# Implementations and evaluation of machine learning algorithms on a microcontroller unit for myoelectric prosthesis control

Jonathan Benitez
2023

**Master's Thesis in Electrical Measurements**

Supervisor: Christian Antfolk
Examiner: Johan Nilsson


LUND UNIVERSITY

Faculty of Engineering LTH
Department of Biomedical Engineering

# Abstract

Using a microcontroller unit to implement different machine learning algorithms for myoelectric prosthesis control is currently feasible. Still there are hardware and timing constraints that need to be accounted for. This master thesis presents results from automatically generated Arduino code for some Neural Networks, Convolution Neural Networks and linear machine learning models that was implemented on a Teensy 4.1 board to see where these constraints were on this specific board. The results show promise that simpler algorithms can be generated for 50 classes with an accuracy of around 40-50%, but more complex algorithms usually run into memory constraints or timing constraints. The results also show that different algorithms are more accurate for different subjects in the used NinaPro database [1]. This suggests that configuring the prosthesis on a patient basis, like the one implemented, is useful.

# Acknowledgements

I want to thank Christian Antfolk for guiding me through this thesis whenever I needed it. I also want to thank Johan Nilsson for helping with all the administration. I want to thank CSN for helping me with my economic problems and I also want to thank William Marnfeldt for giving me helpful tips about how to solve different problems. I also want to thank Emma Jönsson for helping me with simple machine learning tips like knowing when a model is overfitted.

# Contents

# 1

# Introduction

Around 94 000 people in the European community have an upper limb amputation [2]. In the USA, an estimate of 158 000 undergo amputation, with this number increasing over time [3]. The leading cause of amputation in a countries varies. Countries with recent history in warfare have higher trauma related causes for amputation while for countries like Japan, USA, Denmark have a higher disease related causes from tumours, diabetes, etc. [4].

Upper limb amputation affects day to day life and can be debilitating for the person affected [5]. In trying to rehabilitate the consequences of the amputation, different kinds of prosthetic hands are currently used[4]. One type of prostheses, which aims to resemble the human hand is the myoelectric prosthesis. Myoelectric prosthesis can be controlled by measuring the electric activity in muscle contractions, and to measure this activity there are two methods: intramuscular electromyography (iEMG) and surface electromyography (sEMG). iEMG sensors give more accurate readings while sEMG sensors are used as they are non-invasive [2]. Despite the benefits of using a prosthesis, most upper limb amputees do not use them. Usually people with amputations from the elbow upwards tended to use prosthesis less. The documented number of upper limb amputees using prosthesis vary between 27 to 56 percent. [3].

A person with an amputated hand still has muscles that control the hand motions throughout the arm which means that by measuring these muscles movements one can differentiate between which motion the hand would make [6]. Machine learning algorithms (MLAs) is commonly used evaluate which muscle motions generate what hand motions. [7] By training with a data set of which muscle readings corresponds to what motion one can implement the finished trained MLA on a smaller device such as a microcontroller unit (MCU) to then do the calculation with the muscle readings. By doing so, a cheaper and more energy efficient system can be implemented on a portable system [8]. This of course also means that the algorithm cannot be too computational heavy as the MCU might not be able to handle all the processing in real time.

## 1.1 Aim of Thesis

The aim of this thesis is to be able to implement different kinds of MLA to a smaller MCU from the model given after training. The goal is to see if it is feasible to design a configurable system that could handle different kinds of input sizes, sampling rates and window sizes. It will be investigated if an autonomous system can be implemented that can handle different MLA algorithms, convert them to C/C++/arduino code, and then look over what the limits of this systems are. Once this is done different machine learning algorithms will be implemented using this program and evaluated on complexity of implementation and usage on the MCU.

# 2

# Background

For this thesis, the background will be split up in different sections. The first part will be about Electromyography, afterwards Feature extraction will be brought up, and lastly different machine learning algorithms will be explained.

## 2.1 Electromyography

Electromyography (EMG) is the recording of the electrical activity from a muscle. In the medical field EMG can be used to detect and give additional diagnostics to different muscular diseases [9]. To get the EMG recording, either invasive, intramuscular EMG (iEMG), or non-invasive, surface EMG (sEMG), electrodes can be used [10]. EMG signals have an amplitude from 0 to 10 mV. The frequency of the signal is from 0 to 1000 Hz, but the most dominant energy frequencies are concentrated between 20 to 500 Hz [11]. Multiple channel sEMG recordings have been shown to be able to control myoelectric prosthesis with a high accuracy using a machine learning based control system [12].

### 2.1.1 Windowing of sEMG data

Windowing is used to analyze the EMG recording from smaller chunks. By doing so the characteristics of the window can be further analysed and be used as inputs for the control schemes for myoelectric prosthesis [13]. There are two types of windowing: disjointed and overlap windowing. Disjointed windowing are only characterized by a window size, while overlap windowing has a window size and an overlap size [13][14]. This is represented in figure 2.1.

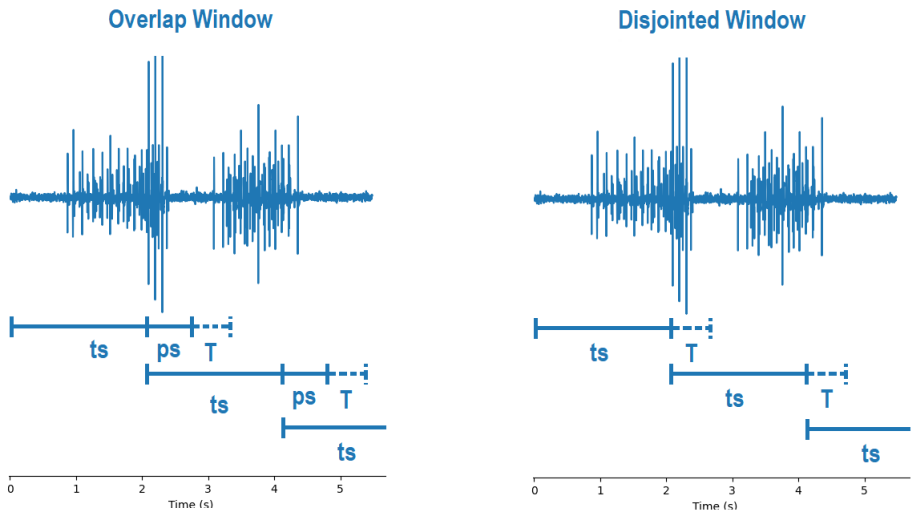


**Figure 2.1** Overlap and disjointed window where ts is window size, ps is overlap size and T is processing delay.

For the disjointed window, the optimal size ranges from 200-300 ms while for the overlap window ranges from 225-300 ms with an overlap size of 10%-30% of the window length [13]. For time domain features the window size has less significant effect on the EMG parameters than the frequency domain features [15]. For larger window sizes, the time required to process the EMG recording increases. This time delay can make the prosthesis feel unresponsive if the delay is to high, for 90th percentile of users this time delay is only excessive above around 100 - 125 ms of delay[14].

To note, the process delay $T$ is constrained by the window size $ts$ and the overlap size $ps$. Thus $T$ cannot be too large or the system will not function, this constraint is described in equation 2.1.

$$ts - ps > T \tag{2.1}$$

## 2.2 Feature extraction

Feature extraction are different algorithms that extract the hidden information in sEMG recordings thus removing the unnecessary information. The different feature extraction algorithms can be grouped up into 3 domains: time domain, frequency domain, and time-frequency or time-scale representation. Of these time domain feature extractions are usually the easiest to implemented as there is no need for any transformation to implement these algorithms. [16]

All the following time domain feature extraction algorithms were taken from [16]. In the following equations, N is the number of samples of the window size, $x_i$ is the value of the sEMG recording of sample i.

### 2.2.1 Integrated EMG

Integrated EMG (IEMG) is the sum of the absolute values of the EMG signal, described in equation 2.2.

$$IEMG = \sum_{i=1}^{N} |x_i| \tag{2.2}$$

### 2.2.2 Mean absolute value

Mean absolute value (MAV) is a feature that takes the average of the absolute value of the EMG signal, this is described in equation 2.3.

$$MAV = \frac{1}{N} \sum_{i=1}^{N} |x_i| \tag{2.3}$$

### 2.2.3 Modified mean absolute value type 1

Modified mean absolute value type 1 (MAV1) is similar to MAV but uses a weight function $w_i$, this is described in equation 2.4.

$$MAV1 = \frac{1}{N} \sum_{i=1}^{N} w_i |x_i|, \quad w_i = \begin{cases} 1, & 0.25N \leq i \leq 0.75N \\ 0.5, & i < 0.25N \text{ or } i > 0.75N \end{cases} \tag{2.4}$$

### 2.2.4 Modified mean absolute value type 2

Modified mean absolute value type 2 (MAV2) is similar to MAV1 but uses a continues functions for the weight function $w_i$, this is described in equation 2.5.

$$MAV2 = \frac{1}{N}\sum_{i=1}^{N} w_i|x_i|, \quad w_i = \begin{cases} 1, & 0.25N \leq i \leq 0.75N \\ \frac{4i}{N}, & i < 0.25N \\ \frac{4(i-N)}{N}, & i > 0.75N \end{cases} \tag{2.5}$$

### 2.2.5 Variance of EMG

Variance of EMG (VAR) is the average of the squared values from the EMG signals. The feature is similar to variance, but as the mean value of the EMG signals is usually very close to zero, it can be negligible, this is described in equation 2.6

$$VAR = \frac{1}{N-1}\sum_{i=1}^{N} x_i^2 \tag{2.6}$$

### 2.2.6 Root mean square

Root mean square (RMS) is the square root of the average squared EMG signal amplitude, this is described in equation 2.7

$$RMS = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2} \tag{2.7}$$

### 2.2.7 Waveform length

Waveform length (WL) is the length of the waveform over the window, this is described in equation 2.8

$$WL = \sum_{i=1}^{N-1} |x_{i+1} - x_i| \tag{2.8}$$

### 2.2.8 Average amplitude change

Average amplitude change (AAC) is similar to WL, but it is averaged out, this is described in equation 2.9

$$AAC = \frac{1}{N}\sum_{i=1}^{N-1} |x_{i+1} - x_i| \tag{2.9}$$

### 2.2.9 Simple square integral

Simple square integral (SSI) is the summed square of the EMG signals, this is described in equation 2.10

$$SSI = \sum_{i=1}^{N} x_i^2 \tag{2.10}$$

### 2.2.10 Temporal moment 3rd, 4th and 5th

Temporal moment 3rd, 4th and 5th are very similar to MAV and VAR, but with a higher order. Temporal moment 3rd is described in equation 2.11

$$TM3 = |\frac{1}{N}\sum_{i=1}^{N} x_i^3| \tag{2.11}$$

Temporal moment 4 (TM4) is described in equation 2.12

$$TM4 = \frac{1}{N}\sum_{i=1}^{N} x_i^4 \tag{2.12}$$

Temporal moment 5 (TM5) is described in equation 2.13

$$TM5 = |\frac{1}{N}\sum_{i=1}^{N} x_i^5| \tag{2.13}$$

## 2.3 Machine learning algorithms

There are many different Machine learning algorithms (MLAs) that can be implemented. This section will be split up into Linear models, and Artificial Neural Networks.

### 2.3.1 Linear models

Linear models are models where the output $y$ is a linear combination of input $x$. This is done by fitting the values of the weights $w = w_1, ..., w_N$ and the intercept $w_0$. For models with binary classification the output $y$ can be calculated according to equation 2.14. From the output $y$ the prediction $\hat{y}$ can be calculated according to the equation 2.15 [17].

$$y(w,x) = w_0 + w_1x_1 + ... + w_Nx_N \tag{2.14}$$

$$\hat{y} = \begin{cases} 0, & y \leq 0 \\ 1, & y > 0 \end{cases} \tag{2.15}$$

There are also linear models that can predict multiple different classes. The output matrix $Y$ is thus dependent on the weight matrix $W$, one dimensional input matrix $X$ and intercept matrix $B$. This is described in equation 2.16. The predicted output $\hat{Y}$ is index of the largest value of $Y$. This is described in equation 2.17 [17].

$$Y(X) = XW + B \tag{2.16}$$

$$\hat{Y} = argmax(Y) \tag{2.17}$$

### 2.3.2 Artificial Neural Networks

Artificial Neural Network (ANN) artificially models how neurons work. One of these models is the perceptron. The perceptron is a linear model, which can be described by equation 2.14. Multiple perceptron can be connected to each other making a feedforward neural network. Feedforward neural networks are neural networks that have three parts: input layer, hidden layers, and output layers. A figure of a feedforward neural network can be seen in figure 2.2 [18][19].

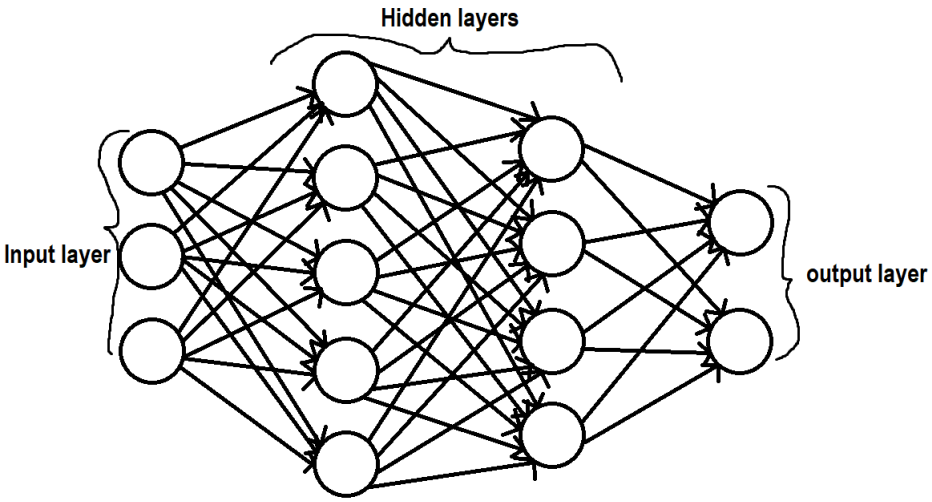


**Figure 2.2** An example of a feedforward neural network, includes three parts: input layer, hidden layers and an output layer.

#### 2.3.2.1 Dense Neural Networks

The dense layer is a linear model that is used to model one layer of perceptron. An ANN using only Dense layers is usually called Dense Neural Networks (DNN), when they have a lot of dense layers it is also considered "deep" [20]. To calculate the output of the dense layer $\hat{Y}$, the weights $W$, bias $b$, input $X$ is added together as in equation 2.16 and the result is the input to an activation function $\sigma$. The activation functions will be more thoroughly specified under section 2.3.2.6. The dense layer is fully described in equation 2.18 [21].

$$\hat{Y} = \sigma(XW + b) \tag{2.18}$$

For the dense layer, the amount of perceptron or "dimension" of the output layer $\hat{Y}$ can be specified between layers. Afterwards the activation function of each dense layer can also be chosen.

#### 2.3.2.2 Convolution Neural Networks

Convolution Neural Networks (CNN) is a type of Artificial Neural Network that uses convolution. CNN are mainly used in image processing but can also be used in processing raw sEMG data by modelling the data as an image [7][22]. Compared to DNN, Convolution Neural Networks can work with inputs of 2 to 3 dimensions. A visual example of Convolution, or more specifically 2-dimensional convolution, that performs a convolution operation on an input matrix can be seen in figure 2.3. This is done following equation 2.19. Where $w$ is the weight, $b$ is the bias, $x_{j,k}$ is the input value at position $(j,k)$, $n_1$ and $n_2$ are the kernel sizes of the convolution layer, $\sigma$ the activation function and finally $\hat{y}_{s,t}$ is the output in position $(s,t)$ and is dependent on the kernel size of the weight matrix and position $(j,k)$ of the input matrix [19].

$$\hat{y}_{s,t}(x_{j,k}) = \sigma(b + \sum_{l=0}^{n_1} \sum_{m=0}^{n_2} W_{l,m} x_{j+l,k+m}) \tag{2.19}$$

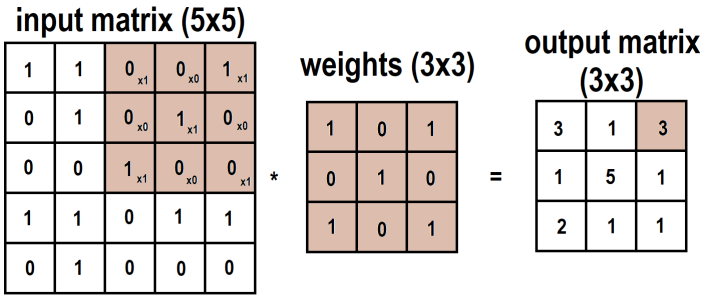


**Figure 2.3** Example of convolution, with no bias, the kernel is the weight matrix; in brown is a highlighted example of the element wise calculation.

There are also different parameters that can be changed in a convolution layer. Firstly, the kernel size of the convolution layer can be specified. Thus, changing the dimensions of the weight matrix. The amount of output layers can be increased or decreased. Convolution layers also have different activation functions [21]. These are presented in section 2.3.2.6.

#### 2.3.2.3 Pooling layers

Pooling layers are usually used in CNN to decrease the spatial dimension of the feature maps and therefore also decreases computational costs. There are different kinds of Pooling layers, average and max pooling are the most common ones. Average pooling takes the average of the pooled values, which is shown in figure 2.4, while max pooling takes the highest number in the pooled values, shown in figure 2.5. [23]

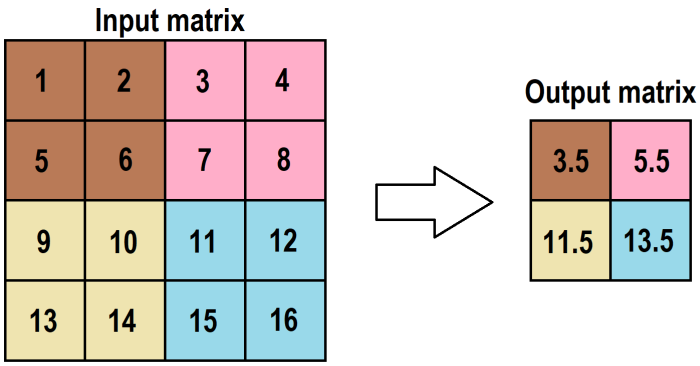


**Figure 2.4** Example of Average pooling, with a 4x4 input matrix and a pooling layer of size 2x2.

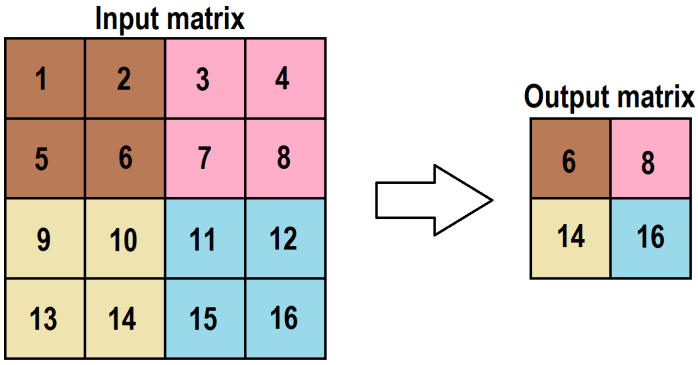


**Figure 2.5** Example of Max pooling, with a 4x4 input matrix and a pooling layer of size 2x2.

#### 2.3.2.4 Dropout

Dropout is a layer used for training. Dropout randomly "deletes" neurons of a layer by setting their weight value to zero. Dropout is used in training to not make the MLA overfitted by relying on the population of neurons instead of the activity of other specific neurons [21][24].

#### 2.3.2.5 Flatten

The flatten layer is used to make a multi-dimensional matrix into a 1 dimensional matrix. This is most often used to connect a CNN with a dense layer for the output prediction. It can also be used just to connect a CNN with a DNN [21].

#### 2.3.2.6 Activation functions

There are many different kinds of activation functions, and different activation functions are used in different ways. Two of these are the ReLU activation function and the linear activation function.

The linear activation is the easiest to understand, it returns the input value of the activation function. The linear activation function is usually the default one when no activation function is specified. The equation for Linear activation function can be seen in equation 2.20 [19][21].

$$f(x) = x \tag{2.20}$$

ReLU is also called Rectified Linear Unit. The ReLU activation function gives out the input if it is larger than zero and if it is lower than zero it instead will output zero. ReLU can also have a set zero point, but in this case ReLU will be defined as is described in equation 2.21 with a fixed zero point [19][21].

$$f(x) = \begin{cases} x, & x > 0 \\ 0, & x <= 0 \end{cases} \tag{2.21}$$

# 3

# Methodology

The methodology was split up into three sections: Resources, Implementation, and Tests.

## 3.1 Resources

There were many different software resources and hardware resources that were used during this project. From databases to microcontroller units. For this specific implementation, the different resources that were used will be described below.

### 3.1.1 Scikit-learn

Scikit-learn also called sklearn is a python library for machine learning. Scikit-learn includes different kinds of trainable models and different ways to train these models. Sklearn have many different linear models than can be implemented, from models that can only perform binary classification to multi label classification. One of these linear models that can do multi label classification with a high degree of accuracy is Linear Discriminant Analysis (LDA). LDA was used to assess most of the feature extraction algorithms [25].

To be able to assess the linear model, or other MLA, sklearn has a function that checks the accuracy and balanced accuracy of the model. The accuracy is the percentage of correct predictions against total predictions. Balanced accuracy is used if there is an imbalance in the amount of datasets for the different classes. If there are more datasets for one motion than another, the balanced accuracy will take this into account [26][25].

### 3.1.2 TensorFlow Keras

TensorFlow Keras, or just Keras is a python library that trains and implements different MLAs. It is most well known for being able to implement feedforward neural networks and convolution neural networks. Keras can implement different layers such as Dense, Convolution, Pooling layers, Dropout, Flatten and has different activation function for these layers [21].

During the project, different kinds of Keras models were tested and implemented in the hardware. Keras was mainly used to generate and train different kinds of DNN and CNN models. Once these models were generated, the Keras models were also used to crosscheck the validity of the generated model and to assert that they performed equally.

### 3.1.3 NinaPro Database

The sEMG data used was taken from NinaPro. NinaPro is a public database used to foster the research in robotic and prosthesis hands. NinaPro has many different databases, but the one used was db2, which has 50 recorded hand motions sampled with 12 different sEMG channels with a sampling frequency of $2kHz$. Each hand motion has 6 repetitions [1]. An example of stimulus data for 18 hand motions and EMG data from one channel for person 5 can be seen in figure 3.1.

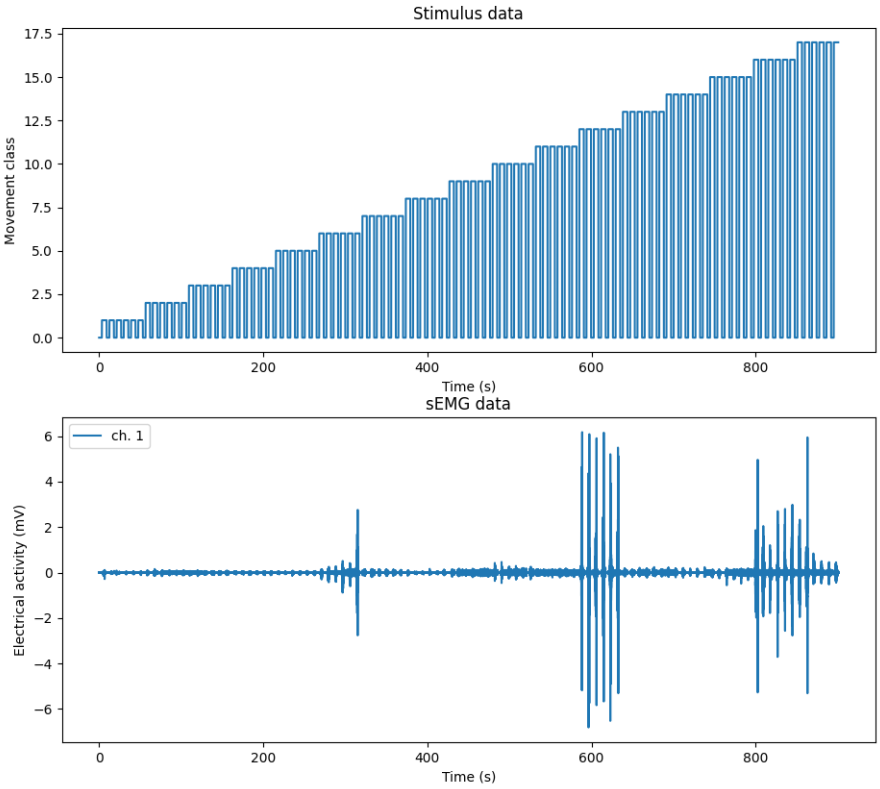


**Figure 3.1** Stimulus data (upper plot) and EMG data (lower plot) of test person 5.

The different values from the stimulus data correspond to different hand motions. So value 1 corresponds to hand motion 1. This is the case for all motions except stimulus value 0, which in this case is the rest hand motion, also motion 50. All the different motions in the database can be seen in figure 3.2.

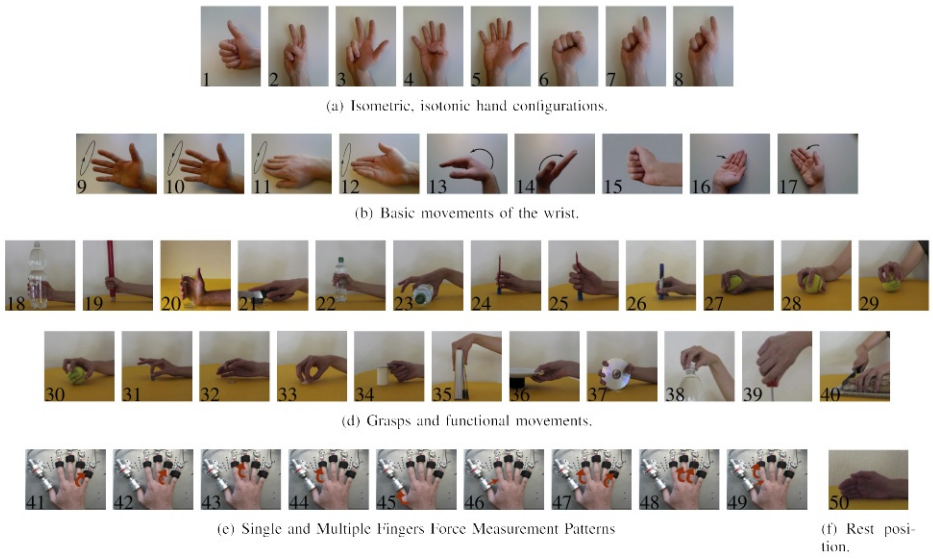


**Figure 3.2** All the different hand motions in NinaPro database 2 [1].

### 3.1.4 Teensy 4.1

Teensy 4.1 is a microcontroller that can use the Arduino IDE with teensy add-ons. Teensy 4.1 board uses an ARM Cortex-M7 CPU with a clock speed of 600 MHz. It can perform floating point arithmetic with 64 or 32 bits and has a 7936 Kbytes of Flash memory. For all the specifications of the Teensy 4.1 board see figure 3.3 [27].

| Feature | Teensy 4.1 | Units |
|---|---|---|
| Processor | IMXRT1062DVJ6 | |
| Core | Cortex-M7 | |
| FPU | 32 & 64 | bits |
| Rated Speed | 600 | MHz |
| Overclockable | 912 | MHz |
| Flash Memory | 7936 | kbytes |
| Bandwidth | 66 | Mbytes/sec |
| Cache | 65536 | Bytes |
| RAM | 1024 | kbytes |
| EEPROM | 4284 (emu) | bytes |
| Direct Memory Access | 32 | Channels |
| Digital I/O | 55 | Pins |
| Breadboard I/O | 42 | Pins |
| Voltage Output | 3.3V | Volts |
| Current Output | 10mA | milliAmps |
| Voltage Input | 3.3V Only | Volts |
| Interrupts | 55 | Pins |
| Analog Input | 18 | Pins |
| Converters | 2 | |
| Usable Resolution | 10 | Bits |
| Prog Gain Amp | - | |
| Touch Sensing | - | Pins |
| Comparators | 4 | |
| Analog Output | - | Pins |
| DAC Resolution | - | Bits |
| Timers | 49 Total | |
| PWM, 32 bit | 3 | |
| PWM, 16 bit | 32 | |
| PWM, 8-10 bit | - | |
| Total PWM Outputs | 31 | Pins |
| PDB Type | - | |
| CMT Type | - | |
| Quadrature Enc | 4 | |
| LPTMR Type | - | |
| PIT/Interval | 4 | |
| IEEE1588 | 4 | |
| Systick | 1 | |
| RTC | 1 | |
| Communication | | |
| USB | 2 | |
| Serial | 8 | |
| With FIFOs | 8 | |
| High Res Baud | - | |
| SPI | 2 | |
| With FIFOs | 2 | |
| I2C | 3 | |
| CAN Bus | 3 | |
| With CAN-FD | 1 | |
| Digital Audio In | 5* | stereo pins |
| Digital Audio Out | 5* | stereo pins |
| S/PDIF Input | 1 | |
| S/PDIF Output | 1 | |
| MQS Output | 1 | |
| SD Card | 1 | |
| Ethernet | 1 | uu |

**Figure 3.3** Hardware specifications for Teensy 4.1 board [27].

The Teensy 4.1 board was used as a microcontroller were all the models were generated to. To test that the models were the same, a python script sent the sEMG data to the Teensy 4.1 board through Serial communication. Afterwards, the Teensy 4.1 board would send back the predicted results and the values of the last output layer as cross validation.

## 3.2 Implementation

The code for the Teensy 4.1 board was implemented as a function with 4 different states, excluding specific debug and printing states. The four different states implemented were *__init__*, where it waits until the input memory buffer is full. The *FeatureExtraction* state, where it performs the feature extraction for the sEMG data once the input memory buffer is full. The feature extracted data is then put in another memory buffer called feature extracted data. *MLA* state, where it performs the different implemented Machine Learning Algorithms on the feature extracted data with the weights and bias of the model and outputs the data in an MLA memory buffer. The *classification* state is where it classifies the results and outputs the prediction for the given sEMG data. The state diagram and the block diagram for the given implementation is described in figure 3.4.

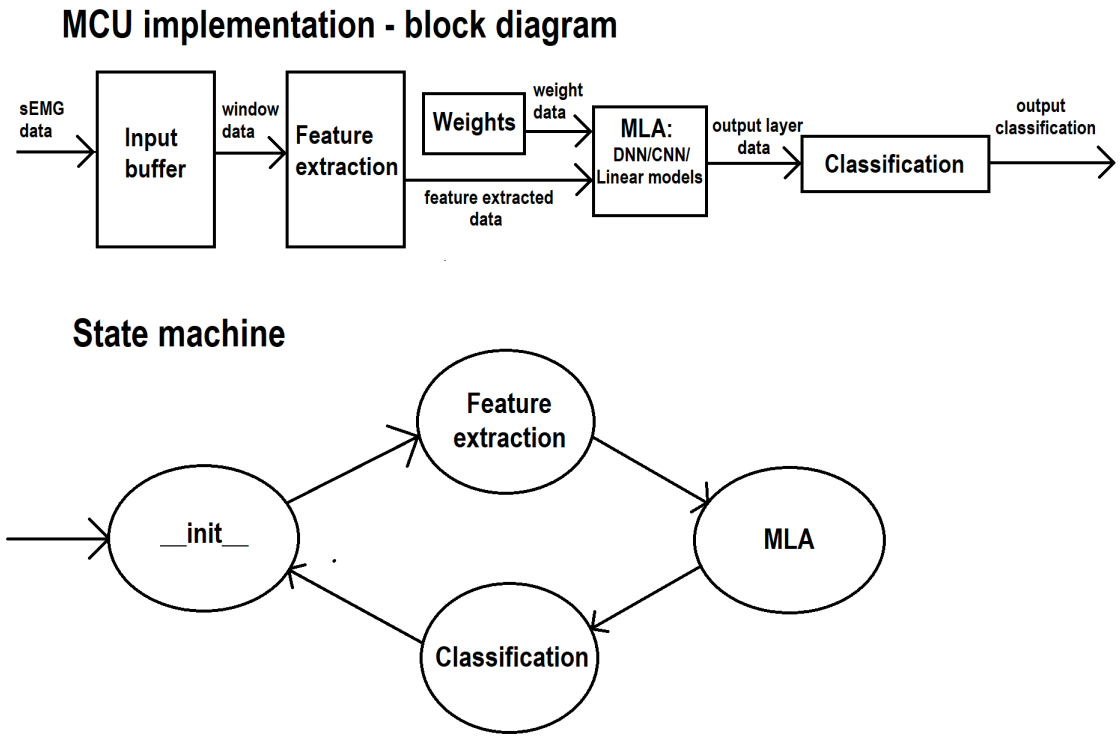


**Figure 3.4** Block diagram and state-machine for the Teensy implemented code.

To generate the Arduino code for a given machine learning model and to validate that the generated model worked as intended, a python model was implemented. The python model consisted of different scripts that worked similarly to the teensy 4.1 model. The first script was *process_data.py* where the Ninapro data was processed by extracting the stimulus and sEMG data. The processed sEMG data from the first script was then passed through the second script $feature\_extraction.py$ where the relevant features were processed, which was then used as the training, testing or validation data for either the sklearn linear models or the Keras

models with the stimulus data being the classifier. From the model and the called features, the teensy code can be generated with *mla_model_parser.py*. The *mla_model_parser.py* script uses some helper scripts, one for the called features called *feature_parser.py* and one for the MLA model called *layer_parser.py*. The teensy code can then be compiled into the Teensy 4.1 board. All of this is described in figure 3.5.

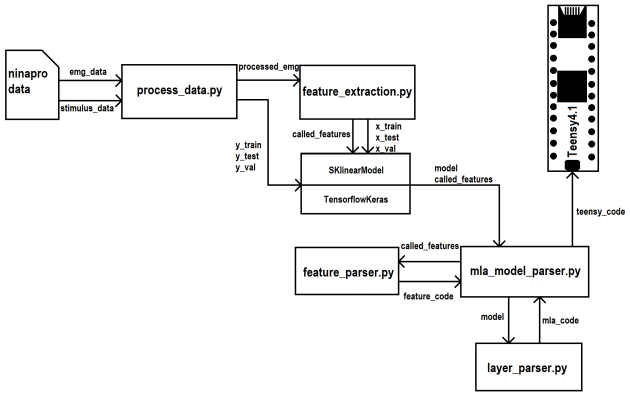


**Figure 3.5** A top down chart of the different scripts and the data passed through these scripts.

The first step in *mla_model_parser.py* is to read the sEMG data. To be able to read the sEMG data with disjointed windows a circular memory buffer was implemented in *mla_model_parser.py*. The circular memory buffer can be implemented by having a memory array that circles back to the first element when reaching the end of the memory array. By doing so, one can start reading new values from the end of the last window, and do so for samples N, where $N =$window size$-$overlap size. A representation of the circular memory buffer can be seen in figure 3.6.

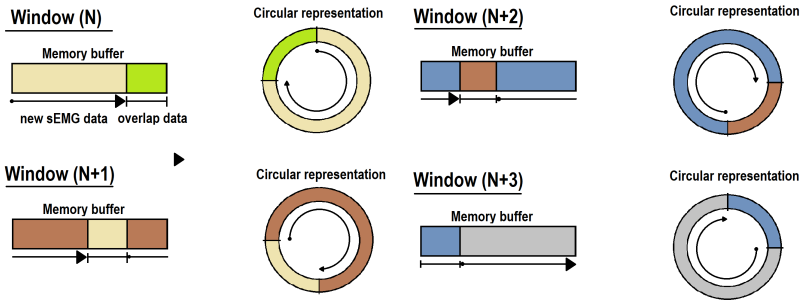


**Figure 3.6** Example of a circular memory buffer where the overlap size is $1/4$ of the window size, for window N where $N \neq 0$.

Afterwards the Feature extraction was implemented. Here all the equations from section 2.2 were implemented in both python code, in the *feature_extractions.py* script, and in C++ code for the Teensy code generation in *feature_parser.py* script. The next step was to make the Arduino code for the different models, these were implemented according to the description in section 2.3 and were implemented in *layer_parser.py*. The output classification, which is the last step, was implemented according to equation 2.17 for binary classification and according to equation 2.20 for multiclass classification. All floating-point variables in the Teensy 4.1 code used for memory or temporary variables by the Teensy 4.1 board were implemented by the datatype float32.

## 3.3 Tests

For testing, another script called *arduino_communication.py* was implemented that could send and read data from the Teensy 4.1 board. Different kinds of tests were devised, mostly looking at how different parameters change the execution time on the Teensy 4.1 board. This was done to see how the execution time scales with the increase of different parameters. If the execution time is too large, it will not hold the timing constraint. Some tests also looked at how the accuracy of the model changed with the different feature extraction methods used. During all the tests, 500 test windows were sent to the Teensy 4.1 board and the output layer were compared with the python model to assert that they predicted the same value. The execution time was measured from when the Teensy 4.1 board exited the *__init__* state until it had made a prediction. Even though there are some variances in execution time between runs, the highest execution time for the 500 windows were used in the results. All the tests will be described below.

### 3.3.1 Linear Discriminant Analysis test: window size

Firstly a test was performed looking how different feature extraction algorithms with varying window sizes affects the execution time and the accuracy of a Linear Model trained using Linear Discriminant Analysis. The tests were done by using a disjointed window and the window size was increased in steps of 100 samples, from 100 to 800 samples. The overlap size was set to $1/4$ of the window size. All the implemented feature extraction algorithms were tested. The split for the dataset was set so 4 repetitions were used as training data, and 2 repetitions as testing data. For rest, or class 0, 66% of the repetitions was used as training data and the 34% was used as testing data. The tests were performed on person 1 in the dataset, with the first 40 movement classification and rest, and had all the available channels used. The accuracy, balanced accuracy and execution time were measured.

### 3.3.2 Linear Discriminant Analysis test: accuracy for all persons

Another Linear Discriminant Analysis test was performed looking at how the different feature extraction algorithms affected the accuracy for each person in the dataset. Again, a disjointed window was used, the window size and overlap size were set to 450 and 120 samples respectively. For the first 40 classes and rest were used in the classifications, with all available channels used. The splits were set the same as the previous test. The accuracy and balanced accuracy were measured for all 40 persons in the dataset.

### 3.3.3 Convolution Neural Network test: comparing results between the two models

To see that the code could successfully generate the same model in the Teensy 4.1 board as in the python model, a CNN test model was used. In the input the test model used all 12 channels and all 12 features. The MLA consisted of 2 layers of convolution, one with ReLU activation and the other with linear activation. Afterwards the MLA had a max and average pooling layer, which was then followed by a flatten layer and two dense layers, one with ReLU the other with linear activation. One of the dense layers also had a dropout layer. Only the first 10 classifications were used. The output for these models was compared, and some values will be shown as an example. The model is described in table 3.1.

**Table 3.1** Summary of the layers in the CNN for Teensy 4.1 model and python model comparision

| Layer | Output Shape | Number of Parameters |
|---|---|---|
| Conv2d with ReLU | (11, 11, 25) | 125 |
| average pooling 2d | (5, 5, 25) | 0 |
| Conv2d | (3, 3, 25) | 5650 |
| Max pooling 2d | (1, 1, 25) | 0 |
| Flatten | 25 | 0 |
| Dense with ReLU | 64 | 64 |
| Dropout | 64 | 0 |
| Dense | 11 | 715 |

### 3.3.4 Dense Neural Network test: output dimension

To see how the dense layer dimensions affects execution time a two layer DNN was implemented. The first dense layer had a varying amount of output dimension starting at 64 and incremented with 64 for each step. All available feature extraction algorithms were used. The window size was set to 225 ms with an overlap size of 62.5 ms. The accuracy of this model using all 50 classifications and the execution time was measured. The different layers, the output shape and the number of parameters is shown in table 3.2.

**Table 3.2** Summary of the layers used in the DNN, where N went from 1 to 8

| Layer | Output Shape | Number of Parameters |
|---|---|---|
| Flatten | 144 | 0 |
| Dense with ReLU | 64N | (144*64N)+64N |
| Dense with ReLU | 50 | 64N+50 |

### 3.3.5 Convolution Neural Network test: output dimension and window size

To see how the input size and output dimension of a CNN affects execution time a CNN was implemented using raw EMG data instead of the feature extracted data. The model used all 12 channels and classified the first 10 hand motions. The window size, ts, was set from 25 to 100 samples with 25 samples increments. The output dimension parameter N was incremented with 25, starting at 50 and ending at 300 samples. A pooling layer, a flatten layer and a Dense layer was also used. The different layers, the output shape and the number of parameters for this CNN model is shown in table 3.3.

**Table 3.3** Summary of the layers used in the CNN with varying window size ts and output shape N

| Layer | Output Shape | Number of Parameters |
|---|---|---|
| Convolutional with ReLU | ts-2, 10, N | 10N |
| Max Pooling | 1, 1, N | 0 |
| Flatten | N | 0 |
| Dense Layer | 10 | 10 +10*N |

### 3.3.6 Convolution Neural Network test: kernel size

Another CNN was similarly implemented where the kernel size was increased from 1 to 5 steps. The window size, ts, was set from 25 to 650 samples. The CNN had a convolution layer with a parameter N that was incremented with 25 samples. It also had a pooling layer, a flatten layer and a Dense layer. The different layers, the output shape and the number of parameters for this CNN model is shown in table 3.4

**Table 3.4** Summary of the layers used in the CNN test with incremental kernel size and window size ts

| Layer | Output Shape | Number of Parameters |
|---|---|---|
| Convolutional with ReLU | ts-kernel size+1, 12-kernel size, 25 | 25*$kernelsize^2$+25 |
| Average Pooling | 1, 1, 25 | 0 |
| Flatten | 25 | 0 |
| Dense Layer | 10 | 260 |

### 3.3.7 Two layer Convolution Neural Network test: kernel size

Afterwards a similar test was done but with 2 2 dimensional Convolution layers, where the window size, ts, was set from 25 to 650 samples with 25 samples increment. The kernel size of the second convolution layer was increased from 1 to 5 with one step increments. Afterwards a flatten and Dense layer was used, the model is shown in table 3.5.

**Table 3.5** Summary of the layers used in the CNN with incriminating kernel size and window size ts

| Layer | Output Shape | Number of Parameters |
|---|---|---|
| Convolutional with ReLU | ts, 12, 25 | 50 |
| Convolutional with ReLU | 50-kernal size , 300 - kernel size, 25 | 625*$kernelsize^2$ + 25 |
| Average Pooling | 1, 1, 25 | 0 |
| Flatten | 392 | 0 |
| Dense Layer | 10 | 3930 |

### 3.3.8 Memory constraint estimate

Lastly, during all the testing, the maximum number of memory usage was estimated. These results will be presented as an estimate to the complexity of the model compared to memory usage.

# 4

# Results

Below, the different results will be described in different sections named similarly as their corresponding section under section 3.3.

## 4.1 Linear Discriminant Analysis result: window size

For the varying window sizes test the different feature extraction algorithms had different execution times. The execution time can clearly be seen to be linearly dependent on window size. The fastest being Variance of EMG and Root Mean Square, while the slowest was Modified Mean Absolute value type 2. Even for the slowest algorithm, the execution time was very low, only taking a maximum of 0.8 ms. This is shown in figure 4.1.

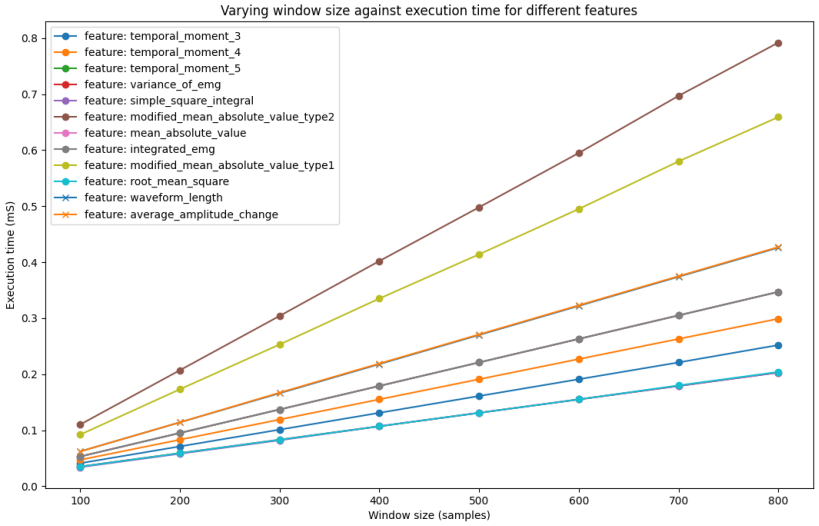


**Figure 4.1** Window sizes in samples against the execution time (ms) for different features.

The accuracy and balanced accuracy of these models can also be seen to be dependent on the window size, with higher window size usually corresponding to a higher accuracy. The accuracy of the model was much higher than the balanced accuracy. The difference was around 30%, which is reasonable as there was a lot more data for rest than the other movements. For the accuracy the feature resulting in the highest accuracy was from average amplitude change. It could also be seen that the feature which resulted in the highest balanced accuracy varied. Average Amplitude Change and Root Mean Square gave the highest accuracy dependent on window size, where one performed better for some window sizes and the other performed better for other. The accuracy is shown in figure 4.2, while the balanced accuracy is shown in figure 4.3.

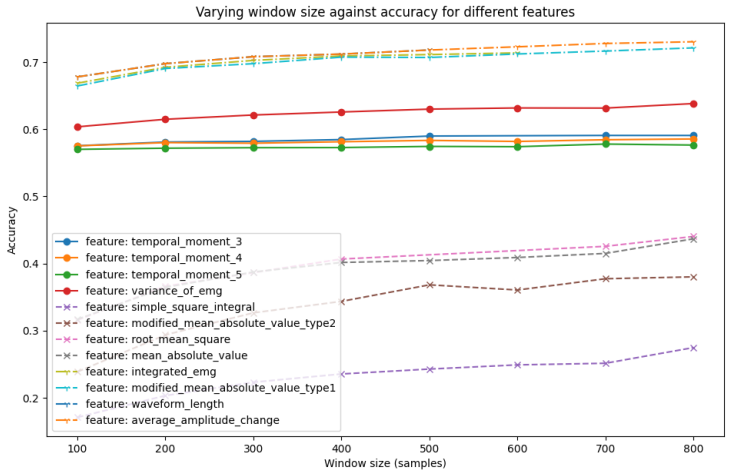


**Figure 4.2** Different window sizes in samples against the accuracy for different features.

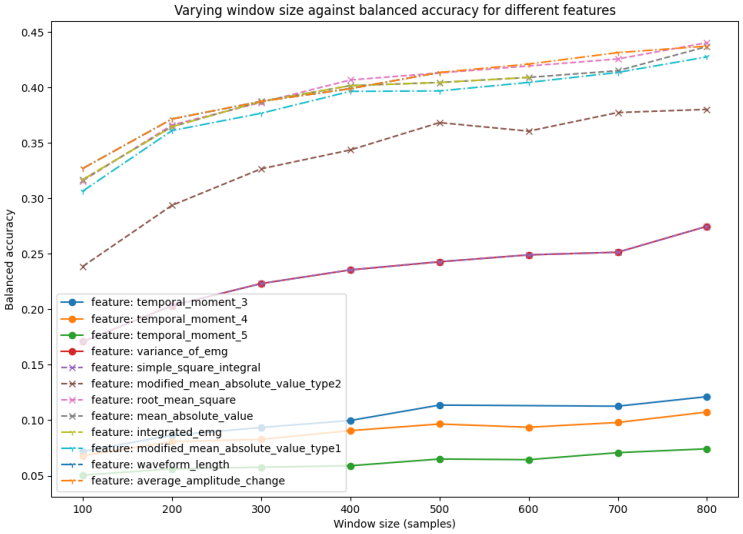


**Figure 4.3** Different window sizes in samples against the balanced accuracy for different features.

## 4.2 Linear Discriminant Analysis result: accuracy for all persons

The accuracy of the feature extraction algorithms using Linear Discriminant Analysis can be seen in figure 4.4 and the balanced accuracy can be seen in figure 4.5. As the figures show, some feature extraction algorithms performed better for different subjects. The feature extraction algorithms were unequal in performance, where some performed better and other worse. The three worst performing algorithms were Temporal Moment 3, 4 and 5. Average Amplitude Change and Root Mean Square usually performed the best between the test subjects for the balanced accuracy test and Average Amplitude Change usually performed the best for the accuracy test. The difference between the accuracy and the balanced accuracy could be seen to be around 20% to 30%.

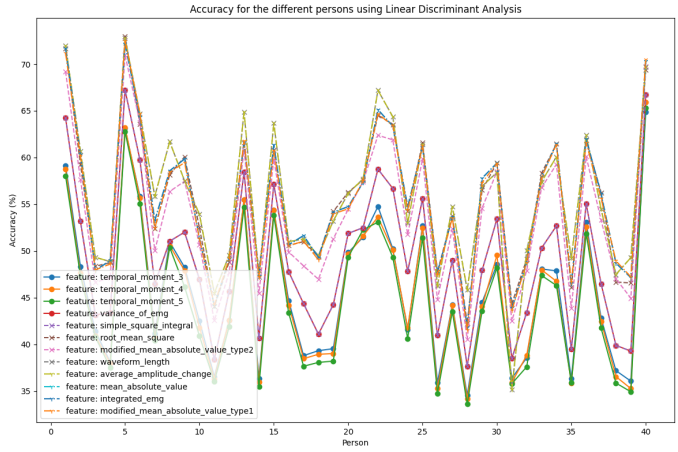


**Figure 4.4** Accuracy of the different implemented feature extraction model using Linear Discriminant Analysis.

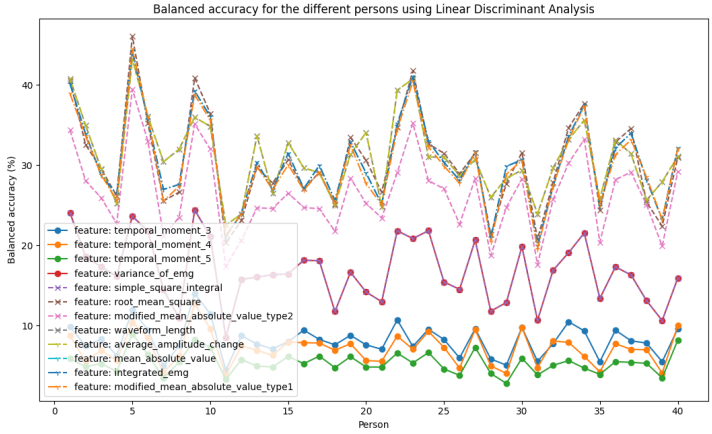


**Figure 4.5** Balanced accuracy of the different implemented feature extraction model using Linear Discriminant Analysis.

## 4.3 Convolution Neural Network result: comparing results between the two models

The results from the comparison between the test model and the Teensy 4.1 model is shown in table 4.1. The results show slight difference between the two models, in the range of $10^{-12}$. Why there is a difference between these results will be further discussed in section 5.

**Table 4.1** Difference between Teensy model and MLA model for last output layer before classification

| element | Teensy 4.1 board result | python model result | difference between the two models |
|---|---|---|---|
| 1 | 0.000617405981756 | 0.000617405923549085854840087890 6 | $5.8 * 10^{-12}$ |
| 2 | 0.000091110487119 | 0.000091110458015464246273040771 48 | $2.9 * 10^{-12}$ |
| 3 | -0.000272167089860 | -0.000272167118964716792106628417 97 | $2.9 * 10^{-12}$ |
| 4 | -0.000382805068511 | -0.000382805039407685399055480957 03 | $2.9 * 10^{-12}$ |

## 4.4 Dense Neural Network result: output dimension

The results show that both the parameters and the execution time scale linearly with the output dimension of the dense layer. Figure 4.6 shows how the execution time in micro seconds scales with the output dimension size. While figure 4.7 shows how the execution time scales with the amount of parameters. After reaching 100 000 params count, the Teensy 4.1 board could not compile the code due to insufficient memory. This means that the timing constraint holds with a lot of spare room when using dense layers for window size, $ts = 225ms$ and an overlap size of 30%. This can be said with certainty as the maximum amount of delay received was under $4ms$.

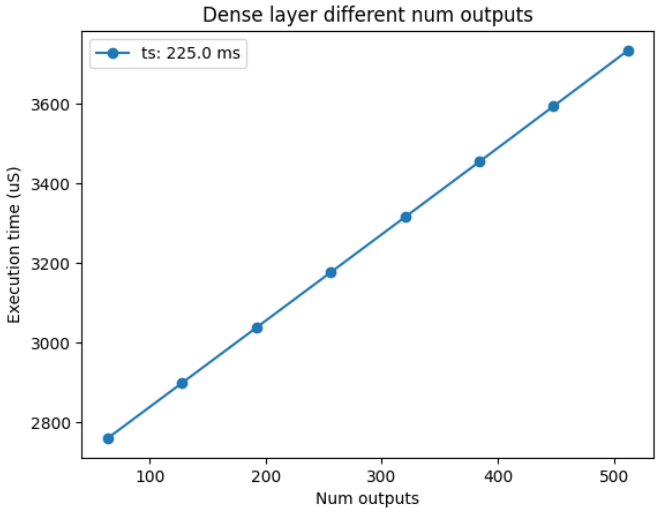


**Figure 4.6** Dense test showing how the number of the output dimension correlates to the execution time.

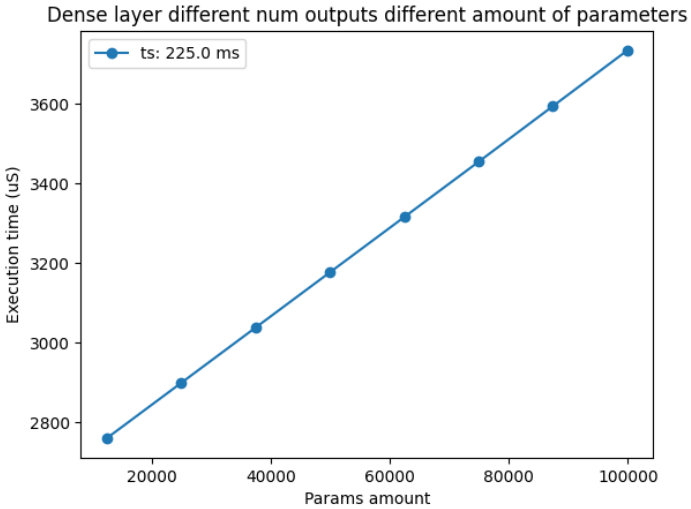


**Figure 4.7** Dense test showing how the number of dense parameters correlates to the execution time.

## 4.5 Convolution Neural Network result: output dimension and window size

The result of this test shows that the execution time scales linearly with output dimension. This is shown in figure 4.8. As the figure also shows, even though there is an increase in execution time between this model and the Dense model, the execution time is still small enough that it will handle the timing constraints for most model having a window size between 200-300 ms with an overlap size of 10-30%.

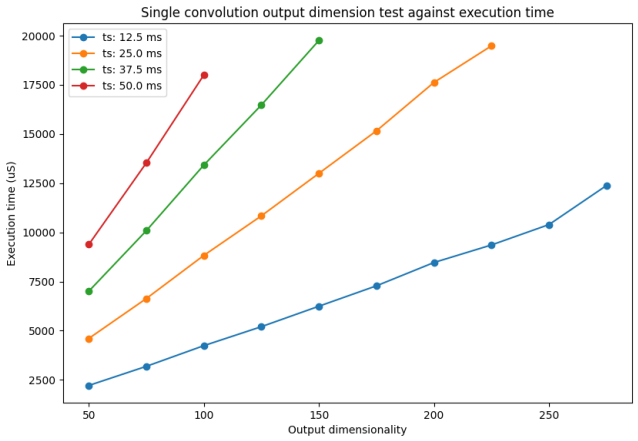


**Figure 4.8** CNN test over how the output dimension corresponds to the execution time.

The total storage elements also seems to scale linearly with the execution time, this is shown in figure 4.9. Here it can be seen that before the execution time starts to be larger than the timing constraints, it will run out of memory. Of the models tested, it seems that the Teensy 4.1 board should be able to handle most of these models.

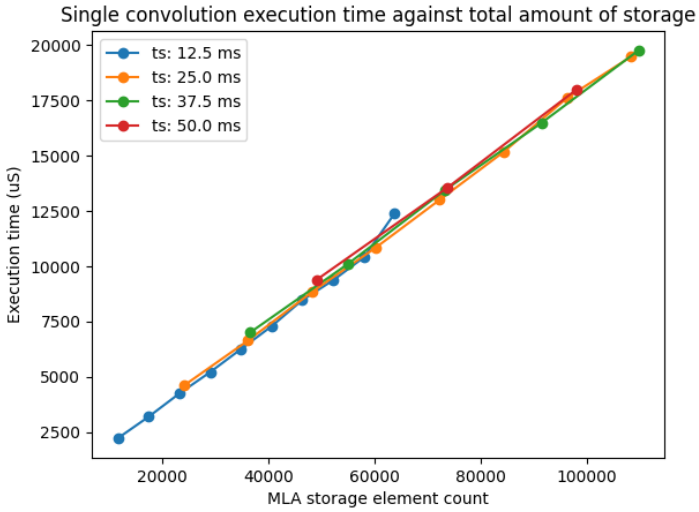


**Figure 4.9** CNN test over how the output dimension correspond to the total MLA memory usage.

## 4.6 Convolution Neural Network result: kernel size

How the kernel size of the CNN affects the execution time can be seen in figure 4.10. Here it can be seen that there is a larger increase in execution time between the second and third kernel size, and then seems to be quite linear towards the end. The largest measured execution time was slightly below 100 ms. For these tests, the timing constraints holds for windows between 200 - 300 ms, with an overlap of 30%.

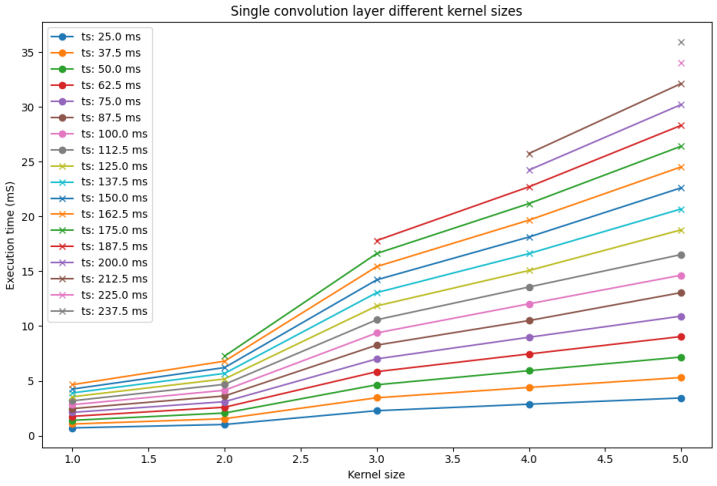


**Figure 4.10** CNN test over how the kernel size corresponds to the execution time.

The amount of MLA storage is dependent on kernel size. This can be seen in figure 4.11. It can also be seen that with a larger kernel size the amount of storage decreases decreases. This is reasonable as there is less intermediary storage, even though more parameters need to be stored for the higher kernel size count. As it could be seen in the figure before, the execution time seems to increase with the kernel size, and as the total amount of memory decreases, it means that higher kernel sizes than the ones tested could result in timing errors. This can be seen in figure 4.12.

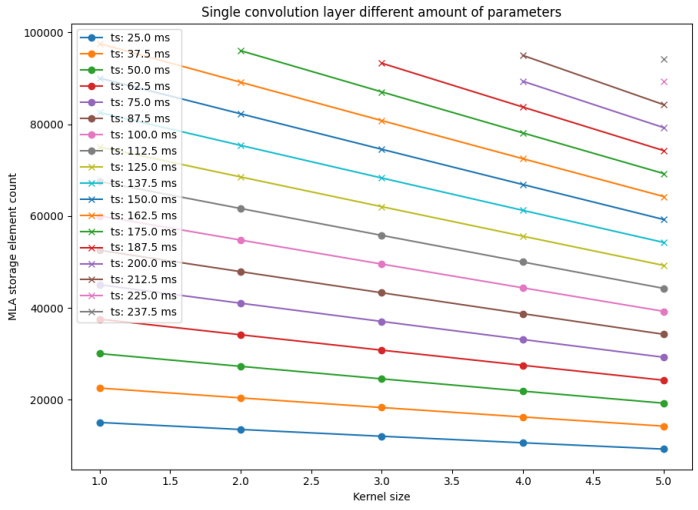


**Figure 4.11** CNN test over how the kernel size correspond to the total MLA memory usage.

## 4.7 Two layer Convolution Neural Network test: kernel size

For the two layer convolution layer it can be seen that the execution time greatly increases with the kernel size and the window size. For a window size of 75*ms* and a kernel size of 5, the execution time is above 250*ms*. This means that here there will be timing errors for most window sizes between 200 - 300 ms. This can be seen in figure 4.12.

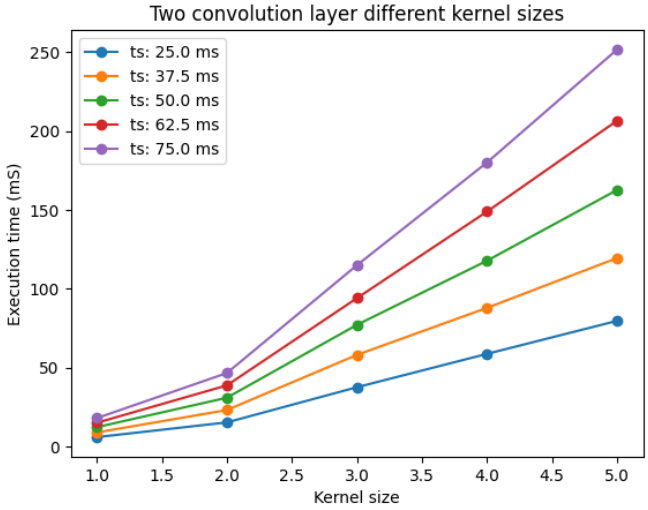


**Figure 4.12** CNN test over how the second convolution layers kernel size corresponds to the execution time.

The total amount of MLA storage units used was also plotted against the kernel size which can be seen in figure 4.13. It can also be seen that for larger kernel sizes, the total amount of memory storage decreases. This means that there could be larger timing errors for larger kernel sizes than the ones tested as the execution time increased with the kernel size.

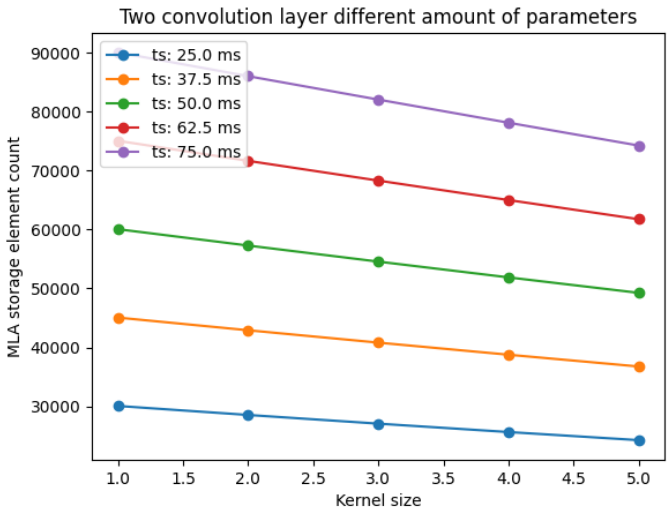


**Figure 4.13** CNN test over how the second convolution layers kernel size correspond to the total MLA memory usage.

## 4.8 Memory constraint estimate

It was found that the models complexity accounting for memory storage could roughly be estimated by the following equation: MLA storage units + params units $<$ 100 000 memory units, where the MLA storage was the storage needed for the intermediary steps and params where the total amount of parameters needed to implement the different weights for the models. There were some models that could be implemented with larger values than 100 000 memory units, but no model was found to not fit this constraint.

# 5

# Discussion

The discussion was split up into different sections. First the accuracy of the different algorithms was evaluated from the results of section 4.1 and 4.2. Secondly, the hardware constraints and implementation constraints were discussed from the result of section 4.3 to 4.8. Afterwards, further work will be discussed and finally the conclusion of this thesis will be presented.

## 5.1 Accuracy evaluation

As it could be seen Temporal Moment 5 had the worst accuracy out of the different feature extraction methods, as seen in figure 4.5. This is likely due to Temporal Moment 5 producing results in the $10^{-25}$ range. This can be a problem and should be handled better especially when using float32 to represent the data. This was also a problem but to a lesser extent for Temporal Moment 3 and 4, as they also had low accuracy.

The balanced accuracy for Linear Discriminant Analysis seen in figure 4.5 also shows that different feature extraction performed better for different subjects in the database. This is probably because every person controls their muscles in different ways. Making some methods better for some people and other better for other people. This means that there were no "clear winners". The same result can be seen when only comparing the accuracy of the different models, which is shown in figure 4.4. Having a system that can implement the best model for each person is thus preferable and having this configurable system that can implement all these models can be helpful to increase the accuracy of myoelectric prosthesis for each person.

## 5.2 Implementation constraints

The compared results between the Teensy 4.1 board and the python model shows that there was at least a difference around $10^{-12}$ range. This result was shown in table 4.1. It is important to note the received data from the Teensy 4.1 board had a precision of 16 decimals, so more discrepancy could not be shown. As the error of the value is relatively low the accuracy should not be lessened compared to the python model. The underlying issue was not found for the discrepancy, but it was assumed that the discrepancy appeared due to some rounding being missing in the python model.

During testing, it was noticed that the biggest constraint for the different designs were memory constraints. While a DNN could be implemented using feature extracted data; once raw data was used most models would run into memory problems. This is because the Teensy 4.1 board did not have enough memory available. This could be solved adding an SD-card to the Teensy 4.1 board to give it more memory but accessing this memory also could take more time dependent on the SD-card used and the boards specifications.

Even though the memory constraint was estimated, the actual constraint depends on different factors. For an example, something that was not accounted for was a larger window size would consume more memory as it would be stored twice, in the input buffer and as the input array for the feature extraction. The number of lines of code could also affect the memory constraint, as these lines needs to be stored in memory.

For the timing constraint, it was noticed that most of the tested models could be implemented with a window size between 200 to 300 ms except for models using two convolution layers with larger kernel sizes. Most other models ran out of memory before the time constraint became relevant. This was seen for single convolution layers, dense layers, and other linear models. Even though all these models could be implemented with a large window size and a large overlap size, this would result in a bad performance for the myoelectric prosthesis control unit. The same thing also applies for very small window sizes for the models, as they will lose accuracy.

## 5.3 Further work

There were some model parameters that were not implemented for the different TensorFlow Keras layers. Firstly, one can currently not choose different padding for convolution or pooling layers. Padding sets the output shape to either the same as the input shape or smaller output shape. Currently it can only implement smaller output shape to the Teensy 4.1 model. There are also "stripes" which can also be implemented for these layers. When performing convolution or pooling layer instead of checking a matrix with indices next to each other "stripes" can choose to compare indexes with a further step than one. There are also a lot more activation functions that could be added to the system, like SoftMax or tanh. By implementing these features, more complex CNN and DNN could be implemented that hopefully would produce better results.

Currently all intermediate memory in the MLA have different memory buffers. Some of these are redundant. To make the code work better, a shared intermediate memory system could be implemented to free more redundant memory usage from the system. Which would also allow for more complex models to be implemented as the memory constraint is lessened.

All the memory systems are currently implemented with float32, but being able to implement some models as float64 could be preferable for models using Temporal moment 3, 4 or 5 even if it would cost more in processing time. To the same extent, being able to implement everything as fixed point arithmetic's instead of floating point arithmetic's could also help with performance when doing different arithmetic's operations. Quantization of the memory could also be investigated as a trade-off between accuracy and memory usage could be useful.

CMSIS was not used which is a library with inbuilt Digital signal processing for the Teensy 4.1 CPU. Implementing the different arithmetic's by using this system could increase the performance of the Teensy 4.1 model. Allowing for the Teensy 4.1 board to be able to implement more complex models without reaching the time constraint.

Currently only time domain feature extraction algorithms are implemented. Having more feature extraction algorithms could be helpful, but implementing frequency domain feature extraction algorithms could produce better results and should be looked into. Implementing frequency domain feature extraction algorithms is more computational heavy than time do-

main feature extraction algorithms, as they need to implement Fourier Transform to work. This could cost a lot in execution time but the added accuracy gain could be worth it.

Lastly, there are more different models in the sklearn library that could be implemented.

### 5.3.1 Conclusion

The results show that simpler models could be generated to a high degree of accuracy, but that more complex models, like the two layer convolution model usually ran into memory or timing constraints problems. For the given implementation, different feature extraction algorithms were more useful than others, the discrepancy was dependent on the memory type used. The results also shows that different models generated different accuracy for the different people in the NinaPro database [1], which points towards this code generating system to be useful for people in need of myoelectric prosthesis as they can get a system that works best for them. Even though further work can be done in this field, like more optimization and implementation of more algorithms, the results are satisfying enough to be useful.

# Bibliography


[1] Manfredo Atzori, Arjan Gijsberts, Claudio Castellini, Barbara Caputo, Anne-Gabrielle Mittaz Hager, Simone Elsig, Giorgio Giatsidis, Franco Bassetto, and Henning Müller. Electromyography data for non-invasive naturally-controlled robotic hand prostheses. *Scientific Data*, 1(1):140053, Dec 2014. ISSN 2052-4463. doi:10.1038/sdata.2014.53. URL https://doi.org/10.1038/sdata.2014.53.

[2] Silvestro Micera, Jacopo Carpaneto, and Stanisa Raspopovic. Control of hand prostheses using peripheral information. *IEEE Reviews in Biomedical Engineering*, 3:48–68, 2010. doi:10.1109/RBME.2010.2085429.

[3] Katherine A Raichle, Marisol A Hanley, Ivan Molton, Nancy J Kadel, Kellye Campbell, Emily Phelps, Dawn Ehde, and Douglas G Smith. Prosthesis use in persons with lower- and upper-limb amputation. *J Rehabil Res Dev*, 45(7):961–972, 2008.

[4] Alberto Esquenazi. Amputation rehabilitation and prosthetic restoration. from surgery to community reintegration. *Disabil Rehabil*, 26(14-15):831–836, 2004.

[5] Kristin Østlie, Per Magnus, Ola H. Skjeldal, Beate Garfelt, and Kristian Tambs. Mental health and satisfaction with life among upper limb amputees: a norwegian population-based survey comparing adult acquired major upper limb amputees with a control group. *Disability and Rehabilitation*, 33(17-18):1594–1607, 2011. doi:10.3109/09638288.2010.540293. URL https://doi.org/10.3109/09638288.2010.540293. PMID: 21166612.

[6] Guanglin Li and Todd A Kuiken. Emg pattern recognition control of multifunctional prostheses by transradial amputees. In *2009 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 6914–6917, 2009. doi:10.1109/IEMBS.2009.5333628.

[7] Domenico Buongiorno, Giacomo Donato Cascarano, Irio De Feudis, Antonio Brunetti, Leonarda Carnimeo, Giovanni Dimauro, and Vitoantonio Bevilacqua. Deep learning for processing electromyographic signals: A taxonomy-based survey. *Neurocomputing*, 452: 549–565, 2021. ISSN 0925-2312. doi:https://doi.org/10.1016/j.neucom.2020.06.139. URL https://www.sciencedirect.com/science/article/pii/S0925231220319020.

[8] Sidharth Pancholi and Amit M. Joshi. Electromyography-based hand gesture recognition system for upper limb amputees. *IEEE Sensors Letters*, 3(3):1–4, 2019. doi:10.1109/LSENS.2019.2898257.

[9] K R Mills. The basics of electromyography. *Journal of Neurology, Neurosurgery & Psychiatry*, 76(suppl 2):ii32–ii35, 2005. ISSN 0022-3050. doi:10.1136/jnnp.2005.069211. URL https://jnnp.bmj.com/content/76/suppl_2/ii32.

[10] Roberto Merletti. *Electromyography : Physiology, Engineering, and Non-Invasive Applications*, volume 11. 08 2004. ISBN 0471675806. doi:10.1002/0471678384.

[11] Jun-Uk Chu, Inhyuk Moon, Yun-Jung Lee, Shin-Ki Kim, and Mu-Seong Mun. A supervised feature-projection-based real-time emg pattern recognition for multifunction myoelectric hand control. *IEEE/ASME Transactions on Mechatronics*, 12(3):282–290, 2007. doi:10.1109/TMECH.2007.897262.

[12] Massimiliano Zecca, Silvestro Micera, M.C. Carrozza, and Paolo Dario. Control of multifunctional prosthetic hands by processing the electromyographic signal. *Critical reviews in biomedical engineering*, 30:459–85, 02 2002. doi:10.1615/CritRevBiomedEng.v30.i456.80.

[13] Hassan Ashraf, Asim Waris, Syed Omer Gilani, Amer Sohail Kashif, Mohsin Jamil, Mads Jochumsen, and Imran Khan Niazi. Evaluation of windowing techniques for intramuscular EMG-based diagnostic, rehabilitative and assistive devices. *J Neural Eng*, 18(1), February 2021.

[14] Todd R Farrell. Determining delay created by multifunctional prosthesis controllers. *J Rehabil Res Dev*, 48(6):xxi–xxxviii, 2011.

[15] Sherif M. Waly, Shihab S. Asfour, and Tarek M. Khalil. Effects of time windowing on the estimated emg parameters. *Computers Industrial Engineering*, 31(1):515–518, 1996. ISSN 0360-8352. doi:https://doi.org/10.1016/0360-8352(96)00188-X. URL `https://www.sciencedirect.com/science/article/pii/036083529600188X`. Proceedings of the 19th International Conference on Computers and Industrial Engineering.

[16] Angkoon Phinyomark, Pornchai Phukpattaranont, and Chusak Limsakul. Feature reduction and selection for emg signal classification. *Expert Systems with Applications*, 39 (8):7420–7431, 2012. ISSN 0957-4174. doi:https://doi.org/10.1016/j.eswa.2012.01.102. URL `https://www.sciencedirect.com/science/article/pii/S0957417412001200`.

[17] Athanasopoulos G Hyndman, R.J. *Forecasting: principles and practice*, 2nd edition, otexts: Melbourne, australia, 2018. URL `https://otexts.com/fpp2/`. (accessed: 2022-11-16).

[18] B. YEGNANARAYANA. *ARTIFICIAL NEURAL NETWORKS*. PHI Learning, 2009. ISBN 9788120312531. URL `https://books.google.se/books?id=RTtvUVU_xL4C`.

[19] Micheal A. Nielsen. "neural networks and deep learning", determination press, 2015. URL `https://www.http://neuralnetworksanddeeplearning.com/index.html`.

[20] Pau Farré, Alexandre Heurteau, Olivier Cuvier, and Eldon Emberly. Dense neural networks for predicting chromatin conformation. *BMC Bioinformatics*, 19(1):372, October 2018.

[21] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[22] Xiaolong Zhai, Beth Jelfs, Rosa H. M. Chan, and Chung Tin. Self-recalibrating surface emg pattern recognition for neuroprosthesis control based on convolutional neural network. *Frontiers in Neuroscience*, 11, 2017. ISSN 1662-453X. doi:10.3389/fnins.2017.00379. URL https://www.frontiersin.org/articles/10.3389/fnins.2017.00379.

[23] Pravendra Singh, Prem Raj, and Vinay P. Namboodiri. Eds pooling layer. *Image and Vision Computing*, 98:103923, 2020. ISSN 0262-8856. doi:https://doi.org/10.1016/j.imavis.2020.103923. URL https://www.sciencedirect.com/science/article/pii/S026288562030055X.

[24] Pierre Baldi and Peter J Sadowski. Understanding dropout. In C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL https://proceedings.neurips.cc/paper/2013/file/71f6278d140af599e06ad9bf1ba03cb0-Paper.pdf.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[26] Kay Henning Brodersen, Cheng Soon Ong, Klaas Enno Stephan, and Joachim M. Buhmann. The balanced accuracy and its posterior distribution. In *2010 20th International Conference on Pattern Recognition*, pages 3121–3124, 2010. doi:10.1109/ICPR.2010.764.

[27] Paul J Stoffregen and Robin C Coon. Teensy® 4.1 development board. URL https://www.pjrc.com/store/teensy41.html. (accessed: 2022-11-21).