. “Stable myoelectric control of a hand prosthesis using non-linear incremental learning”. In: *Frontiers in neurorobotics* 8 (2014), p. 8.

[67] H. Avron, M. Kapralov, C. Musco, C. Musco, A. Velingker, and A. Zandieh. “Random Fourier features for kernel ridge regression: Approximation bounds and statistical guarantees”. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 253–262.

[68] Z. Li, J.-F. Ton, D. Oglic, and D. Sejdinovic. “Towards a unified analysis of random Fourier features”. In: *arXiv preprint arXiv:1806.09178* (2018).

[69] A. Rahimi and B. Recht. “Random features for large-scale kernel machines”. In: *Advances in neural information processing systems*. 2008, pp. 1177–1184.

[70] J. Runge and R. Zmeureanu. “Forecasting energy use in buildings using artificial neural networks: a review”. In: *Energies* 12.17 (2019), p. 3254.

[71] R. E. Uhrig. “Introduction to artificial neural networks”. In: *Proceedings of IECON’95-21st Annual Conference on IEEE Industrial Electronics*. Vol. 1. IEEE. 1995, pp. 33–37.

[72] A. K. Jain, J. Mao, and K. M. Mohiuddin. “Artificial neural networks: A tutorial”. In: *Computer* 29.3 (1996), pp. 31–44.

[73] S. Albawi, T. A. Mohammed, and S. Al-Zawi. “Understanding of a convolutional neural network”. In: *2017 International Conference on Engineering and Technology (ICET)*. IEEE. 2017, pp. 1–6.

[74] Medium. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`, Last accessed on 2020-3-19. 2018.

[75] M. Coşkun, A. Uçar, Ö. Yildirim, and Y. Demir. “Face recognition based on convolutional neural network”. In: *2017 International Conference on Modern Electrical and Energy Systems (MEES)*. IEEE. 2017, pp. 376–379.

[76] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[77] D. J. Sargent. “Comparison of artificial neural networks with other statistical approaches: results from medical data sets”. In: *Cancer: Interdisciplinary International Journal of the American Cancer Society* 91.S8 (2001), pp. 1636–1642.

[78] V. Sharma, S. Rai, and A. Dev. “A comprehensive study of artificial neural networks”. In: *International Journal of Advanced research in computer science and software engineering* 2.10 (2012).

[79] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[80] Intel. *Export Compliance Metrics for Intel Microprocessors*. `https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Core-Processors.pdf`, Last accessed on 2020-3-20. 2020.

[81] M. Fourment and M. R. Gillings. "A comparison of common programming languages used in bioinformatics". In: *BMC bioinformatics* 9.1 (2008), p. 82.

[82] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi. "GNU scientific library". In: *No. Release* 2 (1996).

[83] Z. Parveen and F. Nazish. "Performance comparison of most common high level programming languages". In: *Int. J. Comput. Acad. Res* 5.5 (2016), pp. 246–258.

[84] I. Strazzulla, M. Nowak, M. Controzzi, C. Cipriani, and C. Castellini. "Online bimanual manipulation using surface electromyography and incremental learning". In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 25.3 (2016), pp. 227–234.

[85] S. Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[86] D. P. Kingma and J. Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[87] F. Chollet. *Deep learning with Python*. New York, NY: Manning Publications, 2018. URL: `https://cds.cern.ch/record/2301910`.

[88] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. "Caffe: Convolutional architecture for fast feature embedding". In: *Proceedings of the 22nd ACM international conference on Multimedia*. 2014, pp. 675–678.

[89] ARM. *CMSIS-NN CIFAR10 Example*. `https://github.com/ARM-software/ML-examples/tree/master/cmsisnn-cifar10`, Last accessed on 2020-3-20. 2020.

[90] C. Castellini, P. Artemiadis, M. Wininger, A. Ajoudani, M. Alimusaj, A. Bicchi, B. Caputo, W. Craelius, S. Dosen, K. Englehart, et al. "Proceedings of the first workshop on peripheral machine interfaces: Going beyond traditional surface electromyography". In: *Frontiers in neurorobotics* 8 (2014), p. 22.

[91] J. Kocić, N. Jovičić, and V. Drndarević. "An end-to-end deep neural network for autonomous driving designed for embedded automotive platforms". In: *Sensors* 19.9 (2019), p. 2064.

[92] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu. "A survey of deep learning techniques for autonomous driving". In: *Journal of Field Robotics* (2019).

[93] NXP. *OM13088: LPCXpresso4367 Development Board*. `https://www.nxp.com/design/designs/lpcxpresso4367-development-board:OM13088`, Last accessed on 2020-3-20. 2020.