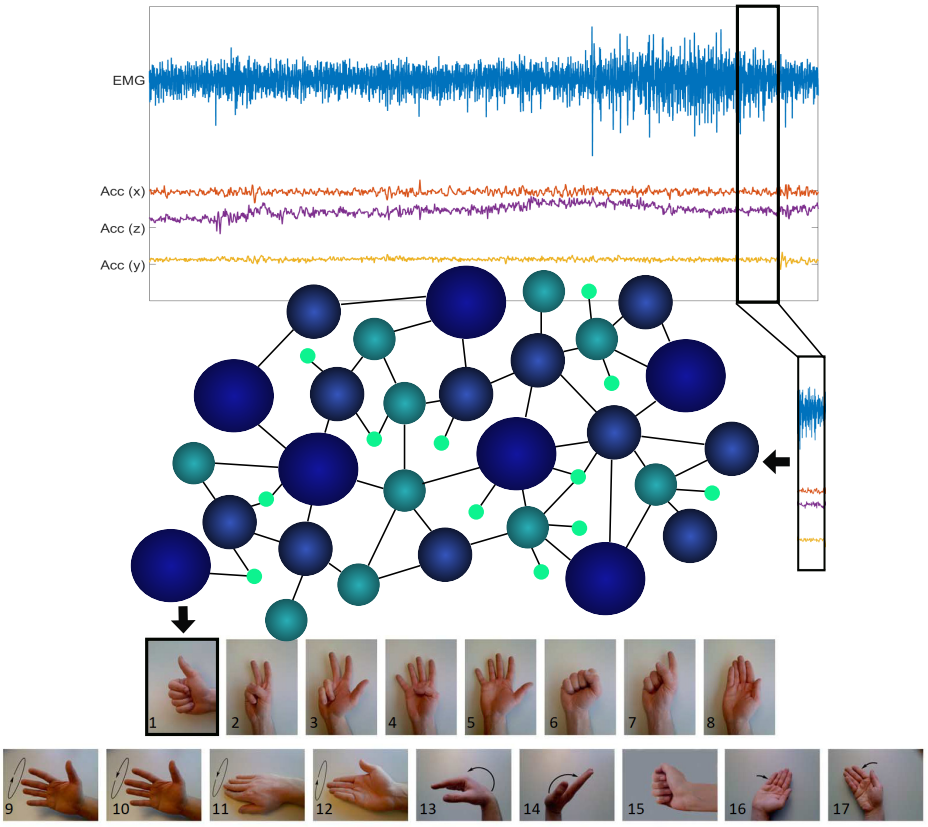
CHALMERS
UNIVERSITY OF TECHNOLOGY



Deep Neural Networks for Myoelectric Pattern Recognition

An Implementation for Multifunctional Control

Master's thesis in Biomedical Engineering

Rita Laezza

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis 2018:01

Deep Neural Networks for Myoelectric Pattern Recognition

An Implementation for Multifunctional Control

RITA LAEZZA

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Signal Processing and Biomedical Engineering
Biomechatronics and Neurorehabilitation Laboratory
Chalmers University of Technology
Gothenburg, Sweden 2018

Deep Neural Networks for Myoelectric Pattern Recognition
An Implementation for Multifunctional Control
RITA LAEZZA

© RITA LAEZZA, 2018.

Supervisor: Max Ortiz Catalan
Examiner: Sabine Reinfeldt

Master's Thesis 2018:01
Department of Electrical Engineering
Division of Signal Processing and Biomedical Engineering
Biomechatronics and Neurorehabilitation Laboratory
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Abstract illustration of deep learning flow. From the input signals at the top, also shown in figures 2.1 and 2.2. To the classification output at the bottom, movement 1 from figure 3.2b

Typeset in LaTeX
Gothenburg, Sweden 2018

Deep Neural Networks for Myoelectric Pattern Recognition
An Implementation for Multifunctional Control
RITA LAEZZA
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Myoelectric control has many applications, such as robotic prosthetics or phantom limb pain treatment. Decoding of muscle signals in order to infer the underlying movement lie at the basis of myoelectric control, for which pattern recognition strategies are commonly used to decode complex movements. The ultimate goal is to create intuitive control systems where an artificial effector is controlled as naturally as a biological limb.

A significant amount of work done in myoelectric pattern recognition research focuses on the pre-processing stage of feature extraction. This aims to reduce the electromyography (EMG) signal to a set of descriptive values. However, these engineered features may not be ideal for pattern recognition applications.

The aim of this thesis was to evaluate the performance of deep learning algorithms for myoelectric control, without prior feature extraction. Three different network architectures were tested: a Convolutional Neural Network (CNN), a Recurrent Neural Network (RNN) and a combination of the two in sequence, noted as CNN-RNN. The RNN models contained Long Short Term Memory (LSTM) units.

Data was obtained from the open source Ninapro database 7 to evaluate the networks performance. The experiments showed that the RNN provided the best result, with a median accuracy of 91.81%, compared with 89.01% for the CNN and 90.4% for the CNN-RNN. Accuracy was averaged across ten different subjects and two groups of movements.

The time an input window took to be classified was approximately 20 ms. Even though these tests were offline, the computation should be fast enough to implement in online control systems as well. Future work should be done to test these algorithms online, since that is a more valuable measure of the algorithms' performance

Keywords: AI, deep learning, ANN, CNN, LSTM, pattern recognition, myoelectric control, EMG.

Acknowledgements

I would like to thank first and foremost my supervisor, Max Ortiz, for giving me the opportunity to work in his amazing research group at the Biomechatronics and Neurorehabilitation Laboratory. I'm grateful for the great work environment that I had throughout my thesis as well as the people I got to know there. Furthermore, I would like to thank Sabine Reinfeldt, my examiner, for her patience when I opted to shift topic within pattern recognition and dive into the field of deep learning. For this, I must also thank my friends whose work motivated me into exploring one of the most interesting and disruptive areas of modern technology.

Por fim, gostaria de aproveitar esta oportunidade para agradecer aos meus pais, não pela tese, mas por todo o seu apoio durante os passados 23 anos. Sem vocês, não teria chegado tão longe. Espero que saibam o quão grata eu me sinto e que passarei o resto da minha vida a tentar demonstrá-lo.

Rita Laezza, Gothenburg, January 2018

Contents

List of Figures

List of Tables

1
Introduction

Myoelectric control is a multidisciplinary field which aims at controlling a certain actuator, based on electromyographic (EMG) signals. Control of robotic prostheses and exoskeletons as well as phantom limb pain treatment software are a few of the most notable applications of this control scheme. These have the potential to give back some of the quality of life lost due to a variety of physical impairments [1].

1.1 Background

Despite several technological advancements in current prosthetic devices, the fundamental control strategies have not significantly improved from the first approaches. Initial control strategies involved on/off control of specific movements, based on carefully chosen thresholds. This meant that, for EMG signals above a certain threshold, the prosthesis would actuate a given movement [1]. Nowadays, a very common approach is to place the recording electrodes over agonist-antagonist pairs of muscles, where each will control opposing movements of a degree of freedom (DoF). This is referred to as direct control (DC), since there is a one to one relationship between electrode and movement. Naturally, this limits the number of movements that can be controlled, specially because the intended user may not have many muscles still intact [2].

In recent years, the research community has shifted its focus to a promising, more intuitive control strategy namely, pattern recognition (PR) based control. By using PR algorithms to recognise the underlying myoelectric pattern for each movement, the number of controllable movements is greatly increased. But the most significant improvement brought by these strategies is the possibility of intuitive control [1, 3]. As opposed to DC strategies, the EMG signals are recorded from a collection of electrodes and are then used to train a PR algorithm. There are numerous types of algorithms that can be implemented for the recognition of movement patterns in EMG signals. They all have the same objective of providing a correct classification, given a desired movement. In this manner, users can simply think about executing a natural movement (e.g. open hand) and by contracting the residual limb the same way they would contract their healthy limb, the signals can be recognised by the PR and that movement will be executed. [3].

Pattern recognition applications typically require a number of transformations to the raw EMG signals before a classification output can be computed. The most common stages found in myoelectric pattern recognition implementations can be seen in Figure 1.1.

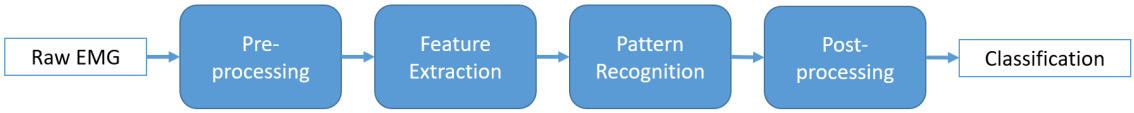


Figure 1.1: Flow diagram of pattern recognition based control. The process begins with the raw EMG signals and finishes with a control output.

Within each stage there can be numerous different data transformations that influence the results down the line. In order to find the best strategy, one would have to try every possible combination of pre-processing, feature extraction, pattern recognition and post-processing methods, which is unfeasible. In the following sections the two centre stages will be introduced, since these have the greatest impact on the outcome of the classification.

1.1.1 Feature Extraction

Feature extraction consists of taking short time windows of the EMG signal and transforming them in some way to obtain a more informative measure. Extensive research has been done on feature selection, of both time domain and frequency domain methods [4]. Additionally, coupled time-frequency domain features based on wavelet transforms have also been explored [5]. New measures, such as cardinality and sample entropy, have been shown to improve classification accuracies [6, 7].

More than just selecting the best features, the effect of feature combinations has also been studied. Perhaps the most commonly used feature selection is the Hudgins' set, which consists of four time domain measures: mean absolute value (MAV), waveform length (WL), zero crossings (ZC) and slope sign changes (SSC). Despite having been proven to be robust when used with Linear Discriminant Analysis (LDA) classifiers, it has been reported that different feature groups provide better results. Furthermore, the best features for one algorithm are not necessarily the best for others [7]. For this reason, it is important to take feature extraction into account when implementing a new PR algorithm.

1.1.2 Pattern Recognition

Similarly to feature extraction techniques, numerous pattern recognition algorithms have been applied to the problem of myoelectric signal classification. Typical PR algorithms include LDA, Multi-Layer Perceptrons (MLP), Support Vector Machines (SVM), k-th Nearest Neighbours (k-NN) and Random Forests, to name a few [8].

The work by Englehart using LDA classifiers has been considered to be the current state-of-the-art approach for myoelectric pattern recognition [9]. There is a considerable number of studies done using LDA in conjunction with the Hudgins' set, which makes this approach valuable for benchmarking purposes [6]. Despite providing robust performances, LDA has several drawbacks. Since an LDA classifier can only produce a single output, for simultaneous pattern recognition, where more than one movement is executed at the same time, a classifier for each movement is needed [10]. Moreover, there is evidence to support that for a larger amount

of movements, the classification accuracy of LDA algorithms is outperformed by MLPs [11].

As well as for LDA algorithms, there have been several studies applying Artificial Neural Networks (ANN), such as MLPs, to myoelectric pattern recognition. More recently, Atzori *et al.* [12]. implemented a Convolutional Neural Network (CNN) for the classification of more than 50 hand movements, obtaining an average accuracy of $66.59 \pm 6.40\%$ for the Ninapro dataset 1. Although the best result, on this particular dataset, was achieved using Random Forests ($75.32 \pm 5.69\%$), the authors claim that the CNN still has potential for improvement.

Unlike the algorithms mentioned above, Recurrent Neural Networks (RNNs) are a class of ANN capable of processing sequential information, by dynamically changing an internal state. They have successfully been used for both time series prediction and classification [13]. Recently, Long Short-Term Memory (LSTM) units and Gated Recurrent Units (GRUs) have become the two prevailing RNN architectures [14, 15]. Since EMG signals are sequential in nature, it is valuable to evaluate recurrent architectures on a myoelectric classification problem. It is important to note that, while CNNs can't handle temporal data explicitly, by windowing time series, they can learn some local dependencies [16].

The motivation behind implementing CNNs for the myoelectric pattern recognition problem comes from their ability to learn features, without the design input from human engineers. Instead, the relevant features are learned directly from data using general-purpose learning procedures. Deep Neural Network (DNN) architectures, such as CNNs and LSTMs, have been making major advances towards solving problems in the field of artificial intelligence. These methods have beaten other machine learning techniques, as well as human performance, in several domains, like image and speech recognition [17].

1.2 Aim

The aim of this thesis is to implement and evaluate Deep Neural Network (DNN) models, for myoelectric pattern recognition, without any prior feature extraction., thus simplifying the processing steps before recognition. Two major architectures will be explored, namely Convolutional Neural Networks and Recurrent Neural Networks, as well as a combination of the two. To that end, the necessary EMG data will be extracted from the Ninapro benchmark database [18].

The evaluation will be based on the classification accuracy of the models when applied to the EMG and accelerometer data obtained from the Ninapro project. This will allow for the evaluation of the pattern recognition algorithms, without the influence of feature extraction techniques, thereby narrowing the research focus.

1.3 Scope and limitations

Perhaps the biggest drawback of pattern recognition based control, when compared with direct control, is that it is less robust. Therefore, a trade-off between functionality and robustness of the control scheme usually arises. By attempting to increase

the number of controllable movements, the reliability of pattern recognition control schemes further deteriorates. For that reason, a great number of publications limit their classification problem to a small number of movements (e.g. six movements and rest). Nevertheless, there is still an ongoing effort to increase the classification accuracy in augmented dexterity applications [1].

Another important avenue in the field of myoelectric pattern recognition is the ability to detect simultaneous movements. However, this thesis will focus on classifying a large number of hand and wrist postures executed in sequence, to achieve what is referred to as sequential control [2].

Finally, depending on the type of electrodes (e.g. surface, intramuscular, etc.) and the measurement configuration (e.g. electrode placement, number of electrodes, etc.) the quality of the signals may vary substantially. There are different filtering techniques to reduce noise from power line interference and movement artefacts. To narrow the scope of the thesis, these factors will not be explored. However, in the Ninapro database, many of these steps have already been performed to improve the quality of the EMG data [1].

1.4 Thesis outline

This thesis is organised into four chapters. The theory chapter begins with a summary of the types of signals to be used for pattern recognition. This is followed by a comprehensive introduction to artificial neural networks and the series of developments that gave rise to deep learning. Moving on from MLPs, there is a description of CNNs and the LSTM architecture. This chapter concludes with a series of relevant theoretical concepts that were implemented in the experimental work.

Once the foundation is set, the methods chapter introduces the methodology used for the work carried out in this thesis. A description of the Ninapro database is included along with the chosen dataset. The hand movements to be classified by the pattern recognition algorithms are also presented. For clarity, there is a data processing subsection that includes all transformations applied to the dataset before feeding it to the machine learning algorithms. Finally, it is also in this chapter that the studied architectures are presented.

The results chapter is divided into two parts. First the performance of the three DNN architectures are compared. Once an architecture is determined to outperform the others, a hyperparameter search is performed to analyse the role of four major parameters on the learning curve. A brief discussion chapter was added following the results, to reflect on the findings and compare with similar research done in this field.

In the final chapter, a short conclusion is presented with reflections on the work carried out throughout the thesis, and some personal comments on what should be done in the future.

2
Theory

This theoretical introduction aims at familiarising the reader with the core concepts surrounding the practical work explained in Chapter 3. By the end, there should be a clear understanding of the signals used for the classification problem and more importantly, the basic machine learning concepts implemented in this thesis.

2.1 Signals

There are two main types of signal to be introduced, namely electromyograms and inertial measurements. While the first is a type of biosignal and is recorded by an electrocardiograph, the latter may be one or a combination of signals from accelerometers, gyroscopes and magnetometers. These components make up a typical inertial measurement unit (IMU).

2.1.1 Electromyography

Electrocardiograms consist of the signals obtained by electromyography. This medical technique to record the electrical activity of the muscles, can be performed using different types of electrodes, and in various electrode configurations, such as monopolar and bipolar. The latter configuration produces a signal with higher signal-to-noise ratio (SNR), coupled with a differential amplifier [19]. For the purpose of this thesis, only surface electromyography will be introduced.

Though surface electrodes are more convenient and less invasive than intramuscular electrodes, they suffer from motion artefacts caused by skin displacement and provide, in general, significantly weaker signals, with peak amplitudes of the order of 0.1 mV to 1 mV. In comparison, using intramuscular needle electrodes provides stronger signals, of up to an order of magnitude higher amplitudes [19].

Another key difference is that needle electrodes are capable of recording action potentials of single motor units, *i.e.* the group of skeletal muscle fibres innervated by a single motorneuron. When the contraction level increases, the signals reach what is referred to as the interference pattern. This interference occurs due to the increase of motor units that get recruited as the contraction level increases. However, in surface electromyography, the signals consist of this interference pattern no matter the contraction level, due to the large area covered by the electrodes [19].

Figure 2.1 shows an example recording of an surface EMG signal, were its stochastic and nonstationary nature is evident.

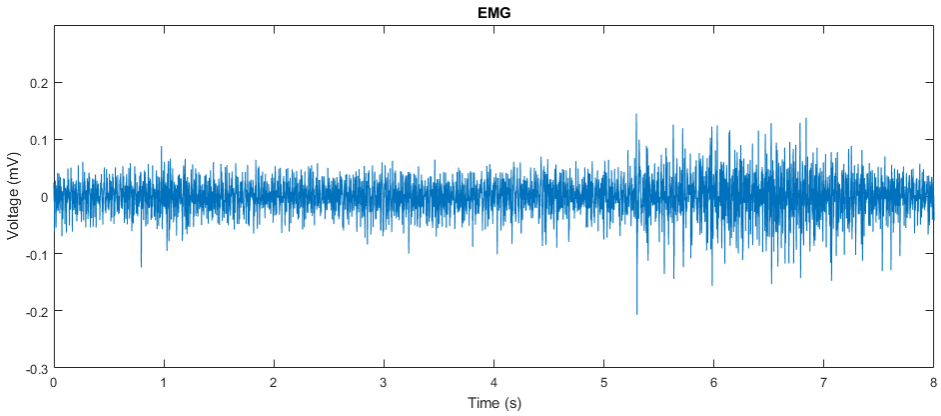


Figure 2.1: Sample EMG signal extracted from the Ninapro database (see Section 3.1). It consists of the first 8 seconds of recording from the first subject, movement and electrode respectively.

2.1.1.1 EMG Rectification

Rectification is a common processing technique for EMG signals, where the negative part of the signal is either made positive (full-wave rectification) or set to zero (half-wave rectification). Some electrodes, such as the well established 13E200 MyoBock from Ottobock, output a Root Mean Square (RMS) rectified signal. Although for these particular electrodes, the rectification is executed analogically at the hardware level, following the work by Atzori *et al.* [12], a similar digital post-processing stage will by applied in this thesis (see Section 3.2).

2.1.2 Inertial Measurements

Inertial measurements are often used for the estimation of orientation relative to a reference frame. Two main sensors are typically employed in inertial measurements units (IMUs), namely the accelerometer and the gyroscope. These can be seen as complementary to each other since the first measures linear acceleration and the latter provides angular speed. Both sensors have distinct problems, the first tends to be noisy and the seconds accumulates errors generating an inevitable drift [20].

Several sensor fusion techniques have been attempted to generate a more reliable orientation estimation, however, this falls out of the scope of the thesis and only raw accelerometer readings will be used as input to the networks. In Figure 2.2 a sample signal can be observed for all three axes. As can be seen from the accelerometer plots, the signals are quite noisy. Note that the y-axis of each plot, indicating acceleration, has no units because the sensor outputs normalised acceleration, with respect to the gravitational acceleration, $g = 9.81\ m/s^2$. In Figure 2.2, the sensor is positioned in such a way that the x-axis is aligned with Earth's gravity, and is thus outputting values close to one. On the other hand, the z-axis is closer to zero, meaning that it is parallel to the direction of acceleration.

There have been several implementations of myoelectric pattern recognition that have been complemented by using inertial measurement data. The work by Krasoulis *et al.* [21], shows that by the concurrent use of EMG and IMU measure-

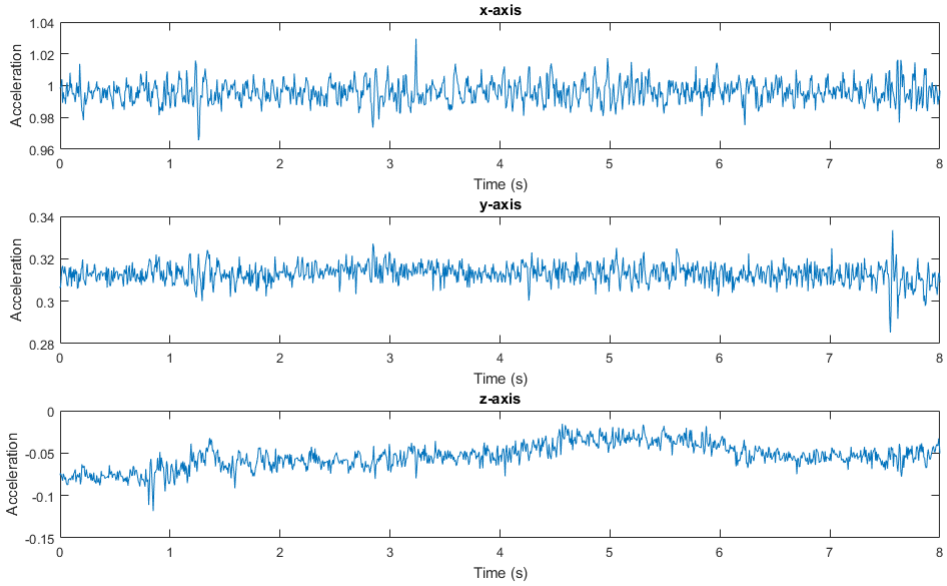


Figure 2.2: Sample accelerometer signals extracted from the Ninapro database (see Section 3.1). It consists of the first 8 seconds of recording from the first subject, movement and sensor respectively.

ments, there is an increase in accuracy with a relatively low number of recording electrodes.

2.2 Artificial Neural Networks

Artificial Neural Networks can be described as computational tools which are composed of interconnected adaptive processing elements commonly referred to as neurons. The structure of the artificial neuron was inspired by its biological counterpart, though it is still quite a distant abstraction. As shown in Figure 2.3, similarly to a biological neuron, there are a set of inputs (synapses) with different weights (dendrites) that get summed together (in the cell body). After the weighted sum, the result is passed through an activation function, thereby exciting the neuron (firing an action potential through the axon). The first model of an artificial neuron, developed by McCulloch and Pitts in 1943 [22], consisted of a mathematical model with binary inputs, restricted weights and a step activation function, g, resulting in binary outputs. An additional threshold value determined the position of the step function [23]. This threshold, θ, is referred to as the bias term.

In 1949, Hebb [24] introduced a theory for neuron adaptation while learning, which provided the first method to train ANNs. The postulate referred to as Hebb's rule, has been summarised as "neurons that fire together, wire together", meaning that if two neurons are activated by the same input, the connection weight between them increase [25].

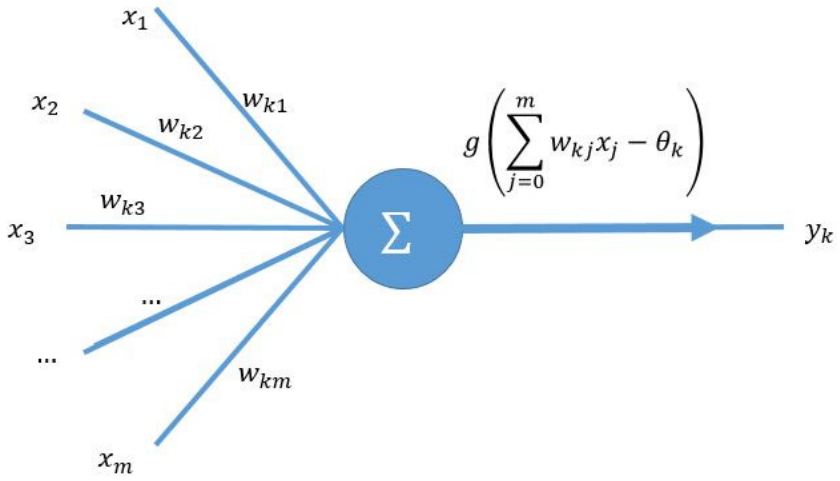


Figure 2.3: Representation of artificial neuron k, with x_j inputs, and w_{kj} weights. The output of the neuron, y_k, is the result of the activation function, g, applied to the weighted sum of all m inputs.

In 1957, Rosenblatt [26] developed the perceptron algorithm, perhaps the simplest type of neural network. With inputs $x_j \in \{-1, 1\}$, the algorithm takes a set of examples, (x, y), to learn how to map inputs to their respective output. Because there is a known output for each input, this is considered a supervised learning method. By iteratively feeding an input and computing the output $y_{predicted} = sign(\mathbf{wx})$, this can be compared to the desired output, y_{target} . For each prediction, the weights are updated according to Hebb's rule [23]:

$$\mathbf{w}^{current} = \mathbf{w}^{previous} + \eta \left(y_{target}^{(\mu)} - y_{predicted}^{(\mu)}\right) \mathbf{x}^{(\mu)} \tag{2.1}$$

where η is the learning rate and μ is the example index. If the predicted output is equal to the target output, then the weights do not change. Otherwise, this learning rule attempts to minimise the error by adjusting the weights after each input-output example [23].

In 1960, Widrow and Hoff [27] created the ADALINE (Adaptive Linear Element) algorithm, which was one of the first industrial applications using ANNs. In addition, it also introduced the "delta rule" as the network's learning rule. It performed the minimisation of the squared error between $y_{predicted}$ and y_{target} to adjust the weights, $\mathbf{w}$ [23]. The error function can be seen in equation (2.2).

$$E(\mathbf{w}) = \frac{1}{2} \sum_{\mu=1}^{p} \left(y_{predicted}^{(\mu)} - y_{target}^{(\mu)}\right)^2 \tag{2.2}$$

Both the perceptron and the ADALINE algorithm have the same output equation, which can be seen in Figure 2.3. However, the index k can be ignored since both of these algorithms constitute a single linear unit, where g is a linear activation function. The main difference between algorithms is how the weights are updated. While the perceptron algorithm used Hebb's rule, the ADALINE algorithm uses the

delta rule, which can be seen in the following equation [23].

$$\mathbf{w}^{current} = \mathbf{w}^{previous} - \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \tag{2.3}$$

Equation (2.3) is the same update rule used for gradient descent. At the end of each epoch (after passing through the entire training set), the weights are adjusted in the opposite direction of the error gradient, noted as $\Delta E(\mathbf{w})$. This delta value can also be multiplied by a learning rate constant, as in equation (2.1). By introducing gradient descent to train neural networks, the ADALINE algorithm was the precursor of current optimisation algorithms used to train ANNs [23].

Despite all of these developments, the perceptron was still a linear classifier and therefore could only learn linearly separable classes. In 1969, the book "Perceptrons" by Minsky and Papert [28], set back the field of neurocomputing, by exposing these limitations. Little notable research was done on ANNs until the mid 1980s, when the backpropagation algorithm was developed, which permitted to train networks with more than one hidden layer [23].

With the advent of backpropagation, the multi-layer perceptron was developed. Though counter intuitive, the MLP consists of layers of perceptrons connected to form a feedforward network, and not a single perceptron with more than one layer, as shown in Figure 2.4. Furthermore, the perceptrons found in MLPs typically have nonlinear activation functions, g [29]. Section 2.2.1.1 will introduce the most commonly used activation functions.

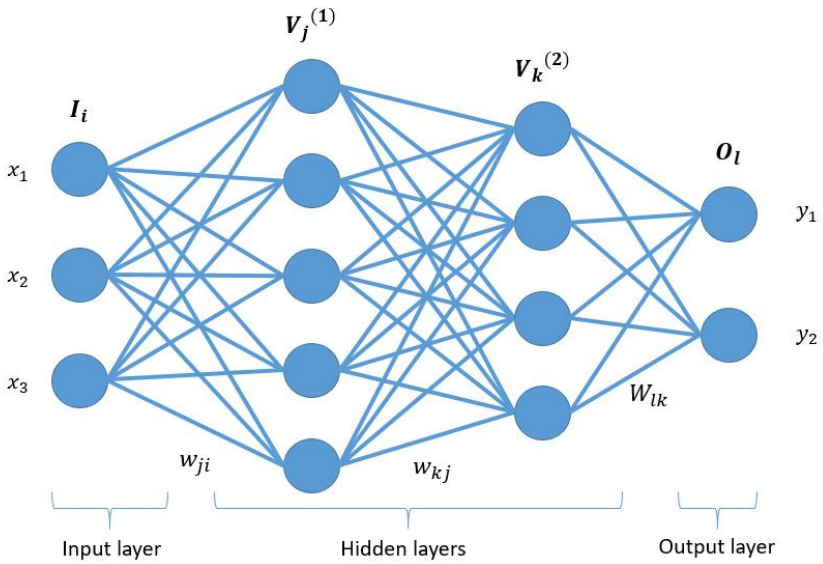


Figure 2.4: Representation of an MLP architecture with three inputs x_i, two outputs y_l and two hidden layers, V_j and V_k, with five and four nodes respectively. The weight matrix connecting the layers, noted as $w_{ji}^{(1)} \in \mathbb{R}^{5\times3}$, represents the connection strength from layer i to layer j, namely the input weights to hidden layer (1). The same applies to matrices $w_{kj}^{(2)} \in \mathbb{R}^{4\times5}$ and $W_{lk} \in \mathbb{R}^{2\times4}$, for hidden layer (2), and output layer, respectively.

2.2.1 Backpropagation

Though initially developed by Werbos in 1971, the backpropagation algorithm gained momentum after it was rediscovered by Rumelhart, Hinton, and Williams in 1986 [30]. This groundbreaking algorithm, allowed for the propagation of errors between target and predicted outputs, backwards through multi-layer networks. Using the chain rule of differentiation, the algorithm iteratively computes the gradients of each layer from the output until the input.

For the MLP represented in Figure 2.4, the training process starts by feeding an input, $I^{(\mu)}$, to the network, passing through the hidden layers until an output, $O^{(\mu)}$, is computed. This is called the forward pass, which is why this architecture is often referred to as feed-forward neural network (FFNN). The forward pass can be written as:

$$y_l^{(\mu)} = g(b_l^{(\mu)}), \qquad b_l^{(\mu)} = \sum_k W_{lk} V_k^{(2,\mu)} - \theta_l \tag{2.4}$$

$$V_k^{(2,\mu)} = g(b_k^{(\mu)}), \qquad b_k^{(\mu)} = \sum_j w_{kj}^{(2)} V_j^{(1,\mu)} - \theta_k^{(2)} \tag{2.5}$$

$$V_j^{(1,\mu)} = g(b_j^{(\mu)}), \qquad b_j^{(\mu)} = \sum_i w_{ji}^{(1)} x_i^{(\mu)} - \theta_j^{(1)} \tag{2.6}$$

The next step is to compute the loss, $\mathcal{L}$, given in equation (2.7).

$$\mathcal{L}\left(O_{predicted}^{(\mu)}, O_{target}^{(\mu)}\right) = \frac{1}{2} \sum_{l,\mu} \left(y_{l,target}^{(\mu)} - y_{l,predicted}^{(\mu)}\right)^2 \tag{2.7}$$

This is a simple error measure known as the mean squared error (MSE) between the predicted and the target output. The loss is computed with respect to all input-output pairs, μ, for gradient descent. In Section 2.3.1, other loss functions will be introduced and Section 2.3.2 includes other optimisation algorithms.

Once the loss has been computed, the gradients can be obtained with respect to the weights between the output and the previous layer, $V_k^{(2)}$.

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W_{lk}} &= \frac{\partial \mathcal{L}}{\partial y_l^{(\mu)}} \cdot \frac{\partial y_l^{(\mu)}}{\partial b_l^{(\mu)}} \cdot \frac{\partial b_l^{(\mu)}}{\partial W_{lk}} \\
&= \frac{\partial \mathcal{L}}{\partial y_l^{(\mu)}} \cdot \frac{\partial y_l^{(\mu)}}{\partial b_l^{(\mu)}} \cdot V_k^{(2,\mu)} \\
&= \frac{\partial \mathcal{L}}{\partial y_l^{(\mu)}} \cdot g'(b_l^{(\mu)}) \cdot V_k^{(2,\mu)} \\
&= \sum_{\mu} (y_{l,predicted}^{(\mu)} - y_{l,target}^{(\mu)}) \cdot g'(b_l^{(\mu)}) \cdot V_k^{(2,\mu)}
\end{aligned} \tag{2.8}$$

The same principle applies for the other layers:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_{kj}^{(2)}} &= \frac{\partial \mathcal{L}}{\partial y_l^{(\mu)}} \cdot \frac{\partial y_l^{(\mu)}}{\partial b_l^{(\mu)}} \cdot \frac{\partial b_l^{(\mu)}}{\partial V_k^{(2,\mu)}} \cdot \frac{\partial V_k^{(2,\mu)}}{\partial b_k^{(\mu)}} \cdot \frac{\partial b_k^{(\mu)}}{\partial w_{kj}^{(2)}} \\
&= \sum_{\mu,l} (y_{l,predicted}^{(\mu)} - y_{l,target}^{(\mu)}) \cdot g'(b_l^{(\mu)}) \cdot W_{lk} \cdot g'(b_k^{(\mu)}) \cdot V_j^{(1,\mu)}
\end{aligned} \tag{2.9}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_{ji}^{(1)}} &= \frac{\partial \mathcal{L}}{\partial y_l^{(\mu)}} \cdot \frac{\partial y_l^{(\mu)}}{\partial b_l^{(\mu)}} \cdot \frac{\partial b_l^{(\mu)}}{\partial V_k^{(2,\mu)}} \cdot \frac{\partial V_k^{(2,\mu)}}{\partial b_k^{(\mu)}} \cdot \frac{\partial b_k^{(\mu)}}{\partial V_j^{(1,\mu)}} \cdot \frac{\partial V_j^{(1,\mu)}}{\partial b_j^{(\mu)}} \cdot \frac{\partial b_j^{(\mu)}}{\partial w_{ji}^{(1)}} \\ &= \sum_{\mu,k,l} (y_{l,predicted}^{(\mu)} - y_{l,target}^{(\mu)}) \cdot g'(b_l^{(\mu)}) \cdot W_{lk} \cdot g'(b_k^{(\mu)}) \cdot w_{kj}^{(2)} \cdot g'(b_j^{(\mu)}) \cdot x_i^{(\mu)}\end{aligned} \tag{2.10}$$

To obtain the bias equations the partial differential equations are solved with respect to θ_{lk}, θ_{kj} and θ_{ji}. These equations can be further simplified by defining the following δ terms:

$$\delta_l^{(3,\mu)} = (y_{l,predicted}^{(\mu)} - y_{l,target}^{(\mu)}) \cdot g'(b_l^{(\mu)}) \tag{2.11}$$

$$\delta_k^{(2,\mu)} = \sum_l \delta_l^{(3,\mu)} \cdot W_{lk} \cdot g'(b_k^{(\mu)}) \tag{2.12}$$

$$\delta_j^{(1,\mu)} = \sum_k \delta_k^{(2,\mu)} \cdot w_{kj}^{(2)} \cdot g'(b_j^{(\mu)}) \tag{2.13}$$

These terms repeat themselves as the the errors get propagated further back, which is why the backpropagation algorithm is computationally efficient even for deeper networks. Finally, the set of simplified backpropagation equations needed to compute the gradients, are summarised in equations (2.14), (2.15) and (2.16).

$$\frac{\partial \mathcal{L}}{\partial W_{lk}} = \sum_\mu \delta_l^{(3,\mu)} \cdot V_k^{(2,\mu)} \qquad \frac{\partial \mathcal{L}}{\partial \theta_{lk}} = \sum_\mu \delta_l^{(3,\mu)} \cdot (-1) \tag{2.14}$$

$$\frac{\partial \mathcal{L}}{\partial w_{kj}^{(2)}} = \sum_\mu \delta_k^{(2,\mu)} \cdot V_j^{(1,\mu)} \qquad \frac{\partial \mathcal{L}}{\partial \theta_{kj}^{(2)}} = \sum_\mu \delta_k^{(2,\mu)} \cdot (-1) \tag{2.15}$$

$$\frac{\partial \mathcal{L}}{\partial w_{ji}^{(1)}} = \sum_\mu \delta_j^{(1,\mu)} \cdot x_i^{(\mu)} \qquad \frac{\partial \mathcal{L}}{\partial \theta_{ji}^{(1)}} = \sum_\mu \delta_j^{(1,\mu)} \cdot (-1) \tag{2.16}$$

These equations can be used by any gradient based optimisation algorithm, to minimise a differentiable loss function and update the weights and biases [31]. For gradient descent the update equations for each weight and bias are given by:

$$\mathbf{w}^{current} = \mathbf{w}^{previous} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{w}^{previous}} \tag{2.17}$$

$$\theta^{current} = \theta^{previous} - \eta \frac{\partial \mathcal{L}}{\partial \theta^{previous}} \tag{2.18}$$

where η is the learning rate.

The backpropagation equations derived for the MLP model in Figure 2.4, may seem trivial, however it is the same fundamental method that is used in more complex models such as CNNs and RNNs, to compute the loss function.

2.2.1.1 Activation function

In the derivation of the backpropagation equations there is no mention of which type of activation function, g, was used. In truth, the only requirement for an activation function is that it has to be differentiable in order to apply backpropagation. These

are a characteristic of the nodes in the network, as seen in Figure 2.3, and act as a transfer function for the weighted sum of the inputs of that neuron, to its output.

Numerous activation functions have been used in the literature. In table 2.1 the three most widely used activation functions are presented.

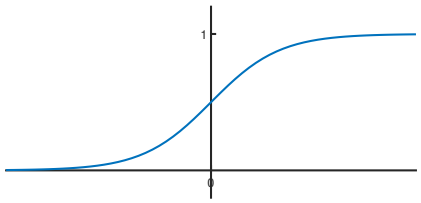
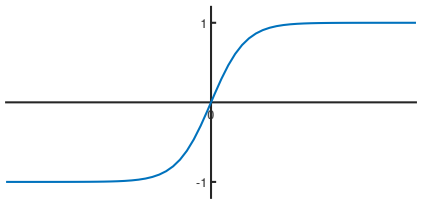
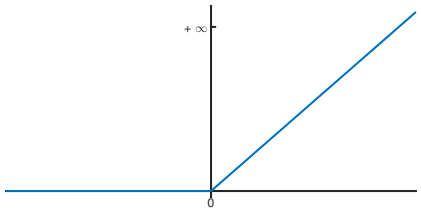
Plot	Formula
	Sigmoid: $g(x) = \frac{1}{1+e^{-x}} \in (0,1)$
	Hyperbolic tangent: $g(x) = tanh(x) \in (-1,1)$
	ReLU: $g(x) = max(0,x) \in [0,+\infty)$

Table 2.1: Activation function plots and respective formulas.

Both the sigmoid function and the hyperbolic tangent have been implemented in ANN nodes as nonlinear activations, for a long time. More recently, the Rectified Linear Unit (ReLU) has been shown to improve generalisation as well as speed up training, and is the most common activation function applied in CNNs [32]. Note that the ReLU is not differentiable at $x = 0$, therefore, for backpropagation, subgradients are computed instead.

2.2.2 Convolutional Neural Networks

Convolutional neural networks are models that revolutionised the field of deep Learning, due to their ability to efficiently process image-like data by resorting to local connections, shared weights, pooling and using many layers [17].

Although CNNs have had the biggest impact in computer vision applications, using 2D images as inputs, for this thesis a 1D CNN will be presented since the relevant data for the desired application consists of a set of one-dimensional input vectors. In Figure 2.5, a graphical representation of the CNN sequence is presented, followed by the description of each layer and the relevant equations.

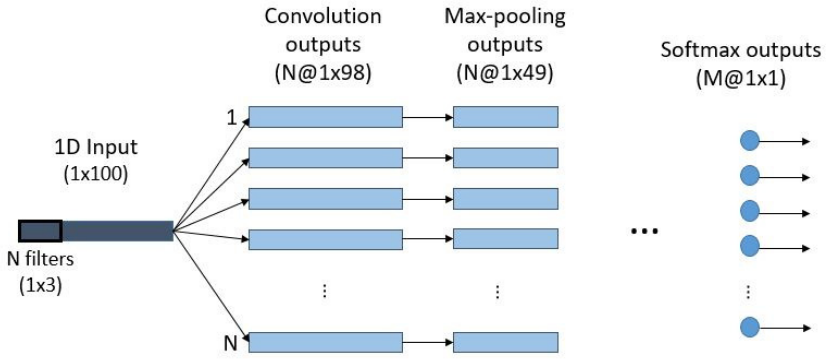


Figure 2.5: Graphical representation of 1D CNN architecture. The input is fed through a series of convolution layers followed by max-pooling layers, to form arbitrarily deep structures. N represents the number of filters in each layer. Here, a stride of one was considered for the convolution step and a stride of two for the max-pooling step. No padding was employed, resulting in a progressive shortening of the input sequence. The classification output is a result of a softmax layer, with M outputs, equal to the number of classes.

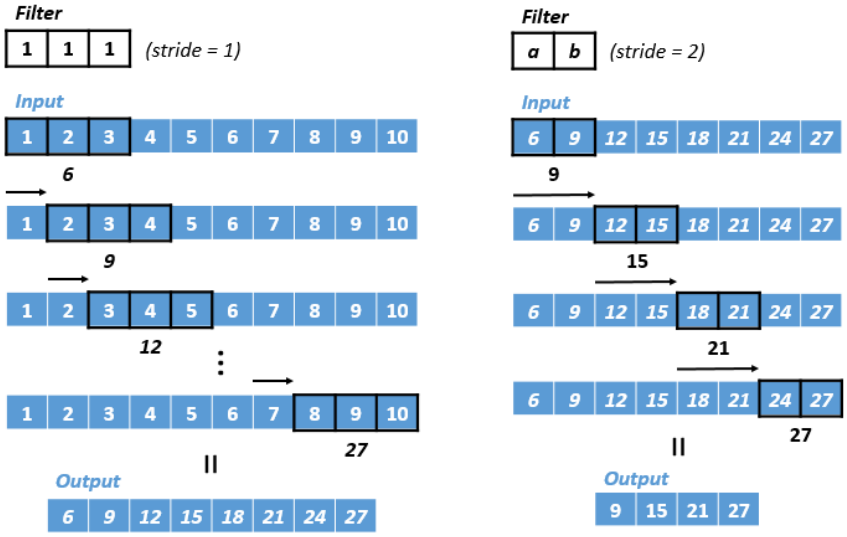
For a better understanding of the principle layers that make up CNNs, two examples were given in Figure 2.6, where 2.6a demonstrates the filtering process of the convolution layers, and Figure 2.6b illustrates the process of max-pooling, also called sub-sampling. However, the term max-pooling is more precise for this specific example since it consists of the $max(a, b)$ function. There are also other sub-sampling functions, *e.g.* mean-pooling.

Each CNN layer has a set of filters which extract locally connected information and pass it to the next layer. The weights from the previous layer are therefore connected to the weights of the following layers. With this arrangement, CNNs are able to detect translation invariant features, which are progressively more detailed as the network depth increases [17]. It is due to this capacity that CNNs have progressively replaced the use of human-engineered feature extractors, such as the ones typically applied for myoelectric pattern recognition.

After a stack of convolution and pooling layers, there can be one ore more fully connected layers before applying the softmax function. This is equivalent to adding an MLP at the end of the CNN. For a softmax activation function, the final layer needs to have the same number of outputs as there are classes. For multi-class problems, the last layer performs a softmax operation which squashes the output into the range of [0,1] in such a way that the sum of all the outputs adds up to 1. This way the classification output can be seen as a probability measure, set by equation (2.19).

$$g(\mathbf{b})_j = \frac{e^{\mathbf{b}_j}}{\sum_{k=1}^{K} e^{\mathbf{b}_k}} \tag{2.19}$$

The above equation consists of output $j \in \mathbb{R}^K$ of the softmax activation function, g, given a K-dimensional vector $\mathbf{b}$ of the output layer. Though the softmax is an activation function, as the examples given in section 2.2.1.1, the derivative has the unique characteristic of being dependant on the output index, since equation (2.19) is computed for all j.



(a) **(b)**

Figure 2.6: Simple example of convolution and max-pooling operations in 1D CNNs. In Figure **a**, the filter was convolved with the input vector, with a stride of 1, meaning that the filter moves in steps of one through the input vector. The centre value is then updated to be the weighted sum of the input segment and the filter weights. In Figure **b**, the max-pooling filter – $max(a, b)$ – with a stride of 2, was applied to the output of the convolution layer, resulting in an output vector of half the size.

2.2.3 Recurrent Neural Networks

When the backpropagation algorithm was introduced, its most exciting application was to train RNNs. These networks differ substantially from MLPs and CNNs, since they possess dynamic memory, in the form of an internal state which can be altered by recurrent connections. Due to this unique feature, they have been shown to be more powerful than FFNN when dealing with temporally dependant signals, such as speech or text [17].

It is possible to view RNNs has a very deep FFNN which has the same shared weights for each layer. This is better visualised from Figure 2.7, where a simple recurrent network is unfolded through time.

It is through this same unfolding that the backpropagation through time (BPTT) algorithm was developed to train recurrent networks. As the name suggests, by treating the recurrent network like a feed-forward network, it becomes possible to apply backpropagation. However, while FFNNs have different parameters that

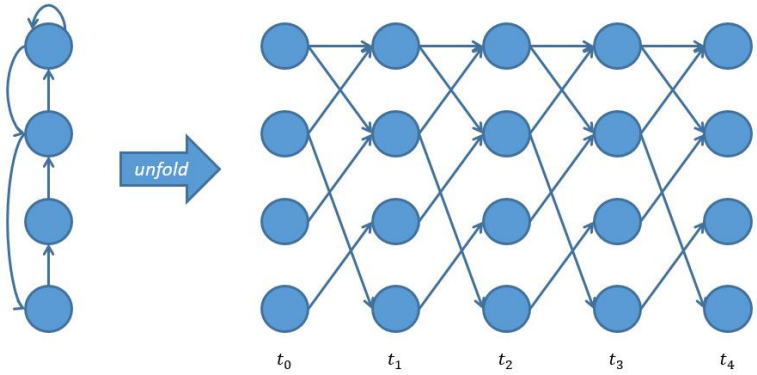


Figure 2.7: Illustration of general recurrent network which was unfolded through time. On the left hand side, the network architecture is shown, with arrows representing the recurrent connections. On the right hand side, these connections are represented in space, where each time step forms a new layer.

have to be updated for each layer, RNNs have the same parameters between each "temporal" layer.

Although RNNs share the efficiency of deep FFNN, they also share the issues that come from large networks, namely the exploding/vanishing gradients problem. With the conventional BPTT, as the gradient computations flow backwards in time, there is a tendency to have increasingly high or low values, leading to either an exploding or vanishing cost. The temporal evolution of the error depends on the size of the weights, in an exponential manner. Therefore, if the gradients explode, this will lead to oscillating weights. On the other hand, if the gradients vanish, it will lead to prohibitively slow learning or even failure [14].

However, the Long Short-Term Memory (LSTM) network architecture, developed by Hochreiter and Schmidhuber [14], enforced constant, non-exploding and non-vanishing error flow, thus solving the problem.

2.2.3.1 Long Short-Term Memory

The architecture of a vanilla LSTM cell is depicted in Figure 2.8. The term vanilla was added since variants of the same architecture have been created, *e.g.* LSTM with peephole connections. As shown in the schematic of the LSTM cell, the output is connected back to the block input and all of the gates, through recurrent (lagged) connections [33]. For clarity, Figure 2.9 shows a graphical representation of two time steps of the recurrent update.

The architecture presented here consists of a revised version of the initial LSTM, which did not yet include a forget gate. With the addition of this gate, the network becomes able to learn continuous tasks, since it can at times forget and thus release memory [33].

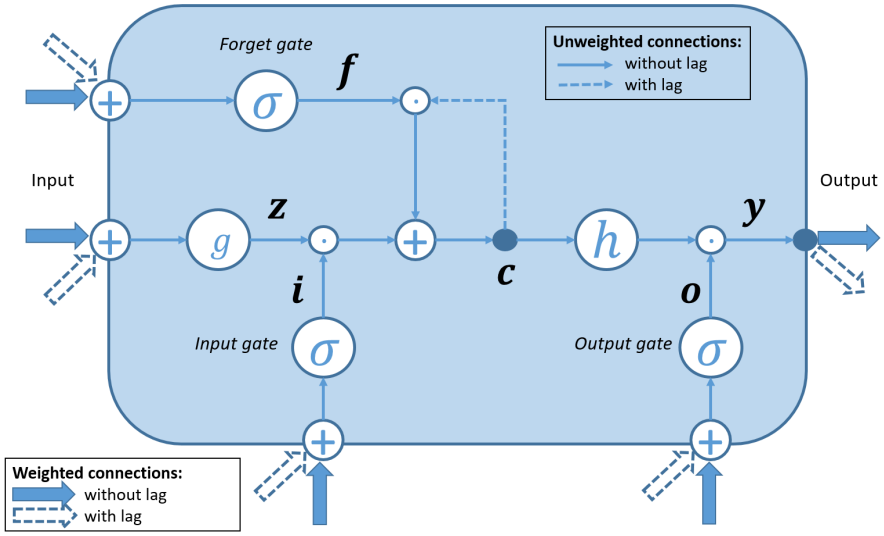


Figure 2.8: Schematic of vanilla LSTM architecture. The recurrent connections are noted by arrows with time-lags. The gate activation functions, noted by g, are always sigmoids, while the input g and output h usually have *tanh* activation functions.

The formulas for the forward pass in a vanilla LSTM are given in equations (2.20) through (2.25). For clarity, the notation is vectorised and follows the order in Figure 2.8.

$$z^{current} = g\left(W_z x^{current} + R_z y^{previous} + \theta_z\right) \qquad input \qquad (2.20)$$

$$i^{current} = g\left(W_i x^{current} + R_i y^{previous} + \theta_i\right) \qquad input\ gate \qquad (2.21)$$

$$f^{current} = g\left(W_f x^{current} + R_f y^{previous} + \theta_f\right) \qquad forget\ gate \qquad (2.22)$$

$$c^{current} = i^{current} \odot z^{current} + f^{current} \odot c^{previous} \qquad cell\ state \qquad (2.23)$$

$$o^{current} = g\left(W_o x^{current} + R_o y^{previous} + \theta_o\right) \qquad output\ gate \qquad (2.24)$$

$$y^{current} = o^{current} \odot h\left(c^{current}\right) \qquad output \qquad (2.25)$$

Here $\odot$ denotes point-wise multiplication of two vectors. W and R are the weights matrices. They are noted differently since W consists of the input weights rectangular matrices and R are square recurrent weight matrices.

2.3 Learning

As previously mentioned, backpropagation is the fundamental principle which allows deep networks to learn. However, it is merely a computation step taken by a certain optimisation algorithm, such as gradient descent. It is the optimiser that determines how the network updates the weights towards the minimisation of the loss function. In section 2.3.1 there will be a short description of the relevant loss functions, followed by the optimisation algorithm used in this thesis (section 2.3.2), and finally some of the regularisation strategies to improve the network's generalisation capabilities, in section 2.3.3.

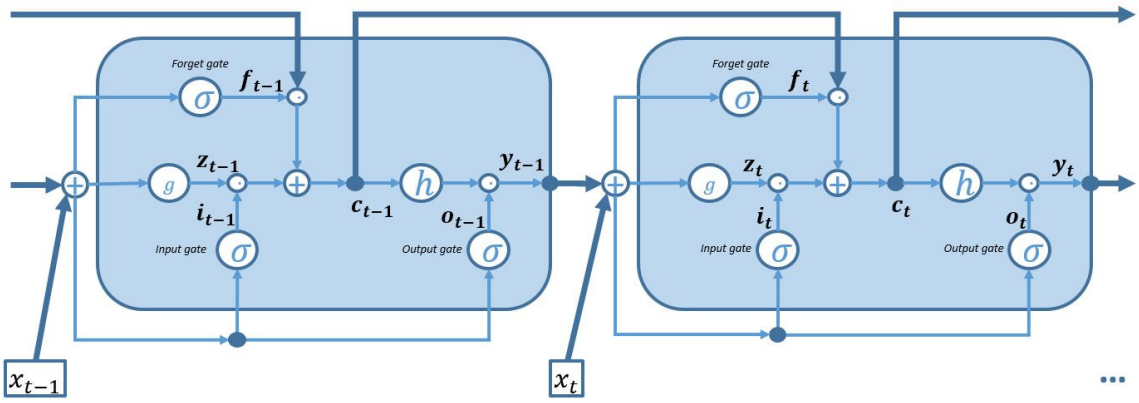


Figure 2.9: Schematic of two time steps of the LSTM update. The notation $t-1$ represents the *previous* time step and t the *current* time step. With a single LSTM cell, there is a sequence of inputs, x and outputs y, while typically only the last output is considered for classification applications.

The goal of the ANN training process is to learn how to map the input data to the output data, so that a high classification accuracy is reached. This is also referred to as achieving a **low bias** network. Conversely, if a network fails to learn and produces low accuracies, it has a **high bias**. However, for a classifier to be useful, besides having a low bias, it needs to perform equally well on unseen data as on the training data, and therefore have **low variance**. Since the training data usually consists of a small sample from a certain pool of data, ANNs can learn the entire sample with an accuracy of 100%, and then perform poorly with other samples from the same pool. When this occurs, the network is said to have **high variance** [34].

Further, when a network is said to have high bias, it means that there was **underfitting** to the training data, and that it could not learn efficiently. On the other hand, when there is high classification accuracy and the test accuracy is significantly lower, the network may suffer from high variance, meaning that **overfitting** occurred [34, 35].

2.3.1 Loss Function

There are several possible loss functions to be employed for training neural networks. The choice of loss depends on the problem at hand. For regression problems, where the network needs to learn a specific function, the appropriate loss would be similar to the one in equation (2.7), based on the distance between input and output (residual). However, for classification problems, such as myoelectric pattern recognition, there are more efficient entropy-based loss functions.

Depending on the desired output format of the classification problem, it may be better to use binary or categorical cross-entropy, where the first is merely a special case of the second. In problems where there are only two output classes, binary cross-entropy is the obvious choice. However, if there is a multi-class problem where multiple outputs can be present for a given input, it becomes a multi-label situation. In such cases, binary cross-entropy can be used. However, in order to use a binary

loss with more than two output nodes, the output activation cannot be a softmax function, but an alternative such as the sigmoid function. This will result in a classifier made up of a group of binary classifiers.

The binary cross-entropy loss function is presented in equation (2.26), with $\hat{y}$ representing the output of the network and y the target value $\in 0, 1$.

$$\mathcal{L}(\hat{y}, y) = -\left[y log(\hat{y}) + (1-y)\, log(1-\hat{y})\right] \tag{2.26}$$

where $\hat{y}$ can be seen as the probability of the output being 1, and $(1-\hat{y})$ the probability of the output being 0.

In order to employ such a loss function to a multi-class problem, it is necessary to encode the outputs into binary arrays, with size equal to the number of classes, K. For example, in a problem with five possible outputs, the one-hot encoded vector for the second class would be: $\mathbf{y_2} = \{0, 1, 0, 0, 0\}$. In equation (2.27), the categorical cross-entropy loss is presented.

$$\mathcal{L}(\hat{y}, y) = -\sum_{k=1}^{K} y_k log(\hat{y}_k) \tag{2.27}$$

where y_k is the k^{th} output node and log indicates the natural logarithm, which is why this is also sometimes called the log loss. Note that the output represents a probability distribution computed by the softmax layer.

Upon choosing the appropriate loss function, the optimisation goal can be written in the form of a loss function to be minimised with respect to the whole dataset:

$$J\left(w^{[1]}, \theta^{[1]}, ..., w^{[L]}, \theta^{[L]}\right) = \frac{1}{p}\sum_{\mu=1}^{p} \mathcal{L}\left(\hat{y}^{(\mu)}, y^{(\mu)}\right) \tag{2.28}$$

where $\mathcal{L}$ may represent any of the loss functions introduced, or even another specially designed for a certain problem, as long as it is differentiable. Note that the cost function is minimised with respect to all network parameters and averaged by all p examples. This is called batch gradient descent because the whole batch of training examples is used to compute the cost function.

2.3.2 Optimisation Algorithm

Though a gradient descent step will always lead to a decrease of the cost function, propagating the entire training set trough the network to obtain a single step is computationally expensive. Furthermore, if the gradient gets smaller, the algorithm becomes increasingly slow. Finally, for deep neural networks it may even be unfeasible to keep all that data in the main memory. Consequently, the most extensively used optimisation algorithm for training large networks is stochastic gradient descent (SGD).

Though the update rule for SGD is essentially the same, the network parameters get updated after each input-output example, similarly to the perceptron algorithm. Additionally, the inputs are ramdomly shuffled before every epoch. However, due to its stochastic nature, the cost is no longer guaranteed to decrease monotonically. Nonetheless, this may come as an advantage, since gradient descent will get stuck in local minima and the stochasticity provides a way to escape them.

Perhaps as a way to include the best of both algorithms, the mini-batch training method was developed. By dividing the training set into smaller batches and using these to update the network, better estimates of the gradient are obtained. It is easy to understand that with a batch size of 1, the mini-batch algorithm is simply SGD, and with a batch size equal o the training set it becomes gradient descent. By using the mini-batch SGD algorithm, there are two parameters that need to be chosen for optimisation, namely the **learning rate**, η, and the **batch size**.

Recently, more efficient stochastic optimisation methods have been developed, such as RMSprop and AdaGrad. The optimisation algorithm implemented in this thesis was the *Adam* algorithm, which was developed by Kingma and Ba [36] and has shown to be a comparably better optimiser since it attempt to combine the advantages of the other two algorithms. The name derives from Adaptive moment estimation, because the algorithm estimates the first and second moments of the gradients and then computes individual adaptive learning rates, for the update rule. This rule can be seen in equation (2.29).

$$\mathbf{w}^{current} = \mathbf{w}^{previous} - \alpha \frac{\hat{m}^{current}}{\sqrt{\hat{v}^{current}} + \epsilon} \tag{2.29}$$

Where $\hat{m}$ and $\hat{v}$, are estimates of the first and raw second moments of the gradients, noted as g. The are computed in accordance with equations (2.30) and (2.31).

$$m^{current} = \beta_1 \cdot m^{previous} + (1 - \beta_1) \cdot g^{current} \tag{2.30}$$

$$v^{current} = \beta_2 \cdot v^{previous} + (1 - \beta_2) \cdot (g^{current})^2 \tag{2.31}$$

These equations consist of exponential moving averages of the gradient, m, and the squared gradient, v. Where the first is an estimate of the 1^{st} moment (the mean) and the latter is the 2^{nd} raw moment (the uncentred variance). The hyperparameters β_1 and $\beta_2 \in [0, 1)$ control the rates of exponential decay of each moving average. However, because these vectors are initialised to zero, they are biased towards zero. The algorithm them uses bias correction terms, shown in equations (2.32) and (2.33), to compute $\hat{m}$ and $\hat{v}$.

$$\hat{m}^{current} = \frac{m^{current}}{(1 - \beta_1^c)} \tag{2.32}$$

$$\hat{v}^{current} = \frac{v^{current}}{(1 - \beta_2^c)} \tag{2.33}$$

where c is the current iteration number. This means that at each step, both β^c parameters get increasingly smaller and this initialisation bias correction has a progressively lower impact on the update [36].

The authors included in their paper the set of advised default parameters: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ [36].

2.3.3 Regularisation Methods

The goal of regularisation methods is to improve the generalisation ability of the network, by reducing the difference between the training error ans the test error. This difference can be considered as the generalisation error, and there are several strategies to reduce it [34].

2.3.3.1 L2-Regularisation

The main regularisation method to be presented here is the L2-regularisation of the cost function. This works by adding an extra penalty term to the cost function, as can be seen in equation (2.34), where $\mathcal{L}$ is the loss function, *e.g.* categorical cross-entropy.

$$J\left(w^{[1]}, \theta^{[1]}, ..., w^{[L]}, \theta^{[L]}\right) = \frac{1}{p}\sum_{\mu=1}^{p}\mathcal{L}\left(\hat{y}^{(\mu)}, y^{(\mu)}\right) + \frac{\lambda}{2p}\sum_{l=1}^{L}\left\|w^{[l]}\right\|^2 \tag{2.34}$$

Where the norm is computed as in equation (2.35). Corresponding to the Frobenius norm of a matrix, it is equal to the sum of squared elements of a matrix.

$$\left\|w^{[l]}\right\|^2 = \sum_{i=1}^{n^{[l+1]}}\sum_{j=1}^{n^{[l]}}\left(w_{ij}\right)^2 \tag{2.35}$$

The cost function is minimised with respect to all weights and biases of the network with L layers. Because of the penalty that will increase the cost if the weights are too large, this regularisation method will force the weights to be small. For this reason, it is sometimes referred to as *weight decay*. The **regularisation parameter**, λ, is therefore another hyperparameter to be tuned.

2.3.3.2 Dropout

In recent years, a new type of regularisation method has been introduced, namely the dropout method. It has shown to reduce overfitting at a low computational cost. By sequentially dropping out a set of nodes, it becomes possible to combine exponentially many smaller architectures within the same model. The choice of which nodes to drop is random, and determined by a probability parameter [35]. Though it seems to have become common practice to apply dropout on deep CNN models, it will not be implemented in this thesis.

2.3.3.3 Data Augmentation

One of the key requirements for deep learning applications is the availability of large quantities of data. Deep models, with more trainable parameters than number of training examples can still have relatively low generalisation error [34]. Nonetheless, when there is an insufficient amount of training data, the models tend to overfit to that data, and end up having bad generalisation capability. Though the ideal is to get additional data, when that is not possible, alternative regularisation strategies can be employed to expand from the available data. Such methods are referred to as data augmentation.

Though more sophisticated methods exist, a simple way of obtaining more training data is to modify the available data. In image classification problems this is done by applying geometric transformations and adding noise to the available images [37]. In one-dimensional signals, adding white Gaussian noise can produce a similar effect, by generating significantly different signals. This is the strategy employed by Atzori *et al.* [12] when using the Ninapro database for a CNN implementation.

2.3.3.4 Early stopping

Finally, the simplest strategy to make sure that the network can generalise well is to split the dataset into three sets: training, validation and test set. The reasoning behind this is to be able to use the largest portion of the data for training the model, but keep a smaller sample of it to track the network's performance on unseen data, *i.e* on the validation set. During training, if the error of the training set keeps decreasing, but the validation error stagnates or even rises, it means that overfitting has begun to occur. Early stopping is simply to check when or if this occurs and to stop training then.

The goal of the test set, independent to the validation set, is to make sure the training was not biased towards the validation set. This is due to the hyper-parameters (*e.g.* number of layers and nodes) that can be fine-tuned based on the learning curve of the validation set . This also includes the stopping point, meaning the epoch at which the learning stopped, since it is based on the accuracy for the validation data. The test set provides a truly unseen group of examples that will give an unbiased evaluation of the model's performance.

2.4 Performance Measures

The most straight forward way of evaluating the performance of a machine learning algorithm is to measure the classification accuracy, which is equal to the fraction of correct classifications with respect to the total number of classifications. However, this measure can give rise to deceivingly high performance results. To illustrate this, consider a binary classification problem, where 80% of test examples are positive and the rest negative. If the classification algorithm were to classify all test examples as positive, the accuracy would be 80%, even though it effectively could not recognise the negative examples. This situation arises from an unbalanced class distribution, but can be avoided by having equal number of examples for each class.

To continue with the example above, let us introduce a few important concepts by looking at the confusion matrix in table 2.2.

Table 2.2: Confusion matrix of a binary classification problem.

$\mathbf{y}_{\mathbf{predicted}}$ / $\mathbf{y}_{\mathbf{target}}$	**Positive**	**Negative**
Positive	TP	FN
Negative	FP	TN

For binary classification problems, the confusion matrix gives a clear summary of the algorithm's performance. The terms TP, TN, FP and FN refer to true positives, true negatives, false positives and false negatives, respectively. From these values many different accuracy measures can be extrapolated, such as precision (true positive rate) and recall (true negative rate) [38].

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \times 100 \tag{2.36}$$

$$Precision = \frac{TP}{TP + FN} \times 100 \tag{2.37}$$

$$Recall = \frac{TN}{TN + FP} \times 100 \tag{2.38}$$

The measures from equations (2.37) and (2.38) can further be used for more complex measures such as the F1-score (harmonic mean of precision and recall).

$$\text{F1-score} = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.39}$$

However, the F1-score has some drawbacks, especially when dealing with unbalanced classes. A measure that is more robust to class imbalance is the Matthew's Correlation Coefficient (MCC) [38]. Equation (2.40) shows the formula for computing the MCC, using values from the confusion matrix.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{2.40}$$

The MCC will be 1 for a perfect classifier and 0 for a random classifier. Additionally, an inverse classifier will have an MCC of -1, which gives another dimension to the performance measure. This score can be applied for multi-class classification problems however, all classes must be present in the classification output, otherwise the measure becomes undefined [38, 39] .

3
Methods

In this chapter, there will be a detailed description of the Ninapro data, followed by the processing steps and finally the experiments. The aim is to clearly state all processing executed before implementing the pattern recognition algorithms in order to recreate the input to the models. The DNN models to be tested are a CNN, an RNN and a combination of the two types of networks, referred to as CNN-RNN. More details about the architectures as well as the experimental setup are presented in section 3.4.

3.1 Dataset

The EMG data used for this thesis was obtained from the Ninapro repository, which has the aim of being a benchmark database for the field of non-invasive adaptive hand prosthetics [18, 40]. To date, the repository includes seven different databases, with distinct acquisition protocols and subjects, but following a general guideline. The most recent database, is the only to include both intact and amputee subjects, which is why it was chosen as the source of data to be used here [21]. Though it would have been possible to collect new data in the lab, it has become common practice to resort to publicly available databases, such as the Ninapro, for benchmarking deep learning algorithms [12]. If new recordings would have been made, that would also require more time while also making the comparison with other methos more difficult. The following sections will introduce how the chosen dataset was recorded and processed.

3.1.1 Signal Acquisition

For the 7$^{\text{th}}$ Ninapro database, myoelectric recordings were made using a Delsys® Trigno™ IM Wireless System, with a total of 12 active double–differential wireless electrodes. Each of the Trigno™ sensors also includes an Inertial Measurement Unit (IMU), capable of recording tri-axial accelerometer, gyroscope and magnetometer measurements. While the EMG signals were acquired at a 2000 Hz sampling rate, the IM data was sampled at a much lower rate of 128 Hz. To compensate for this discrepancy, the IM data was later up-sampled to a frequency of 2 kHz, by linear interpolation [21]. While additional measurements are available on the database, these fall out of the scope of this thesis.

3.1.2 Sensor Placement

Following the original Ninapro protocol, eight sensors were equally spaced around the proximal section of the forearm, 3 cm from the elbow, as shown in Figure 3.1a. This strategy is often described as untargeted electrode placement, since there is no specific muscle that it aims to cover. The remaining four sensors were placed in a targeted manner, above the extensor digitorum communis (EDC), digitorum superficialis (FDS), biceps brachii and triceps brachii muscles [21].

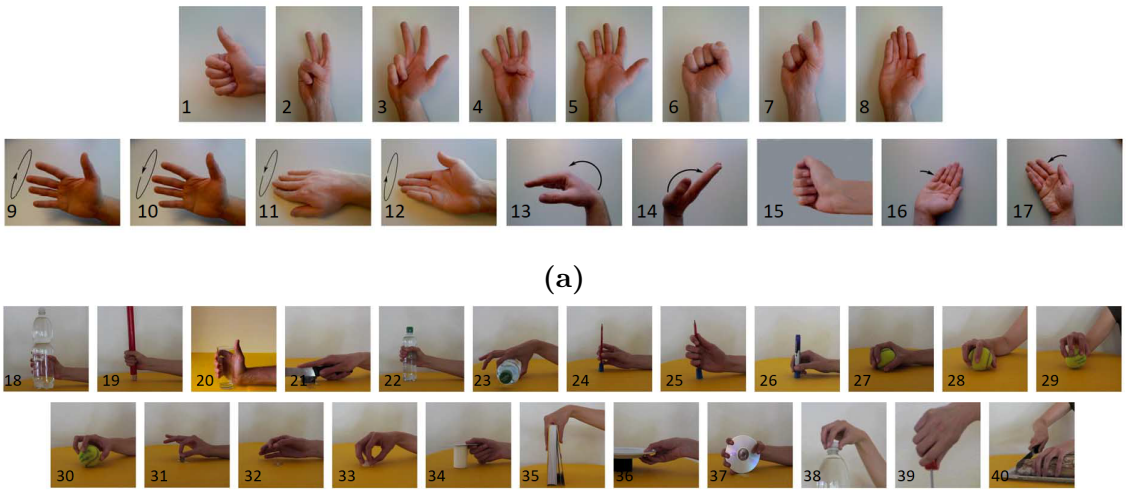
(a) **(b)**

Figure 3.1: Photographs of electrode placement in an amputee subject. In picture **a**, four of the eight untargeted Trigno™ sensors can be seen. Picture **b** additionally shows the targeted electrode placement, above and below the untargeted region. All sensors were covered by an adhesive tape to secure them in place during recordings. Both pictures were taken from Krasoulis *et al.* [21], and are licensed under a Creative Commons Attribution 4.0 International License (https://creativecommons.org/licenses/by/4.0/).

3.1.3 Exercises

The database contains recordings from two different types of exercises, one with basic movements of the hand and wrist, and another with grasps and functional movements. In total they make up 40 different movements that could be controllable by an amputee [21]. These can be seen along with their numbering, in Figure 3.2.

For a better understanding of the movements represented in Figure 3.2, a list of detailed descriptions for each class is presented in tables 3.1 and 3.2. From these, it becomes clear that some of the movements are quite similar to each other, which naturally hinders the performance of any classification algorithm. For that reason, a decision was made to reduce the number of classes by ignoring somewhat redundant movements. In addition, functional movements were excluded, since one of the amputee subjects was interrupted and did not perform the last two [21]. With this selection, a total of 30 movements will be considered, and they are renumbered on the right-hand side of tables 3.1 and 3.2.



(a)

(b)

Figure 3.2: Collection of movements executed by subjects in database 7 of the Ninapro repository. Pictures in group **a**, correspond to movements from exercise 1 and pictures in group **b** consists of the variety of functional grasps recorded for exercise 2. These photographs © [2015] IEEE were taken and rearranged from Atzori *et al.* [18].

Table 3.1: Detailed description of movements from exercise 1. Sections in gray represent the original numbering found in Figure 3.2, with excluded movements.

Isometric, isotonic hand postures	1	Flexing all fingers except thumb (Thumb up)	1
	2	Extension of index and middle finger while flexing others (V-sign)	2
	3	Flexion of ring and little finger while extending others (German three)	3
	4	Thumb opposing base of little finger (Four)	4
	5	Abduction of the fingers (Open hand)	5
	6	All fingers flexed (Close hand)	6
	7	Extended index, with remaining fingers flexed (Index pointer)	7
	8	Adduction of extended fingers (Joined fingers)	8
Basic movements of the wrist	9	Wrist supination (rotation axis through middle finger)	9
	10	Wrist pronation (rotation axis through middle finger)	10
	11	Wrist supination (rotation axis through little finger)	
	12	Wrist pronation (rotation axis through little finger)	
	13	Wrist flexion	11
	14	Wrist extension	12
	15	Wrist extension with closed hand	13
	16	Wrist radial deviation	14
	17	Wrist ulnar deviation	15

Table 3.2: Detailed description of movements from exercise 2. Sections in gray represent the original numbering found in Figure 3.2, with excluded movements.

Grasps, with every-day objects	18	Large diameter grasp (1L bottle)	16
	19	Small diameter grasp (tube)	
	20	Fixed hook grasp (glass)	17
	21	Index finger extension grasp (knife)	18
	22	Medium grasp (0.5L bottle vertical)	19
	23	Ring grasp (0.5L bottle horizontal)	20
	24	Prismatic four fingers grasp (pencil)	
	25	Stick grasp (pencil)	21
	26	Writing tripod grasp (pen)	22
	27	Power grasp (sphere)	
	28	Three finger grasp (sphere)	
	29	Precision grasp (sphere)	
	30	Tripod grasp (sphere)	23
	31	Prismatic grasp (coin)	24
	32	Tip pinch grasp (pill)	25
	33	Quadpod grasp (bottle cap)	26
	34	Lateral grasp (card)	27
	35	Parallel extension grasp (book)	28
	36	Extension type grasp (plate)	29
	37	Power grasp (disk)	30
Functional movements	38	Open a bottle with tripod grasp	
	39	Turn a screw (grasp the screwdriver with a stick grasp)	
	40	Cut something (grasp the knife with an index finger extension grasp)	

3.1.4 Data Collection

Data was collected from a total of 20 intact and two amputee subjects, with exactly the same procedure. However, while the first group preformed each movement monolaterally, the second was instructed to perform bilateral mirrored movements, to obtain a ground truth. This is a common practice which helps amputee subjects to visualise the phantom limb moving in synchrony with their intact hand [18]. Each session required the subject to perform each movement six times, with 3 seconds rest followed by 5 seconds contraction. This was done by providing a visual stimulus on a computer screen indicating which movement to perform [21]. Due to time limitations, it was decided to use only the first 8 intact subjects for the experiments, which gives a total of 10 test subjects.

3.1.5 Signal Processing

In addition to linearly interpolating the IM data, further processing stages were applied. First, to reduce power line interference, a Hampel filter was used on the EMG signals. Subsequently, a relabelling procedure was performed to align the visual stimulus and the respective contraction [21]. This is a vital step for classification purposes, since the presence of mislabelled examples in training data can lead to poor results. Because there is an inevitable misalignment between the stimuli and the contraction periods, due to variable reaction time and attention span of subjects, the relabelling was done in accordance with the Ninapro protocol. For a description of the procedure, refer to section III B of the paper by Atzori *et al.* [18].

3.2 Data Processing

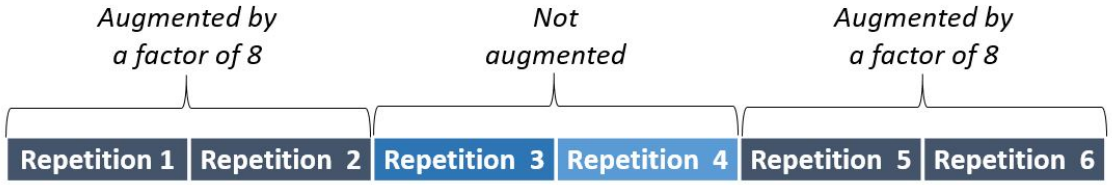
In order to apply the desired machine learning algorithms to the Ninapro dataset, it was necessary to structure the data appropriately. The dataset was provided in a separate Matlab file for each subject and exercise, where all measured movements were stacked into long vectors of non-contiguous signals. To produce the comma separated value (CSV) file containing pairs of input and output vectors, some data processing was performed.

The first step was to remove the redundant movements as mentioned in Section 3.1.3, as well as measurements other than EMG and accelerometer values. Then data augmentation by a factor of 8 was performed as introduced in Section 2.3.3.3. This was done by iteratively taking the original signals and adding white Gaussian noise with increasing SNR from 10 to 40, with an increment of 5. The augmentation magnitude was chosen by verifying the amount of data generated from the process. With a factor of 8 the size of the files reached approximately 4 GB per subject and exercise. If the networks were to be trained for both exercises simultaneously, this value would therefore reach a total of 8 GB, which proved difficult to handle by the hardware. Though there are methods to overcome these limitations, none were explored in this thesis, thus each exercise was evaluated independently. Furthermore, to add a reasonable amount of noise, two separate SNR values were used based on the average signal power of the EMG and the accelerometer signals. To achieve

this, the inbuilt Matlab function `awgn` was used and seven new complete recordings obtained. This process resulted in a total of 48 repetitions for each movement.

Subsequently, the accelerometer signals were down-sampled to 500 Hz, in an attempt to reduce the amount of computation time needed. Based of the work by Atzori *et al.* [12], the EMG data was RMS rectified to the same frequency as the accelerometer data, so that more information was preserved in the final signal while also attenuating the effects of outliers.

The next stage of the script was responsible for splitting the data into **training**, **validation** and **test** sets. To that end, the first, second, fifth and sixth repetitions, and the augmented data obtained from these, were selected to make up the training set. The third and fourth repetitions were saved for the validation and test sets, respectively. While the training set included augmented data, the validation and test sets consisted only of real signals. Therefore, the data obtained by augmenting the original third and fourth repetitions was discarded.



Set	Real	Augmented	Total repetitions
Training	4	28	32
Validation	1	0	1
Test	1	0	1

Figure 3.3: Illustration of the separation of training, validation and test data.

A seen in figure 3.3, with this separation, the training set included a total of 32 repetitions for each movement, making the other two sets much smaller, containing only one repetition. The use of only true signals in these sets is crucial because their purpose is to evaluate how well the network performs. If artificial data were to be used for testing and validation, the results would not reflect the performance for real signals.

Furthermore, having the central recordings in these sets, assures that if there is any changes in the signals as the number of contractions increases, for example due to fatigue, then the intermediate recordings should represent the median contraction. Such changes have been previously reported by Atzori *et al.*, in the Ninapro project (dataset 1) [18]. There was a decreasing trend through repetitions of each movement in both EMG amplitude and range of motion.

Finally, the signals in each set were segmented with a 200 ms window and 150 ms of overlap between windows. Though bigger intervals will store more information and typicality improve the classification output, it has been shown that windows must be smaller than 300 ms so that the user does not experience control delays [41].

Once all the data was processed, it was stored in a separate CSV file for each set. Groups of three sets were stored in separate folders according to exercise and subject to be accessible for testing.

3.3 Software and Hardware

Once the data was structured and stored appropriately, it could be imported to the Python program developed to run the different deep learning algorithms. This was implemented in a virtual environment created using Anaconda. In this environment Microsoft's Cognitive Toolkit (CNTK) was installed along with other necessary libraries. CNTK is an open source deep learning framework that can be imported has a library to Python, C++ or C# programs, as well as a standalone tool in the model description language, BrainScript [42]. Python was chosen due to its simplicity as well as the compatibility with a well established library supported by other major deep learning frameworks, such as Theano and TensorFlow. The Keras library provides a simple high level method of programming neural networks, with an intuitive and well documented API. Therefore, Keras is an ideal choice for a high level implementation of complex networks, such as CNNs and LSTMs [43].

For deep learning applications, there has to be a careful choice of hardware as well as software, since they require computational power superior to most commercially available computers. The system used for the following experiments was a 64-bit Windows based computer, with a CUDA-enabled GPU by NVIDIA, namely the GeForce GTX 970 model, with 4 GB dedicated memory. Other relevant system specifications include a 16 GB RAM and an Intel® Core™ i7 CPU. In the results there will be an evaluation of the execution time for the different networks which highly depends on the system used together with the selected software.

3.4 Experiments

For the experimental stage of this thesis, the CSV files were loaded into a Python script and the input-target pairs stored in separate NumPy arrays. Here, as described in section 2.3.1, the target classes had to be transformed from integers into one hot encoded vectors. The final step before training the networks was to balance all the classes within the sets. This was essential because the class of no movement or rest, had three times more samples than the other movement classes. Thus, two thirds of the windows labelled as rest had to be discarded from each set. For that purpose, before each run, windows classified as rest were randomly removed from all sets. In addition, the movement classes were also subjected to random deletion of a small number of windows so that all sets contained exactly the same number of examples as the least represented movement. To improve training, the examples were also randomised, before each run.

To evaluate the DNN models, three different architectures were tested on ten subjects for both exercises. Once a comparison was made between these structures, the one that provided the best result was further optimised by searching the hyperparameter space.

3.4.1 Model Comparison

The three architectures to be compared are a CNN, RNN and CNN-RNN model. Because there are many hyperparameters that influence the performance of DNNs, in this phase of the experiment, a fix set of parameters were chosen and used for all three structures. The only hyperparameter that was different between networks was the number of layers, since these behave differently for RNNs and CNNs. Schematics of the architectures can be found in figures 3.4, 3.5 and 3.6, and the set of parameters used is presented in table 3.3.

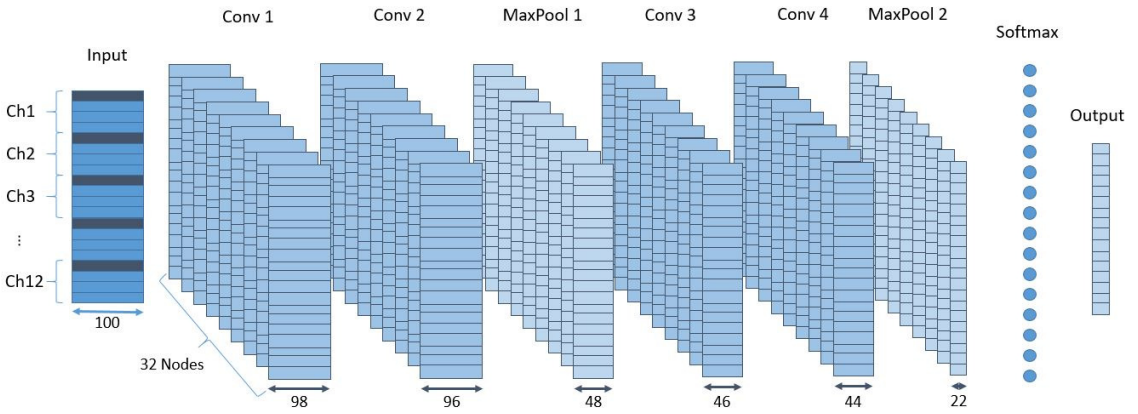


Figure 3.4: Schematic of CNN model. The input is fed to a sequence of convolutional and max-pooling layers. The sizes of the layer outputs are annotated below each layer. At the end there is a softmax layer with 16 nodes, one for each class. Though not represented in the schematic, the outputs of the last max-pooling layer were stacked into one large vector before being fed to the output layer.

The CNN structure was inspired by the VGGNet architecture, due to its simplicity and reported success in other applications. The network developed by the Visual Geometry Group at Oxford, has the key characteristic of containing only 3x3 filters in the convolutional layers, and 2x2 in the max-pooling layers. However, the original paper had a much deeper network with 16-19 weight layers. Of these, three were fully-connected added at the end, just before the softmax layer. Finally, all convolutional layers had ReLUs has activation functions [44].

The RNN model is perhaps the simplest, as it includes only one layer of LSTM units and is followed by a softmax layer. When implementing recurrent networks, there is a possibility to predict several time steps from each input window, since the nodes have temporal depth (*i.e.* many-to-many configuration). However, for the purpose of immediate movement classification this was not necessary, therefore a many-to-one configuration was used, meaning that each node outputs a single value to the softmax layer.

There have been several attempts to create hybrid architectures with CNNs and RNNs. Perhaps the most relevant example is the recent work by Xia *et al.* which employed a CNN-RNN model for EMG based estimation of limb movements [45]. The strategy proposed in this thesis consists of a quite similar model with an initial CNN architecture, responsible for feature extraction and dimensionality

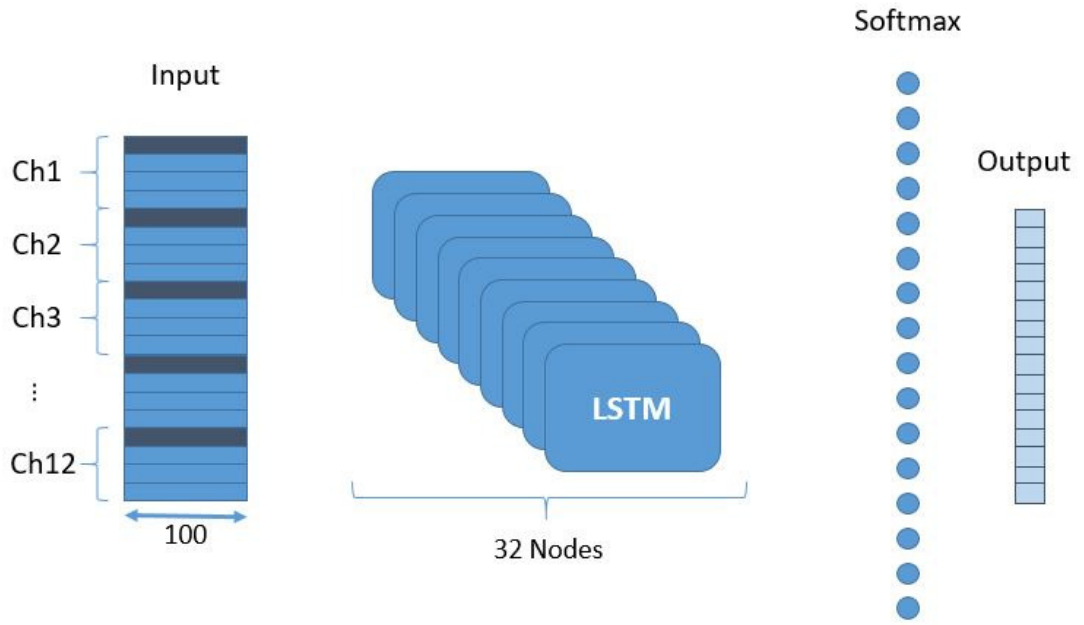


Figure 3.5: Schematic of RNN model. The input is fed to a 32 LSTM nodes. At the end there is a softmax layer with 16 nodes, one for each class.

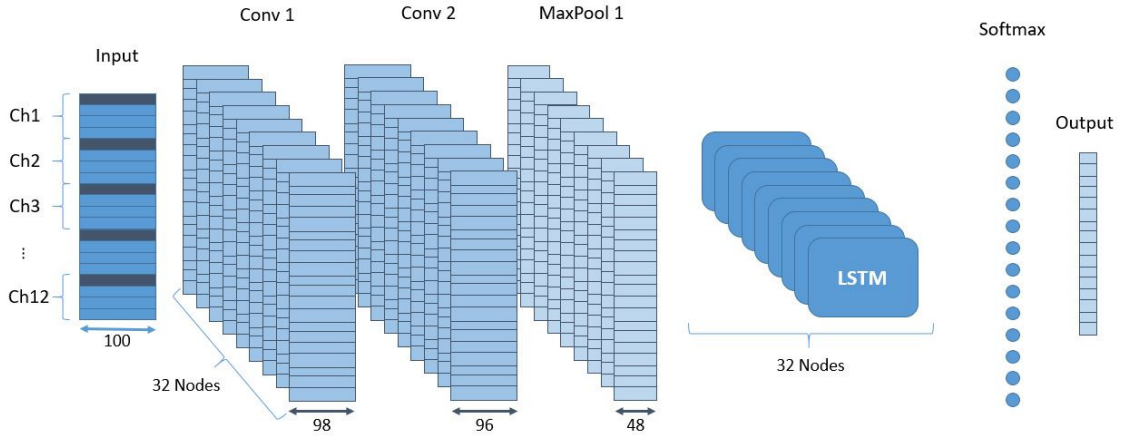


Figure 3.6: Schematic of CNN-RNN model. The input is fed first to two stacked convolutional layers, followed to a max-pooling layer. The feature vectors extracted by these will then be the input to the recurrent network with 32 LSTM nodes. The sizes of the layer outputs are annotated below each layer. At the end there is a softmax layer with 16 nodes, one for each class. Though not represented in the schematic, the outputs of the last max-pooling layer were stacked into one large vector before being fed to the output layer.

reduction, followed by a recurrent layer for learning time-dependent characteristics of the signals. In accordance with the other two architectures, the CNN layers used ReLU units, while the LSTM nodes maintained the activation functions shown in figure 2.8.

Table 3.3: Hyperparameter values kept constant, for model comparison.

Nodes	32
Batch size	128
Learning rate, η	0.001
Regularisation parameter, λ	0.001

The choice of initial parameters was, for the most part, arbitrary. The number of nodes repeats for each convolutional, pooling and recurrent layers in each architecture. Therefore, depending on the depth of the architectures, it results in a different number of trainable parameters (*i.e.* weights and biases). With 32 nodes per layer, the CNN contains the most trainable parameters with a total of 37040, followed by the CNN-RNN with 16592, and finally the RNN containing 10896 trainable parameters.

Starting with a relatively high batch size of 128, helps keep the training process short and therefore speed up the model comparison experiment. On the other hand, the learning rate and other optimiser parameters do not affect the total training time. The learning rate was set as the default value for the *Adam* optimisation algorithm. While this was later changed for the hyperparameter search, all remaining optimisation parameters (β_1, β_2 and ϵ) were left as the default values. Finally, the initial regularisation parameter was chosen by trial and error from a few test runs.

In order to evaluate the performance of each algorithm, during training of a model, after each epoch if there was an improvement in validation accuracy, the weights and biases were stored. At the end of 100 epochs, the model to provide the best validation accuracy was used for testing. This can be seen as a sort of early stopping as described in section 2.3.3.4.

3.4.2 Hyperparameter Search

In deep learning applications, even after a good architecture is found, there is still a need to search for potentially better hyperparameters, that improve the performance of the algorithm. There are several possible strategies to achieve this, such as performing a grid search where several combinations of parameters are tested and the results compared.

For the purpose of this thesis, a sequential search was performed to see the effects of each parameter on the learning curves. Each parameter was changed at a time, while keeping the others constant. The test model was updated for 100 epochs and the resulting accuracies recorded. At the end of each test, the parameter value that provided the best result was kept for the subsequent tests. Here, as for the model comparison experiment, the accuracies were obtained using the test set. Therefore, by executing the parameter search using a constant set for testing, the hyperparameters chosen become biased towards this set. To guarantee that a set of

parameters outperforms another for unseen data, the test set should be chosen at random for each run, and the accuracy averaged over many runs.

Ideally there would also be a more comprehensive and rigorous hyperparameter search, with a heavily sampled search space, for more significant conclusions. However, since the goal of this experiment is mainly to explore the effects of each hyperparameter on the learning process, only four new values were tested for each parameter, in three separate trials. The values selected were picked at random from a constrained range.

Typically the learning rate takes values between 0.0001 and 1, for that reason a logarithmic scale was used for randomly selecting the test values. Nodes and batch size, were limited by the memory capacity of the system. Following convention, both parameters took only powers of two as vales. Finally, the regularisation parameter was sampled in a similar range as the learning rate. Because it belongs to the penalty term, it could take any value equal or larger than zero. However, if λ would be too large, the penalty would overpower the cost function.

Due to the stochastic nature of the training, an individual run carries little information on the efficacy of the chosen parameters. By running the same model repeatedly, the results can be averaged to reduce variability. Though three runs is still not enough for statistically significant conclusions, it gives an intuition on how the different parameters affect the learning process.

4

Results

This chapter summarises the findings from the model comparison experiment as well as the hyperparameter search. All results presented are discussed in an attempt to evaluate the potential of implementing these deep neural networks for myoelectric pattern recognition applications. Furthermore, the effects of each hyperparameter were discussed to gain a better insight on their influence during training.

4.1 Model Comparison

In order to compare the three architectures, each model was trained once for each of the ten subjects, first for the movement classes from exercise 1, and then from exercise 2. This resulted in 20 runs per algorithm, for a total of 60 trained models. Tables 4.1, 4.2 and 4.3 contain the results from this experiment. In addition to accuracy and MCC score, three other measures were recorded: the time it took to train the network, t_{Train}, the time it took to evaluate one window and get a classification output, t_{Test}/w, and the epoch at which the model was obtained (best validation accuracy).

In terms of classification performance, the best network, with an average accuracy of 89.42% was the RNN model. Table 4.2 shows the cases in which the CNN model outperformed the RNN with the † symbol, and ‡ denotes the cases where the CNN-RNN model performed better. Naturally, since the models are of a stochastic nature, these five samples in which the RNN did not outperform both models may have been a result of an unfortunate run. To be certain, this experiment should have included multiple trials to average over, however, there was a time constraint.

While the CNN model took on average, approximately 4 minutes to train, the RNN took around 31 minutes. One of the benefits of having the CNN-RNN hybrid model is that the initial CNN structure reduces the input dimensionality of the LSTM cells, thus speeding up the training to an average of 18 minutes.

When comparing test times of the different models, the CNN was the considerably faster at producing a classification output when given an input window, taking on average only 1.25 ms. The CNN-RNN model followed with 11.82 ms and finally the RNN took 20.20 ms.

There is not a lot that can be concluded about the learning process from the epoch at which the best model was obtained. A better indication is to look at the learning curves of the training and validation set to observe when overfitting starts to occur. However, if the best validation accuracy is obtained in an early epoch, it may indicate that the learning rate is to high. Similarly, if the best model is

Table 4.1: CNN results for ten subjects, separated by exercise. The mean accuracy was 86.27% (E1:87.70%, E2:84.84%), while the average MCC was 0.8558 (E1:0.8709, E2:0.8406). Training times (t_{Train}) were between 3 and 5 minutes, and the average test time (t_{Test} per input window $\mathbf{w}$) was 1.25 ms. The early-stopping epoch is noted for each subject and was based on the validation accuracy. The early-stopping epoch is noted for each subject and was based on the validation accuracy. The † symbol denotes the results obtained with the CNN that outperformed the RNN for the same subject and exercise.

(a) Exercise 1

Subject	**Accuracy (%)**	**MCC**	$\mathbf{t_{Train}}$ (min)	$\mathbf{t_{Test}/w}$ (ms)	**Epoch**
1	94.33	0.9399	3	1.1926	53
2	92.47	0.9201	3	1.2356	76
3	91.86	0.9160	3	1.2391	58
4	96.88	0.9668	4	1.1926	57
5	85.20	0.8451	3	1.2620	80
6	90.00	0.8965	4	1.2449	21
7	92.07	0.9159	3	1.2194	20
8	83.59	0.8298	4	1.3256	92
21	80.83	0.7984	3	1.2267	25
22	69.74	0.6803	3	1.2155	65

(b) Exercise 2

Subject	**Accuracy (%)**	**MCC**	$\mathbf{t_{Train}}$ (min)	$\mathbf{t_{Test}/w}$ (ms)	**Epoch**
1	90.45	0.8990	4	1.1822	89
2	88.02	0.8745	5	1.3150	33
3	93.86	0.9355	4	1.2481	93
4	86.22	0.8557	5	1.2246	90
5	85.51	0.8482	4	1.2296	74
6	93.96†	0.9361†	4	1.2046	95
7	87.66	0.8705	3	1.4025	91
8	90.00	0.8952	4	1.1976	100
21	49.47	0.4675	4	1.1867	100
22	83.24†	0.8237†	3	1.3828	81

obtained near the one hundredth epoch, it may be that the learning rate should be increased or that the training should be extended to more epochs.

To summarise the model comparison, the classification results were compiled into Figure 4.1, in the form of boxplots. From these, the same conclusion was taken, namely that the RNN outperformed the other two models. With a median accuracy of of 91.81% (MCC = 0.9135), compared with 89.01% (MCC = 0.8849) for the CNN and 90.43% (MCC = 0.8990) for the CNN-RNN, the RNN model was chosen for the hyperparameter search. Although, by selection of better parameters for the other models could potentially lead them to outperform the RNN.

Table 4.2: RNN results for ten subjects, separated by exercise. The mean accuracy was 89.42% (E1:91.38%, E2:87.47%), while the average MCC was 0.8889 (E1:0.9088, E2:0.8690). Training times (t_{Train}) were between 24 and 46 minutes, and the average test time (t_{Test} per input window **w**) was 20.20 ms. The early-stopping epoch is noted for each subject and was based on the validation accuracy. The results for which the † and ‡ symbola are appended show instances in which the RNN was outperformed by the CNN and CNNR-RNN models, respectively.

(a) Exercise 1

Subject	**Accuracy (%)**	**MCC**	$\mathbf{t_{Train}}$ (min)	$\mathbf{t_{Test}}$/**w** (ms)	**Epoch**
1	94.91	0.9460	26	21.0467	46
2	93.75	0.9338	29	20.2995	58
3	99.13	0.9907	24	20.0171	84
4	97.72	0.9758	30	20.3820	79
5	92.27	0.9182	22	20.1575	67
6	90.80‡	0.9026‡	31	19.8325	55
7	95.73	0.9548	25	22.3052	91
8	86.99	0.8645	31	20.3809	83
21	89.17	0.8859	25	20.7791	47
22	73.30	0.7160	25	19.5468	96

(b) Exercise 2

Subject	**Accuracy (%)**	**MCC**	$\mathbf{t_{Train}}$ (min)	$\mathbf{t_{Test}}$/**w** (ms)	**Epoch**
1	91.25	0.9075	34	20.1043	93
2	91.35	0.9088	39	20.9211	54
3	94.31	0.9401	32	19.5972	92
4	86.86‡	0.8630‡	46	19.3947	81
5	93.47	0.9305	35	19.3309	86
6	93.33†	0.9300†	37	20.0148	84
7	90.71	0.9024	24	20.4519	48
8	92.31‡	0.9192‡	33	19.5547	100
21	59.71	0.5732	37	20.0887	83
22	81.39†	0.8149†	26	19.6957	65

As seen in Figure 4.1, there are outliers in the data. These correspond to the results from subject 21 on exercise 2. For all networks, the classification performance for this exercise was substantially lower than the other participants and even for the same participant on exercise 1. This may indicate that the amputee subject had more difficulties performing the movements from this exercise, resulting in worse recordings. When looking at Figure 3.2b, with the movements from exercise 2, it is understandable why an amputee would have difficulties recreating these movements.

Table 4.3: CNN-RNN results for ten subjects, separated by exercise. The mean accuracy was 85.69% (E1:87.82%, E2:83.56%), while the average MCC was 0.8490 (E1:0.8715, E2:0.8265). Training times (t_{Train}) were between 13 and 28 minutes, and the average test time (t_{Test} per input window **w**) was 11.82 ms. The early-stopping epoch is noted for each subject and was based on the validation accuracy. The ‡ symbol denotes the results obtained with the CNN-RNN that outperformed the RNN for the same subject and exercise.

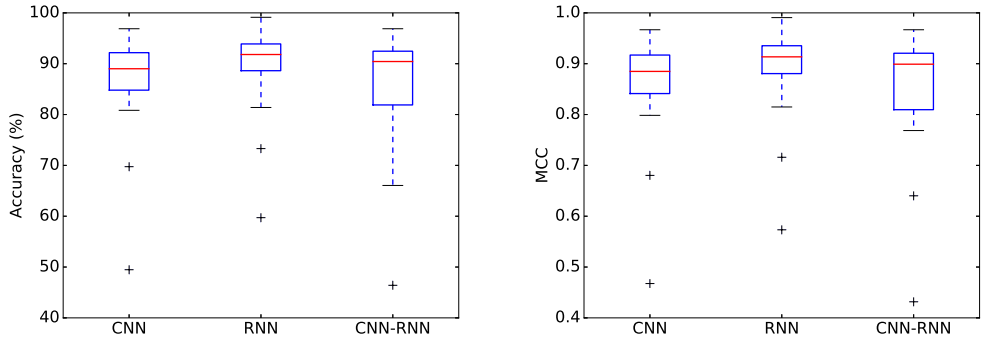
(a) Exercise 1

Subject	**Accuracy (%)**	**MCC**	$\mathbf{t_{Train}}$ (min)	$\mathbf{t_{Test}}$/**w** (ms)	**Epoch**
1	92.44	0.9204	15	11.6438	23
2	92.98	0.9257	17	11.7232	36
3	96.66	0.9645	14	12.0112	47
4	96.88	0.9668	18	11.6034	32
5	87.60	0.8691	13	11.6746	96
6	91.59‡	0.9109‡	18	11.9243	17
7	91.16	0.9065	15	11.6649	23
8	80.48	0.7974	18	12.2664	54
21	82.36	0.8136	15	12.0728	14
22	66.05	0.6401	15	11.6535	82

(b) Exercise 2

Subject	**Accuracy (%)**	**MCC**	$\mathbf{t_{Train}}$ (min)	$\mathbf{t_{Test}}$/**w** (ms)	**Epoch**
1	90.80	0.9028	20	11.3705	42
2	90.83	0.9027	23	12.0714	33
3	92.52	0.9212	19	11.4918	39
4	87.82‡	0.8726‡	28	11.8096	28
5	79.26	0.7808	21	12.0017	84
6	87.08	0.8649	22	11.4452	52
7	90.06	0.8954	14	12.3661	61
8	92.79‡	0.9249‡	20	11.5054	96
21	46.41	0.4316	23	12.4690	96
22	77.98	0.7685	15	11.6596	96

Finally, it can be concluded from the interquartile range (IQR) values that the RNN also provided the lowest variability, as opposed to the CNN-RNN which had the highest IQR of approximately 10% in accuracy. Even though variability is to be expected when dealing with different subjects along with stochastic models, it is clear that the CNN-RNN model was more variable than the other two.



(a) **(b)**

Figure 4.1: Tukey boxplots with classification results for each architecture. The whiskers have length equal to $1.5 \times IQR$ and were computed from the results in tables 4.1, 4.2 and 4.3. These show the RNN outperforming the other two architectures with a median value of 91.81%.

4.2 Hyperparameter Search

The results from the hyperparameter search were compiled in Table 4.4. Since the experiments were done sequentially, a total of five values were actually tested for each parameter. Because the first hyperparameter to be tested was the learning rate, the initial model, with $\eta = 0.001$ had to be tested for two additional trials. This resulted in a reference average initial accuracy of 95.25% and MCC score of 0.9495.

Table 4.4: Hyperparameter search results separated by parameter. The values correspond to averages over three runs.

(a) Learning rate, η

η	**Accuracy (%)**	**MCC**
0.0001	94.81	0.9449
0.0006	**95.93**	**0.9568**
0.0040	92.54	0.9218
0.0253	77.91	0.7680

(b) Number of nodes

Nodes	**Accuracy (%)**	**MCC**
16	93.65	0.9328
64	94.72	0.9439
128	93.07	0.9266
256	92.93	0.9255

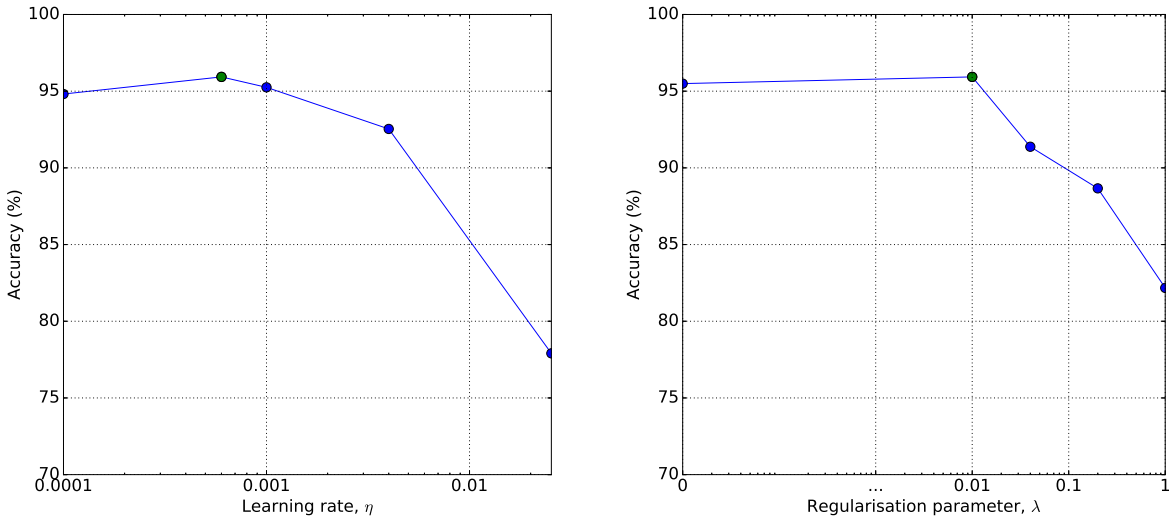
(c) Batch size

Batch	**Accuracy (%)**	**MCC**
16	93.65	0.9339
32	94.62	0.9429
64	94.57	0.9425
256	93.56	0.9318

(d) Regularisation parameter, λ

λ	**Accuracy (%)**	**MCC**
0	95.49	0.9522
0.04	91.38	0.9101
0.2	88.66	0.8807
1	82.17	0.8117

The best result is shown in bold, and was obtained with the initial number of nodes and batch size, namely 32 nodes and batch size of 128. The effect of changing learning rate can be seen in Figure 4.2a. It can be concluded that a learning rate between 0.0001 and 0.001 produces the best results. Because there were insufficient trials, there is no certainty that $\eta = 0.0006$ is in fact the best value, however, from the small sample obtained, it produced the best result.



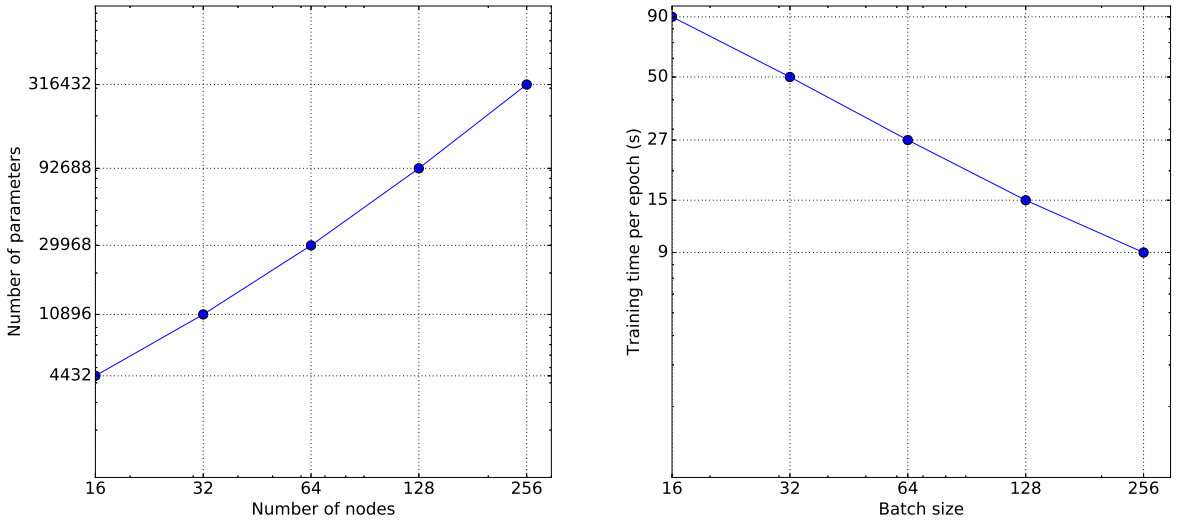
(a) **(b)**

Figure 4.2: Visualisation results from Table 4.4. From these plots, the effect of the learning rate and the regularisation parameter are shown with respect to the accuracy. The data point in green corresponds to the best average accuracy obtained from the hyperparameter search. There is a clear decrease in learning ability as these values become too large.

From tables 4.4b and 4.4c there are little changes in accuracy with the values tested in this experiment. However, it would have been unfeasible to test much higher values, due to the memory limitation of the system. Though the number of nodes did not have any significant effect on the classification accuracy, it can be seen in Figure 4.3a, that with the increase of number of nodes there is a linear increase of number of parameters that need to be trained, has would be expected. In a similar way, Figure 4.3b shows the linear relationship between the batch size and the training time. This was also to be expected since the bigger the mini-batch size, the fewer times the network needs to be evaluated and updated.

The final hyperparameter to be tested was the regularisation parameter. From Figure 4.2b it becomes evident that λ cannot be too large. However, there is no significant improvement from having $\lambda = 0.01$, when compared with having no regularisation at all. In order to gain a better understanding of the effects of this, as well as the other hyperparameters, it is valuable to look at the learning curves shown in figures 4.4 through 4.7.

It becomes clear from Figure 4.4 that the learning rate affects the speed at which the network converges. With $\eta = 0.0001$, the network seems to still be learning when it reaches 100 epochs. By increasing the learning rate to 0.0006,



(a) **(b)**

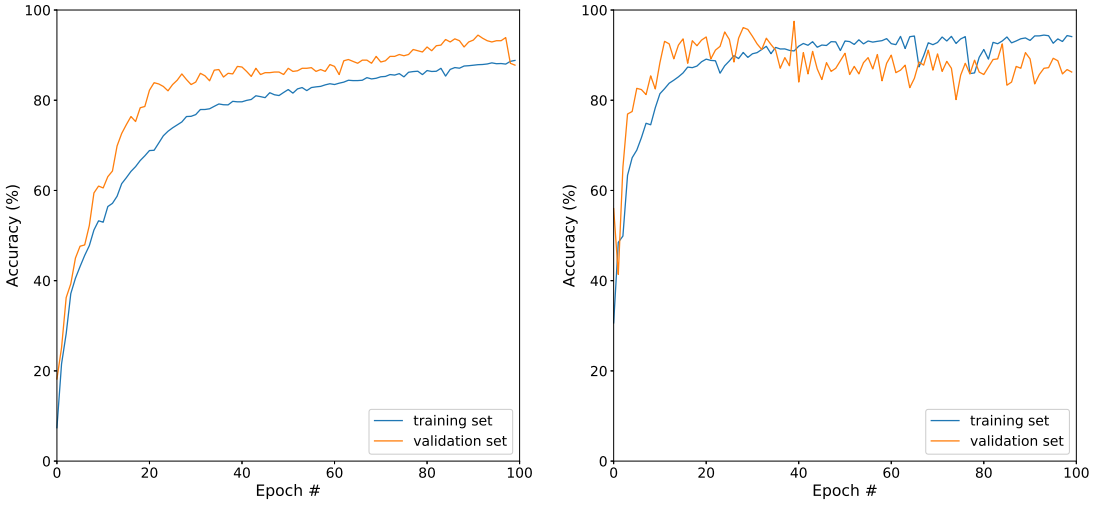
Figure 4.3: Additional effects of hyperparameters. Figure 4.3a shows effect of number of nodes on the total number of trainable parameters. Figure 4.3b shows the effect of increasing batch size on the training time. These results were to be expected since they are directly linked with the computational complexity of the program.

there is already significant improvement and the network rapidly converges to a good accuracy, until it starts overfitting. This turning point is very clear in Figure 4.4b where the training curve intersects the validation curve. With $\eta = 0.004$ the network still converges, however, it seems that initially the rate is too large resulting in large oscillations. Nevertheless, there seems to be no evidence of overfitting. which means that values of this order of magnitude may produce better results. The same cannot be said for $\eta = 0.0253$, which clearly made the network incapable to learn.

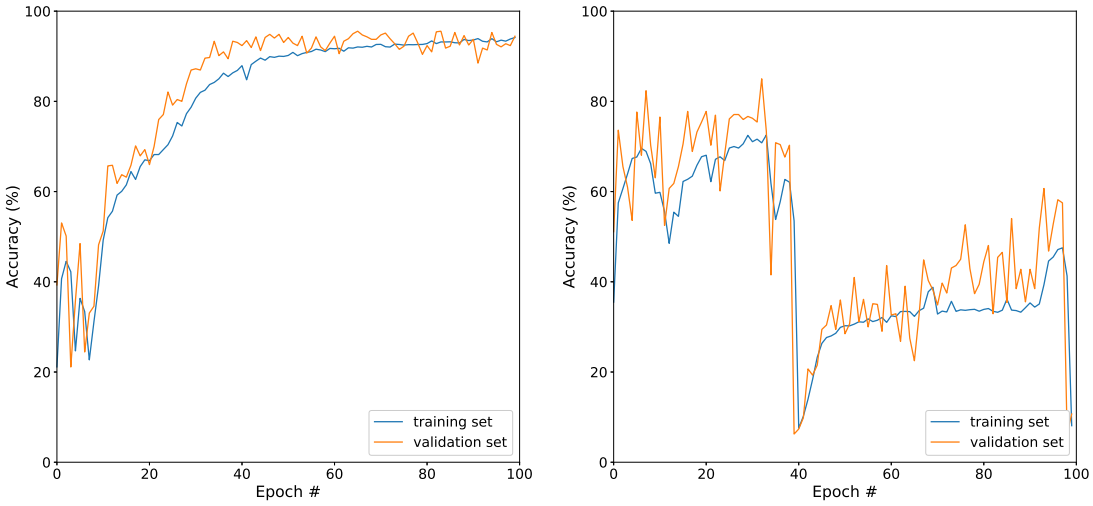
The effect of the number of nodes, seen in Figure 4.5, is not as drastic. It seems that with 16 nodes, the RNN learns slower, since at the end of 100 epochs it still did not converge. On the other hand, with 256 nodes, the network had significantly higher variance at the end of the training. However, since early-stopping was implemented, this did not affect the accuracy measured for the test set.

The batch size also affected the speed of convergence of the network. From Figure 4.6 there is a tendency for the network to converge increasingly slower as the mini-batch size increases. This can be explained by the fact that with a smaller batch, at the end of each epoch the model was updated more times than with a larger batch.

Finally, as had been concluded from 4.2b, the regularisation parameter can have large effects on the learning process. With larger values of λ, the learning curves become more linear. Perhaps by increasing the number of epochs these values would actually lead to good results. Nevertheless, the learning speed becomes too slow for the 100 epoch tests. Moreover, the increase in overlap between the training curve and the validation curve shows that the regularisation is efficient in keeping a low variance.

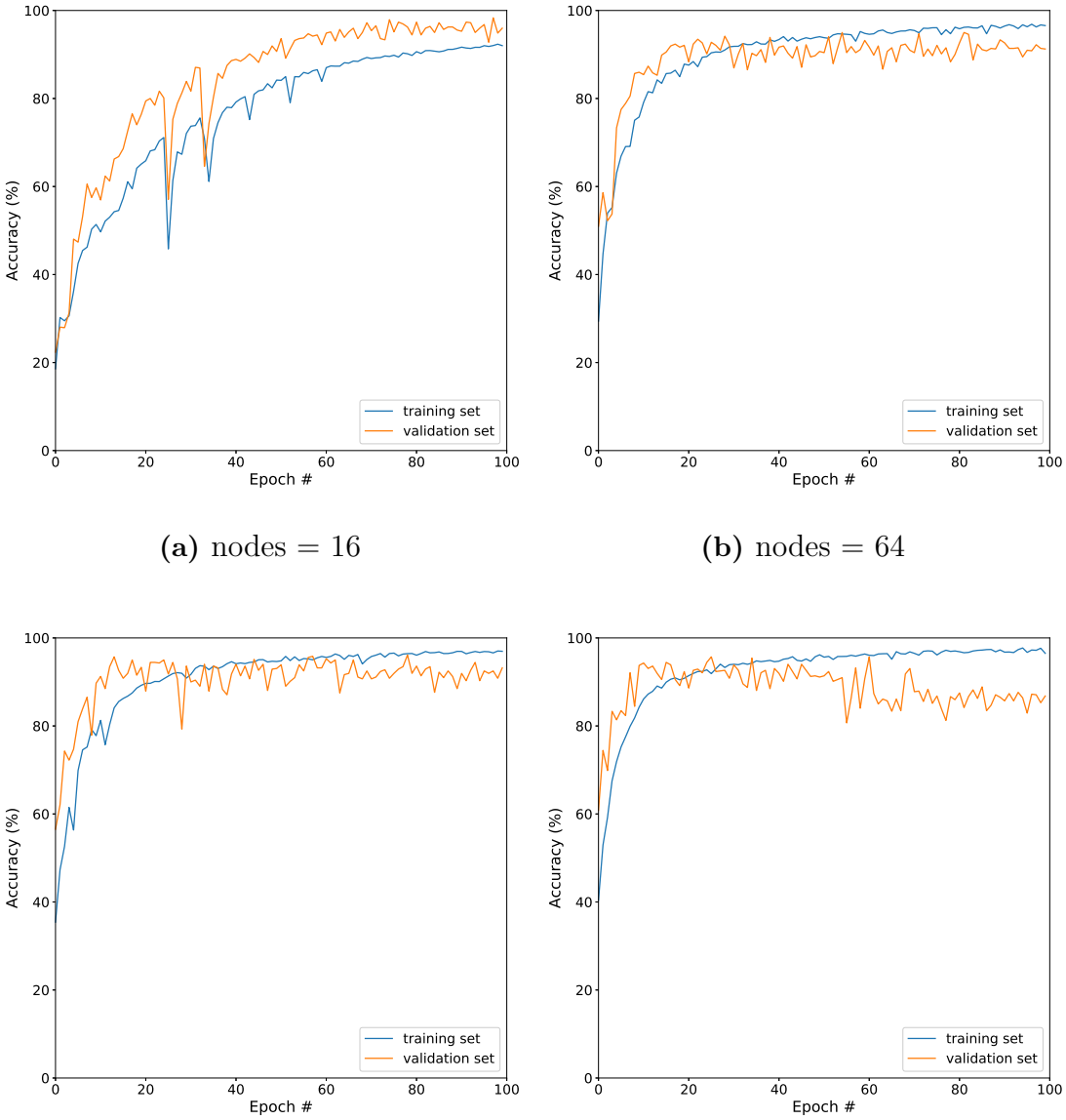


(a) $\eta = 0.0001$ **(b)** $\eta = 0.0006$



(c) $\eta = 0.004$ **(d)** $\eta = 0.0253$

Figure 4.4: Learning curves for different learning rates shown in Table 4.4a. Each plot corresponds to one of the three trials for a given parameter set. With a larger learning rate, the network suffers from high bias.



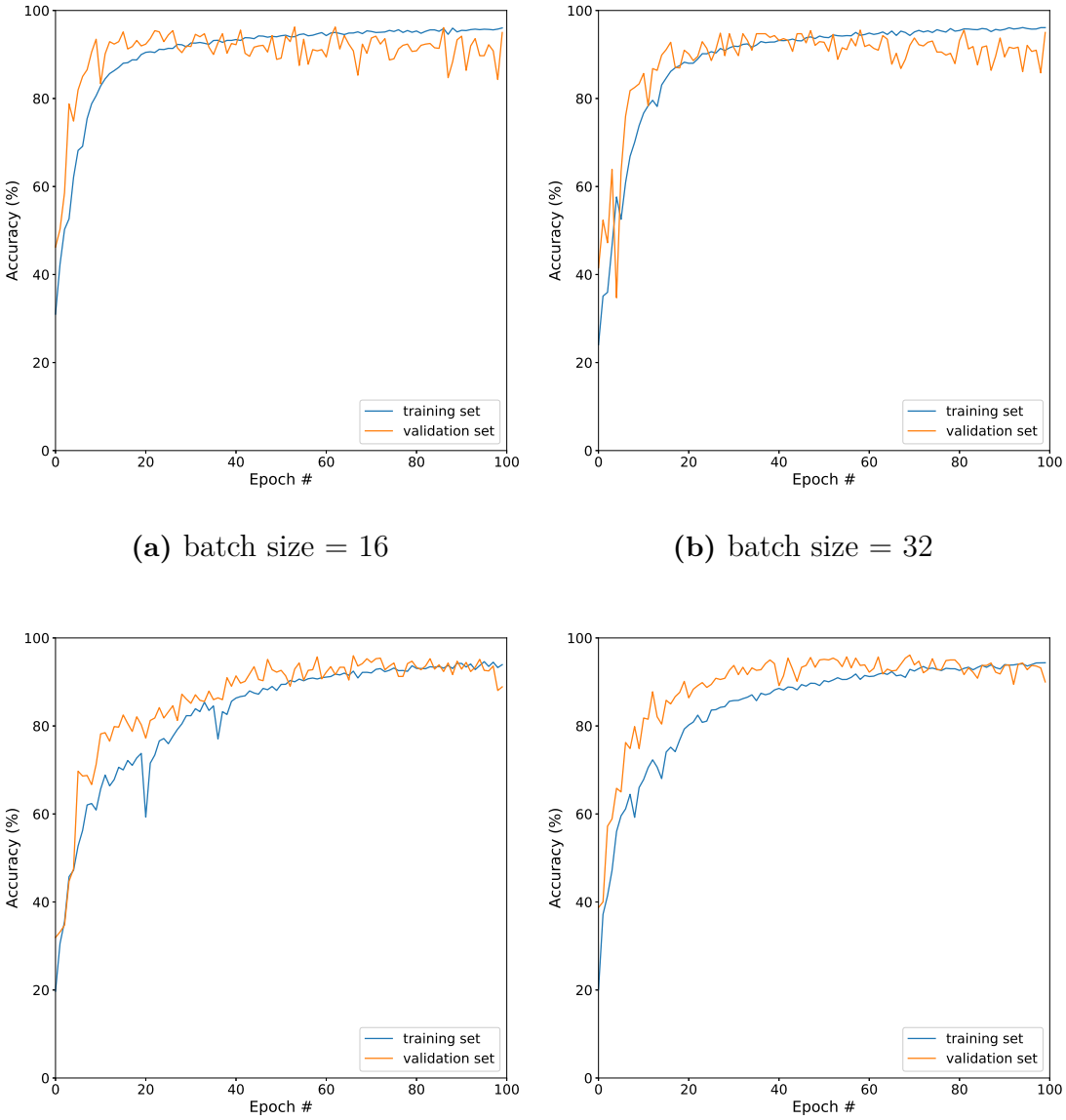
(a) nodes = 16

(b) nodes = 64

(c) nodes = 128

(d) nodes = 256

Figure 4.5: Learning curves for different number of nodes, shown in Table 4.4b. Each plot corresponds to one of the three trials for a given parameter set. With more number of nodes the validation accuracy drops below the training accuracy, which is a sign of high variance.



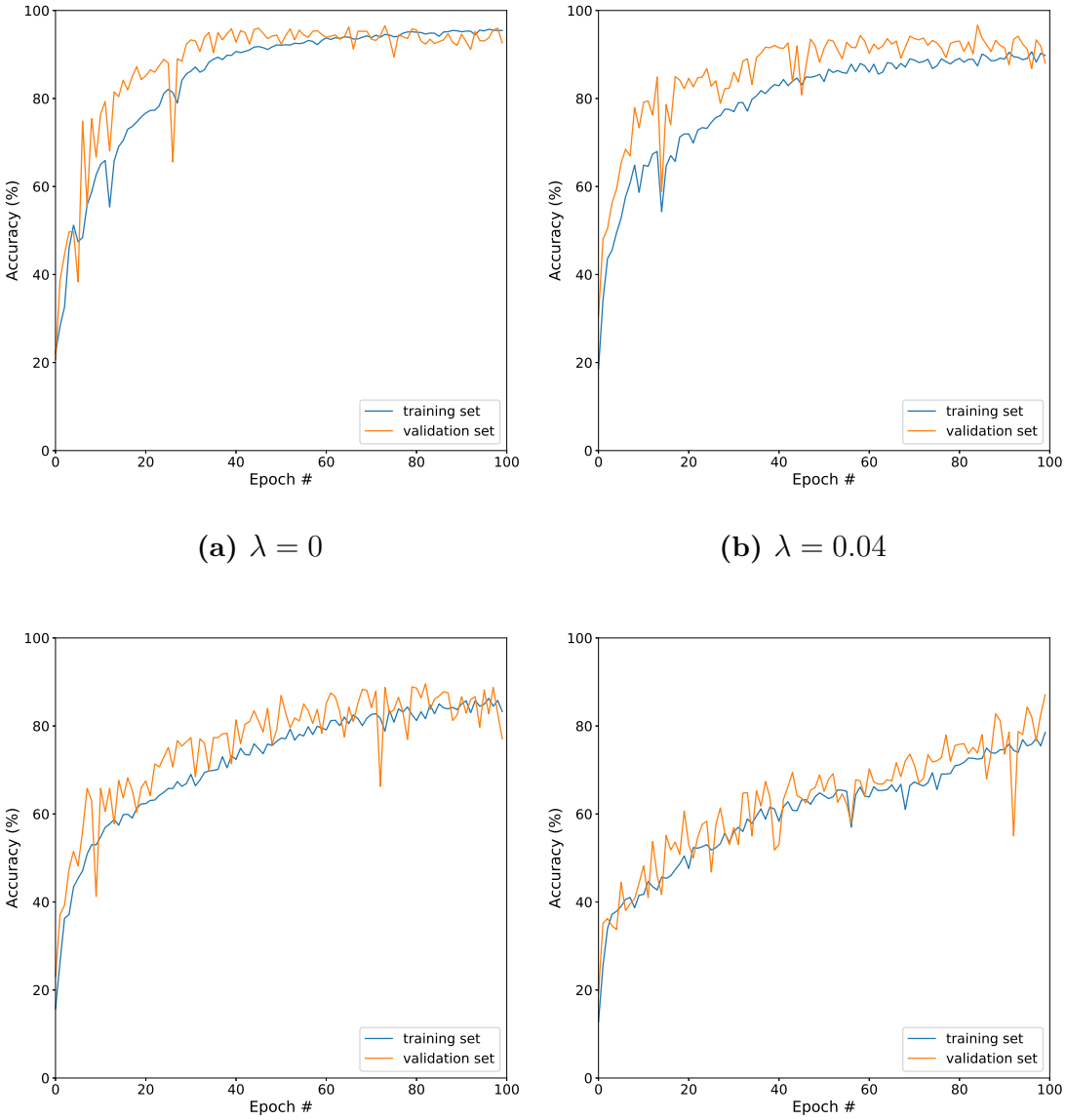
(a) batch size = 16

(b) batch size = 32

(c) batch size = 64

(d) batch size = 256

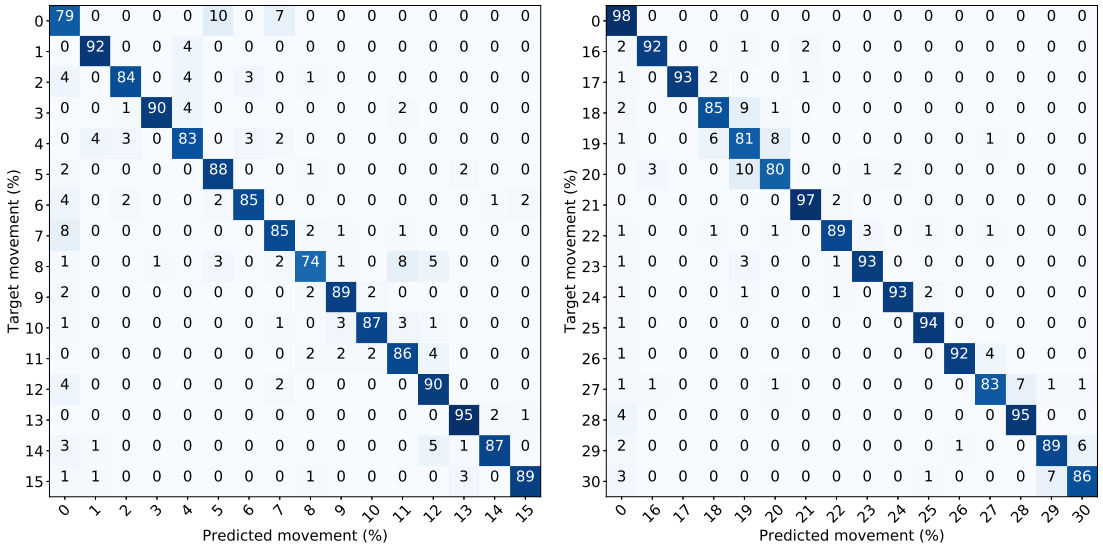
Figure 4.6: Learning curves for different mini-batch sizes shown in Table 4.4c. Each plot corresponds to one of the three trials for a given parameter set. The learning curves for different batch sizes are not significantly different. In general, it seems that a smaller batch size leads to a faster convergence.



(a) $\lambda = 0$ **(b)** $\lambda = 0.04$

(c) $\lambda = 0.2$ **(d)** $\lambda = 1$

Figure 4.7: Learning curves for different levels of regularisation, shown in Table 4.4d. Each plot corresponds to one of the three trials for a given parameter set. Note that plot 4.7a displays the learning curve without regularisation. From $\lambda = 0.2$, the regularisation effect overpowers the learning and leads to high bias.



(a) Exercise 1 **(b)** Exercise 2

Figure 4.8: Confusion matrices for both exercises, averaged over all subjects, with the final set of hyperparameters. The percentages were truncated to the nearest integer for better visualisation. Despite leading to inconsistencies in the percentages, values smaller than 1% can be disregarded for the purpose of this analysis.

As a final evaluation of the results, the RNN model was once again trained using the best set of parameters with the data from all subjects and both exercises. The results were compiled in two separate confusion matrices shown in Figure 4.8. The percentages obtained were computed by averaging the results of the ten subjects, for each exercise.

By looking at Figure 4.8, a few observations can be extracted. For instance, the majority of the output classes were at some point misclassified as the rest class (*i.e.* no movement). This may be due to failure of the relabelling procedure, where a time window during rest was labelled as a certain movement. Furthermore, the class with the lowest accuracy consisted of class number 8 of exercise 1, with 74%. This movement consists of the adduction of extended fingers, which is actuated by the *palmar interossei* muscles, which are located in the hand. Since the electrodes were all placed in the proximal part of the forearm, these were most likely unable to detect the movement effectively.

5

Discussion

This chapter will attempt to build from the discussion initiated in the results section and further discuss the significance of the findings of this thesis.

While the model comparison experiment was done by grouping together both intact and amputee subjects, it is relevant to evaluate the results separately. The mean CNN accuracy was 90.13% for intact subjects and 70.82% for amputees. The same trend is observed for the RNN, with 92.81% and 75.89% for intact and amputee subjects, respectively. The difference was even more noticeable for the hybrid network with 90.06% for intact and 68.20% for amputee subjects. Since the end-users of myoelectric control systems are for the most part amputees, these results reinforce the RNN as the best performing model.

For myoelectric control, the testing time is a limiting factor. In order to implement these networks in real time applications, for online classification, the time it takes to compute an output needs to be sufficiently small. This requirement will depend on the application and the hardware on witch it is implemented. Since the tests were run in an a high performing computer, there will likely be an increase in testing time for other systems. While software applications, such as for phantom-limb pain treatment, may be used via similar systems, prosthetic devices have embedded systems which are significantly more limited. Nonetheless, there is an effort from the research community to increase computing speeds of DNNs which, in the future will allow for larger models to be implemented on weaker devices [46].

Taking test time into consideration, the CNN model would be the most likely candidate to be implemented in embedded devices, with the mean test time of 1.25 ms. As previously stated, it is possible that by searching the hyperparameter space for the CNN network, this would reach or even surpass the performance of the RNN.

Although Atzori *et al.* [12] employed CNNs for classification of the Ninapro movements, they did not exclude redundant movements nor separate the two exercises into different classifiers. Furthermore, they also did not include accelerometer data, therefore it is not possible to do a direct comparison of the accuracies achieved here. On the other hand, the 7^{th} Ninapro database was provided by Krasoulis *et al.* [21], where for the 40 movements and using an LDA classifier a median accuracy of 82.7% for intact subjects and 77.8% was achieved. This however, was including all of the IMU data, namely three-axial accelerometer, gyroscope and magnetometer signals. From this work it was concluded that by employing all the IMU data, significantly improved classification accuracy.

Initially it was attempted to use all the movements at once to train the classifiers, but several issues were encountered. The first being the memory capacity of the system, which did not allow for data augmentation in such a case. In addition, the classifiers proved to have difficulties distinguishing between movements that were too similar (*e.g.* movements 9-12 in Figure 3.2a), leading to poor accuracies. Finally, there was a certain reasoning behind splitting the two exercises into separate classifiers. Due to the significant differences of the exercises, it seemed logical to have two modes between which the user could switch, depending if they wanted to control hand postures or grasps.

While the CNN implementation by Atzori *et al.* [12] was implemented using two-dimensional filters, this was not the approach used for this thesis. The motivation for this modification was based on the characteristics of the input signals. By stacking the EMG signals into a matrix and applying *e.g.* 3x3 filters, the signals from different channels are mixed together through the convolution operation, and valuable information may be lost. Furthermore, CNNs are not rotation or inversion invariant which attributes significant importance to the order in which the different channels are stacked. As mentioned in the theory, the CNNs learn local, translation invariant features. This only makes sense if the signals are kept separate and one-dimensional convolution is used.

Due to the considerable variability between different myoelectric pattern recognition research, it is difficult to compare results. Overall the conclusion that can be derived from the current work is that these networks are yet another type of PR algorithm that can be employed to solve the problem.

Moving on to the hyperparameter search experiment, the results showed that number of nodes and the batch size had little effect on the test accuracy of the RNN. On the other hand, the optimisation parameters can have a significant impact on the learning curves, therefore they should be prioritised when tuning parameters.

6
Conclusion

To conclude, there will be a short summary of results an discussion followed by some comments on the work presented in this thesis as well as possible future work that may be relevant for the field of myoelectric pattern recognition.

With median accuracies above 80% achieved by all three models studied, it can be concluded that these DNNs are powerful pattern recognition tools for myoelectric control applications. However, there have been higher accuracies reported in offline studies with other PR algorithms. Although, contrary to such studies, the classification was done independently of any feature extraction phase and still reached high performances.

The highest performing architecture, with a median accuracy of 91.81%, was the RNN consisting of a layer of LSTM units. This could support the hypothesis that recurrent networks have an advantage when processing temporal data, such as EMG and IMU signals. This can also be supported by the higher median accuracy obtained by the hybrid CNN-RNN model, 90.4% when compared with the CNN, 89.01%. Though, due to the higher variability of the CNN-RNN model, it achieved the lowest average accuracy. Nevertheless, the median performance values indicate that the networks containing LSTM units have an advantage.

Another factor that should be taken in to consideration, is that even though the CNN performed worse, it is an order of magnitude faster at computing an output than the CNN-RNN and RNN models. By passing the input windows through the CNN layers to the LSTM layer, the hybrid architecture became twice as fast as the RNN. This is due to the automatic feature extraction and reduction by convolutional and max-pooling layers, respectively.

The utility of DNNs for myoelectric control applications has not yet been proven. These models are more complex to implement than other well established algorithms such as LDA, and have high computational cost. For simple embedded systems, such as in prosthetic devices, they may not be a viable option until faster algorithms or systems are developed. However, DNNs have been gaining momentum in the research community, and have been tested in a growing number of fields. The high pace in development of both deep learning software and processing power of hardware may lead to more reliable myoelectric control.

References

[1] Aidan D Roche, Hubertus Rehbaum, Dario Farina, and Oskar C Aszmann. Prosthetic myoelectric control strategies: a clinical perspective. *Current Surgery Reports*, 2(3):44, 2014.

[2] Sophie M Wurth and Levi J Hargrove. A real-time comparison between direct control, sequential pattern recognition control and simultaneous pattern recognition control using a fitts' law style assessment procedure. *Journal of neuroengineering and rehabilitation*, 11(1):91, 2014.

[3] Susannah M Engdahl, Breanne P Christie, Brian Kelly, Alicia Davis, Cynthia A Chestek, and Deanna H Gates. Surveying the interest of individuals with upper limb loss in novel prosthetic control techniques. *Journal of neuroengineering and rehabilitation*, 12(1):53, 2015.

[4] Angkoon Phinyomark, Pornchai Phukpattaranont, and Chusak Limsakul. Feature reduction and selection for emg signal classification. *Expert Systems with Applications*, 39(8):7420–7431, 2012.

[5] Kevin Englehart, B Hudgin, and Philip A Parker. A wavelet-based continuous classification scheme for multifunction myoelectric control. *IEEE Transactions on Biomedical Engineering*, 48(3):302–311, 2001.

[6] Max Ortiz-Catalan. Cardinality as a highly descriptive feature in myoelectric pattern recognition for decoding motor volition. *Frontiers in neuroscience*, 9, 2015.

[7] Angkoon Phinyomark, Franck Quaine, Sylvie Charbonnier, Christine Serviere, Franck Tarpin-Bernard, and Yann Laurillau. Emg feature evaluation for improving myoelectric pattern recognition robustness. *Expert Systems with Applications*, 40(12):4832–4840, 2013.

[8] Paul Kaufmann, Kevin Englehart, and Marco Platzner. Fluctuating emg signals: Investigating long-term effects of pattern matching algorithms. In *Engineering in Medicine and Biology Society (EMBC), 2010 Annual International Conference of the IEEE*, pages 6357–6360. IEEE, 2010.

[9] Erik Scheme and Kevin Englehart. Training strategies for mitigating the effect of proportional control on classification in pattern recognition based myoelectric control. *Journal of prosthetics and orthotics: JPO*, 25(2):76, 2013.

[10] Max Ortiz-Catalan, Bo Håkansson, and Rickard Brånemark. Real-time and simultaneous control of artificial limbs based on pattern recognition algorithms. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 22(4):756–764, 2014.

[11] Afarin Nazemi and Ali Maleki. Artificial neural network classifier in comparison with lda and ls-svm classifiers to recognize 52 hand postures and movements. In

Computer and Knowledge Engineering (ICCKE), 2014 4th International eConference on, pages 18–22. IEEE, 2014.

[12] Manfredo Atzori, Matteo Cognolato, and Henning Müller. Deep learning with convolutional neural networks applied to electromyography data: A resource for the classification of movements for prosthetic hands. *Frontiers in neurorobotics*, 10, 2016.

[13] Michael Hüsken and Peter Stagge. Recurrent neural networks for time series classification. *Neurocomputing*, 50:223–235, 2003.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[16] Wenpeng Yin, Katharina Kann, Mo Yu, and Hinrich Schütze. Comparative study of cnn and rnn for natural language processing. *arXiv preprint arXiv:1702.01923*, 2017.

[17] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[18] Manfredo Atzori, Arjan Gijsberts, Ilja Kuzborskij, Simone Elsig, Anne-Gabrielle Mittaz Hager, Olivier Deriaz, Claudio Castellini, Henning Müller, and Barbara Caputo. Characterization of a benchmark database for myoelectric movement classification. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 23(1):73–83, 2015.

[19] John Webster. *Medical instrumentation: application and design*, chapter 4, pages 144–146,242. John Wiley & Sons, 4th edition, 2015.

[20] Melvin M Morrison. Inertial measurement unit, December 8 1987. US Patent 4,711,125 A.

[21] Agamemnon Krasoulis, Iris Kyranou, Mustapha Suphi Erden, Kianoush Nazarpour, and Sethu Vijayakumar. Improved prosthetic hand control with concurrent use of myoelectric and inertial measurements. *Journal of neuroengineering and rehabilitation*, 14(1):71, 2017.

[22] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[23] Ivan Nunes Da Silva, Danilo Hernane Spatti, Rogerio Andrade Flauzino, Luisa Helena Bartocci Liboni, and Silas Franco dos Reis Alves. *Artificial Neural Networks: A Practical Course*, pages 6–7,34,44–47. Springer, 2016.

[24] Donald O Hebb et al. *The organization of behavior: A neuropsychological theory*. New York: Wiley, 1949.

[25] Siegrid Lowel and Wolf Singer. Selection of intrinsic horizontal connections in the visual cortex by correlated neuronal activity. *Science*, 255(5041):209, 1992.

[26] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[27] Bernard Widrow and Marcian E Hoff. Adaptive switching circuits. Technical report, STANFORD UNIV CA STANFORD ELECTRONICS LABS, 1960.

[28] Marvin Minsky and Seymour Papert. Perceptrons. 1969.

[29] Dennis W. Ruck, Steven K. Rogers, Matthew Kabrisky, Peter S. Maybeck, and Mark E. Oxley. Comparative analysis of backpropagation and the extended kalman filter for training multilayer perceptrons. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(6):686–691, 1992.

[30] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[31] Bernard Widrow and Michael A Lehr. 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, 1990.

[32] Matthew D Zeiler, M Ranzato, Rajat Monga, Min Mao, Kun Yang, Quoc Viet Le, Patrick Nguyen, Alan Senior, Vincent Vanhoucke, Jeffrey Dean, et al. On rectified linear units for speech processing. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 3517–3521. IEEE, 2013.

[33] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 2017.

[34] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

[35] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

[36] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[37] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick van der Smagt, Daniel Cremers, and Thomas Brox. Flownet: Learning optical flow with convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2758–2766, 2015.

[38] Pierre Baldi, Søren Brunak, Yves Chauvin, Claus AF Andersen, and Henrik Nielsen. Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics*, 16(5):412–424, 2000.

[39] Jan Gorodkin. Comparing two k-category assignments by a k-category correlation coefficient. *Computational biology and chemistry*, 28(5):367–374, 2004.

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

[41] Lauren H Smith, Levi J Hargrove, Blair A Lock, and Todd A Kuiken. Determining the optimal window length for pattern recognition-based myoelectric control: balancing the competing effects of classification error and controller delay. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 19(2):186–192, 2011.

[42] The microsoft cognitive toolkit. https://docs.microsoft.com/en-us/cognitive-toolkit/. Accessed: 20-09-2017.

[43] Keras: The python deep learning library. https://keras.io/. Accessed: 03-10-2017.

[44] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[45] Peng Xia, Jie Hu, and Yinghong Peng. Emg-based estimation of limb movement using deep learning with recurrent convolutional neural networks. *Artificial organs*, 2017.

[46] Zhisheng Wang, Jun Lin, and Zhongfeng Wang. Accelerating recurrent neural networks: A memory-efficient approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2763–2775, 2017.

[47] Tsvi Achler. Input shunt networks. *Neurocomputing*, 44:249–255, 2002.

[48] Tsvi Achler and Eyal Amir. Input feedback networks: Classification and inference based on network structure. *Frontiers in artificial intelligence and applications*, 171:15, 2008.

[49] Tsvi Achler. Evaluating the role of feedback regulation in recognition processing. *BMC Neuroscience*, 11(S1):P54, 2010.

A

Appendix

Introduction

Originally, the purpose of this thesis was to investigate a different algorithm for pattern recognition, namely Regulatory Feedback Networks (RFN). Since there was a change in topic which caused some delay in the conclusion of the thesis, it seems appropriate to justify it. To that end, there will be a short section summarising RFNs, the reason why they seemed to be a promising topic and ultimately, the reason why they were replaced by the more mainstream DNN.

RFNs constitute a type of neural network structure developed by Tsvi Achler [47], initially named input shun networks. The interest for this algorithm arose from the fact that, in some test cases, RFNs seemed to outperform traditional pattern recognition algorithms, such as MLPs, SVMs, *etc.* when dealing with simultaneous patterns. In the context of multifunction myoelectric control, where the correct recognition of complex movement patterns is essential, this algorithm showed promise.

Theory

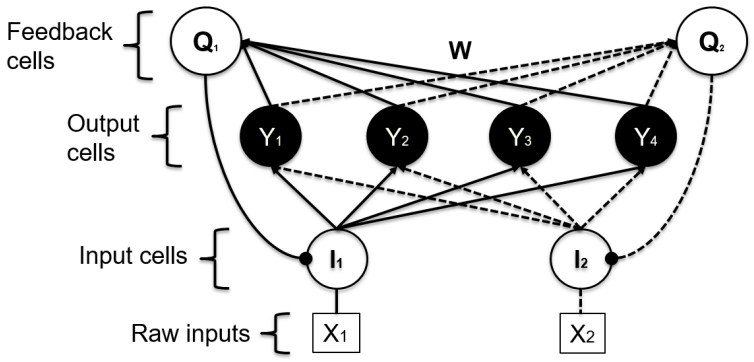


Figure A.1: Regulatory Feedback Network with two input cells and four output cells. The number of feedback cells are always the same as of input cells, since they regulate the state of the input.

In figure A.1 one can see the architecture of a simple RFN. A key differentiating factor for this architecture is that the classification decision is made during test phase by repeatedly inhibiting some inputs in favour of others. While most machine

learning algorithms have long training periods, and fast test phases, this is reversed for RFNs. In reality, there is no learning mechanism to *train* an RFN. Instead, the connectivity matrix, $W \in \mathbb{R}^{m \times n}$, has to be defined, where m is the number of output classes and n is the number of input features.

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} \tag{A.1}$$

As for ANNs, to calculate the output, the connectivity matrix is multiplied with the input vector. However, though the inputs X_i corresponding to the raw feature vectors to be classified, are kept constant, the input cells $I_i(t)$ are updated at each iteration based on salience at the output cells connected to it, $Q_i(t)$ of the previous iteration. Therefore, to compute the output, first the input must be updated:

$$Q_i(t) = \sum_{j=1}^{m} Y_j(t) w_{ji} \tag{A.2}$$

$$I_i(t) = \frac{X_i}{Q_i(t)} \tag{A.3}$$

The whole feedback equation to update the input cell can be seen below:

$$I_i(t) = \frac{X_i}{\sum_{j=1}^{m} Y_j(t)} \tag{A.4}$$

Finally, the complete update rule is presented:

$$Y_j(t + \Delta t) = \frac{Y_j}{n} \sum_{i=1}^{n} I_i(t) w_{ji} \tag{A.5}$$

Here it becomes evident that the output depends not only on the raw input and connectivity matrix. The output will change at each iteration, depending on the output at the previous step and the regulated input. The network was proven to eventually converge to a decision [48].

Discussion

Achler hypothesised in his poster: "Evaluating the role of feedback regulation in recognition processing" [49], that the role of feedback connections in sensory regions of the brain was to regulate or conserve information. He further claimed that RFNs, with their recurrent structure, are able to obtain unparalleled performance with simultaneous patterns, by down-regulating inputs that activate the output too vigorously.

In the same poster, the performance of the RFN algorithm was compared to that of SVM, ANN and Lateral Inhibition algorithms (e.g. winner-take-all) in simultaneous pattern recognition. The results show that while the performance of the RFN stays at 100% with increasing number of overlapping patterns, the other algorithms seem to fail. Despite the impressive results, there are a few thing to take into account:

1. All networks were trained only with the single patterns. Typically, this means that the connectivity matrix of the RFN was generated by averaging the training samples of each target into one representative feature vector. For the other algorithms, this means that there was a learning mechanism to map the input vectors to the respective target outputs.
2. The simultaneous patterns were generated by adding overlapping features of multiple learned patterns.
3. The percentage of correct compositions was computed by checking if the k most-active cells corresponded to the correct k target outputs.

The reason why all of these remarks are extremely important is that in the desired context, namely simultaneous movement recognition, the problem is much more complex than this dummy experiment. In order to achieve such a high performance, one would need:

1. The feature vectors for each single movement would have to be sufficiently different between movements. Which, if this were the case, simpler decision algorithms could be used instead.
2. The feature vector of the simultaneous movements would have to be, if not added together such as in the poster at least some kind of linear combination between the constituting single movements.
3. There would have to be a separate decision mechanism to predetermine how many movements, k, are present for each input vector.

Conclusion

The overall conclusion is that even though the algorithm works under a specific set of conditions, it seems that it cannot be directly applied to pattern recognition based on myoelectric signals, in the context of simultaneous patterns. Although there may be other strategies to improve the performance of RFNs, there is a limited amount of literature available on this type of network, having all related publications been from the same author. Therefore, a decision was made to look for alternative algorithms.