DEEP LEARNING ON A MICROCONTROLLER UNIT IN A HAND-PROSTHESIS CONTROL CONTEXT

Master's Thesis in Electrical Measurements

Hannes Nilsson

June 2021

LUNDS UNIVERSITET

LTH

LUNDS TEKNISKA HÖGSKOLA

Faculty of Engineering LTH
Department of Biomedical Engineering

Main supervisor : Christian Antfolk
Co-supervisor : Alexander Olsson
Examiner : Johan Nilsson

Abstract

This master's thesis investigates in what capacity a typical hand-prosthesis classification ANN (Artificial Neural Network) can be deployed on a microcontroller unit. A working, implemented interface is presented along with extra scripts, applications and functions for easy training and testing. Performance of different architectures as well as model-conversion tools are evaluated and compared. Furthermore, potential drawbacks, advantages of ANN architectures and tools are discussed. A dense neural network, albeit primitive, implementation on current readily available microcontrollers might be feasible as a classifier in a real time hand-prosthesis context. Finally, possible solutions and improvements are proposed.

Keywords: electromyography, deep learning, microcontroller, hand-prosthesis control, artificial neural network

Acknowledgements

I would like to thank my supervisor Christian Antfolk, for providing guidance and valuable feedback throughout this thesis work. I'm also deeply grateful for my family, whose continuous support and encouragement has meant a lot to me.

Contents

Acronyms **7**

1 Introduction **9**
1.1 Aim . . . 10

2 Background **11**
2.1 Machine Learning . . . 11
2.1.1 Classification . . . 12
2.2 Artificial Neural Networks . . . 14
2.2.1 The perceptron . . . 14
2.2.2 Activation functions . . . 15
2.2.3 Training neural networks . . . 16
2.2.4 Deep learning . . . 18
2.3 Machine learning on MCU . . . 19
2.3.1 Fixed-Point Mathematics . . . 19
2.3.2 Teensy . . . 21
2.4 Electromyography . . . 22
2.4.1 Common EMG features . . . 23
2.4.2 Learning Movements/Patterns . . . 24

3 Methodology **25**
3.1 Training the ANN . . . 25
3.1.1 Data acquisition . . . 25
3.1.2 Pre-processing . . . 28
3.1.3 Feature Extraction . . . 29
3.1.4 Model Training . . . 29
3.1.5 Conversion tools . . . 30
3.2 Testing the ANN . . . 31
3.3 Teensy Implementation . . . 32

3.3.1 Robo-limb library 32

4 Results **35**
4.1 Inference Performance 35
4.1.1 Ninapro data 35
4.1.2 Self-obtained data 36
4.1.3 Comparison: Teensy vs Keras 39
4.2 Inference Time 40
4.3 Memory Requirements 41
4.4 Prosthesis control 42

5 Discussion **43**
5.1 Prosthesis control 43
5.2 Performance 44
5.3 Related Work 44
5.4 Future Work 45
5.5 Conclusions 45

References **47**

Acronyms

ANN Artificial Neural Network. 10, 14, 16, 18, 21, 24, 25, 29–31, 35, 43

CAN Controller Area Network. 21
CMSIS Cortex Microcontroller Software Interface Standard. 19
CMSIS-NN Cortex Microcontroller Software Interface Standard - Neural Network. 19
CNN Convolutional Neural Network. 18, 44, 45

DNN Dense Neural Network. 18, 29, 32, 35–38, 41, 44, 45

EMG Electromyography. 5, 9–11, 14, 22, 23, 28, 29, 31, 38, 42–44

ICA Independent Component Analysis. 22
iEMG Intramuscular Electromyography. 9, 22
IoT Internet of Things. 19

MAV Mean absolute value. 23, 29
MCU Microcontroller Unit. 10, 11, 19, 21, 30–32, 35, 40–42, 44, 45
MNF Mean frequency. 23
MV Majority Voting. 38

NN Neural Network. 15, 18, 37, 44, 45

PCA Principal Component Analysis. 22
PR pattern recognition. 44

RMS Root mean square. 23

SD Standard Deviation. 39
sEMG Surface Electromyography. 9, 22, 23
SGD stochastic gradient descent. 17

VAR Variance. 23, 29

Chapter 1

Introduction

The loss of a limb, specifically a hand, can be a severely debilitating condition affecting ones quality of life as well as ability to perform daily tasks. In Europe alone, the number of upper-limb amputees is estimated to be around 100 000 and worldwide this number could be as high as 40 million [1] [2]. Naturally, solutions for restoring limb functionality have been explored. One in three people with a limb deficiency use a passive prosthetic hand although this often times doesn't adequately restore limb function. Thus, active alterative solutions based on Electromyography (EMG) have for a long time been investigated and developed [1][2][3].

EMG interfaces can commonly be found in the biomedical field as assistive- and rehabilitation-technology. Applications extend beyond the medical field though. Most EMG interfaces aim at, through signals picked up by electrodes either on the surface of the skin (sEMG) or intra-muscular (iEMG), control devices such as prosthetic limbs. Moreover, this type of interface has the potential of being non-invasive with a low minimal motor skill requirements [4]. As a result, this makes it especially suitable for people with motor skills impairments. Additionally, it's possible to control prosthetic devices with surface (sEMG) signals from an amputee's stump skin [4]. EMG interfaces can also provide an alternative to other human-machine control methods relying on, for example, voice recognition or eye movements.

Electrically powered prosthesis, if successfully rehabilitated, can consequently improve the quality of life for upper limb amputees. Unfortunately, this doesn't come without issues. Despite many recent advances and attempts at replacing a limb with a dexterous prosthesis, not many solutions have proven to work as a complete daily functional replacement option. Often times, prosthetic rejection is due to the fact that the amputee was not able to developed sufficient skills for daily use [5]. This can partly be attributed to the fact that the degree of motor skill impairment varies from case to case, thus lacking a general solution. Robustness outside a laboratory setting has also been a limiting factor [6].

In recent years, the interest in EMG interfaces has increased swiftly. The main reasons for this is the adaptation of the pattern-based recognition approach. These newer control methods allow, amongst other benefits, a greater number of degrees of freedom (DOF) and more versatile interfaces than conventional non pattern-based control methods [4] [7]. One way to implement this, is with an Artificial Neural Network (ANN). Loosely inspired by the biological brain, an ANN tries, through a series of layers, make intelligent decisions. The *depth*, i.e. number of layers and computational units, of these networks correlate with the potential for solving increasingly complex problems.

Recent technological improvements have made it possible to implement ANNs directly on a Microcontroller Unit (MCU). This brings many benefits relating to latency, bandwidth and privacy [8].

1.1 Aim

The aim of this master thesis is to construct, test and evaluate an interface for controlling a hand-prosthesis with sEMG running on a Microcontroller Unit (MCU). Specifically, how the Teensy[9] MCU performs in the context of myoelectric control, both individually and in comparison with its desktop-computer solution counterpart.

As such, part of the work is to investigate what type of data-processing can be performed under the real-time constraints provided. At later stages, optimization of functionality and accuracy, among other things, will be explored.

Chapter 2

Background

This chapter will cover the fundamental concepts and theory for machine learning, neural networks, edge computing and EMG signal analysis. Machine learning will be presented with MCU implementation in mind. Furthermore, more topics relevant in an embedded context such as number representations and quantization basics will be explored. Lastly, there is a section about Electromyography which addresses signal origin and common signal processing.

2.1 Machine Learning

In broad terms, machine learning is a collection of methods for solving tasks, learning problems or finding patterns in provided data. There are three main types of learning, although they may sometimes be combined in a hybrid manner [10].

- Unsupervised learning
- Supervised learning
- Reinforced learning

The goal of supervised learning is to learn a function that maps inputs to outputs, $\mathbf{x} \rightarrow \mathbf{y}$. In contrast, unsupervised learning tries to learn the underlying structure, without given any value pair examples. Thus, the agent tries to learn without explicit feedback. Finally, reinforced learning aims to maximize some defined future reward function. The algorithm will *learn* by trying to eliminate actions leading to punishment while also increasing the probability of taking actions leading to reward over time.

2.1.1 Classification

A common type of problem in machine learning is that of classification, in which the model has to assign a predicted class labels to a collection of input data. When there is only two possible classes to select from, it's referred to as binary classification and when there are more than two it's known as multi-class classification [11]. A good starting point for evaluating performance is to look at the classification accuracy. Further information can be obtained by creating what is called a confusion matrix. The confusion matrix can illuminate potential problems with whether the model commonly *confuses* two classes with each other. The columns correspond to what classes the model predicted, while the rows correspond to the actual ground truth. Thus, ideally there should only be values along the matrix diagonal as seen in figure 4.3.

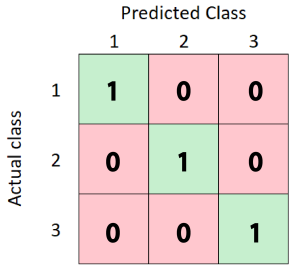


Figure 2.1: Ideal confusion matrix with 3 classes.

For classification accuracy to be relevant and useful, it's important that the input data is balanced. This means that each class should occur roughly the same amount of times in the training dataset. An unbalanced class distribution can potentially introduce a bias towards a certain class. In a worst case scenario where there is a severe unbalance, the model accuracy will only reflect the underlying class distribution [12].

As an alternative or complementary performance measure precision and recall can be used. These are usually presented in the context of binary classification where one class is *positve* and the other *negative*. In this way, precision is defined as the number of positive class predictions that actually belong to this class. Recall, on the other hand, quantifies how many of the positive classes present in the data were correctly classified [13]. Accuracy, precision and recall can also be defined with the amount of true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN):

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.1}$$

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

Lastly, there is a performance metrics that tries to consider and weight both precision and recall. It's known as F1-score and is defined as the following.

$$F_1 - score = 2 * \frac{precision * recall}{precision + recall} = \frac{TP}{TP + \frac{1}{2}(FP + FN)} \tag{2.4}$$

Precision and recall can also be extended to multi-class classification by considering some classes to be positive and others to be negative [12]. In the context of prosthesis control, one might consider the class/classes respresenting *no movement* as negative and *movement*-classes as positive. With this definition precision answers the question "Of all the predicted movements, how many were actually movements?" and recall answers the question "Of all the movements performed by the user, how many were correctly predicted?".

Classification delay

In a real-time system, the time allocated for a classification is pivotal. In general, more input data has the possibility to produce better results or outcomes, although it is important to remember that some methods that improve classification accuracy may not be viable in a real-time system since they introduce too long of a delay. By selecting a larger data acquisition time (window), the model may be able to achieve greater accuracy, although responsiveness is sacrificed. Hence, a balance between accuracy and responsiveness is required in practice.

Multiple studies have suggested a (round-trip) response time of 200 ms as the upper limit for achieving a system that feels "responsive" without noticeable delay. This can be argued to be task dependent though [14]. To achieve this, the signal is commonly segmented into *windows*. In this way, the analysis is performed in an on-line manner, each window at a time. Usually two approaches are considered for segmenting the data: disjointed or overlapping windowing. This is illustrated in figure 2.2, where T_{a1}, T_{a2}, T_{a3} are the window sizes and τ the processing time. The latter approach with overlapping windows requires more computations over equal amount of data, but can improve performance especially when paired with post-processing methods.

It is unclear what the smallest window size for segmentation is. The smallest time between distinct muscle contractions is in the 200ms range, but it may be beneficial to choose a smaller window size when using certain features and/or majority voting post-processing [14].

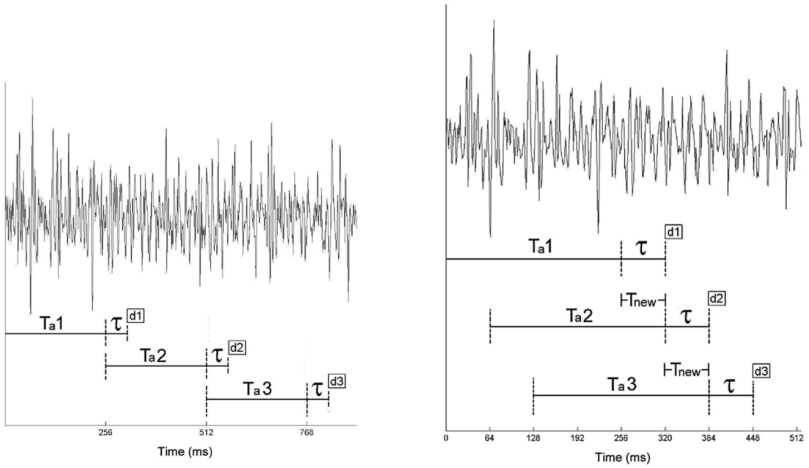


Figure 2.2: Disjoint windowing (left) versus overlapped windowing (right) on a single channel EMG. [14]

2.2 Artificial Neural Networks

The notion of Artificial Neural Network (ANN)s is not a new concept, rather it dates back to the mid 1940s, during which, Warren McCulloch and Walter Pitts presented an early model of the artificial neuron [15]. Shortly after, independently, Alan Turing defined the concept of an *unorganized machine* - a simple form of binary neural net, with interconnected processing units [16]. While many of the ideas for artificial neural networks are originally inspired from neuroscience and biological information processing, one should be cautious in taking these analogies too seriously [17]. In recent years, discoveries of new use-cases and flexible implementations, have brought ANNs to the leading edge of machine learning and artificial intelligence [18][10]. Simplified, ANNs are constructed of multiple computational layers with the goal of providing a rational output or decision. Each layers consists of multiple artificial neurons. The input to the an ANN can either be raw data, or features.Features are defined as individual measurable properties or characteristics present in data.

2.2.1 The perceptron

The basic building block and computational unit of most neural networks is a single neuron, known as the perceptron. It maps input signals $x_1, x_2, ..., x_n$ to a single output y through a series of calculations. Each input x_n is multiplied by its specific unique weight w_n before they are all summed together. An optional bias can also be added to the sum. Finally, it goes through, what is known as an activation function that maps the data to a nonlinear output range. This can be seen as a schematic representation in figure 2.3.

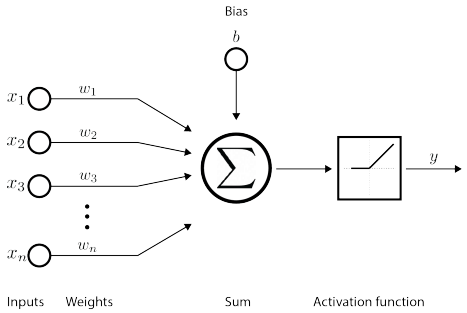


Figure 2.3: Schematic of a single neuron where inputs are denoted $x_1, x_2, x_3, ..., x_n$, bias b and output y

The output can consequently be written as:

$$y = \sigma\left(w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b\right) \tag{2.5}$$

Where σ represents an arbitrary activation function. By recognizing the dot product between $\mathbf{w}$ and $\mathbf{x}$ above, the equation can be rewritten in compact matrix form:

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad \mathbf{W} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

$$y = \sigma\left(\mathbf{X}^\mathbf{T}\mathbf{W} + b\right) \tag{2.6}$$

2.2.2 Activation functions

It is apparent from equation 2.6, that without the activation function, the output would only be a linear with respect to its inputs. As such, the neuron would also only be able to model and solve linear problems. For the ANN to be able to solve more complex problems and modeling arbitrary functions, non-linearities are introduced in the form of activation functions [18].

One of the first activation functions used is the sigmoid function seen to the right in figure 2.4. Due to its bound nature, it's especially suited for probabilistic problems.

A problem later discovered with this particular activation function is, what is often called, the *vaninshing gradient problem*. In short, since the functions derivative becomes very small at either end of the output space, the algorithm updating the weights in the NN cannot work effectively. This is especially prominent as the number of layers are increased.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.7}$$

Quite recently, in June 2010 another activation function was introduced by Vinod Nair & Geoffrey E. Hinton [19]. The *Relu* (eq. 2.8) simply maps values including and below zero to zero, while values above zero become unchanged. This eliminates the vanishing/exploding gradient problem. Relu is one of the most used activation functions in current neural networks due to its simplicity [20].

$$\sigma(x) = max(0, x) \tag{2.8}$$

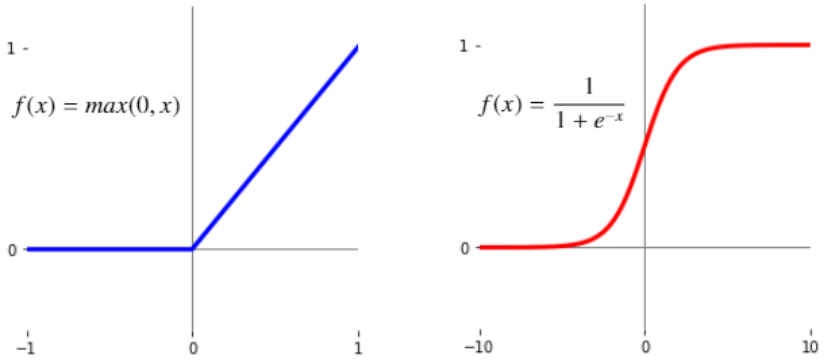


Figure 2.4: Activation functions : Relu (left), sigmoid (right)

In the last layer, often times a sigmoid activation function is used. This results in that the sum of every output value equals 1 and can thus be represented as a probability indication for classification problems.

$$\sigma(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \qquad \text{for } i = 1, \ldots, K \tag{2.9}$$

Say, the output layer of a binary classifying ANN consists of two neurons. The values of these could be 0.2 and 0.8 respectively, which would indicate the second class is much more likely the correct output.

2.2.3 Training neural networks

Essentially, *training* neural networks is finding the optimal weights and biases $\boldsymbol{W}^*$ that minimizes the loss function $\boldsymbol{J}(\boldsymbol{W})$.

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} J(\boldsymbol{W}) \tag{2.10}$$

This can be done in a multitude of ways and a common way to perform training is to use a backpropagation algorithm. The algorithm computes the gradient $\nabla \boldsymbol{J}$ through repeatedly

applying the chain rule. As such, the Gradient vector for the loss function can be written as:

$$\nabla J = \begin{bmatrix} \frac{\partial J}{\partial w^{(1)}} \\ \frac{\partial J}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial J}{\partial w^{(L)}} \\ \frac{\partial J}{\partial b^{(L)}} \end{bmatrix} \tag{2.11}$$

How the weights and biases are updated is another non-trivial problem. One intuitive approach would be to calculate the average gradient $\nabla \hat{J}$ over the whole training set and then taking a step in the negative direction of the calculated gradient - lowering the cost function. An immediate drawback with this method is that it can become very computationally expensive as the training set gets large.

A different approach to the same optimization problem is to handle each training example seperately calculating ∇J_i for each training example i. Furthermore a varible ∇, the step size or learning rate, is introduced. Typically, algorithms implement an adaptive learning rate, for quicker and better convergence. It makes intuitive sense to reduce the step size as the solution approaches the optimal minima [21].

$$w := w - \eta \nabla J_i(w) \tag{2.12}$$

This obviously handles larger training sets better, but doesn't step in the direction of the "true" gradient. Frequently, the training examples are selected at random and the algorithm is then known as stochastic gradient descent (SGD). Naturally, this doesn't come without drawbacks. On top of not stepping in the direction of the true gradient, movement can be erratic since each calculated estimate of the gradient is only based on one training sample. To remedy this, a compromise between both approaches can be used: mini-batch (stochastic gradient descent). This can ensure smoother convergence and performance increase.

One of many, variations and improvements upon the classic SGD algorithm is Adam (Adaptive Moment Estimation) [22]. It should be mentioned that the performance of these different optimization algorithms depend on multiple factors such as the shape of the loss function and training set size, among other things. They each have their inherent advantages and drawbacks.

2.2.4 Deep learning

As touched upon, ANNs are nothing more than interconnected layers of perceptrons/neurons. The first layer is known as the input layer and the last as the output layer. Layers in between are known as hidden layers. Figure 2.5 illustrates a simple ANN with one hidden layer.

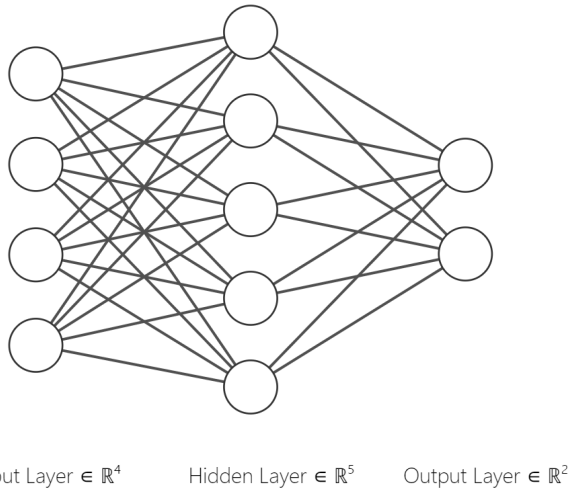


Figure 2.5: Simple ANN with one hidden layer.

Summarizing, the values for each layer $\mathbf{a^n}$ can be calculated from the previous layer as follows.

$$\begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_n^{(1)} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,n} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k,0} & w_{k,1} & \cdots & w_{k,n} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} \right) \tag{2.13}$$

$$\mathbf{a}^{(n)} = \sigma\left(\mathbf{W}\mathbf{a}^{(n-1)} + \mathbf{b}\right) \tag{2.14}$$

While the definition *deep* learning refers to NNs consisting of multiple (hidden) layers [23], one sometimes associates deep learning with a model obtaining "deeper" understanding, through learning on data directly instead of handcrafted features as input.

Up until this point, only network architectures where all the neurons between two subsequent layers are interconnected with each other have been introduces. These are called Dense Neural Network (DNN). There are other kinds of network architectures experimented with such as Convolutional Neural Network (CNN) and Recurrent Neural Networks (RNN). These lay outside the scope of this work though, but can be considered for future improvements and implementations.

2.3 Machine learning on MCU

Machine learning on microcontrollers and other edge devices is currently a hot topic with applications in many fast expanding areas. Among these are, Internet of Things (IoT), sensor fusion and synthetic sensors.

Traditionally, to meet computational requirements, machine learning has been done on the cloud or other host computers [8]. Issues with this approach however include latency, scalability and privacy. By moving or distributing parts of the computing to edge devices these issues can be mitigated to some degree. Firstly, latency is no longer dependent on propagation delays and eventual queuing in the network [8]. This is greatly important in applications with strict low-latency requirements. Moreover, scaling issues are alleviated as the architecture expands and the cloud can become the bottleneck due to increase in data and connections. Lastly, privacy concerns are reduced since less data is being sent to the cloud and more is done locally on the device. As a result of less data being sent, bandwidth is also being saved.

While the benefits are plentiful, machine learning on MCUs isn't the solution to every type of application and there are still limitations to ML on edge devices. Most notably, the training process is still too costly both computationally and memory wise for todays microcontrollers. Typical target applications include sensor analysis, activity recognition, voice recognition and predictive maintenance.

The current approach for implementing machine learning on MCUs is to initially train a model on a desktop computer or the cloud, which is then converted to a simplified, reduced version deployed on the target device. An example of such a tool is Google's Tensorflow Lite platform [24]. For MCUs with Cortex-M processors, there is another useful open source library (Cortex Microcontroller Software Interface Standard - Neural Network (CMSIS-NN)) that implements optimized NN algorithms [24] directly.

2.3.1 Fixed-Point Mathematics

Knowledge about fixed-point data types and arithmetic can be useful for understanding the Cortex Microcontroller Software Interface Standard (CMSIS) library in further depth, which is the basis for the optimized neural network and dsp functions.

Two of the most common ways of representing data are *fixed-point format* and *floating-point format*. For floating point numbers, the decimal point is *floating*, meaning a certain number of bits specify where it will be placed. As an implication of this, a wide range of numbers can be represented with a floating point. In contrast, the fixed-point number representation specifies exactly where the decimal point is placed, thus also defining the amount of bits used for the integer and fractional part of the number. Fixed point numbers are commonly used on MCUs and other DSP for efficiency and memory reduction.

Interestingly, a fixed-point number can assume exactly the same amount of numbers as its floating-point counterpart, the difference being what range they represent. Furthermore, fixed-point numbers always have a linear range. This is useful since the minimal error will always be well-defined and constant as the difference between two consecutive numbers.

Q Number Format & Quantization

The Q format is a type of fixed point number format often used. It is represented as

$$\boldsymbol{QM.N} \tag{2.15}$$

where $\boldsymbol{M}$ specifies the number of integer bits and $\boldsymbol{N}$ fractional bits. As an example, $\boldsymbol{Q}4.11$ would mean a number with 4 integer bits and 11 fractional bits. If using a sign bit to represent positive and negative values, this is sometimes included in the integer bit, which can cause some confusion [25]. The CMSIS-DSP (Cortex Microcontroller Software Interface Standard) defines a couple of different fixed point datatypes. These can be seen in figure 2.6. As an example, the datatype *q15_t* uses 1 bit for the sign part and 15 fractional bits, thus a valid format would be Q5.10, omitting the sign bit.

```
/**
 * @brief 8-bit fractional data type in 1.7 format.
 */
typedef int8_t q7_t;

/**
 * @brief 16-bit fractional data type in 1.15 format.
 */
typedef int16_t q15_t;

/**
 * @brief 32-bit fractional data type in 1.31 format.
 */
typedef int32_t q31_t;

/**
 * @brief 64-bit fractional data type in 1.63 format.
 */
typedef int64_t q63_t;

/**
 * @brief 32-bit floating-point type definition.
 */
typedef float float32_t;

/**
 * @brief 64-bit floating-point type definition.
 */
typedef double float64_t;
```

Figure 2.6: Datatypes defined by the CMSIS library.

It should be clearified however, that in actuality, fixed point numbers are represented as integers in the device memory as seen in fig 2.6. To convert from a QM.N integer to the represented value, one divides by 2 two the power of the fractional part i.e. 2^N. The fractional part, $\boldsymbol{N}$, is therefore often called the scaling factor. As an example, to convert an Q2.13 integer to the actual represented value the following has to be done:

$$Q2.13 = 4915 \longrightarrow \frac{4915}{2^{13}} = 0.59997 \tag{2.16}$$

When adding or multiplying two QM.N with different number of integer/fractional bits, it's necessary to convert them to the same QM.N type. This is easily done by bitshifting either right or left. Furthermore, the scaling factor has the significance that the lowest possible representable value is $\frac{1}{2^N}$, which also defines the lowest error due to quantization.

2.3.2 Teensy

The Teensy 4.0 is a MCU that can easily be programmed over USB in Arduino IDE [9]. It features, amongst other things, an ARM Cortex-M7 at 600 MHz, 2Mb of flash memory, 1MB RAM memory and 14 analog inputs. A relevant excerpt of the specifications can be seen in 2.7.

Feature	Teensy 4.0
Processor	IMXRT1062DVL6A
Core	Cortex-M7
FPU	32 & 64
Rated Speed	600
Overclockable	912
Flash Memory	1984
Bandwidth	66
Cache	65536
RAM	1024
Digital I/O	40
Breadboard I/O	24
Voltage Output	3.3V
Current Output	10mA
Voltage Input	3.3V Only
Analog Input	14
Converters	2
Usable Resolution	10
Comparators	4
Communication	
USB	2
Serial	7
With FIFOs	7
SPI	2
With FIFOs	2
I2C	3
CAN Bus	3
With CAN-FD	1

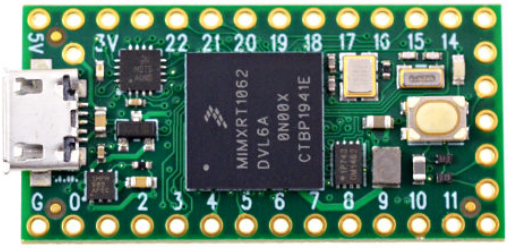


Figure 2.7: Excerpt from the documentation of features [9].

One of the required features for communication with the hand prosthesis (Robo-limb by Touch bionics) used in the system is a Controller Area Network (CAN) bus. More information about the prosthesis is found in section 3.3.1. A CAN is a robust communication protocol often found for automobile systems and control systems. Since the Teensy is equipped with this, it's possible to communicate with the robolimb through a CAN transceiver chip which amplifies the levels [26].

The processor in the Teensy MCU has a floating point unit (FPU), thus supporting floating point precision math in the hardware. The speed of these operations is comparable to standard integer math [9]. Certain calculations of ANNs, such as matrix multiplications, however, have been further optimized and are found in the CMSIS-NN library. To use these functions the datatypes defined in figure 2.6 are required. Consequently, ANN models need to be converted from floating point to fixed point for full optimization. There are conversion tools doing this, covered in the methodology section.

It is worth noting that the Teensy only has two analog-to-digital converters, meaning that might not be taken completely synchronously when sampling more than two channels. For this application, however, the core processor speed is sufficiently fast that no meaningful delay is expected due to this fact.

2.4 Electromyography

Electromyography (EMG) is the practice of capturing and evaluating electrical activity originating from the skeletal muscles. Consequently, electromyography is an important clinical tool for detecting different motor neuron abnormalities, rehabilitation medicine and intelligent prosthesis control [27] [28]. Recently, applications have been extended to gesture recognition, game control, robotics and various other control systems. [28].

The electrical activity originates at junction points between muscle fibers and motor neurons, called motor units. During a voluntary movement, the motor units are activated continuously sending motor action potential trains [29]. Together, these make up the measured EMG signal.

Electrical activity can be captured either with electrodes mounted on the surface (sEMG) of the skin or intramusclular (iEMG). The former method has the advantage of being non-invasive while the latter has the possibly of providing better individual muscle signals.

Overall, pattern recognition during hand movements is a challenge due to prevalence of inter-subject, intermuscle and context dependent variability [30]. Furthermore, signal processing as well as knowledge of how to extract useful information from raw EMG data are critical for proper EMG analysis [27]. Common pre-processing steps include filtering and signal separation methods (PCA/ICA). Multiple studies have concluded the importance of EMG characteristics such as signal intensity, burst duration and temporal patterns [30] [31].

Typical signal amplitude normally lies in the range 0.01mV to 10mV range, and the main energy is concentrated between 0Hz and 500Hz [28]. For complete frequency resolution, a sampling frequency of at least 1kHz would be recommended. Some electrodes, though, will internally rectifying the signal (Full-Wave Rectified EMG in fig. 2.8) and passing it through a moving average filter (Linear Envelope in fig. 2.8).

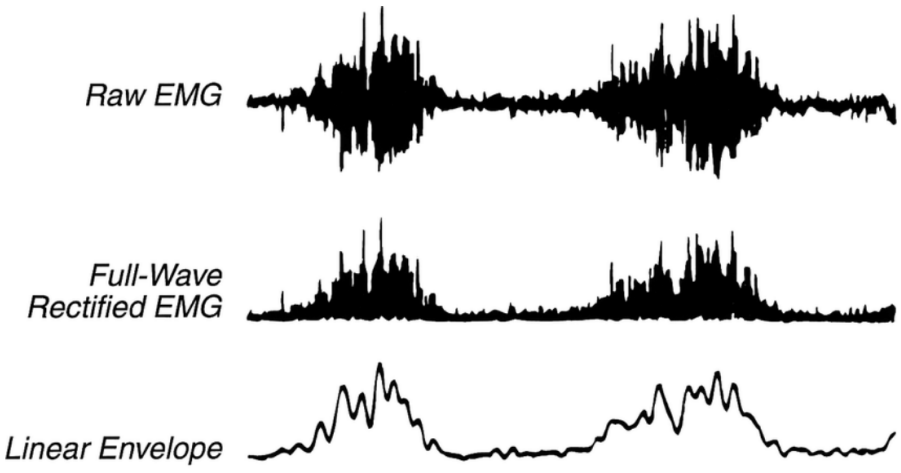


Figure 2.8: Different types of EMG signals. [32]

Naturally, the placement of sEMG electrodes is also of significance. It is recommended to position the electrodes in the middle of the muscle belly somewhere between the motor point (innervation zone) and a myotendon junction [30][33].

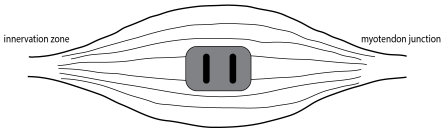


Figure 2.9: Electrode placement for largest signal during muscle contraction

During complete rest, muscular activity should be silent. This is rarely the case though since patients often have problems relaxing muscles completely. To distinguish activity originating from incomplete relaxation versus abnormal spontaneous contractions it's helpful to look at the rhythmicity [27].

2.4.1 Common EMG features

Historically, three types of features have been used to classify movements from EMG data: time-, frequency- and spatial domain features. Time domain features are the most used features for classification since they dont need any transformations and can immediately be calculated from the EMG signal amplitude [34]. Some examples of time domain features and their mathematical defintion are Mean absolute value (MAV), Variance (VAR) and Root mean square (RMS)

$$MAV = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{2.17}$$

$$VAR = \frac{1}{N-1}\sum_{i=1}^{N} x_i^2 \tag{2.18}$$

$$RMS = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2} \tag{2.19}$$

where N is the total number of samples in a given window of data, x_i is the EMG signal value at sample i.

Common frequency domain features are MNF and median frequency [4]. Feature selection also depends on the EMG signal obtained from the electrodes. If the signal is already rectified or a linear envelope has been applied (see figure 2.8 for reference), some features, such as time domain features, are not useful.

2.4.2 Learning Movements/Patterns

The machine learning done in the project will exclusively be with supervised learning. The ANN is trained with input-output pairs, where every movement corresponds to a number arranged in a "one-hot" array. Consequently, the number of neurons in the output layer agree with the number of possible movements. Furthermore the size of the input layer agrees with the number of input features.

Chapter 3

Methodology

This chapter describes the process firstly described how data was collected, prepared and processed. Secondly, the complete training process is described, including training the ANN. The methods of evaluating and verifying the working system along with implementation details is lastly described.

For most of the experiments in this work, both the publicly available dataset[35] Ninapro DB1 and self-obtained data was used. The main difference being the number of electrodes used, resulting in differently sized input layers of the ANN.

3.1 Training the ANN

The training process can be subdivided into five main steps. Data acquisition, pre-processing, feature extraction, model training and lastly conversion. It is essential, that processing steps are done identically on the deployed device. Thus, the capabilities of the device were considered when selecting pre-processing steps, features and ANN architecture.

3.1.1 Data acquisition

Ninapro

The Ninapro DB1 dataset was selected as a reference dataset since it uses the same type of electrodes used in our system. It's available as .mat files that can be imported to Matlab or Python.

Table 3.1: Synthetic description of the first 12 basic finger movements in the Ninapro DB1 dataset.

	#	Description	Ref.
Finger mvts.	1-2	Index flexion and extension	[13]
	3-4	Middle flexion and extension	[13]
	5-6	Ring flexion and extension	[13]
	7-8	Little finger flexion and extension	[13]
	9-10	Thumb adduction and abduction	[13]
	11-12	Thumb flexion and extension	

Self-obtained data

In this project five Myobock 13E200=60 electrodes [36] were used to capture the electrical activity of the forearm muscles in contrast to Ninapro DB1 which uses 10 of the same electrodes. The methods described in this section were used in both the training and test process.

Figure 3.1: Myobock Electrode 13E200=60 [36].

In preparation of signal acquisition with the electrodes a few things were considered. Optimally, the placement should be consistent, and repeatable for every training/test session. Trying to achieve this, the five Myobock electrodes were attached with adhesive tape to a flexible armband. The armband was subsequently fastened slightly below the elbow.

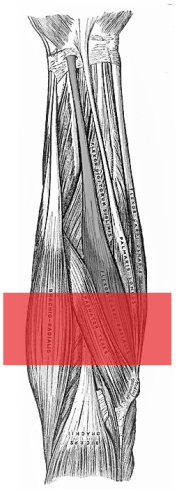


Figure 3.2: Approximate electrode armband placement [37]

The electrodes provide a rectified and filtered signal (similar to the lower signal in figure 2.8) at a sample rate of 100 Hz. Due to this, It takes some seconds of "relaxation" before the signal amplitude reaches its relative noise floor and reliable measurements can be done. Thus, it was important to always let it stabilize before beginning any measurements.

On the exterior of the electrodes, is a small potentiometer located, which was used to adjust the gain of the internal amplifier. An effort was made to set these gain levels equal across the electrodes, while at the same time avoiding any clipping of the signal.

The output of the Myobock eletrodes are in the range 0-4.5V which means it has to be stepped down as input voltages above 3.3V isn't supported on the Teensy. This was accomplished with two $10k\Omega$ resistances in a simple voltage divider configuration (figure 3.3).

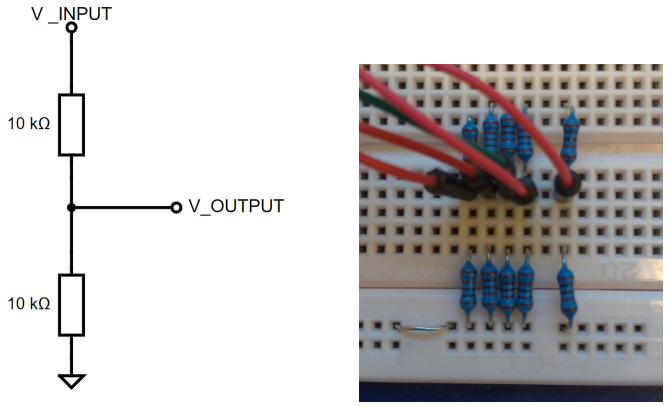


Figure 3.3: Voltage divider schematic (**left**). Voltage dividers for every input of the Teensy (**right**).

For simple analysis and communication, the serial monitor/serial plotter in the Arduino IDE is sufficient. The main issue doing it this way though, was that there was no neat way of saving data generated from the MCU on the computer. Due to this, serial port communication was done through Matlab. In this way, it was possible to efficiently record data while simultaneous instructing the user during training. A sample of the application can be seen in figure 3.4.

Two training programs were devised. The first training program "Basic - 5 movements, 5 repetition" consisting of flexion of all the fingers individually and the second program "10 movements, 10 repetitions" consisting of both flexion and extension for each finger. For the latter program, class labels 1 through 5 corresponded to flexion of each finger starting at the thumb, and labels 6 through 10 corresponded to extension of each figher - also starting at the thumb.

During training, cues were presented to the user and instructions on what movement to perform. Each movement repetition duration was 5 seconds with an identical 5-second rest pause in between repetitions.

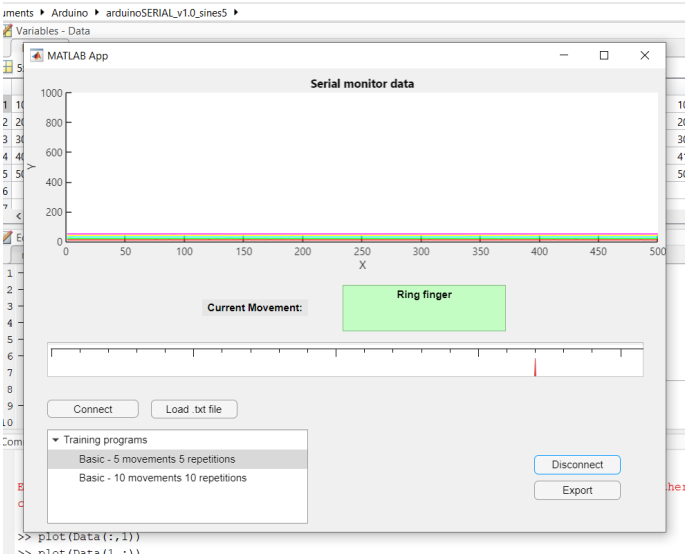


Figure 3.4: Matlab application for serial communication with the Teensy MCU.

The outputed data was formatted similar to the Ninapro dataset. One matrix containing all the "rawdata" of dimension #electrodes x samples. A further matrix, "stimulus", was created that indicates what type of movement was performed, where it starts and where it ends.

3.1.2 Pre-processing

Pre-processing consists firstly of data preparation. Inspired by the Ninapro database, a "stimulus" data array was used to indicate where a movement starts and ends in the EMG data. Every movement is assigned a number. Naturally, *no movement* equals zero and the rest a unique arbitrary integer. An example of how the stimulus array together with the EMG data can look during one movement is shown in figure 3.5 (left), while the whole stimulus vector is shown on the right. Here, some manual adjustments can be made to improve classification accuracy, depending on how good of a "fit" most of the movements have relative to the stimulus vector. After some familiarization with the training process, this wasn't highly necessary though.

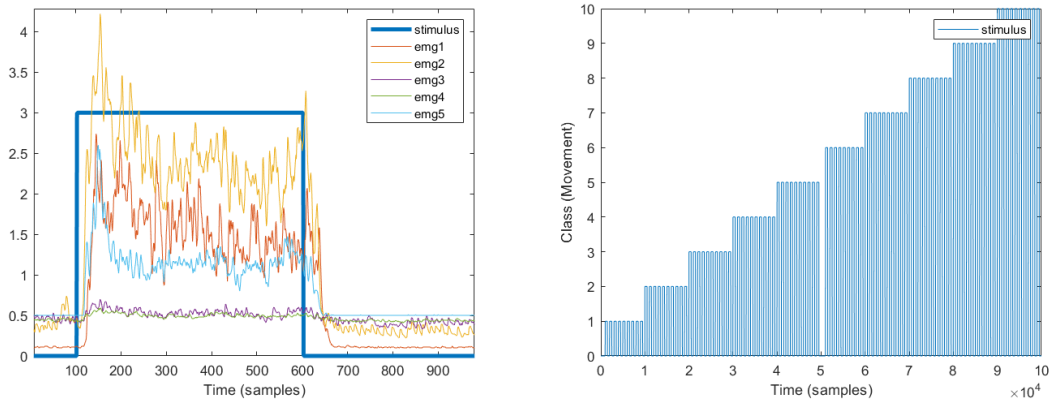


Figure 3.5: A single repetition of class 3 illustrating both the stimulus vector and five emg electrodes **(left figure)**. The complete stimulus vector including all classes 0 through 10, each repeated 10 times (**right figure**).

3.1.3 Feature Extraction

There are a variety of features to extract from EMG data, both time domain and frequency domain. Because of the already filtered output of the electrodes used though, only time domain features were selected. After intial testing, the features MAV and VAR were selected.

The two features were calculated for each electrode, meaning that in total 10 features were calculated for own data, and twice the amount of features were present for the experiments on the Ninapro DB1 data. As such, the size of the input layer to these ANNs are 10 respectively 20.

To calculate the features in practice over the EMG data, a rolling window (10 samples) was selected. Each window was assigned the correct label by taking the most occuring class in the stimulus array at the position of the window.

Moreover, with disjointed 50% segmentation (figure 2.2) features are calculated every 5 samples.

Since the training program contained 5 seconds rest between each movement, the 0 class *no movement* was severy overrepresented. This was accounted for by removing sufficiently many of this class to balance the dataset.

3.1.4 Model Training

The training of the ANNs was done in Python, specifically with the modules Keras and Tensorflow. The datasets were balanced, by reducing the number of number of occurances of the overrepresented class. Datasets were split into training-, test- and validation set with the ratios 0,64/0,2/0,16. Parameters were at first selected based on previous recommendations in articles[38] and then later fine tuned based on iterative testing.

Sequential DNN models of different sizes and structures were constructed and evaluated.

The optimizer algorithm Adam was used to minimize the Cross entropy loss between correct labels and predictions. Batch size was set to 64 in most cases and the learning rate was defaulted to 0.001.

To reduce risk of overfitting, an early stopping function was added to the training phase with a *patience* of 20 epochs, i.e. the training will be halted if validation accuracy has not improved over the last 10 epochs.

3.1.5 Conversion tools

To run a ANN model on the Teensy MCU, it has to be converted. Two different conversion frameworks, k2arm and Tensorflow Lite, were used and evaluated. An overview of some options for deploying ANNs on the Teensy are shown in figure 3.6. Blocks "*Q7 Quantization*" and "*Dynamic range quantization*" are marked yellow, which indicates only partially working implementation exists.

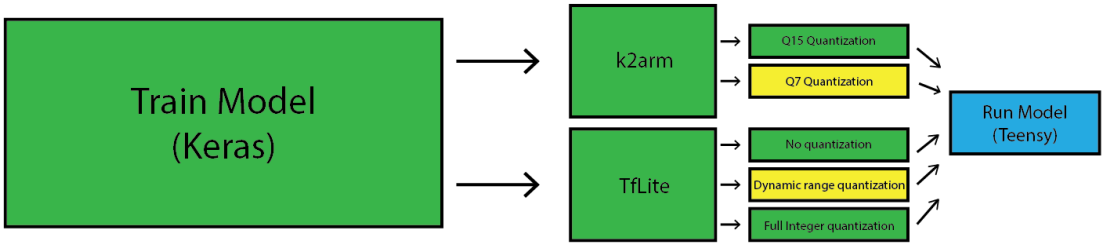


Figure 3.6: Flowchart of training process

Tensorflow lite

There are a few different quantization options when converting a TF lite model and deploying it on a microcontroller. If the target supports floating point numbers, it's possible to deploy the complete unquantized model. In other cases, "Dynamic range quantization" might be of interest. This quantizes only the weight, while the rest still is in float format. Finally, for potentially the fastest performance on MCUs the "Full Integer Quantization" option can be used. When using this option, all numbers are quantized to 8 bit integers.

In all cases, the converted model is saved as a C byte array that is loaded by the interpretor on the target device.

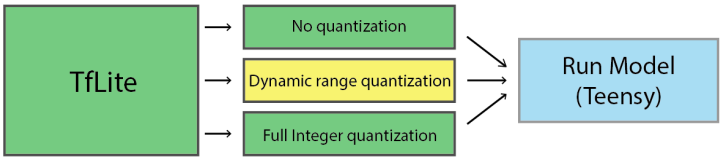


Figure 3.7: Tensorflow Lite Quantization options

The approximated benefits of different quantization techniques is finally summaried in figure 3.8.

Technique	Benefits	Hardware
Dynamic range quantization	4x smaller, 2x-3x speedup	CPU
Full integer quantization	4x smaller, 3x+ speedup	CPU, Edge TPU, Microcontrollers
Float16 quantization	2x smaller, GPU acceleration	CPU, GPU

Figure 3.8: Specifications for the conversion tools provided i the Tensoflow lite toolbox.

k2arm

The k2arm is a different conversion module that was used to directly translate the keras models to C-code. With k2arm it's possible to quantize the model to either Q15 or Q7 numbers. Moreover, all layers and activation functions are directly translated to their function counterpart of the CMSIS-NN library.

The correct QM.N number for weights and biases in the network are selected by iteratively increasing the range (increasing the number of integer bits) until all the values fit in the given range. Quantization of values between layers are based on model accuracy on an evaluation set provided. Finally, correct bias and output shifts are calculated for correct computations between different QM.N formats.

The k2arm conversion tool unfortunately had some stability and accuracy issues for larger network sizes, that were not completely solved. This could however be mitigated to a degree, by switching the evaluation set between the test set, training-set and combined set. Since a greater number of layers exacerbates this issue it was concluded that this problem is related to the inter-layer data quantization.

3.2 Testing the ANN

The ANN was tested on both the MCU and the host computer, including optional post-processing such as majority voting. On the computer performance parameters accuracy, precision, recall and f-score were evaluated for different configurations of the ANN across both the Ninapro dataset and self-obtained data.

On the Teensy MCU, due to the limited memory of the MCU, two approaches were implemented for testing the NN on the device. In the first one, the data was processed and features were calculated on the host-computer. Subsequently, the processed complete test dataset was saved on device memory and classification accuracy was calculated. This allows an isolated comparison of the ANN on the host-computer and the MCU.

The other approach implements continuously streaming the EMG data to the MCU. Preprocessing and feature extraction is then done on the MCU in addition to ANN classification and optional post processing. The benefit of this is to more closely simulating the behavior of

the final system by moving all the calculations to the device. The results from this approach wasn't quantitivly measured as a whole system, but the individual steps were confirmed to be working identical to the computer.

3.3 Teensy Implementation

The program deloyed on the Teensy MCU was written in the Arduino developement environmet in C/C++. Sampling was performed on five analog input pins connected to the electrodes and was implemented with high priority interrupts. This was enabled through the use of the standardlibrary IntervalTimer. The library lets a specific function (in this case, the function controlling the sampling of all the electrodes) run at exact time intervals. Values are continuously inserted into five circular buffers (first in first out), which makes up the window or segmentation of data.

At specified times, corresponding to the amount of overlap desired, feature extraction is performed on the data in the circular buffers. Subsequently, the features serve as the input for the DNN and a movement prediction is made. Each prediction is placed in a further majority voting buffer. At each point in time, the most occuring element in the buffer is decided to be the predicted movement. The movement is then translated, with the help of the robo-limb library, to a certain command that is then sent to the hand prothesis via the CANbus connection.

In addition, the hand prosthesis continously sends feedback about the position of the limbs that can be used for better decision making. This wasn't fully implemented though.

A flowchart of the system can be seen in figure 3.9.

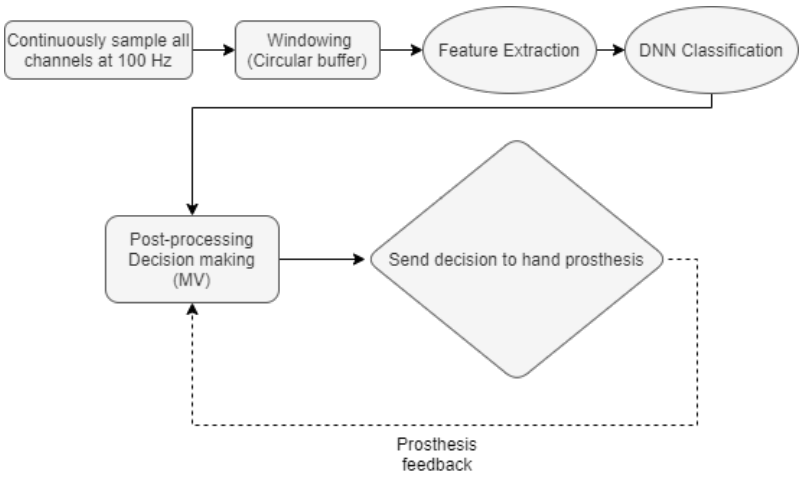


Figure 3.9: Flowchart of implemented Teensy program

3.3.1 Robo-limb library

To ease the communication between the Teensy and the hand prosthesis, an Arduino library (header+source file) was written containing the most important movement commands. The

library was set-up for individual limb control. In short, a message consisting of a four byte payload, and a message ID is sent every time a classification is made. Each controllable limb of the robolimb has a unique mailbox I.D (figure 3.11). Furthermore, the High Word (first 2 bytes) of the payload controls the direction while the Low Word specifies the motor speed with which the movement should be done with. The Robo-limb from touch bionics feature individually controllable fingers and thumb. To control the individual fingers the mailbox I.Ds 0x101,0x102, …, 0x106 were used.

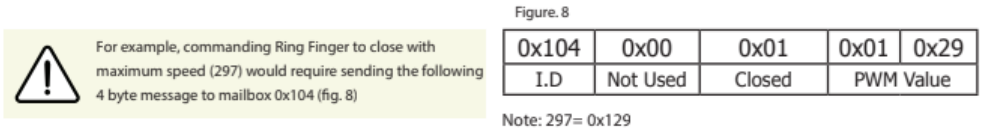


Figure 3.10: Example of the CANbus communication protocol.

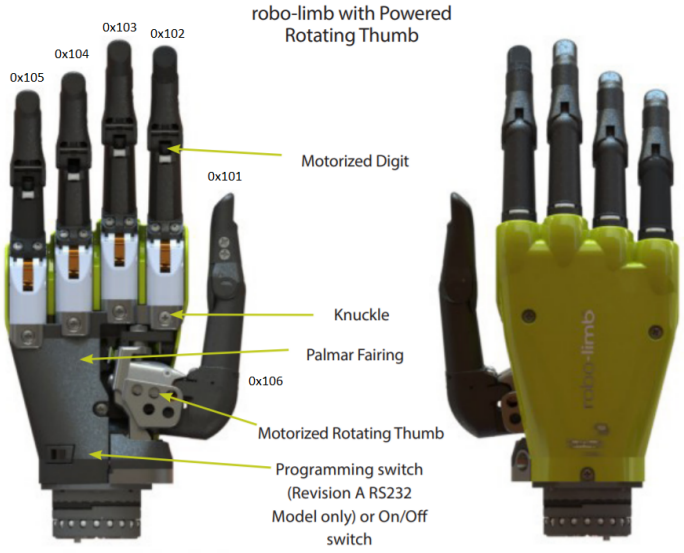


Figure 3.11: Robo-limb hand prosthesis

Chapter 4

Results

This chapter presents the results, both from benchmarks on the publicly available Ninapro DB1 dataset [35][39] as well as on self-obtained data. For convenience, some DNNs are defined in table 4.1 that will recur for different measurements in the result section. For some measurements, only the number of (trainable) parameters are considered. This is defined as sum of all connections layers as well as all biases.

Performance on the Teensy[9] MCU is shown both in the form of comparisons between the result from the different model conversion tools, as well as compared to their unquantized counterparts running on the host-computer.

Table 4.1: ANN architecture definitions

Label name	Layers sizes	Trainable parameters
small	20 | 32 | 13	1 101
medium	20 | 16 | 32 | 32 | 13	2 365
large	20 | 32 | 64 | 64 | 32 | 13	9 453
vlarge	20 | 32 | 64 | 128 | 128 | 64 | 13	36 717

4.1 Inference Performance

4.1.1 Ninapro data

Since the Ninapro database[39] contains datasets of different subjects, it can be useful for benchmarking different ANNs. Results from running four different sizes of DNNs (table

4.1) for every available patient data are presented in figure 4.1. No post-processing is applied to the data and the models are unquantized thus providing a baseline for later comparisons. It can be observed that some subject datasets (5,6,18,19) are more sensitive to changes in DNN size. Others (8, 17) are less affected. A complementary table with some data statistics is available in table 4.2.

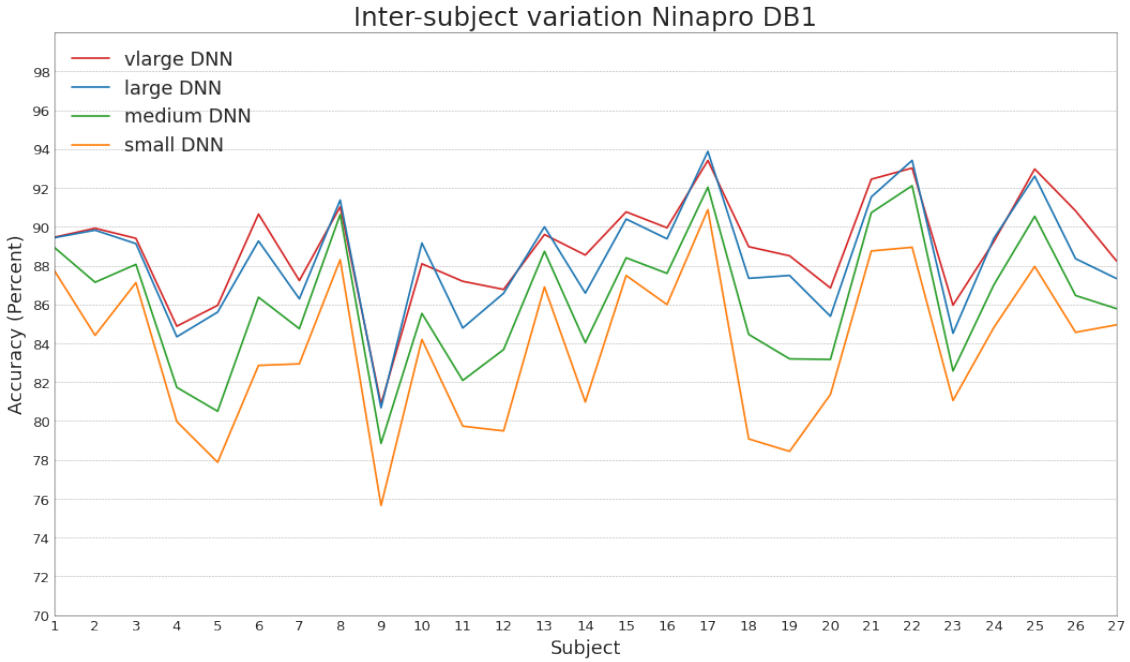


Figure 4.1: Test accuracy variations between different subject in the Ninapro DB1 database[39]. Four different DNNs are evaluated as defined in table 4.1

.

Table 4.2: Table of some data statistics presented in figure 4.1.

	Mean (%)	Median (%)	Max (%)	Min (%)
small	83.79	84.41	90.88	75.65
medium	86.11	86.37	92.11	78.85
large	88.30	89.13	93.88	80.68
vlarge	88.92	89.23	93.41	80.88

4.1.2 Self-obtained data

For precision, recall and f-score in a multiclass classification context, classes 1-10 (finger movements) are considered *positive* and class 0 (no movement) is *negative*. These performance metrics are presented on three different datasets: X train (training dataset), X test (test dataset) and X nomix (complete unshuffled dataset). Furthermore the model is training both with balanced dataset (equal amount of classes) and imbalanced dataset (bias towards class 0). for training on balanced and unbalanced training datasets.

Table 4.3: Performance metrics for balanced and unbalanced training data. Self obtained, 10 individual finger movements.

balanced	Accuracy (%)	Precision (%)	Recall (%)	F-score (%)
X train	92.9	94.3	97.8	96.0
X test	90.2	92.2	96.9	94.5
X nomix	83.3	92.2	78.2	84.6

imbalanced	Accuracy (%)	Precision (%)	Recall (%)	F-score (%)
X train	93.3	91.6	94.9	93.2
X test	91.2	88.4	93.6	90.9
X nomix	92.9	90.9	94.6	92.7

An example of classification accuracy on self-obtained data as a function of the number of parameters in the DNN can be found in figure 4.2. The exact DNN architecture is not considered here, rather only the number of parameters in the DNN. The classification predictions are also presented in an ordered fashion (figure 4.4) as a complement for the confusion matrix in figure 4.3. It should be said that the decrease in accuracy seen in 4.2 for the largest model is most likely due to overfitting, since this measurement was done before early stopping was implemented.

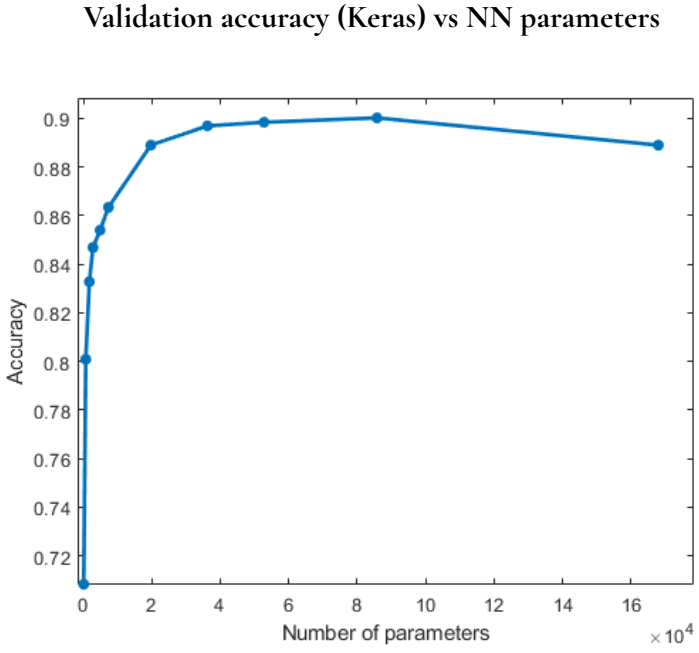


Figure 4.2: Validation accuracy (Keras) vs NN parameters on self-obtained data.

While hard to see in 4.3, the NN has a hard time predicting the 0 class and most commonly confuses classes 1 and 6, and 7 and 10 with eachother. This is more visible in figure 4.4 where all the predictions of the test dataset have been ordered.

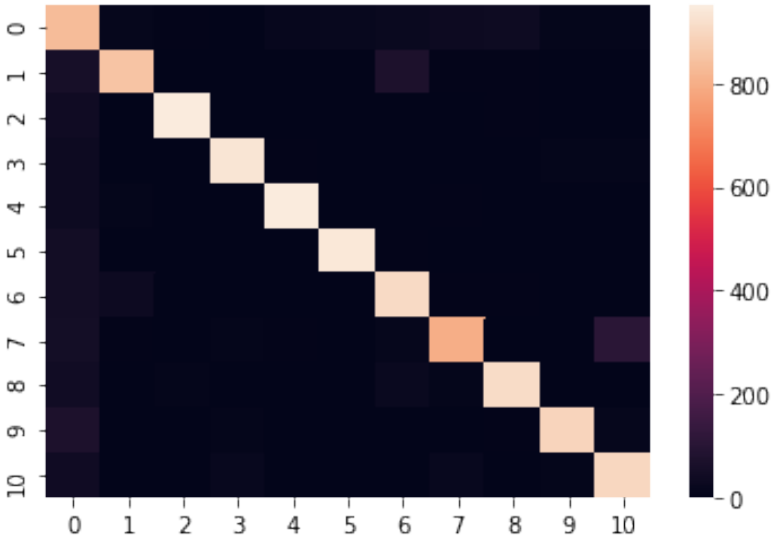
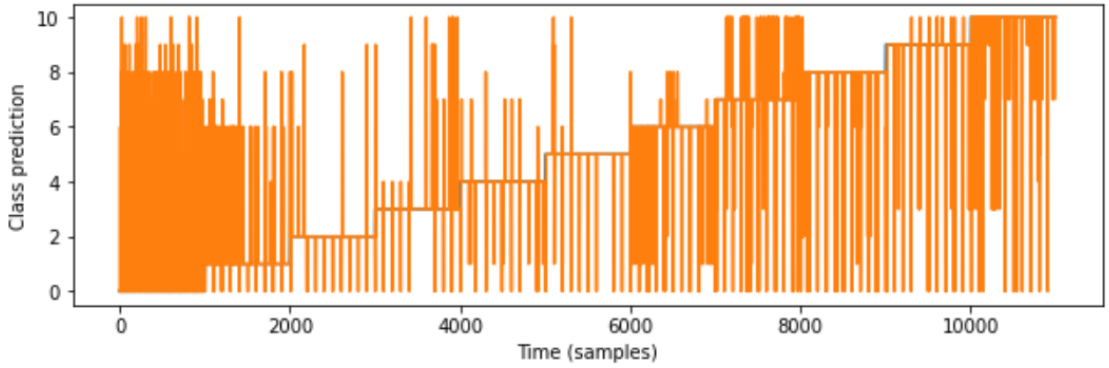


Figure 4.3: Confusion matrix of classifications on self-obtained data.




```
 Occurances of classes :
[1000  999 1000 1000 1000 1000 1000 1000 1000 1000 1000]

Accuracy : 0.898172561141922
```

Figure 4.4: Predicted classifications without post-processing, ordered from 0 to 10

The effect of majority voting post-processing can be seen in figure 4.5. It should be noted though, that the MV accuracy has to be tested on the original unmixed data (the complete training program EMG data). Here, a window of 10 samples was used, and an overlap of 90% meaning the majority vote is calculated for every new sample.

The meaningfulness of the accuracy at higher number of majority votes can be questioned, due to introduction of large delay in the system, thus a limit of 200ms (11 windows) was marked in the figure. When running the complete unquantized DNN model on the Teensy microcontroller, no significant classification accuracy reduction was expected, and this also proved to be the case.

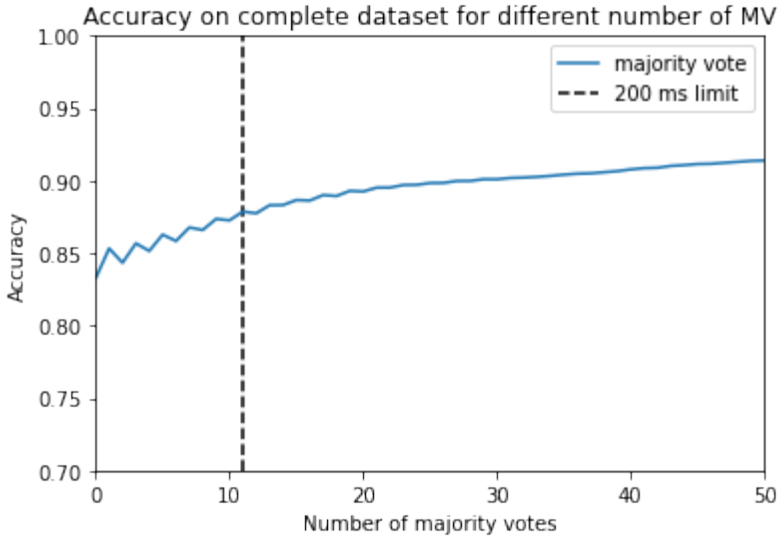


Figure 4.5: Effect of Majority voting post-processing

4.1.3 Comparison: Teensy vs Keras

In this comparison between the unquantized and quantized models, one subject (17), was first selected. Additionally, the differences in accuracy averaged over five subjects are presented in 4.4. The k2arm model had some instabilities which made it hard to quantitatively get values for inference. When models successfully converted with this method, its accuracy matched that of the TFLite Full Integer Quantization method.

Table 4.4: Comparison of accuracy for Unquantized vs Full integer quantized model. Subject 17 in Ninapro DB1.

	TFlite Unquantized (%)	TFlite Int. Quant. (%)	Difference (pp.)
small	90.88	90.41	0.47
medium	92.03	90.98	1.05
large	93.88	92.87	1.01
vlarge	93.41	92.00	1.40

Table 4.5: Difference in accuracy compared to unquantized model, averaged over five subjects in the Ninapro DB1 dataset. Statistics mean, median and Standard Deviation (SD) are considered.

	Mean Acc. Diff (%)	Median Acc. Diff. (%)	SD
small	0.17	0.18	0.17
medium	0.52	0.79	1.00
large	0.65	0.55	0.29
vlarge	0.41	0.77	0.91

4.2 Inference Time

Here, inference times on the Teensy MCU are presented and compared for different conversion tools/quantizations. The following runtime results were obtained by using the micros() function on the Teensy MCU. Figure 4.6 confirms that the use of optimized CMSIS-NN come with an increase in performance speed, both when using k2arm and TFlite quantization.

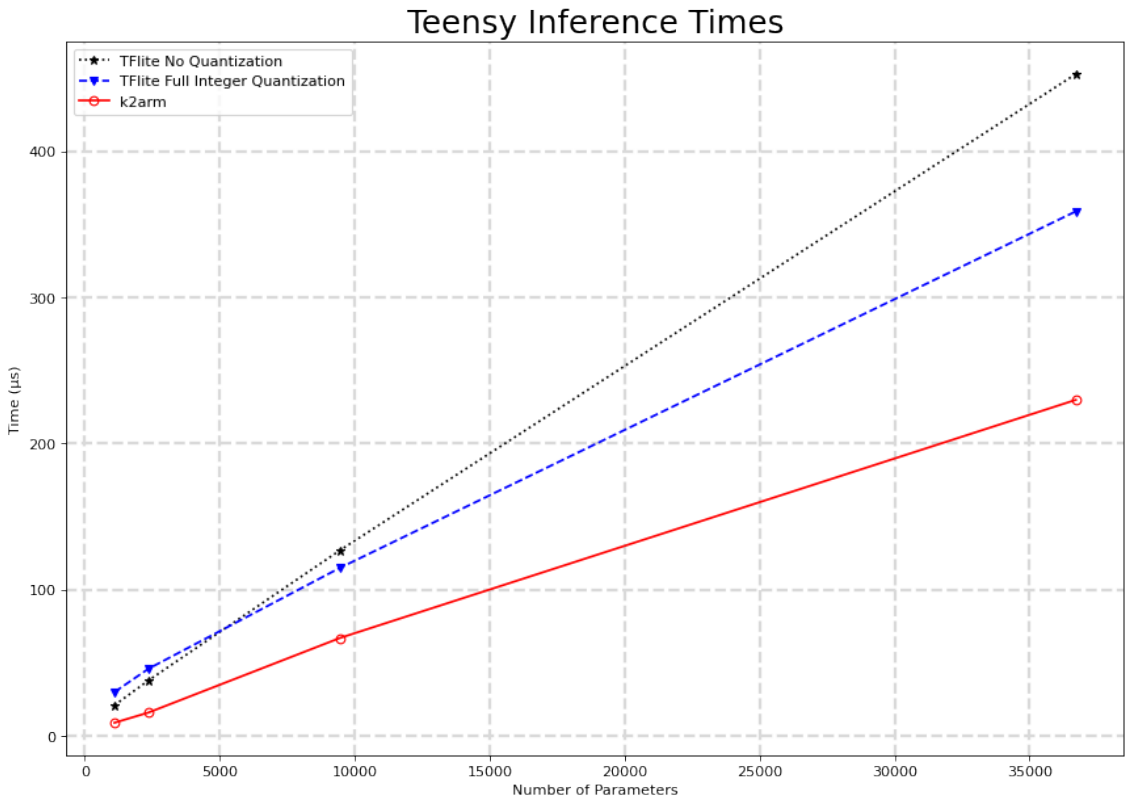


Figure 4.6: Inference times for different conversion methods versus the number of parameters in deployed. The for datapoints for each line corresponds to the networks defined in table 4.1.

Table 4.6: Comparison of Tensorflow lite inference time for unquantized vs full integer quantized models.

	TFlite Unquantized	Full integer quantization	k2arm
small	21 μs	29 μs	9 μs
medium	38 μs	45 μs	16 μs
large	127 μs	111 μs	67 μs
vlarge	454 μs	356 μs	230 μs

4.3 Memory Requirements

Below are the momory consumptions of different models presented. Most of the models implemented in the final complete program used 60-90 % of the dynamic memory on the Teensy 4.0 MCU.

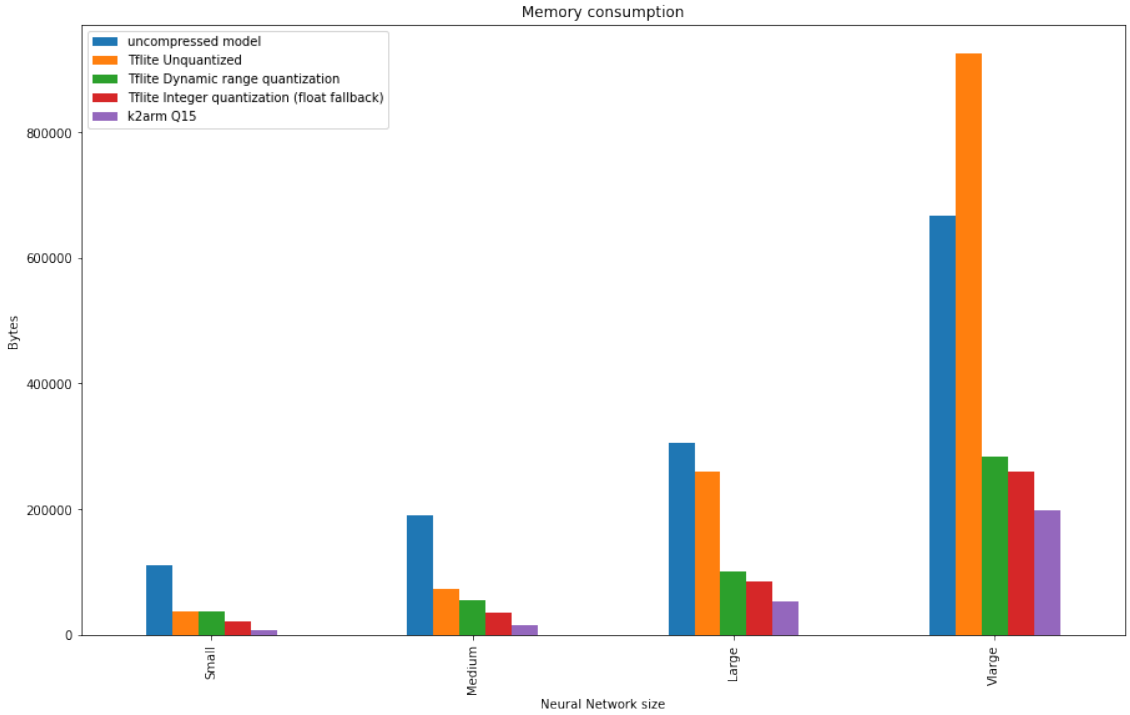


Figure 4.7: Memory consumption for different DNN and model conversion tools.

Table 4.7: Memory consumption for different types of conversion methods on defined DNNs 4.1.

Type of conversion	small	medium	large	Vlarge
Uncompressed model (.pb)	111.4 kB	189.4 kB	306.2 kB	667.7 kB
Unquantized Tflite model	36.5 kB	72.9 kB	250.2 kB	925.5 kB
Tflite Dynamic range quantization	37.1 kB	55.3 kB	100.7 kB	283.7 kB
TfLite (Integer with float fall)	20.7 kB	35.7 kB	85.7 kB	260.5 kB
k2arm model (Q15)	8.1 kB	16.0 kB	53.7 kB	197.6 kB

4.4 Prosthesis control

Simple individual finger control of the prosthetic hand was to some extent achieved, both when streaming Ninapro data to the MCU as well as real-time control with live EMG sampled data. Due to time contraints, no quantitative tests were devised to assess the robustness and ability to control the hand prosthesis.

When streaming EMG data to the MCU, even with post-processing there are frequent miss-classifications hindering the control. This was highly dependent on what delay was deemed acceptable. An example of this is seen in figure 4.8.

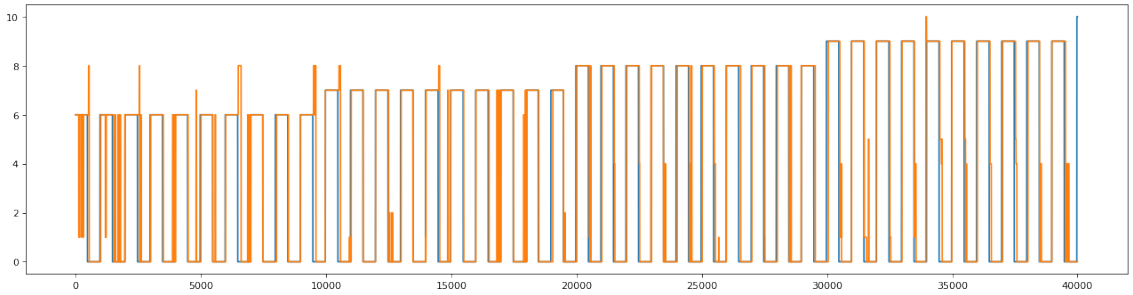


Figure 4.8: Example of majority voting result on complete training protocol dataset. Y-axis are the classification (movement) and X-axis time. The blue line is the true stimulus signal while the orange is the estimated class after postprocessing.

Chapter 5

Discussion

5.1 Prosthesis control

While the accuracy the Artificial Neural Network (ANN) is high, how this is translated into movements leaves a bit to be desired. Additionally, (tactile) feedback, while not necessary under perfect conditions, can be valuable when uncertainties are present [40].

It was noted during training that subsequent redone sessions often yielded in a higher final classification accuracy. This could be due to the fact that I "learned" how to do the movements in a manner that gave the strongest output of the EMG. Also, through some trial and error I got more understanding of where to place the electrodes.

It should also be noted, alot of the performance was based upon the accuracy of the model. This might not be the best performance measure depending on how the final control system is constructed. If it's of importance to predict a movement only when there is one, precision might be a better measure. In hindsight, an simple threshold function of EMG activity before any inference is done would have been a simple solution to elimitate the need to predict the 0 class.

Additionally, other post-processing methods might provide a more robust sytem. In this article[2], a type of FIFO queue where the output only changes when one class label is dominant is proposed.

5.2 Performance

Overall, there were noticable differences both in accuracy, speed and memory consumption present for the same DNN model, but converted and quantized differently. In the tested range, models created with the k2arm conversion tool proved to be superior in both memory footprint and inference speed. This could in part be explained due to a larger overhead in the tensorflow framework.

Another observation is that the "full integer quantized TFlite" model is closer to the "unquantized TFlite" than the "k2arm model" in figure 4.6. It turns out that the "full integer quantized TFlite" model has, if available on the target device, float fallback which potentially is the reason for this anomaly. From, at least the interval tested interval (< 40 000 parameters), the inference time each of quantized models have a roughly linear relationship with the number of parameters and diverge from each other. The implications of this would be a more pronounced difference in inference speed for larger models.

It should be mentioned that the inference speed of the neural network is only part of the contribution towards the total latency in the system. For the tested DNNs and sample rate, even a 90% overlapping data segmentation, i.e. a classification is done every time a new sample is taken once every 10 ms at 100Hz samplerate, the inference speed of every tested model is well below the limit. As such, on the Teensy MCU, for applications similar to this, the bottleneck seems to be the memory required - not the computational time. For different architectures that are potentially more computationally complex such as CNNs and RNNs, the same conclusion cannot immediatly be drawn. Fast inference times allow more post-processing, which could improve the performance of the final interface.

Whether the mean and median accuracy loss of roughly 1 percent (table 4.5) is significant in the final system is hard to quantify. For this, quantitative prosthesis control tests have to be established to determine the final impact.

As for validating the results, the self-obtained data was comparable to the the accuracy of known dataset validating the data acquisition methods, while having half the amount of electrodes. Furthermore, inference times and memory consumption are not dependent on the accuracy of the final network and should thus be the same as long as the architecture stays the same.

5.3 Related Work

Traditionally, the methods used to decode EMG data can be split into pattern recognition (PR) based and non PR based. The focus in this thesis work was in PR-based methods since they have the potential of high accuracy combined with many degrees of freedom control when controlling a hand-prosthesis [41]. An outline of conventional PR-based approaches can be found in [41], and generally includes five steps also used in this work: Pre-processing, Segmentation of data, Feature extraction, Myoelectric classification and Post-processing [41].

For different ANN architectures, an example of a CNN i presented in this article [42]. It is based on viewing the EMG spectrogram as an image which provides the input for the NN.

As for machine learning on microcontrollers, usually referred to as Edge Computing, there are an increasing amounts of articles covering these practices [8].

Complete commercial solutions featuring hand prosthesis controlled by a MCU are at the time of writing few and limited in number of degrees of freedom [43][44]. One example is the device "Sense" created by "Infinite Biomedical Technologies" introduced in 2017 [7].

5.4 Future Work

In this master's these, only DNN type of neural networks were considered. Different deep learning NN architectures should be the subject of further studies. The Tensorflow lite features at the time of writing limited support for deploying CNNs and RNN type of networks on a microcontroller. A CNN type of network eliminates the need to select features and in some state of the art solutions works without preprocessing [2].

Even present state-of-the-art convolutional networks are not deep enough to generalize to multiple individuals, but have the potential to work well when fine-tuning for one subject [2].

Lastly, reinforced learning can be considered as a continuous learning process, possibly implemented on a microcontroller unit.

5.5 Conclusions

Current readily available microcontrollers, paired with open source libraries, might be sufficient and feasible for basic prosthesis control using dense neural networks. There is an inevitable loss of accuracy from quantization, although it is hard to estimate whether this significantly affects performance in the end use case. For Dense Neural Networks (DNNs), inference times are sufficiently fast in a real time hand prosthesis control context, even allowing additional computations such as post-processing.

With that said, neural networks are only as good as the data they are trained with and other factors may have a greater impact on the final result. Furthermore, neither accuracy or other performance metrics are, in isolation, enough to assess the level of prosthesis control achieved in the final system.

References

[1] Silvestro Micera, Jacopo Carpaneto, and Stanisa Raspopovic. Control of hand prostheses using peripheral information. ieee rev biomed eng 3:48-68. *IEEE reviews in biomedical engineering*, 3:48–68, 01 2010.

[2] M. Jafarzadeh, D. C. Hussey, and Y. Tadesse. Deep learning approach to control of prosthetic hands with electromyography signals. In *2019 IEEE International Symposium on Measurement and Control in Robotics (ISMCR)*, pages A1–4–1–A1–4–11.

[3] Mr Kevin J Zuo and Jaret L Olson. The evolution of functional hand replacement: From iron prostheses to hand transplantation. *Plastic Surgery*, 22(1):44–51, 2014.

[4] Maria Hakonen, Harri Piitulainen, and Arto Visala. Current state of digital signal processing in myoelectric interfaces and related applications. *Biomedical Signal Processing and Control*, 18:334–359, 2015.

[5] Upbeat: Augmented reality-guided dancing for prosthetic rehabilitation of upper limb amputees. 2019:2163705. Publisher: Hindawi.

[6] Carles Igual, Luis A. Pardo, Janne M. Hahne, and Jorge Igual. Myoelectric control for upper limb prostheses. *Electronics*, 8(11), 2019.

[7] Alexandre Calado, Filomena Soares, and Demétrio Matos. A review on commercially available anthropomorphic myoelectric prosthetic hands, pattern-recognition-based microcontrollers and semg sensors used for prosthetic control. In *2019 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, pages 1–6, 2019.

[8] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, PP:1–20, 07 2019.

[9] Paul Stoffregen. Teensy® 4.0 development board. `https://www.pjrc.com/store/teensy40.html`. Accessed: 2021-03-17.

[10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.

[11] Jason Brownlee. 4 types of classification tasks in machine learning. `https://machinelearningmastery.com/types-of-classification-in-machine-learning/`. Accessed: 2021-03-17.

[12] Jason Brownlee. 8 tactics to combat imbalanced classes in your machine learning dataset. `https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/`. Accessed: 2021-03-17.

[13] Jason Brownlee. How to calculate precision, recall, and f-measure for imbalanced classification. `https://machinelearningmastery.com/precision-recall-and-f-measure-for-imbalanced-classification/`. Accessed: 2021-04-21.

[14] Todd R. Farrell. Determining delay created by multifunctional prosthesis controllers. *The Journal of Rehabilitation Research and Development*, 48(6):xxi, 2011.

[15] Raúl Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag, Berlin, Heidelberg, 1996.

[16] Levy Boccato, Everton Schumacker Soares, Marcos Fernandes, Diogo Soriano, and Romis Attux. Unorganized machines: From turing's ideas to modern connectionist approaches. *Interanational Journal of Natural Computing Research*, 2:1, 10 2011.

[17] Adam H. Marblestone, Greg Wayne, and Konrad P. Kording. Toward an integration of deep learning and neuroscience. *Frontiers in Computational Neuroscience*, 10:94, 2016.

[18] Alexander Amini. Intro to deep learning. `http://introtodeeplearning.com/slides/6S191_MIT_DeepLearning_L1.pdf`, 2021. Accessed: 2021–02-11.

[19] Vinod Nair and Geoffrey Hinton. Rectified linear units improve restricted boltzmann machines vinod nair. volume 27, pages 807–814, 06 2010.

[20] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. 12 2020.

[21] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2021.

[22] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[23] MS Windows NT kernel description. `https://en.wikipedia.org/wiki/Deep_learning#Definition`. Accessed: 2021-01-10.

[24] Fouad Sakr, Francesco Bellotti, Riccardo Berta, and Alessandro De Gloria. Machine learning on mainstream microcontrollers. *Sensors*, 20(9), 2020.

[25] MS Windows NT kernel description. `https://en.wikipedia.org/wiki/Q_(number_format)`. Accessed: 2021-02-15.

[26] Microchip Technology Inc. High-speed can flexible data rate transceiver. `https://ww1.microchip.com/downloads/en/DeviceDoc/20005284A.pdf`, March 2014. Accessed: 2021–02-11.

[27] Kerry Mills. The basics of electromyography. *Journal of neurology, neurosurgery, and psychiatry*, 76 Suppl 2:ii32–5, 07 2005.

[28] Jiajia Wu, Xiaoou Li, Wanyang Liu, and Z. Jane Wang. sEMG signal processing methods: A review. *Journal of Physics: Conference Series*, 1237:032008, jun 2019.

[29] Muhammad Zahak. Signal acquisition using surface EMG and circuit design considerations for robotic prosthesis. In Ganesh R. Naik, editor, *Computational Intelligence in Electromyography Analysis - A Perspective on Current Applications and Future Challenges*. InTech.

[30] Cinthya Toledo, Josefina Martinez, Jorge Mercado, Ana Martín-Vignon-Whaley, A. Vera, and Lorenzo Leija-Salas. semg signal acquisition strategy towards hand fes control. *Journal of Healthcare Engineering*, 2018:1–11, 03 2018.

[31] S.M. Mane, R.A. Kambli, F.S. Kazi, and N.M. Singh. Hand motion recognition from single channel surface emg using wavelet & artificial neural network. *Procedia Computer Science*, 49:58–65, 2015. Proceedings of 4th International Conference on Advances in Computing, Communication and Control (ICAC3'15).

[32] Ziad Abu-Faraj, Gerald Harris, Peter Smith, and Sahar Hassani. *Human Gait and Clinical Movement Analysis*, pages 1–34. 12 2015.

[33] SparkFun Electronics. 3-lead muscle / electromyographysensor for microcontroller applications. `https://www.digikey.com/htmldatasheets/production/1897318/0/0/1/myoware-muscle-sensor-at-04-001-.html`, 2015. Accessed: 2021–02-11.

[34] S. Negi, Y. Kumar, and V. M. Mishra. Feature extraction and classification for EMG signals using linear discriminant analysis. In *2016 2nd International Conference on Advances in Computing, Communication, Automation (ICACCA) (Fall)*, pages 1–6.

[35] Manfredo Atzori, Arjan Gijsberts, Simone Heynen, Anne-Gabrielle Mittaz Hager, Olivier Deriaz, Patrick van der Smagt, Claudio Castellini, Barbara Caputo, and Henning Muller. Building the ninapro database: A resource for the biorobotics community. In *2012 4th IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics (BioRob)*, pages 1258–1265. IEEE.

[36] Ottobock Scandinavia. 13e200 myobock electrode. `https://professionals.ottobock.se/Products/Prosthetics/Upper-Limb/Adult-Terminal-Devices/13E200-MyoBock-electrode/p/13E200`. Accessed: 2021-06-16.

[37] Flexor carpi radialis picture. `https://de.wikipedia.org/wiki/Datei:Flexor-carpi-radialis.png`. Accessed: 2021-01-15.

[38] M. A. Ozdemir, D. H. Kisa, O. Guren, A. Onan, and A. Akan. EMG based hand gesture recognition using deep learning. In *2020 Medical Technologies Congress (TIPTEKNO)*, pages 1–4. ISSN: 2687-7783.

[39] Ninaweb. Db1 - 27 intact subjects - otto bock electrodes. `http://ninapro.hevs.ch/node/3`. Accessed: 2021-06-16.

[40] Ian Saunders and Sethu Vijayakumar. The role of feed-forward and feedback processes for closed-loop prosthesis control. *Journal of neuroengineering and rehabilitation*, 8:60, 10 2011.

[41] Nawadita Parajuli, Neethu Sreenivasan, Paolo Bifulco, Mario Cesarelli, Sergio Savino, Vincenzo Niola, Daniele Esposito, Tara J. Hamilton, Ganesh R. Naik, Upul Gunawardana, and Gaetano D. Gargiulo. Real-time emg based pattern recognition control for hand prostheses: A review on existing methods, challenges and future implementation. *Sensors*, 19(20), 2019.

[42] Na Duan, Li-Zheng Liu, Xian-Jia Yu, Qingqing Li, and Shih-Ching Yeh. Classification of multichannel surface-electromyography signals based on convolutional neural networks. *Journal of Industrial Information Integration*, 15:201–206, 2019.

[43] Sergey Lobov, Nadia Krilova, Innokentiy Kastalskiy, Victor Kazantsev, and Valeri A. Makarov. Latent factors limiting the performance of semg-interfaces. *Sensors*, 18(4), 2018.

[44] F. Tenore, A. Ramos, A. Fahmy, S. Acharya, R. Etienne-Cummings, and N. V. Thakor. Towards the control of individual fingers of a prosthetic hand using surface EMG signals. In *2007 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 6145–6148. ISSN: 1558-4615.