UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Industrial

# Analysis of feedforward and LSTM deep neural network architectures for self-supervised regression of sEMG signals for multi-grasp robot hand control

Trabajo Fin de Grado

Grado en Ingeniería en Tecnologías Industriales

AUTOR/A: Soto Alapont, Josep

Tutor/a: Picó Marco, Jesús Andrés

CURSO ACADÉMICO: 2022/2023

## Index

# 1. Introduction

In recent years, the area of robotics has made major advances, changing a variety of sectors and domains. Robots have stopped being limited to regulated environments, and are already being used in real-world circumstances to aid people in a variety of jobs. The design and implementation of functional and accessible human-robot interfaces (HRIs) is critical for enabling fluid communication and cooperation between people and robots [1].

This chapter provides the thesis's introduction, offering the relevant background, research context, and motivation for the study of surface Electromyography (sEMG)-based interfaces for operating robotic hands. It describes the scope and boundaries of the study, as well as the methodology and approach that will be used to attain the research aims, and it specifies the research issues and goals that will lead the thesis. A summary of the thesis is given, laying the groundwork for the next chapters, which will go further into the theoretical foundations, experimental procedures, and data analysis.

## 1.1 Research context and motivation

The study background is based on the growing need in the field of robotics for intuitive and natural control platforms. Traditional HRIs frequently employ sophisticated manual input devices like joysticks that need substantial training and coordination, restricting their usability and preventing widespread adoption [1].

sEMG-based interfaces, on the other hand, allow users to operate robotic hands using their own neurological signals by monitoring and interpreting muscle electrical responses. However, developing accurate and stable control algorithms remains difficult, necessitating sophisticated machine learning approaches that are able to address the computational complexity required for the implementation of this technology [2].

This technique has great potential for numerous engineering and medical applications, such as enhancing the quality of life for people with motor impairments or assistive technologies that can be used in hazardous situations, where human workers'

security cannot be guaranteed, as well as the enhancement of productivity systems in the industrial production sector.

### 1.2 Research questions and objectives

The purpose of this thesis is to assess and contrast the effectiveness of two deep neural network architectures, particularly Feedforward Neural Networks (FNNs) and Long Short-Term Memory Neural Networks (LSTMs), for self-supervised regression of sEMG signals in the context of multi-grasp robotic hand control. The following are the major research questions that will lead this study:

- How successfully can FNNs and LSTMs map the relationship between sEMG signals and robotic hand movements?
- What are the advantages and disadvantages of FNNs and LSTMs for self-supervised regression of sEMG signals?
- Which design outperforms the other in terms of accuracy for robotic hand control?

By answering these questions, this study hopes to provide light on the feasibility and efficacy of various deep neural network designs for sEMG-based robotic hand control, therefore contributing to the evolution of human-robot interface technologies.

### 1.3 Scope and limitations

To guarantee reasonable expectations and attainable outcomes, the extent and constraints of the study must be established. The scope of this research is to examine and compare FNNs and LSTMs as deep neural network architectures for self-supervised regression of sEMG data. The study makes use of a collection of sEMG signals acquired from human individuals while doing multi-grasp activities.

However, certain limits have to be acknowledged. The primary focus of the research case will be on three types of grasping activities: tripodal grip, ulnar grasp, and power grasp. These grasping activities are a subset of the extensive spectrum of hand movements and allow for a decrease in computer analysis complexity, simplifying the control task. Furthermore, the study does not implement alternate signal processing

approaches for sEMG feature extraction other than the Non-Negative Factorization Matrix (NMF) algorithm, nor does it investigate other machine learning algorithms beyond FNNs and LSTMs. Additionally, the study is restricted to offline evaluation of the algorithms' performance and does not take into account real-time implementation concerns.

### 1.4 Methodology and approach

The approach adopted for tackling this study case is based on a systematic methodology that addresses the proposed research questions for completing the general objectives of the research, which comprises the following main steps:

1. Reviewing pertinent literature on human-robot interaction, sEMG-based interfaces, and sEMG signal analysis machine learning methods.
2. Creating and executing a data collecting experimental setup for multi-grasp activities.
3. Using non-negative matrix factorization (NMF) to extract features and reduce the dimensionality of sEMG signals.
4. Using both FNN and LSTM models to perform self-supervised regression on preprocessed sEMG data.
5. Once both algorithms are trained and optimized, the performance evaluation and comparison between FNN and LSTM is carried out in terms of their accuracy.
6. Finally, analyzing the findings and evaluating the study's implications, limits, and potential future directions.

Overall, this research seeks to contribute to current knowledge in the field and give significant understanding into the application of deep neural networks for sEMG-based multi-grasp robotic hand control by employing the proposed methodology.

## 2. Theoretical Framework

This thesis' theoretical framework chapter provides a detailed summary of the concepts and theories that constitute the foundation of the research analysis. This chapter

looks into the theoretical elements of human-robot interaction/interface (HRI), surface electromyography (sEMG)-based interfaces, and machine learning methods, all of which are necessary for understanding the next chapters.

First, the HRI domain is investigated, which is concerned with the creation of a platform that enables users and robots to interact. It will be addressed the concept of human-robot interaction (HRI) and its importance to this study, emphasizing the challenges and possibilities involved with building successful interfaces for human-robot cooperation.

Following that, sEMG-based interfaces will be explored, which are used to operate robotic equipment via muscle activity. It is explained in detail how to acquire sEMG signals, identify features, and operate robotic hands with sEMG signals.

Finally, the machine learning methods that are used to interpret sEMG data and generate the appropriate commands to operate robotic devices are looked over in detail. The focus is on the Non-negative Matrix Factorization (NMF), Feedforward Deep Neural Network (FDNN), and Long Short-Term Memory Deep Neural Network (LSTM) algorithms, which are typically used in sEMG-based investigations. The ideas underlying each algorithm are explained, as well as their benefits and drawbacks.

Overall, this chapter establishes the foundations for the rest of the thesis, giving a theoretical foundation for the empirical work in the coming chapters. This chapter attempts to create a basis for the succeeding chapters' research and analysis by reviewing the present level of knowledge in the domains of HRI, sEMG-based interfaces, and machine learning techniques.

## 2.1 Human-robot interaction/interface (HRI)

Because of the growing interest in integrating robots in a variety of applications, the topic of human-robot interaction (HRI) has received a lot of attention in recent years. HRI refers to the collaboration of human operators and robots in industrial and service environments to execute tasks; these platforms can be used to control robots, supply feedback from their activities and supervise their performance. Surface electromyography

(sEMG) technology is one of the most promising ways for achieving effective HRI, but it has considerable practical difficulties [3].

The detection of electrical signals generated by muscular contractions allows sEMG technology to assess muscle activity. It is feasible to deduce human operator motions from sEMG signals and make use of this knowledge to operate robotic equipment. This technology has the potential to improve HRI significantly by allowing for more natural and intuitive communication between people and robots [2],[4].

However, developing viable interfaces for sEMG-based HRIs requires overcoming a number of obstacles. One of the most challenging difficulties is creating simple and user-friendly interfaces that allow operators to properly manage robots. Furthermore, the complexity of sEMG signals, as well as individual variability, offer obstacles in effectively understanding the signals and converting them into robot control [3].

Despite these obstacles, the potential for sEMG-based HRIs is immense. sEMG technology, for example, may be used to operate prosthetic devices, allowing persons with impairments to regain movement and freedom. Furthermore, sEMG-based HRIs can be utilized to improve production processes by allowing robots to adapt to human operators' preferences and motions [3].

Finally, sEMG-based HRIs provide a viable option for furthering the study of human-robot interaction. Building viable interfaces for sEMG-based HRIs, on the other hand, necessitates tackling considerable issues in interface design, signal interpretation, and feature recognition precision. By addressing these obstacles, the potential benefits of sEMG-based HRIs can be achieved, resulting in safer, more efficient, and intuitive human-robot collaboration [2],[4].

### 2.2 Surface Electromyography (sEMG)-based interfaces

Surface electromyography (sEMG) is a non-invasive method that has grown in popularity in the field of human-robot interaction (HRI) because of its use as a tool for human intention identification and gesture-based control. sEMG is a method that uses

surface electrodes put on the skin to collect and record the electrical activity of skeletal muscle contractions. The analysis of sEMG signals has been demonstrated to be an excellent method for building intuitive and natural control interfaces for robotic devices, particularly for applications requiring a high level of skills in performing certain manual tasks, such as multi-grasp hand control [1].

The usage of sEMG-based interfaces has grown in popularity in recent years, thanks to developments in technology and algorithms that enable real-time processing and interpretation of sEMG signals. The desire to build natural, intuitive control interfaces for robotic systems, particularly for those with motor impairments or disabilities, has fueled the development of sEMG-based interfaces[1].

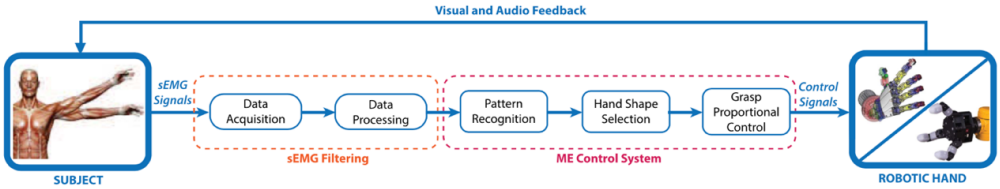


Figure 1: sEMG-based HRI diagram [2]

This part of the thesis gives an overview of sEMG-based interfaces, such as signal collection, feature labeling, and robotic hand control. The section looks at the several methods for acquiring and processing sEMG signals, such as signal filtering, noise reduction, and feature extraction. Its overall goal is to give a thorough knowledge of sEMG-based interfaces and their potential for designing natural, intuitive control interfaces for robotic systems.

### 2.2.1 sEMG signal acquisition

Data gathering is essential in sEMG-based human-robot interaction (HRI) systems. The quality of sEMG signals is significant for the robotic control precision. Hence, it is critical to employ reliable acquisition equipment that can effectively enhance the signal-to-noise ratio (SNR). To keep as much relevant signal information as possible, the acquisition frequency is often adjusted to 500–1000 Hz [1].

The location of the sEMG signal acquisition is also crucial. The midline of the muscular belly between the closest innervation zone and the myotendinous junction is the preferred position. To decrease noise, the skin in this region should be washed with alcohol before signal collection, and hair should be removed if necessary. The electrodes used in sEMG signal collection are also important. Wet electrodes, which require an electrogel layer between the skin and the electrode, are better suited for fundamental theoretical study, whereas dry electrodes, which are put directly on the skin, are more suited for practical applications [1].

The number of electrode channels used for sEMG signal capture is determined by the number of muscles associated with the desired human limb motions. It is critical to choose an appropriate number of electrode channels to achieve adequate recognition accuracy for human movement intentions while minimizing the amount of calculation necessary [1].

In a nutshell, the quality of the obtained sEMG signals has a significant impact on the precision of robotic control in sEMG-based HRI systems. High-quality sEMG signals may be acquired by employing reliable collection equipment and following precise acquisition protocols, providing a solid data foundation for future pattern identification and robotic control in sEMG-based HRI systems [1].

### 2.2.2 sEMG feature labeling

The act of giving semantic meaning to sEMG signals, known as feature labeling, is critical for detecting specific gestures or motions. This entails matching certain sEMG patterns to related hand motions.

The Non-Negative Matrix Factorization algorithm (NMF) is one way for collecting grip labels. For this particular study case, NMF is based on the concept of muscular synergies, which are patterns of coordinated muscle activation that are related with specific motions or grasps. The underlying muscle activation patterns may be recovered by decomposing the sEMG signals into a linear combination of muscle synergies using the NMF factorization method (the technical aspects about how the algorithm works will be explained in the following sections). Each muscular synergy is a

unique mix of muscle activations linked with a certain grip. These muscle synergies may subsequently be associated with certain grasps, thereby labeling the sEMG signals. This method has the benefit of capturing the underlying interdependence of muscle activations, resulting in a more robust and interpretable grip categorization [4].

For sEMG categorization, the NMF method has some particular characteristics that makes them to outperform conventional factorization-based labeling algorithms in various ways:

- Non-Negativity Constraint: since muscle activations are intrinsically non-negative, it matches well with the physiological features of sEMG signals; it gives a more interpretable and valid depiction of muscle synergies.
- Shortened representation: it tends to employ a small number of muscle synergies for analyzing the sEMG signals; it is favorable in terms of computing efficiency and interpretability (more compact data representation and lower dimensionality of the feature space).
- Computational efficiency: it is especially important in real-time applications where low-latency processing is critical for responsive control of robotic devices; the NMF algorithm allows quicker analysis and identification of sEMG signals, enabling real-time control and interaction between humans and robots.

Overall, the specific properties of the NMF algorithm makes it a suitable method for this particular study case, as its performance for different error metrics when carrying out the grasps' labeling beats those of other possible algorithms as showed in [4].

Furthermore, since they record the interaction between muscle activations and joint configurations, postural synergies can aid in the learning of grab labels. Substantial relationships can be established by studying this connection throughout various gripping activities. According to [4], it is reasonable to suppose that a unique synergistic DoF is generated during the opening/closing action of a hand, which significantly decreases the level of sophistication of the sEMG generative model.

### 2.2.3 sEMG-based control of robotic hands

Regarding the control of the robotic device, several stages must be completed before the sEMG-based HRI can get the correct commands to operate the robotic hand.

First, the neural drives that are triggered to perform the various movements must be identified. They correspond to the labels created by the NMF algorithm, as described in the preceding section. Once the various grasping motions have been encoded, a Deep Neural Network may be trained using the obtained labels to learn the mappings for the defined grasps [5].

This step will be addressed with two different approaches, using two supervised machine learning algorithms whose performance will be compared to determine the best solution; the mappings will be computed using first a typical Feedforward DNN and then a Long-Short Term Memory NN; this second choice is justified due to the time-series nature of the sEMG dataset being used. Finally, once trained, the NN-based grip classification model may be used to categorize real-time sEMG signals and control the robotic hand accordingly. The anticipated grasp labels are converted into control commands, which activate the robotic hand and allow it to make the necessary grasping actions.

## 2.3 Machine learning algorithms

Surface electromyography-based interfaces for human-robot interaction systems rely heavily on machine learning techniques. These methods allow relevant information to be extracted from sEMG signals, allowing for reliable interpretation and control of robotic equipment. This paper focuses on the following machine learning algorithms due to their notorious compatibility with the study case that it addresses: Non-negative Matrix Factorization, Feedforward Deep Neural Networks, and Long Short-Term Memory Neural Networks.

NMF, a matrix factorization approach, has recently acquired prominence for its capacity to breakdown sEMG signals into their underlying components, known as muscle synergies. This method results in a more compact representation of the sEMG data and

makes it easier to extract useful characteristics for control purposes. The use of NMF in sEMG-based HRI systems has led to encouraging results in terms of enhancing control interface accuracy and resilience [4].

Feedforward Deep Neural Networks have also been widely used in sEMG-based HRI systems. These neural networks may learn complicated correlations between sEMG signal patterns and associated robotic operations. They can accurately categorize and interpret sEMG signals by training DNNs on huge datasets, allowing precise control of robotic equipment. DNNs have made significant advances in the field, notably in the areas of motion control [2].

Long Short-Term Memory Neural Networks have also developed as a strong tool for processing time series data, making them well-suited to studying sEMG signals. The temporal relationships contained in sequential sEMG data may be successfully captured by LSTM networks, allowing for reliable prediction and control of robotic devices across time. Because of this, LSTM networks are particularly useful for applications involving dynamic and continuous movement [6].

It can be improved the performance and usability of sEMG-based HRI systems by evaluating and comprehending these machine learning methods. This section will dive into the concepts, implementation, and possible uses of NMF, Feedforward DNNs, and LSTM networks in the context of sEMG-based interfaces, emphasizing their value to the field of HRI and their potential to strengthen human-robot control and communication capacities.

### 2.4.1 Non-negative Matrix Factorization (NMF)

Non-Negative Matrix Factorization (NMF) is a technique for reducing dimensionality in machine learning and data analysis. Its goal is to decompose a given matrix into two smaller matrices with non-negative entries. NMF seeks a low-rank estimate of the initial matrix which reflects relevant trends or characteristics in the data [7].

NMF works on the principle of representing the starting matrix as a linear combination of non-negative basis vectors, each weighted by non-negative coefficients. Assume an input matrix X with dimensions m x n, where m is the number of samples or observations, and n is the number of features or variables. NMF's purpose is to factorize X into two non-negative matrices, W and H, such that $X \approx W \cdot H$. This enables a representation by parts of the data, in which each basis vector records a separate feature or pattern, and the coefficients indicate the contribution of each feature to reconstructing the original matrix [7].

The NMF algorithm is generally updated in an iterative way. The process begins by randomly initializing the matrices W and H, which contain the basis vectors and coefficients, with non-negative values. The matrices are then iteratively updated to reduce the reconstruction error between the original matrix and its estimate. The update procedure involves optimizing one matrix while correcting the other: first, fix H to update W by reducing the reconstruction error among X and WH, and then, fix W to update H by lowering the reconstruction error among X and WH. Optimization methods such as multiplicative updates and gradient descent can be used. The method updates the matrices until convergence, or a predetermined stopping condition happens; NMF converges when the matrices W and H have been modified to as nearly approach the input matrix X as feasible while ensuring non-negativity [4],[7].

The generated basis vectors and coefficients can be understood once the NMF algorithm has converged: matrix W (m x r) represents the basis vectors or components that capture the underlying structure or features in the data, with each column of W corresponding to a different feature; matrix H (r x n) contains the weights that determine the contribution of each basis vector in reconstructing the original matrix X, with each row of H corresponding to the activation of the associated basis vector [4],[7].

In a nutshell, NMF can find relevant patterns or features in data by deconstructing the original matrix X into non-negative components. It has been employed in tasks such as feature extraction and pattern recognition.

### 2.4.2 Deep Feedforward Neural Network (DFNN)

Deep Feedforward Neural Networks (DFNNs), which are also referred to as multilayer perceptrons (MLPs), have become known as effective models for tackling challenging machine learning problems. DFNNs have transformed the area of machine learning by discovering detailed patterns and correlations in data.

DFNNs are made up of numerous layers of artificial neurons that are linked together. A DFNN architecture has three basic components [8]:

- The input layer is made up of neurons that receive input data. Each neuron reflects a different aspect or characteristic of the input. The dimensionality of the input data determines the number of neurons in the input layer.
- One or more hidden layers can be placed between the input and output layers in DFNNs. Each hidden layer is made up of several neurons that analyze and transform data from the preceding layer. These layers are in charge of identifying and learning complex trends and representations from input data.
- The output layer is responsible for the network's final predictions or outputs. The amount of neurons in the output layer is determined by the type of problem being handled. A binary classification job, for example, requires a single neuron, but multi-class job classification may require several neurons, each representing a different class.

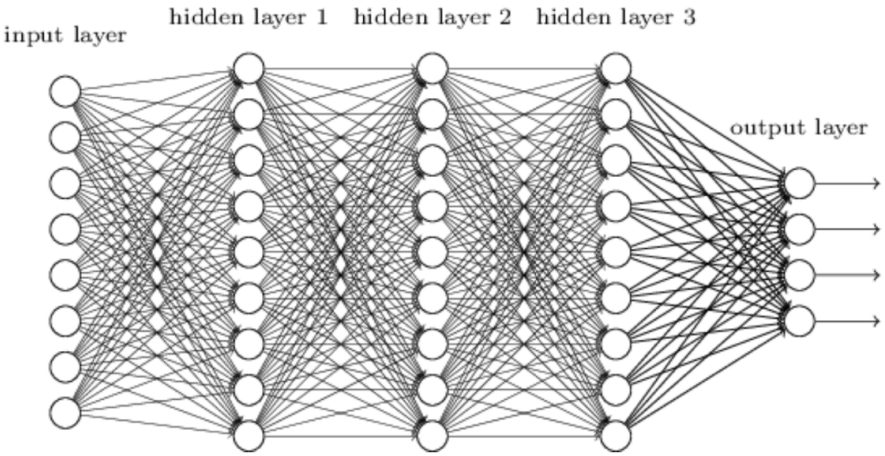


Figure 2: Deep Neural Network diagram [8]

A supervised learning strategy is widely used to train a DFNN. It entails feeding the network a labeled dataset and repeatedly changing the weights to minimize the discrepancy between the network's anticipated outputs and the real labels. The training procedure may be broken down into the phases that follow [8]:

- Forward Propagation: The input data is supplied into the network during forward propagation, and the activations of every neuron in consecutive layers are calculated. Each neuron computes the weighted total of its inputs, implements a function of activation, and sends the result to neurons in the following layer. This procedure is continued up to the output layer has been reached, at which point the final predictions are generated.
- Loss Calculation: The activations of the output layer are evaluated against the ground truth labels employing a specified loss function. Mean squared error (MSE) is a common loss function for regression tasks and cross-entropy for classification tasks. The loss measures the difference between the expected and real labels.
- Backpropagation: Applying the chain rule of calculus, the loss is transmitted backwards across the network's structure, layer by layer. This procedure determines how much each weight contributes to the total inaccuracy. The gradients of the loss are computed with respect to the weights, and the weights are changed as needed to minimize the loss.
- Weight Update: Optimization methods like as stochastic gradient descent (SGD) or its derivatives (e.g., Adam) are used to update the weights linking neurons. These techniques gradually improve the network's capacity to generate correct predictions by updating the weights according to the gradients obtained during backpropagation.

Activation functions are important in DFNNs because they introduce non-linearities into the network's structure. They enable the network to simulate complicated and nonlinear data interactions. Among the most commonly utilized activation functions are [8]:

- The sigmoid function converts the weighted total of inputs into a number between 0 and 1. It is smooth and limited, making it suitable for binary classification jobs at the output layer.
- The Rectified Linear Unit (ReLU), when activated, sets negative inputs to zero while positive inputs remain intact. Its appeal stems from its simplicity and computing effectiveness; ReLU activation is frequently used in the hidden layers of DFNNs.
- The hyperbolic tangent (tanh) function, similarly to the sigmoid function, translates values to an interval between -1 and 1. It has greater gradients than the sigmoid function, which makes it helpful in situations when stronger activations are requested.

In summary, Deep Feedforward Neural Networks (DFNNs) have showed outstanding potential by enabling the effective modeling of complicated connections in data through their layered design, iterative training process, and integration of activation functions.

### 2.4.3 Long Short-Term Memory Neural Network (LSTM)

Long Short-Term Memory (LSTM) neural networks are establishing themselves as a strong tool for processing sequential data in the field of artificial intelligence and machine learning. When faced with long-term dependencies, LSTM networks are meant to overcome the constraints of classic neural networks, such as Feedforward NNs, by using their capacity to acquire and store information over long periods of time [9].

Feedforward neural networks, for example, struggle to grasp long-term dependencies in sequential input. LSTM networks solve this issue by including memory cells and gating systems that allow for long-term information retention as well as efficient transmission and control of the flow [9].

An LSTM network is made up of linked memory cells that function as independent processing units. Each memory cell must have three components: an input gate, a forget gate, and an output gate; they control the flow of data into and out of memory cells, allowing for selective memory preservation and access [9].

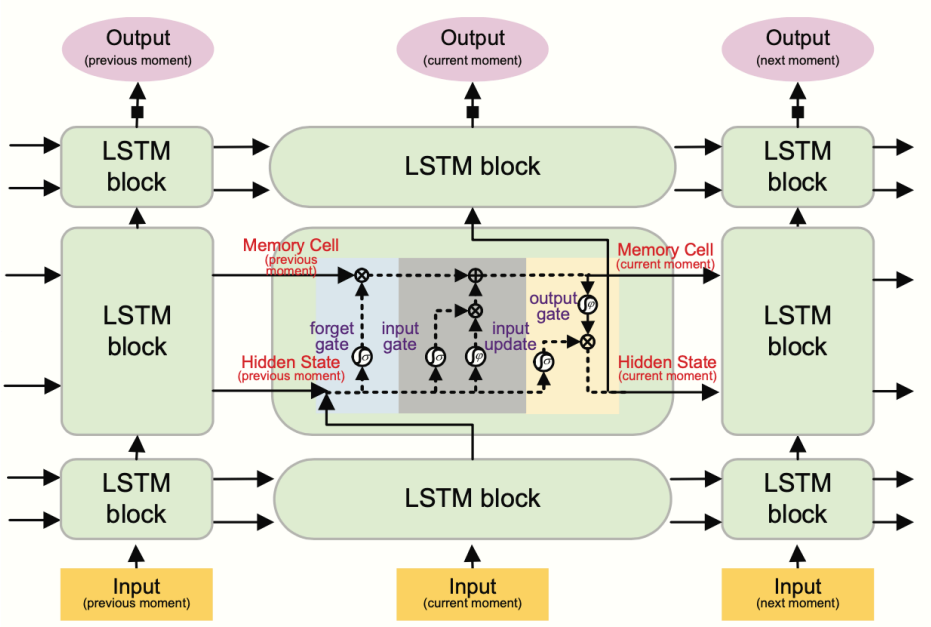


Figure 3: LSTM Neural Network diagram [6]

The LSTM memory cell gets two inputs at each time step: the present input data and the prior memory cell state, which includes short-term as well as long-term memory. The input gate sends them via a sigmoid activation function, and the result is multiplied by a candidate cell state, representing the new information that might be added to the memory cell; it preferentially saves new information by weighing the relevance of the current input [9].

The forget gate controls what information from the memory cell state should be deleted, enabling the system to forget useless or obsolete information. It uses the prior hidden state and the current input, runs them across a sigmoid activation function, and returns a value between 0 and 1 for each cell state item. Depending on the forget gate's outcome, multiplying this forget gate value component by component with the prior cell state eliminates or maintains the necessary information [9].

The output gate determines the amount of data gathered from the memory cell state is accessible to the following layers. It considers the current input and the prior hidden state, runs them through a sigmoid activation function, then multiplies the result

by the candidate cell state that has been run through a tanh activation function. This gating method can regulate the quantity of information that a memory cell provides to the broader prediction process [9].

LSTM networks use backpropagation through time (BPTT) during the training phase to adjust the network's parameters and maximize its performance. The error signals are transmitted not just from the current time step but also from subsequent time steps, complicating the learning process but allowing the network to successfully understand long-term dependencies [9].

To summarize, LSTM networks provide selective information storage and retrieval by adding memory cells and gating processes, allowing for greater modeling of complicated temporal interactions.

## 3. Materials and Methods

The materials and methods chapter describes in detail the experimental setup, data gathering, and preprocessing methods, as well as the development and assessment of the machine learning models used in this work. The goal of this chapter is to discuss the techniques used to acquire and process data, as well as the algorithms used to evaluate sEMG signals and operate the robotic hand.

First, the experimental setup, including some general aspects about hardware and software components employed to gather the sEMG signals, will be described. The processes performed to preprocess the sEMG data, including filtering and normalization approaches, will be next explained.

The chapter will next go into the various machine learning techniques employed in this work, beginning with non-negative matrix factorization (NMF) for feature extraction and dimensionality reduction. The implementation of the NMF algorithm, including the selection and optimization of algorithm parameters, will be described.

Finally, it will also go through the implementation and training of the feedforward deep neural network (DNN) and long short-term memory (LSTM) NN models, which are exploited for performing a self-supervised regression of the sEMG data to compute the mapping of the different grasp gestures. Each model's performance evaluation, including the measures employed, will be discussed.

### 3.1 Experimental setup

A sEMG acquisition system was built in order to collect the user's data for developing the sEMG-based HRI for the control of a robotic hand. In the purpose of this project, eight sEMG channels (eight pairs of electrodes in differential arrangement) from the user's forearm muscles are collected. Disposable surface skin electrodes with conductive gel are employed. The electrodes are spread evenly across the forearm, making an armband (gForcePro). The sEMG bracelet is designed to offer data on hand movements as well as grip closure. As a result, it is focused on the *Flexor Digitorum Superficialis* and *Extensor Digitorum Communis* muscles, which play a role in digit flexion and extension [2].

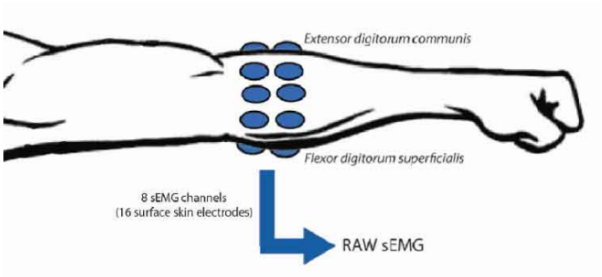


Figure 4: sEMG electrodes placement representation [4]

This application's wearable sensor node is built on a 6-layer printed circuit board. The node is intended for the capture of analog biological signals in wearable multisensory technologies using Cerebro, a high-performance analog front end (AFE) coupled to an ARM Cortex M4 microprocessor via serial peripheral interface. Data is collected at 1 kHz and sent to a PC using an average 2.0 Bluetooth interface [2].

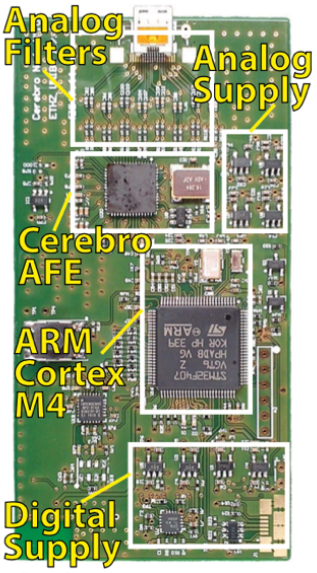


Figure 5: Circuit board architecture diagram [2]

Finally, this particular study case is developed for a subsequent experimental simulation to try the designed sEMG-based HRI using the AR10 Robot Hand by Activat8 Robots, a lightweight anthropomorphic robot hand with 5 fingers and 10 degrees of freedom (DoFs), as mentioned in [10].

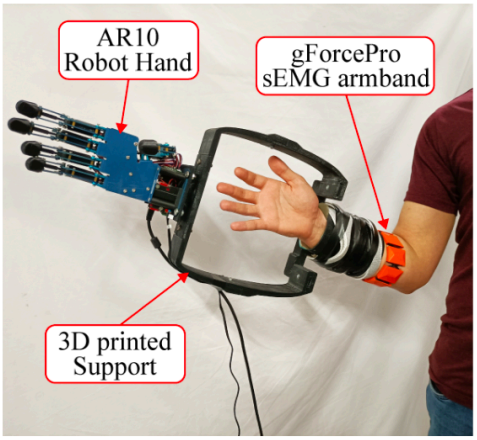


Figure 6: sEMG armband for data acquisition and AR10 robot hand simulative setup [10]

This section seeks to lay a strong basis for the following examination and understanding of the collected sEMG data by providing a complete review of the

experimental setup. It emphasizes the significance of proper experiment design and execution in delivering trustworthy and relevant results.

### 3.2 Data acquisition and preprocessing

Acquiring and preparing sEMG data correctly is critical for getting reliable and useful information regarding muscle activation. These processes set the groundwork for further analysis, feature extraction, and control algorithms.

The wearable sensor node mentioned in the preceding section is used to collect the electrical activity of the muscles during the data acquisition step. The acquisition system captures analog sEMG signals, which are subsequently transformed to digital signals using the previously stated Cerebro AFE. Afterwards, a preprocessing procedure based on signals' filtering is applied to the input data to enhance the quality of the readings [2].

A training session is required to acquire the raw data for building subsequently a training dataset, which is consisting of 8-dimensional samples of the RMS values of the sEMG channels, in order to develop the classifier. Open hand, three fingers position, fist, ulnar squeeze, and neutral pose are the five motions to be categorized. A grab transition logic is utilized to employ two motions that are not related with any form (open hand and neutral position). The training dataset to be gathered consists of 6 repetitions of each motion (excluding the neutral stance). Every move must be performed for 3 seconds, followed by 3 seconds in which the user must rest his fingers (neutral position). Between two separate gesture repetition groups, the neutral stance length becomes 6 s. This procedure is done by 5 different subjects to have a more diverse dataset to cover the possible discrepancies that could appear due to the users' physiological variability [2].

The filtering method, applied in the preprocessing stage, that is built for every input channel of the sEMG data consists on: 1) a 50-Hz notch filter for powerline interference cancellation; 2) a 20-Hz band-pass filter that achieves the best compromise between reducing initial noise (primarily thermal, chemical, and motion item noises) and obtaining the intended data material; and 3) the signal's root mean square (RMS) value estimated on a 200-ms window with no overlap [2].

## 3.3 Non-negative matrix factorization for sEMG signals

In the following section, it will be examined how NMF may be used to analyze and decode sEMG signals in the framework of human-robot interaction (HRI) systems. NMF is an effective method for collecting relevant information from sEMG data, interpreting muscle activity patterns, and improving instinctive control of robotic systems [7].

NMF can be used in the context of sEMG-based HRI systems to decipher the user's intended motions or gestures from the captured sEMG signals. NMF facilitates the conversion of sEMG data into control commands for robotic devices by detecting the underlying muscle synergies and their activation levels. This opens the door to the development of more accessible and productive control interfaces, hence improving the usability and efficacy of HRI systems [4].

In this chapter, it will be analyzed how NMF may be used for sEMG signal breakdown and feature extraction from the point of view of the algorithm implementation. Moreover, NMF algorithmic intricacies will be addressed, such as optimization approaches and parameter choices. This section tries to illustrate the value of NMF in sEMG-based HRI systems by exploiting its power. The use of NMF in sEMG signal processing helps to develop HRI systems by allowing for more organic and smooth interactions among humans and robots.

### 3.3.1 Algorithm implementation

The complete Python script of the NMF algorithm implemented (labeling script) for carrying out the feature labeling of the sEMG dataset can be found in the corresponding appendices' subsection (see Section 8.2). Subsequently, the script is broken down step by step to explain how it works:

The following needed libraries and modules are imported: **numpy** for numerical computations [11], **matplotlib.pyplot** for plotting [12], **sklearn.decomposition** for the

NMF (Non-negative Matrix Factorization) algorithm [13], **pickle** for object serialization [14], **os** for operating system-related operations [15], **scipy.io** for loading MATLAB files [16], and **random** for random number generation [17]. It also loads the **AlphaMatrix** module from the **mapping file** (see section 8.2 for a detailed description of the script functioning and the nature of its content).

The **NMF_Routine** class handles Non-Negative Matrix Factorization (NMF) on a specified input matrix **X**. The NMF approach transforms a matrix into two non-negative matrices: the basis matrix (**W**) and the coefficient matrix (**H**). The following methods are available in the **NMF_Routine** class:

- **_nmf(X):** it is a method for performing NMF on the input matrix **X**. The **NMF** class from the **sklearn.decomposition** package is used. It creates an NMF model with **n_components=2**, implying that it attempts to breakdown **X** into two components. The **init** argument is set to **'random'** to randomly initialize the factorization. The **solver** parameter is set to **mu** to represent the multiplicative updating algorithm. To utilize the Kullback-Leibler divergence as the goal function, the **beta_loss** option is set to **kullback-leibler**. The maximum number of iterations for the NMF method is specified by the **max_iter** option, which is set to 1000. The basis matrix **W** and the coefficient matrix **H** are returned by the procedure [13].
- **compute(X, n_rep=10, diff_signal=True):** this method performs the NMF routine on the input matrix **X** for feature extraction. It takes additional parameters **n_rep** and **diff_signal** which control the number of repetitions of the NMF routine and whether to compute the difference signal or not. The method first calls the **_nmf** method to obtain the coefficient matrix **H** by applying NMF to **X**.

  Next, the NMF routine is repeated **n_rep** times for better smoothing of the features. Each repetition applies NMF to the current coefficient matrix **H** to obtain the updated **H**.

  Following the repetitions, the approach evaluates the shape of **H** to confirm that the signal order is correct (high, low, high). If it is incorrect, it swaps the rows of **H** to restore the right order.

  The normalization step scales each signal in **H** to a value between 0 and 1; **Hn** then, stores the normalized matrix.

The technique either computes the difference signal (**S = Hn[1,:] - Hn[0,:]**) or utilizes **Hn** directly as the extracted features, depending on the value of **diff_signal**.

Finally, the calculated features are normalized to a range of 0-1 before being returned.

Overall, the **NMF_Routine** class's aim is to wrap the NMF feature extraction procedure, offering a reusable and modular method for computing features from myoelectric measurements.

The script defines another class named **Labeler**. This class is in charge of labeling the features retrieved from myoelectric data using the **NMF_Routine** class's Non-Negative Matrix Factorization (NMF) procedure. It handles signal merging and normalization of input data, labels, and reference signals. The following methods are available in the **Labeler** class:

- **__init__(self, data_path, alpha_type, diff_signal, n_rep_nmf):** this is the **Labeler** class's constructor function. It initializes the **Labeler** object with the following parameters: **data_path** (path to the dataset), **alpha_type** (type of alpha matrix for signal merging), **diff_signal** (boolean indicating whether the difference signal should be computed or not), and **n_rep_nmf** (number of repetitions for the NMF routine). The alpha property is also set relying on the **alpha_type** option.
- **merge_1_signal(self, H_pn, H_ul):** using a predetermined alpha matrix, this approach blends the labeled signals for power, pinch, and ulnar grasps. It accepts as input the retrieved characteristics for power (**H_pw**), pinch (**H_pn**), and ulnar (**H_ul**). The alpha matrix is chosen depending on the **Labeler** object's alpha property. The approach multiplies each signal element by element with the relevant alpha values and concatenates the results to generate the merged signal matrix **T**, which is then returned by the method.
- **merge_2_signals(self, H_pn, H_ul):** using a modified alpha matrix, this approach integrates the labeled signals for power, pinch, and ulnar grasps. It accepts as input the retrieved characteristics for power (**H_pw**), pinch (**H_pn**), and ulnar (**H_ul**). The **alpha_bar** modified alpha matrix is constructed by duplicating rows of the original alpha matrix. The procedure multiplies each signal element by element with the matching alpha values from **alpha_bar** and

concatenates the results to generate the merged signal matrix **T**, which is then returned by the method.

- **load_data(self, subj_id):** Loads the myoelectric measurements and reference signals for a certain subject ID. Using the **scipy.io** package, it reads data from the MATLAB (**.mat**) files that contain arrays of the acquired sEMG data. The loaded data is returned in the form of NumPy arrays [11],[16].
- **run_subj(self, subj_id, plot=True):** This function does the feature labeling for a given individual's readings. The subject ID and an optional parameter plot are used to determine whether or not to plot the signals. The method initially invokes the **load_data** function to load the subject's myoelectric values and reference signals. It then iterates through each iteration of the data, doing the following steps:
    1. Uses the **NMF_routine.compute** method to extract characteristics for power, pinch, and ulnar grasps. The procedure receives the arguments **diff_signal** and **n_rep_nmf**.
    2. It invokes the **merge_1_signal** or **merge_2_signals** method to merge the labeled signals depending on the **diff_signal** value.
    3. Normalizes the input, merged and reference signals to a range between 0 and 1.
    4. Saves the data in a dictionary with multiple signal versions.
    5. If **plot** is set to **True**, the plot technique is used to depict the signals.
    6. Finally, for each repeat, it provides the dictionary containing the labeled and normalized data.
- **plot(self, X, T, R, title**): For visualization, this method graphs the myoelectric measurements (**X**), merged signals (**T**), and reference signals (**R**). It employs the **matplotlib** library to generate line plots with labels and legends [12].

Summarizing, the **Labeler** class offers an interface for labeling myoelectric characteristics using NMF and merging the labeled signals for further analysis or robotic device control. It contains the essential stages and allows for the selection of several alpha matrices for signal merging.

Finally, the **if __name__ == "__main__"** block is the entry point of the script. It configures **PLOT** (whether to plot the signals), **SEED** (random seed), **N_REP_NMF**

(number of repetitions for NMF), **ALPHA** (alpha matrix type), and **DATA_PATH** (path to the dataset). Then, it generates an output directory in which to save the labeled dataset and, afterwards, it iterates through each subject from 1 to 5 and labels them using the **Labeler** class. Each subject's labeled data is subsequently saved in a pickle file.

Overall, the script applies NMF to the myoelectric readings, merges and labels the signals, and stores the labeled dataset for further analysis and usage. In the following figure, it can be found an example of the graphical representation of the myoelectric readings, as well as the generated labels for the three different grasps with their corresponding reference signals:

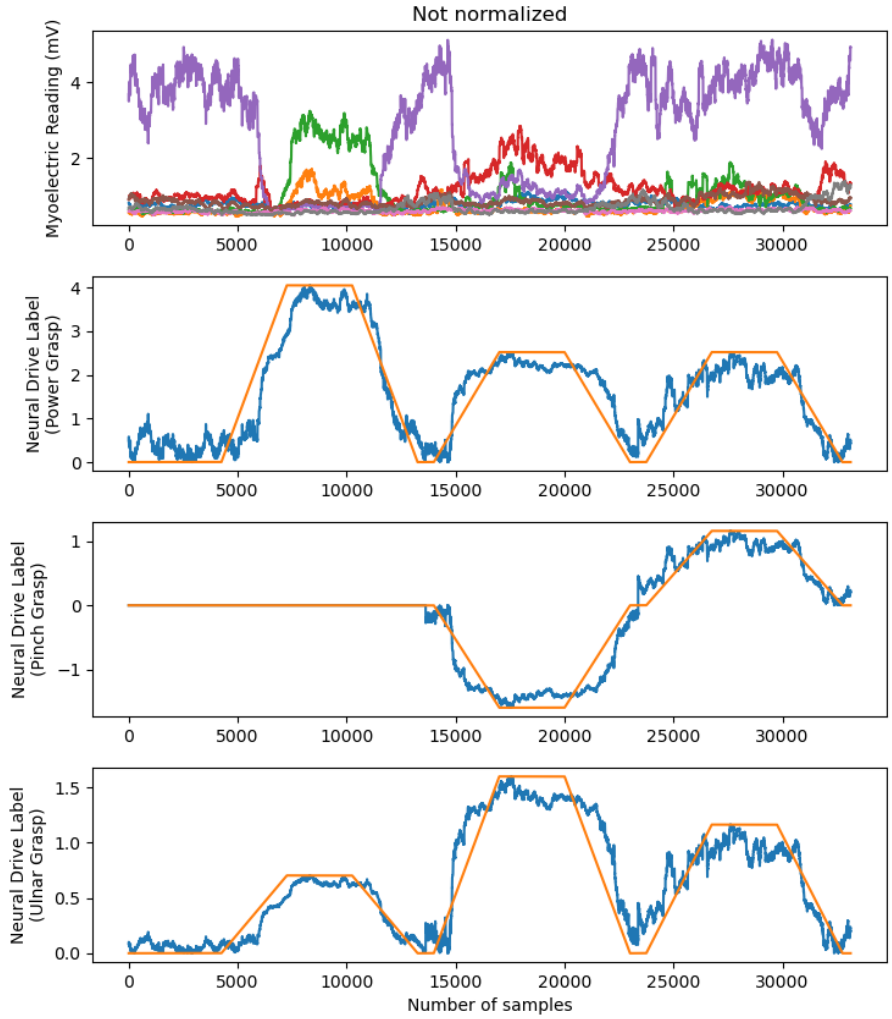


Figure 7: Myoelectric readings and grasps' labels from subject 1 (1st repetition)

### 3.3.2 Parameters selection and optimization

Several settings in the feature labeling script may be adjusted and tweaked to enhance the labeling process. Here's a more detailed explanation of these parameters and how they might be chosen and optimized [13]:

- **n_components**: This parameter determines the number of components (or features) to be extracted by NMF. It represents the desired dimensionality of the latent matrix. In this particular case, the sEMG generative model considers only 2 muscular synergies. Therefore, this parameter has to be set to take the same value in order to fix the synergistic matrix dimensionality.
- **solver:** The algorithm used to solve the NMF problem is determined by this parameter. It can be set to "cd" for coordinate descent or "mu" for multiplicative update. The solver selected is determined by the size of the dataset and the unique characteristics of the problem. Coordinate descent is faster for sparse data, but multiplicative updating is more efficient for dense data. Taking into account the time series nature of the ME signals read at a high frequency, a multiplicative approach seems to be a more suitable choice.
- **max_iter**: It sets the maximum number of iterations required for the NMF algorithm to converge. Increasing the number of iterations can enhance decomposition quality, but it also increases computing time. A reasonable amount of 1000 iterations were chosen that allowed the method to converge to a good answer without requiring excessive computation.
- **beta_loss**: The beta divergence utilized as the cost function in the NMF method is determined by this parameter. It can be adjusted to different choices like "kullback-leibler", "frobenius" or "itakura-saito". This parameter is dependent on the individual features of the data and the desired decomposition qualities, but since the "kullback-leibler" divergence is generally employed for non-negative data, it was the most logical option to settle on.
- **n_rep_nmf**: It defines the number of repetitions for the Non-Negative Matrix Factorization procedure. The NMF routine is used to smooth the data and improve the label quality. More steady and dependable results can be achieved by performing the NMF method several times. It is possible to experiment with different N_REP_NMF values to achieve the best balance of computing resources

and labeling accuracy. Low values were taken as a starting point, and they were progressively increased until a total of 10 repetitions, where the labeling quality was substantially improved.

- **seed**: It controls the script's random seed. Randomness is used in several processes, including initialization and random selection. Setting a specified integer value of the seed ensures that the labeling procedure is reproducible in order to iteratively tune other parameters. Different seed values can be tested to observe how they affect the final labeled data. In this particular study case, the seed value of 1 worked reasonably good for the labeling of the sEMG signals.

### 3.4 Deep Feedforward and Long Short-Term Memory Neural Networks

When it comes to feedforward DNNs, it is well known that they have great learning capabilities for complicated mappings between sEMG signals and related control inputs, making them a potential solution for improving HRI system performance and usability. Training a feedforward DNN entails optimizing the network's parameters for minimizing the difference between expected control instructions and ground truth values. The DNN eventually develops the ability to identify the complicated mappings between the input sEMG signals and the intended control outputs through recurrent optimization [5].

LSTM NNs, on the other hand, have grown in popularity because to their ability to handle sequential and temporal data. Unlike standard feedforward neural networks, they have a memory component that allows them to store information over long time periods. LSTM NNs are well-suited for modeling complicated temporal relationships in sequential data because of this memory mechanism and the capacity to selectively forget and update information. They may provide a helpful way for capturing the temporal dynamics and patterns inherent in muscle activation sequences in the context of sEMG-based HRI systems, allowing for more accurate and precise control of robotic devices [6].

In this chapter, it will be explored how feedforward DNNs and LSTM NNs may be used to decode sEMG signals and provide intuitive control of robotic equipment. How to construct and train these network architectures for sEMG signal processing will be explained; network structure layout, hyperparameter selection, and training techniques

will be covered. In addition, the assessment measures used to analyze the precision, reliability, and responsiveness of the models will be presented.

### 3.4.1 Deep Feedforward model implementation

The complete Python script of both Feedforward and LSTM NN algorithms implemented (**models** script) can be found in the corresponding appendices' subsection (see Section 8.2). Subsequently, the script is broken down step by step to explain how it works.

The **models** script defines first a feedforward neural network algorithm using the **PyTorch** library, an established deep learning tool for creating and training neural networks [18]. Before anything else, it imports the mentioned library and then, it constructs a class called **NN**, which derives from the **torch.nn.Module** class; it serves as the neural network model's template. The following methods are available in the **NN** class [18]:

- The **__init__** function is the class's constructor. It demands various arguments:
    - **input_dim:** It corresponds to the input dimension or the number of features in the dataset (the default value is set to 8, since the acquiring system has 8 built-in input channels).
    - **hidden_units:** It defines the number of neurons or units in the hidden layers (the default value is set to 32).
    - **out_dim:** It matches the number of output dimensions or classes to use (the default value is fixed to 3, since the NN is designed to learn the mappings for 3 different grasps).
    - **num_layers:** The total number of neural network layers, including input and output layers (the default value is set to 3).

  The **super().__init__()** line invokes the parent class's constructor (**torch.nn.Module**). The next lines assign the specified values for input dimensions, hidden units, output dimensions, and the number of layers to instance variables of the **NN** class. Afterwards, the **layers** variable is initialized as an empty list and them, layers are added based on the number of levels and dimensions initially defined [18]:

- The first layer is built, which is a fully connected layer that maps the input dimensions to the hidden units; this layer's activation function is **ReLU**.
- The same fully linked layer is added **num_layers - 2** times for the intermediate layers (except the first and last layers), and **ReLU** activation is done after each layer.
- The final layer is then generated, which represents the completely linked layer that maps the hidden units to the output dimensions; this layer's activation function is **Sigmoid**.

- The **forward** method implements the neural network's forward pass function. It starts with an input tensor **x** and successively sends it across each layer, applying the layer's transformation on the input. The ultimate output of the forward pass is the output of the final layer.

In a nutshell, this part of the **models** script constructs a feedforward neural network with adjustable input dimensions, hidden units, output dimensions, and layer count. It employs fully linked layers for the hidden layers with ReLU activation functions and a Sigmoid activation function for the output layer, while the forward method executes the network's forward pass.

### 3.4.2 Long Short-Term Memory model implementation

The **models** script defines a Long Short-Term Memory neural network using the **PyTorch** library imported [18]. First, it defines the **LSTM** class, which derives from the **torch.nn.Module** class. It serves as the LSTM network model's foundation. The following methods are available in the **LSTM** class [18]:

- The LSTM class's constructor is the **__init__** function. It demands various arguments:
    1. **input_dim:** It corresponds to the expected number of features in the dataset (the default value is set to 8, since the acquiring system has 8 built-in input channels).
    2. **hidden_units:** It defines the number of features contained in the hidden state (the default value is set to 32).

3. **out_dim:** It matches the number of output dimensions or classes to be returned (the default value is fixed to 3, since the NN is designed to learn the mappings for 3 different grasps).
4. **num_layers:** The total number of repeating layers in the neural network (the default value is set to 1).

The **super().__init__()** line invokes the parent class's constructor (**torch.nn.Module**). The next lines assign the specified values for input dimensions, hidden units and the number of layers to instance variables of the **LSTM** class [18].

Afterwards, the **lstm** variable is initialized as an instance of **torch.nn.LSTM**, which represents the network's LSTM layer. While created, the **input_dim**, **hidden_units** and **num_layers** parameters are passed, as well as **batch_first** is set to true, meaning that the batch dimension of the input tensor will be the first dimension [18].

Moreover, as an instance of **torch.nn.Linear**, the **linear** variable is initialized, denoting a completely linked layer. Because the output of the LSTM layer is transmitted through this linear layer, the **in_features** is set to **hidden_units**. Also, the **out_features** parameter has been set to **out_dim**, which represents the number of output classes [18].

- The **forward** method implements the neural network's forward pass computation. It takes a tensor **x** as input and performs the following:
    1. Based on the geometry of the input tensor, determine the batch size.
    2. Initialize the original hidden state h0 and cell state c0 to zeroes; the LSTM layer requires these states in order to maintain track of the temporal dependencies.
    3. The input tensor and starting states are passed to the LSTM layer (**self.lstm**). The result is a tuple that contains the output tensor as well as the final hidden and cell states.
    4. To retrieve the final output, extract the final hidden state **hn** from the tuple and feed it through the linear layer (**self.linear**).
    5. The output tensor is returned.

Overall, this part of the **models** script constructs an LSTM network with adjustable input dimensions, hidden units, output dimensions, and layer count. For feature

labeling in the myoelectric dataset, it employs an LSTM layer followed by a linear layer, while the forward method executes the network's forward pass.

### 3.4.3 Neural Networks' training

The complete Python script containing the code corresponding to the training of both Feedforward and LSTM NN algorithms (**train** script) can be found in the corresponding appendices' subsection (see Section 8.2). Subsequently, the script is broken down step by step to explain how it works.

The **train** script is developed so that a predefined NN can learn the mappings of the sEMG signals from different grasping actions, given a training dataset (this script is shared for both **Feedforward** and **LSTM** training implementation). The script first imports the necessary modules and classes for the model training and assessment; it imports modules such as **numpy** for numerical computations [11], **torch** for neural network training [18], **random** for random number generation [17], and **tqdm** for progress bars [19], as well as essential classes defined in the **models** and **dataset** scripts (the dataset script can be found at the section 8.1 from the appendices chapter, with an attached description of its purpose). Then we can find the following methods for performing the NN training:

- The **set_seeds** function is in charge of generating random seeds for various libraries in order to assure repeatability of findings. The following random seeds have been defined [11],[17],[18]:
    - **torch.backends.cudnn.deterministic = True:** This line assures that **CuDNN** (CUDA Deep Neural Network library) actions are deterministic; it is a GPU-accelerated deep neural network library used by **PyTorch**. By setting this value to **True**, it makes **cuDNN** operations predictable, which means that the same input will yield the same result every time the code is performed, which is critical for repeatability.
    - **torch.backends.cudnn.benchmark = False:** Disables the benchmark mode in **cuDNN**, which automatically identifies the optimum method for the present hardware configuration and input size. However, the method used may differ across runs, resulting in non-deterministic

behavior. Therefore, the benchmark mode is disabled, providing consistent behavior across runs.

- **torch.manual_seed(seed):** It sets the random seed for **PyTorch**'s CPU-based random number generator. By using the same seed, the sequence of random integers created by **PyTorch** will be consistent between code runs.
- **torch.cuda.manual_seed_all(seed):** It configures the random number generator used by **PyTorch** on the GPU (if one is available). By using the same seed, it assures that the random number generation on the GPU is likewise repeatable.
- **np.random.seed(seed):** It sets the random seed for **NumPy**'s random number generator. Because many operations in the code use **NumPy** functions, establishing the seed guarantees that these actions provide consistent results across runs.
- **random.seed(seed):** Sets the random seed for Python's built-in random number generator. It assures that any further operations in the code involving random integers that are not covered by the preceding seeds are repeatable.

Overall, the **set_seeds** function generates a deterministic environment for random number generation by establishing the seeds in this manner, guaranteeing that the results acquired during model training and assessment are consistent and reproducible.

- The **train_model** function is in charge of training the neural network model using the training data given. Four arguments are required by the function: the **data_loader** object that performs batch iterations on the training dataset, the **model** object, which corresponds to the neural network model that will be trained, the **loss_function** that is used to compute the training loss and the **optimizer**, that is in charge of adjusting the model's parameters depending on the obtained gradients.

  To begin, the function determines the total number of batches in the training data (**num_batches**). It also creates the variable **total_loss** to keep track of the entire loss during training. The **model.train()** call places the model in training mode, allowing features such as dropout and batch normalization if they are present [18].

Afterwards, the function runs a training loop across the data loader's batches of data. It iterates through the following stages for each batch [18]:

1. By running the input data (**x**) through the model, the model's output (**output**) is computed.
2. Using the supplied loss function, computes the loss between the model's output and the target labels (**y**).
3. The gradients of the model's parameters are reset (**optimizer.zero_grad()**).
4. The loss is backpropagated to compute the gradients of the model's parameters (**loss.backward()**).
5. The model's parameters are updated depending on the estimated gradients (**optimizer.step()**).
6. The loss value (**loss.item()**) is added to the **total_loss** variable.

The function determines then, the average training loss after finishing the training loop by dividing the **total_loss** by the total number of batches (**num_batches**), which is finally printed and returned.

In summary, the **train_model** function trains the neural network model using the training data supplied. It iterates over the data batches, computes the model's output, calculates the loss, backpropagates, and adjusts the model's parameters using the supplied optimizer. It adds the loss values together and delivers the average training loss.

Finally, the **if __name__ == "__main__"** block is the entry point of the script. It comprises the fundamental logic that is implemented when the script is run right away. The block begins by establishing a base configuration in the form of a dictionary: the batch size, learning rate, number of hidden units, random seed, number of training iterations, device used (CPU or GPU), downsample rate, dataset name, subject, and network type (either "lstm" or "nn") are all included. These settings can be changed for customizing the training process.

Then, the random seed from the basic configuration is sent to the **set_seeds** method. This assures that the script's random number generation is repeatable, as mentioned in the earlier description of the **set_seeds** function. Afterwards, the pickle file path for uploading the dataset is defined using the dataset name and subject from the basic configuration.

Once the previous steps are completed, both training and validation datasets can be generated using the **cross_validation_out** function included in the **DataPreProcessing** class in a “for loop”, which allows to create different divisions of the whole available data, providing every time one of the six different gestures’ repetitions as the validation dataset and the other five as the training set. The processed datasets are then returned and assigned to the variables **dataset_train** and **dataset_val** according to the selected network type, which are generated with the corresponding imported classes from the **dataset** generator script.

Moreover, the **DataLoader** class from **PyTorch** is used to develop data loaders. They perform batch iterations on the training and validation datasets, being the batch size determined by the initial setup. The **num_workers** option sets the number of subprocesses used for data loading, and **shuffle** controls whether or not the data is shuffled during training [18].

Subsequently, an NN (feedforward) or LSTM model is constructed based on the network type chosen in the basic configuration, as well as the number of hidden units and layers supplied. Then, **Torch.nn.MSELoss()** is used to generate the mean squared error (MSE) loss function, as well as the model's parameters, that are defined to be optimized using the Adam criterion, where the learning rate is set by the base configuration [18].

To finish the loop, depending on the number of training iterations (**base_config["iters"]**) and the size of the training dataset, the total number of epochs is determined. To keep the training and validation losses for each epoch, a dictionary **log_dict** is established.

Using the range function, the script begins a loop across the epochs. The **train_model** function is invoked for each epoch to train the model on the training data, passing the data loader, model, loss function, and optimizer, while the training loss is saved in **loss_train**. Similarly, the **val_model** function is invoked to assess the model against the validation data, and the validation loss is saved in **loss_val**; training and validation losses are stored in the **log_dict** dictionary for each epoch.

Last but not least, the script generates a dictionary state that includes both model's state and training log dictionaries as well as the basic initial configuration and, employs **torch.save** to store this state dictionary as a checkpoint file [18].

After executing the train script successfully, the chosen neural network can be considered to be trained, generating six different trained models where each one was trained using a different training dataset created with the **cross_validation_out** function.

The performance of this process is measured throughout the training loop, and it is addressed in the next section.

### 3.4.4 Performance evaluation

The script's training procedure focuses on developing a neural network model for sEMG-based myoelectric control of a robotic hand. To ensure the efficacy of the training, the performance of the training process must be evaluated.

The training process is evaluated by computing two essential metrics: the training loss and the validation loss. These metrics indicate how effectively the model learns during training and how well it generalizes to previously unknown data (note that this evaluation is done in order to have an instant insight of the model's performance so that it can be easily tuned after running the script, a further general analysis of the models' performance after the training process is done in the following results chapter).

The training loss, calculated within the **train_model** function, quantifies the difference between the model's anticipated outputs and the true labels for the training data. It is calculated using the Mean Squared Error (MSE) loss function in the script; it is calculated batch per batch, and the average loss across all batches is presented for the six different generated models after training them.

The validation loss, evaluated by the **val_model** function, measures how well the model performs on a different validation dataset that is not used for training. It provides an assessment of how effectively the model generalizes to previously unknown data. The MSE loss function, similar to the training loss, is used to compute the validation loss; it is also calculated batch by batch, with the average loss provided across all batches for the six different generated models after training them.

Furthermore, the training and validation losses for each epoch are saved in a log dictionary, which documents the training process's progress. This log keeps a thorough record of how the losses change over time, providing insights into the model's learning dynamics which will be used later to determine if the model actually learns the mappings improving the labeling predictions after more epochs are performed.

In a nutshell, the script's evaluation of training process performance is based on a study of training and validation loss. By monitoring these losses throughout the training loop, it is possible to make educated judgments about model selection, hyperparameter tuning, and the overall performance of the myoelectric control neural network for controlling a robotic hand using sEMG signals.

# 4. Results

In this chapter, the outcomes of the machine learning algorithms mentioned in the previous chapter's implementation are provided; there can be found the findings of the conducted experiments and analyses. The chapter is divided into two sections, given in the following order: the performance on sEMG signals of the developed machine learning models, and afterwards, the comparison of both machine learning techniques' performance.

The first part exposes the evaluation results of Feedforward and LSTM deep neural networks performance for the self-supervised regression of sEMG data, separately. On the other hand, the second part compares both machine learning methods' functioning in order to decide which would be a better approach for the given study case.

Overall, this chapter gives a thorough analysis of the findings from the tests carried out in this study. This work adds to the understanding of the effectiveness of machine learning methods for self-supervised regression of sEMG signals and its possible potential in robotic hand control.

## 4.1 Performance of ML models on sEMG signals

The whole Python script containing the code corresponding to the evaluation of the trained NNs' performance for predicting the neural drive features given a validation sEMG signals dataset (**predict** script) can be found in the corresponding appendices'

subsection (see Section 8.3). Subsequently, the graphical results generated by the script are displayed for their analysis.

First, the training and validation losses from the last trained Feedforward model evaluation are plotted, as well as the actual and predicted labels by the algorithm, generating the following graphs:

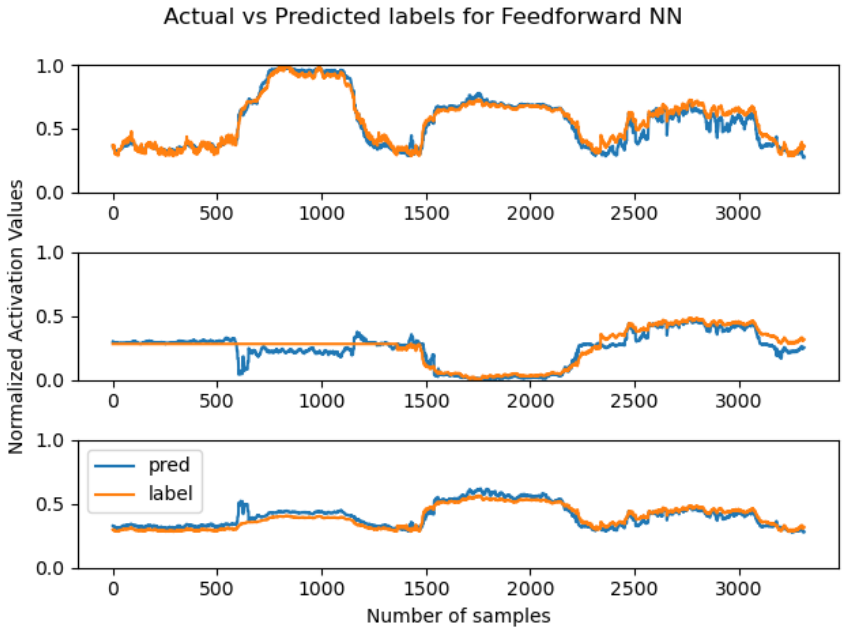


Figure 8: Actual vs Predicted neural drives for trained Feedforward DNN

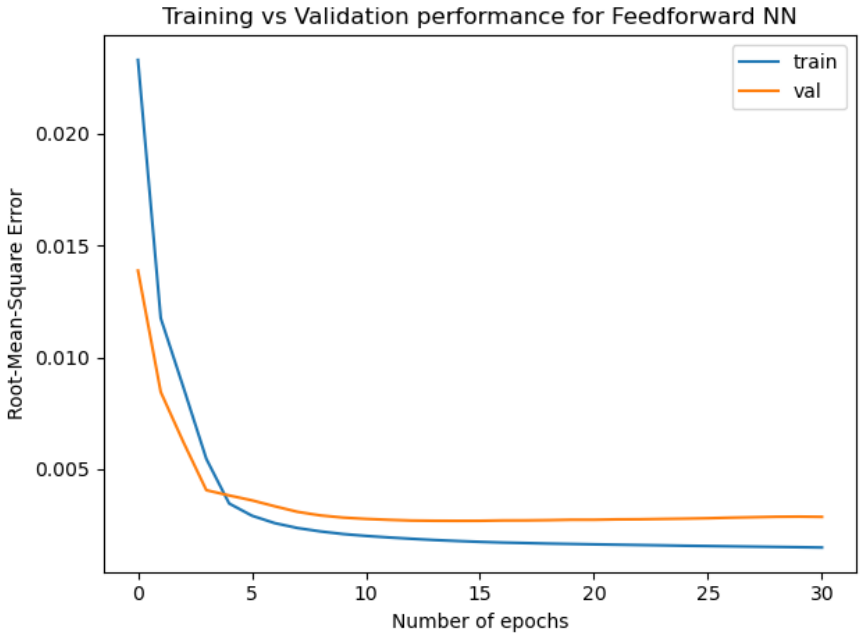


Figure 9: Training vs validation RMSE losses for Feedforward DNN

In the plot shown in figure 9, a decreasing tendency of the computed training and validation MSE losses can be seen, meaning that the algorithm actually works as it should, improving the feature extraction's precision between epochs of the method. The fact that the algorithm indeed does the job reasonably is also depicted in the plot of figure 8, where the generated neural drive follows the tendency of the label previously generated with the NMF method, that is considered the ground-truth for the ML application.

It is noticeable that the training loss starts at bigger values than the validation loss which actually makes sense, taking into account that, even though the validation dataset contains completely new data that is given to the algorithm for the first time, the NN has already been trained with "similar" data that can differ due to the nature of the signals, such as the physiological variability for the different sEMG recordings, making it easier for the ML method at the beginning, while having a similar precision outcome at the end.

Afterwards, the performance assessment of the Long Short-Term Memory algorithm is done in the same way as described for the Feedforward algorithm's analysis, leading to generate the following plots:

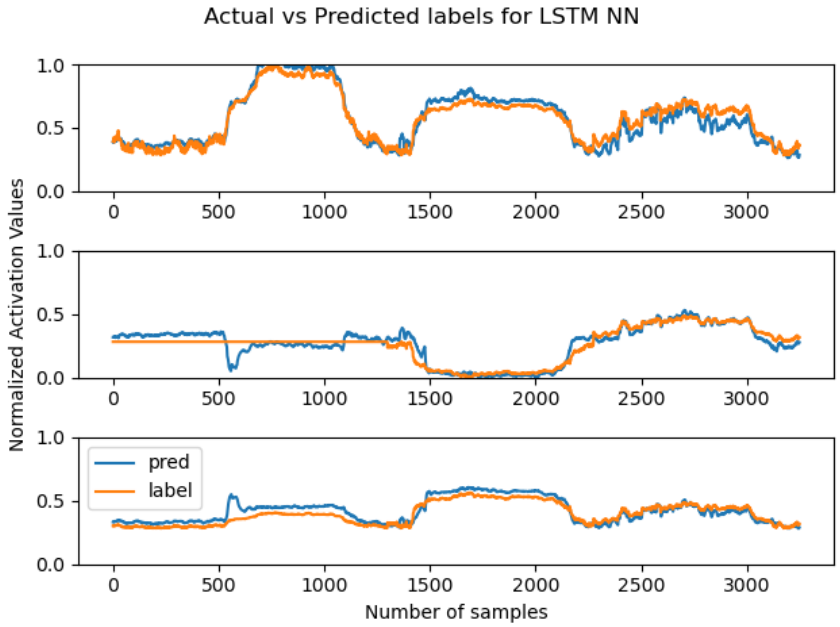


Figure 10: Actual vs Predicted neural drives for trained LSTM NN

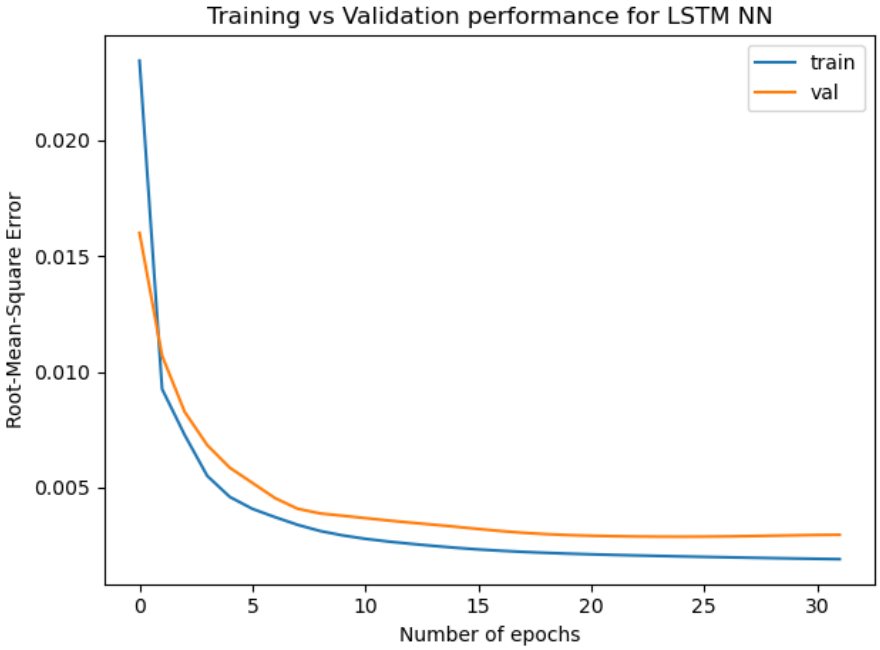


Figure 11: Training vs validation RMSE losses for LSTM NN

In the plot shown in figure 11, a decreasing tendency of the computed training and validation MSE losses is also seen as in the case of the DFNN, meaning that the algorithm

actually works as well, enhancing the feature extraction's accuracy between epochs of the method. The fact that the algorithm does a reasonable job is also demonstrated in figure 10, where the produced neural drive is comparable to the label previously created using the NMF approach, which serves as the verification fact for the ML application.

Similarly to the DFNN performance, the training loss begins at larger values than the validation loss, which has again a logical reasoning considering that, while the validation dataset includes completely new data that is provided to the algorithm for the first time, the NN has already been trained with "comparable" data that can diverge because of the characteristics of the signals, such as physiological variability for the different subjects' sEMG recordings, making it more simple for the ML method to learn at the starting point, while achieving a similar level of accuracy in the final stages.

Overall, both ML algorithms have been able to compute a suitable mapping for decoding the neural drives given a set of sEMG signals for a posterior control of a robotic hand. In the next section, both performances will be compared in order to make a founded decision about which method would be more fitting in order to tackle this particular study case.

### 4.2 Comparison of the machine learning algorithms

The **predict** script performs a validation test on both Feedforward and LSTM algorithms, where the Root-Mean-Square Error is computed for each different trained models and the average value of the six results is then represented as a bar plot for each algorithm, as well as a confidence interval that is displayed in black for both cases, allowing a graphical comparison of both machine learning algorithms' performance by means of the validation loss parameter. Subsequently, the graphical results generated by the script are displayed for their analysis.

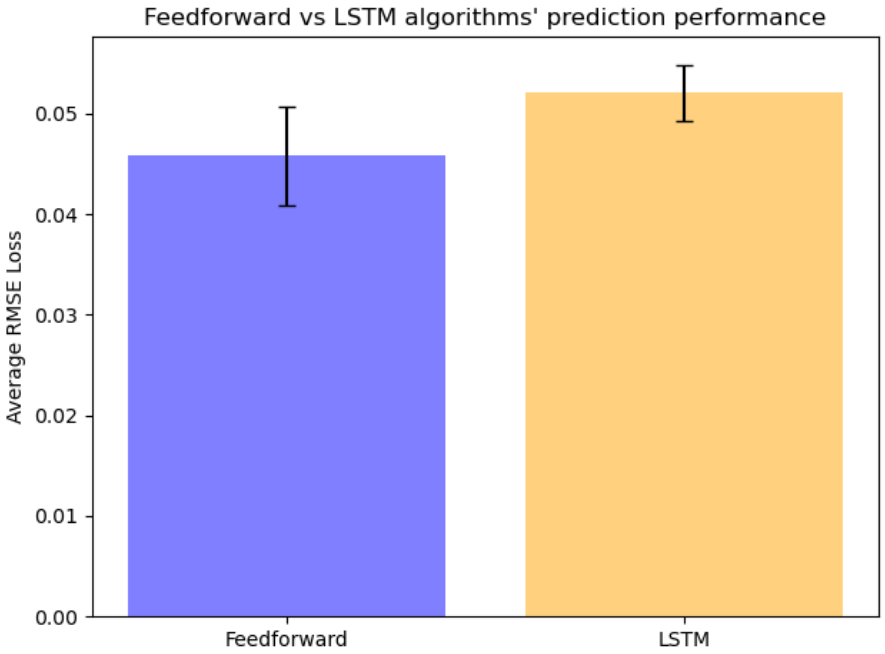


Figure 12: Feedforward vs LSTM NNs' average RMSE Loss

As it can be seen in figure 12, both algorithms performed pretty similarly, even though they have such different architectures. On the one hand, the typical Deep Feedforward Neural Network was able to predict the neural drives from the sEMG signals with a precision higher than 95% with respect to the actual one, which is indeed reflected in the blue bar plot where the average RMSE loss is lower than 0.05 (5% error). On the other hand, the proposed alternative of the Long Short-Term Memory Neural Network actually scored a slightly higher average RMSE loss, of value barely higher than 0.05 (5% error), meaning that the algorithm almost reached a 95% precision in predicting the neural drives from the sEMG signals that would be then translated into control commands for a robotic hand.

To sum up, it was shown that given the hyperparameters' selection and the number of epochs performed during the training sessions among other factors, both neural architectures were able to learn the mappings for predicting the desired neural drives from sEMG readings with a remarkable accuracy, giving a special mention to the Feedforward NN for achieving a hardly better performance doing this job. In the following section, the

results that were just exposed will be discussed for providing an answer to this study case, which will lead then to concluding the work done.

# 5. Discussion

The present chapter of the thesis, which refers to the discussion, acts as a complete examination of the research project's important results. It strives to critically examine the experimental data and offer a detailed interpretation of their relevance, while taking into account the study's limitations.

The relevance of the findings acquired from the various machine learning algorithms examined in the previous chapter will be investigated. The causes behind the observed performance variations across the algorithms will be studied, while recommending areas for development.

Furthermore, this chapter recognizes the study's shortcomings and proposes possible topics for further research in the subject. Overall, this chapter presents a critical assessment on the research effort and its consequences, with the goal of adding to the knowledge in the field of sEMG-based robotics.

## 5.1 Implications of the findings

When it comes to the findings previously exposed about the training of both Feedforward and LSTM Neural Networks, an exponential decay of the training loss can be seen (as shown in figures 9 and 11), meaning that epoch after epoch that was performed throughout the architecture of the algorithms, a substantial improvement was achieved when it comes to learning the mappings between the provided sEMG signals and the desired neural drives' predictions.

On the one hand, the Feedforward NN architecture was comprised of the following elements (notice that the following parameters' definition is flexible depending

on the algorithm's application and the selection criteria should be focused on the optimization of the output's precision):

- An input layer with a dimension of 8, meaning that it contains 8 different nodes where each one represents one input channel corresponding to the 8 pairs of electrodes that were used for the data acquisition.
- A hidden layer containing a total of 32 units, meaning that the intermediate step for learning the mappings of the signals contains 32 neurons that are used to compute the weights associated with the relationship between the input and output values.
- An output layer with a dimension of 3, where each node represents a neural drive corresponding to the 3 different grasp motions that are considered in the study to be then performed by a robotic hand.

By defining this structure in particular, the algorithm was able to be trained progressively in such a way that when a different/new set of data was provided (that is the validation dataset generated during the training process), it was able to predict also with an improving accuracy over the iterative process, meaning a decrease of the computed loss between the output and the provided labels for each grasp, the wanted neural drives (as shown in figure 9).

On the other hand, the Long Short-Term Memory NN architecture was comprised of the following elements (again, notice that the following parameters' definition is flexible depending on the algorithm's application and the selection criteria should be focused on the optimization of the output's precision):

- An input layer with a dimension of 8, meaning that there are 8 different features or input values in the NN where each one represents one input channel corresponding to the 8 pairs of electrodes that were used for the data acquisition.
- An LSTM layer containing a total of 32 units, meaning that the intermediate step for learning the mappings of the signals contains 32 memory cells that are used to analyze the sequential input data and record the data's long-term relationships for computing the final output values of the algorithm.
- An output layer with a dimension of 3, where the hidden states returned by the LSTM layer are received to generate 3 different final outputs that represent the

neural drives corresponding to the 3 different grasp motions that are considered in the study to be then performed by a robotic hand.

By establishing this structure specifically, the algorithm managed to be trained gradually in such a manner that when a different/new set of data was given (that is, the validation dataset generated during the training process), it also became able to predict with an increasing precision over the course of the iterations, indicating a decrease in the estimated loss between the output and the provided labels for each grasp, the desired neural drives (as shown in figure 11).

Overall, both algorithms showed a reasonable capacity to learn the mappings between the acquired sEMG signals and the neural drives corresponding to the performed grasp activities, making them possible solutions to tackle the problem of sEMG signals regression for the control of a robotic hand.

Finally, a graphical comparison of both architectures' performance was generated in order to discriminate between them to decide which would be a better fit for achieving the goal of a robotic hand control with the best possible accuracy.

According to the bar plot showed in figure 12, the prediction's precision of the neural drives executed by both algorithms is tightly similar, with a slight advantage of the Feedforward NN that scored a hardly lower Root-Mean-Square Error loss in average between the six different trained models for each case.

Apparently, the LSTM NN architecture would be less appropriate for this study case, but given the fact that several factors regarding the particular structure selection may have a decisive impact on the results, which did not differ that much from the ones obtained by the Feedforward NN, there is no clear and absolute proof that the latter is indeed the best option. Nevertheless, strictly analyzing the obtained results, it could be said that the Feedforward algorithm may be the best approach given the limitations of the study, with room for improvement in both cases that will be discussed throughout the following section.

## 5.2 Future research directions

According to the performed work along with its limitations, an extended refinement of several aspects regarding the architectures' structure, algorithms implementation and parameters' selection could be carried out in succeeding research efforts about the sEMG-based control of robotic hands using the proposed approach. Some directions to be taken into account could be the following:

- The general architecture of both algorithms could be improved by increasing the number of hidden layers, as well as the number of neurons/units with which these are constituted, for creating deeper networks capable of learning more complex mappings for the neural drives' regression. Notice that this approach could escalate the computational cost for training the algorithm, to an extent in which it could be unviable to generate a response feasibly.
- The implementation of an inner cross-validation loop would be the next step when it comes to selecting the algorithms' parameters, since it would allow the optimization of the values that take the variables defining the structure of the selected methods.
- Another sensitive factor to take into account would be the sequence length with which the time series dataset is broken down into batches to be fed to the LSTM algorithm. A fine tuning of this parameter could lead to an enhancement of the neural network's outcome quality.
- Moreover, regarding the LSTM algorithm implemented in this study case, a basic version of the architecture was taken into account but, according to [9], several variations have been developed which have shown an enhanced performance in a variety of applications. This path could also be explored for trying to implement a version that better fits the nature of sEMG signals.

Overall, several possible directions have been stated for improving the performance of both algorithms and carrying out a more precise analysis and educated judgement of which approach could be more suitable for this particular study case.

# 6. Conclusion

This final chapter summarizes the discussion of the introduction's questions and objectives of the study, being the main goal of the section to expose if these aims were met. It also intends to show how the research findings contribute to the area of sEMG-based robotics, as well as how the insights acquired from this study may be used to actual real-world applications.

## 6.1 Summary of the key findings

The research efforts accomplished in this paper focus on the report of the performance achieved by the proposed Machine Learning algorithms, both a Feedforward Neural Network and a Long Short-Term Memory Neural Network, in the job of carrying out a regression technique on a pre-acquired set of surface electromyographic signals for the further estimation of the neural drives corresponding to different hand-grasping movements for the myoelectric control of a robotic hand.

The obtained sEMG dataset, containing the readings of the electrodes' input channels, was divided in both training and a validation sets by means of an outer cross-validation loop. Several models based on the intended ML algorithms were trained with the corresponding datasets, which showed a promising ability learning the mappings between the sEMG signals and the expected neural drives' labels that would be then used for predicting neural drives given a new validation dataset.

Once the models were trained, the performance of both architectures was then measured by means of the average between the Root-Mean-Square Error losses of the different models' outputs when predicting the neural drives given the mentioned sEMG validation dataset. A high and consistent accuracy was shown by both ML algorithms, where the Feedforward NN barely outperformed the LSTM NN for the particular architecture design and parameters' selection done in this study case.

To conclude, the experimental evaluation of the algorithms' performance was utterly useful to shed light on the profitability and convenience that can be gained from

the usage of the proposed architectures for the task of regression regarding sEMG signals for developing a natural and user-friendly Human-Robot Interface capable of controlling a robotic hand's multi-grasp gestures.

### 6.2 Practical applications and contributions to the field

The development of this thesis is intended to have a positive impact on the topic of sEMG-based robotic hand control, expanding the current knowledge available for the community and promoting the persistence in developing new interfaces that achieve greater results.

Regarding the functional applications that could benefit from the stated implementation and analysis of the chosen Machine Learning architectures, several fields of domain can be taken into account: prosthetic devices and assistive technologies, where enhanced control algorithms can increase the efficiency and naturalness of prosthetic hands, allowing for a wider range of grip patterns with more precision, making patients' everyday tasks easier and thereby enhancing their quality of life; industrial production systems in which the profitability, precision, and quality of automated operations in industrial settings that demand dexterous manipulation may be improved while decreasing recurring human labor, resulting in cost savings and overall production enhancement; finally, teleoperation in hazardous environments, in which the algorithms might enable and improve an accurate remote control of robotic hands for functioning in potentially dangerous environments for people, such as radioactive locations, disaster response scenarios, underwater operations, or space exploration.

Weighing up the facts stated throughout the preceding chapters, some contributions that could be extracted from this thesis work may be the following: the design of more accurate and solid feature extraction algorithms for sEMG-based control interfaces, an understanding of the appropriateness and efficacy of the chosen algorithms in multi-grasp robotic hand control, and the statement that applying deep neural networks to sEMG-based control can result in more smooth and sensitive interactions, improving the overall user experience and accessibility of the systems.

## 7. References


[1] LI, Kexiang, et al. A review of the key technologies for sEMG-based human-robot interaction systems. *Biomedical Signal Processing and Control*, 2020, vol. 62, p. 102074.

[2] MEATTINI, Roberto, et al. An sEMG-based human–robot interface for robotic hands using machine learning and synergies. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 2018, vol. 8, no 7, p. 1149-1158.

[3] BERG, Julia; LU, Shuang. Review of interfaces for industrial human-robot interaction. *Current Robotics Reports*, 2020, vol. 1, p. 27-34.

[4] MEATTINI, Roberto, et al. Design and evaluation of a factorization-based grasp myoelectric control founded on synergies. En *2019 12th International Workshop on Robot Motion and Control (RoMoCo)*. IEEE, 2019. p. 252-257.

[5] KANG, Tae Gyoon, et al. NMF-based target source separation using deep neural network. *IEEE Signal Processing Letters*, 2014, vol. 22, no 2, p. 229-233.

[6] HUA, Yuxiu, et al. Deep learning with long short-term memory for time series prediction. *IEEE Communications Magazine*, 2019, vol. 57, no 6, p. 114-119.

[7] SEUNG, D.; LEE, L. Algorithms for non-negative matrix factorization. *Advances in neural information processing systems*, 2001, vol. 13, p. 556-562.

[8] NIELSEN, Michael A. *Neural networks and deep learning*. San Francisco, CA, USA: Determination press, 2015.

[9] CHRISTOPHER, Olah. Understanding LSTM networks [online]. *Understanding LSTM Networks - Colah's Blog.* (August 27, 2015). Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[10] MEATTINI, Roberto, et al. Self-supervised regression of sEMG signals combining non-negative matrix factorization with deep neural networks for robot hand multi-grasp control. *IEEE Robotics and Automation Letters*, 2023.

[11] NumPy Reference [online]. numpy.org, December 18, 2022. Available: https://numpy.org/doc/stable/reference/index.html#reference

[12] Matplotlib API Reference [online]. matplotlib.org, 2023. Available: https://matplotlib.org/stable/api/index.html

[13] Scikit-learn API Reference [online]. scikit-learn.org, 2023. Available: https://scikit-learn.org/stable/modules/classes.html

[14] pickle - Python object serialization [online]. docs.python.org, 2023. Available: https://docs.python.org/3/library/pickle.html

[15] os - Miscellaneous operating system interfaces [online]. docs.python.org, 2023. Available: https://docs.python.org/3/library/os.html

[16] Input and output (scipy.io) [online]. docs.scipy.org, 2023. Available: https://docs.scipy.org/doc/scipy/reference/io.html

[17] random – Generate pseudo-random numbers [online]. docs.python.org, 2023. Available: https://docs.python.org/3/library/random.html

[18] PyTorch Documentation [online]. pytorch.org, 2023. Available: https://pytorch.org/docs/stable/index.html

[19] tqdm documentation [online]. tqdm.github.io, 2022. Available: https://tqdm.github.io

# 8. Appendices

## 8.1 Code implementation and description of dataset building and preprocessing

**Dataset Python script** (Dataset generation and prior preprocessing)

```
import numpy as np
import torch, pickle, copy
import matplotlib.pyplot as plt


class DataPreProcessing(object):
    def __init__(self, pickle_path):
        self.data_dict = self.load(pickle_path)

    def get_data(self):
        data = np.hstack([self.data_dict[k]["Xn"] for k in self.data_dict.keys()])
```

```
        label = np.hstack([self.data_dict[k]["Tn"] for k in self.data_dict.keys()])
        return data, label

    def cross_validation_out(self):
        keys = list(self.data_dict.keys())
        for k in keys:
            train_data, val_data, train_label, val_label = [], [], [], []
            for z in keys:
                if z != k:
                    train_data.append(self.data_dict[z]["Xn"])
                    train_label.append(self.data_dict[z]["Tn"])
                else:
                    val_data = self.data_dict[z]["Xn"]
                    val_label = self.data_dict[z]["Tn"]

            yield copy.deepcopy(train_data), copy.deepcopy(val_data),
copy.deepcopy(train_label), copy.deepcopy(val_label)

    def load(self, path):
        with open(path, "rb") as handle:
            b = pickle.load(handle)
            return b



class EMGData(torch.utils.data.Dataset):
    def __init__(self, data, label, downsample=True, rate=4):
        self.data = data
        self.label = label

        if downsample:
            self.data, self.label = self.downsample(self.data, self.label, rate)

        self.data = copy.deepcopy(self.data.T)
        self.label = copy.deepcopy(self.label.T)

    def downsample(self, data, label, rate):
```

```
        data = data[:, ::rate]
        label = label[:, ::rate]
        return data, label

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        datum = torch.from_numpy(self.data[idx].T).float().unsqueeze(0)
        label = torch.from_numpy(self.label[idx].T).float().unsqueeze(0)
        return datum, label


class EMGDataSeq(torch.utils.data.Dataset):
    def __init__(self, data, label, downsample=True, rate=10, sequence_length=64):
        self.data = data
        self.label = label
        self.sequence_length = sequence_length

        if downsample:
            self.data, self.label = self.downsample(self.data, self.label, rate)

        self.data_ready = []
        self.label_ready = []
        for i in range(self.data.shape[2]):
            i_end = i + self.sequence_length
            if i_end < self.data.shape[2]:
                x = self.data[:, i:i_end]
                y = self.label[:, i_end]
                self.data_ready.append(x)
                self.label_ready.append(y)

        self.data_ready = np.array(self.data_ready).T
        self.label_ready = np.array(self.label_ready).T

    def downsample(self, data, label, rate):
```

```
        data = data[:, ::rate]
        label = label[:, ::rate]
        return data, label

    def load(self, path):
        with open(path, "rb") as handle:
            b = pickle.load(handle)
            return b

    def __len__(self):
        return self.data_ready.shape[-1]

    def __getitem__(self, idx):
        x = self.data_ready[:, :, idx]
        y = self.label_ready[:, idx]

        x = torch.from_numpy(x).float()
        y = torch.from_numpy(y).float()
        return x, y


if __name__ == "__main__":
    # DATA
    with open("dataset_ar10/subj1.pickle", "rb") as f:
        data_subj = pickle.load(f)
    data = data_subj[0]["Xn"]
    label = data_subj[0]["Tn"]
    print(f"data shape: {data.shape}, label shape: {label.shape}")

    # DATASET LSTM
    dataset = EMGDataSeq(data, label, downsample=True, rate=10, sequence_length=64)
    print("dataset length: ", len(dataset))

    # DATASET NN
    dataset = EMGData(data, label, downsample=True, rate=10)
    print("dataset length: ", len(dataset))
```

The supplied Python script addresses data preparation and dataset generation for training the ML algorithms used for the sEMG-based myoelectric control of robotic hands. Following that, the various elements of the script are discussed in order to comprehend how they work.

The script begins by loading the required libraries, which include **numpy** for numerical calculations [11], **torch** for deep learning [18], **pickle** for serialization [14], and **matplotlib** for charting [12]. Data loading and preparation are handled by the **DataPreProcessing** class. During the initial setup, it accepts a path to a pickle file as input. The pickle file contains a data dictionary as well as labels for various samples. The following methods are available in the class:

- **get_data():** Concatenates the data and labels from all the pickle file samples and returns them as independent arrays.
- **cross_validation_out():** A cross-validation outer loop is implemented. It generates six different training and validation data packages, as well as labels, where out of the six available recorded grasp repetitions, one is taken as the validation recording and the other five as the training recording, varying within the loop which one is the validation one.
- **load():** Loads the pickle file from the specified directory and returns the data as a dictionary.

The dataset generation for training a neural network is represented by the **EMGData** class. During startup, it accepts preprocessed data and labels as the input. It employs the following techniques:

- **downsample():** By a given factor, reduces the sampling rate of the data and labels.
- **__len__():** This function returns the number of samples in the dataset.
- **__getitem__():** It retrieves a single sample from the dataset.

The **EMGDataSeq** class provides a sequential dataset used to train an LSTM recurrent neural network. During startup, it accepts preprocessed data and labels as input, along with additional settings such as downsampling rate and sequence length. It employs the following methods:

- **downsample():** By a given factor, reduces the sampling rate of the data and labels.

- **load():** Loads the pickle file from the specified directory and returns the data as a dictionary.
- **__len__():** This function returns the number of sequences in the dataset.
- **__getitem__():** Retrieves a particular sequence from the dataset.

Finally, the main script gets started by reading data and labels from a pickle file and then, they are extracted and printed. Afterwards, the data and labels are sent to an instance of the **EMGDataSeq** class, along with additional parameters such as downsampling rate and sequence length. Then, using the same data and labels but without the sequence information, an instance of the **EMGData** class is constructed, executing the building of a dataset for both LSTM and Feedforward NNs.

### 8.2 Code implementation of factorization and deep learning algorithms

**Mapping Python script** ( Precalculated synergy matrix and activation vectors )

```
import numpy as np

class AlphaMatrix():
    ar10 = np.array([[4.0504, 2.5198, 2.5198],
                     [-0.0000, -1.5981, 1.1627],
                     [0.7030, 1.5981, 1.1627]])

class SynMatrix():
    ar10 = np.array([[0.3386, 0.3536, -0.1017],
                     [0.3386, 0.3536, -0.1017],
                     [0.3386, 0.3536, -0.1017],
                     [0.3386, 0.3536, -0.1017],
                     [0.3386, -0.3536, -0.1017],
                     [0.3386, -0.3536, -0.1017],
                     [0.3386, -0.3536, -0.1017],
                     [0.3386, -0.3536, -0.1017],
                     [0.3386, -0.3536, -0.1017],
                     [0.2649, -0.0000, 0.8819],
                     [0.1122, -0.0000, 0.3735]])
```

These arrays correspond to a previous necessary analysis of the robotic hand that is intended to be used after developing the sEMG-based HRI, which in this case is the AR10 Hand as mentioned in the experimental setup section (see section 3.1).

According to [10], the grasp synergy matrix (stored in the **ar10** variable inside the **SynMatrix()** class) was computed in line with the notion of postural synergies in order to allow management of the closure level of power, tripodal and ulnar grasps. This was done by applying a method called Principal Component Analysis on a matrix containing the vectors corresponding to the different joint angles configurations of each maximum closure level regarding each grasp type.

Since, the synergy matrix corresponds to an orthonormal basis of the robot hand configuration space, the vectors of synergy activations corresponding to the maximum closure level of power, tripodal and ulnar grasps (stored in the **ar10** variable inside the **AlphaMatrix()** class) can be then computed as the product between the pseudo-inverse of the synergy matrix and the joint angles configuration matrix [10].

This script is intended to contain the synergy and synergy activation matrixes for different robotic hands, so that a customized study can be performed as it is reflected in the subsequent scripts, where different **alpha_types** can be used according to the desired robotic hand analysis implementation.

**Labeling Python script** (NMF algorithm implementation)

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import NMF
import pickle, os, scipy.io, random
import copy

from mapping import AlphaMatrix


class NMF_Routine():
```

```
    @staticmethod
    def _nmf(X):
        model = NMF(n_components=2, init='random', random_state=None, solver="mu",
max_iter=1000, beta_loss="kullback-leibler")
        W = model.fit_transform(X)
        return W, model.components_  # latent

    @staticmethod
    def compute(X, n_rep=10, diff_signal=True):

        _, H = NMF_Routine._nmf(X)

        # repeat NMF routine n_rep times for better smoothing
        for _ in range(n_rep):
            _, H = NMF_Routine._nmf(H)

        # check shape H high/low -> signal should be high -> low -> high
        if H[0, 0] - H[1, 0] < 0:
            H = H[[1, 0], :]  # swap

        # normalization of each signal
        Hn = (H.T / np.max(H, axis=1)).T  # 2 x timestep

        if diff_signal:
            S = Hn[1, :] - Hn[0, :]
        else:
            S = Hn

        # normalization 0-1
        Sn = (S - np.min(S)) / np.ptp(S)
        return Sn

class Labeler():

    def __init__(self, data_path, alpha_type, diff_signal, n_rep_nmf):
```

```
        if "ar10" in alpha_type:
            self.alpha = AlphaMatrix.ar10
        elif "ub" in alpha_type:
            self.alpha = AlphaMatrix.ub
        elif "berrett" in alpha_type:
            self.alpha = AlphaMatrix.berrett
        else:
            NotImplementedError("AlphaMatrix type not available!")

        self.diff_signal = diff_signal
        self.n_rep_nmf = n_rep_nmf
        self.data_path = data_path

    def merge_1_signal(self, H_pw, H_pn, H_ul):
        # column 1 of T
        alpha1 = np.repeat(self.alpha[:, 0].reshape(-1, 1), repeats=H_pw.shape[-1], axis=1)  #
repeat signal k times along columns to match h1 dims
        T1 = np.multiply(alpha1, np.tile(H_pw, (3, 1)))

        # column 2 of T
        alpha2_rep = np.repeat(self.alpha[:, 1].reshape(-1, 1), repeats=H_pn.shape[-1], axis=1)
        T2 = np.multiply(alpha2_rep, np.tile(H_pn, (3, 1)))

        # column 3 of T
        alpha3_rep = np.repeat(self.alpha[:, 2].reshape(-1, 1), repeats=H_ul.shape[-1], axis=1)
        T3 = np.multiply(alpha3_rep, np.tile(H_ul, (3, 1)))

        return np.hstack([T1, T2, T3])

    def merge_2_signals(self, H_pw, H_pn, H_ul):

        alpha_bar = np.zeros((self.alpha.shape[0] * 2, self.alpha.shape[2]))
        alpha_bar[0:2, :] = np.repeat(self.alpha[0, :].reshape(1, -1), repeats=2, axis=0)
        alpha_bar[2:4, :] = np.repeat(self.alpha[1, :].reshape(1, -1), repeats=2, axis=0)
        alpha_bar[-2:, :] = np.repeat(self.alpha[2, :].reshape(1, -1), repeats=2, axis=0)
```

```
        # column 1 of T
        alpha1 = np.repeat(alpha_bar[:, 0].reshape(-1, 1), repeats=H_pw.shape[-1], axis=1)  #
repeat signal k times along columns to match h1 dims
        T1 = np.multiply(alpha1, np.tile(H_pw, (3, 1)))

        # column 2 of T
        alpha2_rep = np.repeat(alpha_bar[:, 1].reshape(-1, 1), repeats=H_pn.shape[-1], axis=1)
        T2 = np.multiply(alpha2_rep, np.tile(H_pn, (3, 1)))

        # column 3 of T
        alpha3_rep = np.repeat(alpha_bar[:, 2].reshape(-1, 1), repeats=H_ul.shape[-1], axis=1)
        T3 = np.multiply(alpha3_rep, np.tile(H_ul, (3, 1)))

        # T
        T = np.hstack([T1, T2, T3])
        print("T matrix: ", T.shape)
        return T

    def load_data(self, subj_id):
        # load mat files
        F_pw =
np.array(scipy.io.loadmat(f"{self.data_path}/subj{subj_id}_pwr.mat")[f"subj{subj_id}_pwr"
]).squeeze()  # 6 x 1 -> 6 repetitions
        F_pn =
np.array(scipy.io.loadmat(f"{self.data_path}/subj{subj_id}_pin.mat")[f"subj{subj_id}_pin"])
.squeeze()
        F_ul =
np.array(scipy.io.loadmat(f"{self.data_path}/subj{subj_id}_uln.mat")[f"subj{subj_id}_uln"])
.squeeze()

        F_pw_ref =
np.array(scipy.io.loadmat(f"{self.data_path}/subj{subj_id}_pwr_ref.mat")[f"subj{subj_id}_p
wr_ref"]).squeeze()  # 6 x 1 -> 6 ripetizioni
        F_pn_ref =
np.array(scipy.io.loadmat(f"{self.data_path}/subj{subj_id}_pin_ref.mat")[f"subj{subj_id}_pi
```

```
n_ref"]).squeeze()
        F_ul_ref =
np.array(scipy.io.loadmat(f"{self.data_path}/subj{subj_id}_uln_ref.mat")[f"subj{subj_id}_ul
n_ref"]).squeeze()
        return F_pw, F_pn, F_ul, F_pw_ref, F_pn_ref, F_ul_ref

    def run_subj(self, subj_id, plot=True):
        F_pw, F_pn, F_ul, F_pw_ref, F_pn_ref, F_ul_ref = self.load_data(subj_id)
        data_dict = {}
        for rep in range(len(F_pw)):
            print("-> rep =", rep)

            print("X | power: {}, pinch: {}, ulnar: {}".format(F_pw[rep].shape, F_pn[rep].shape,
F_ul[rep].shape))
            print("R | power: {}, pinch: {}, ulnar: {}".format(F_pw_ref[rep].shape,
F_pn_ref[rep].shape, F_ul_ref[rep].shape))
            assert F_pw[rep].shape[2] == F_pw_ref[rep].shape[2] and F_pn[rep].shape[2] ==
F_pn_ref[rep].shape[2] and F_ul[rep].shape[2] == F_ul_ref[rep].shape[2]

            # NMF LABELING
            H_pw = NMF_Routine.compute(F_pw[rep], n_rep=self.n_rep_nmf,
diff_signal=self.diff_signal)  # power
            H_pn = NMF_Routine.compute(F_pn[rep], n_rep=self.n_rep_nmf,
diff_signal=self.diff_signal)  # pinch
            H_ul = NMF_Routine.compute(F_ul[rep], n_rep=self.n_rep_nmf,
diff_signal=self.diff_signal)  # ulnar

            # merge label signals
            if self.diff_signal:
                T = labeler.merge_1_signal(H_pw, H_pn, H_ul)
            else:
                T = labeler.merge_2_signals(H_pw, H_pn, H_ul)

            # emg input signals
            X = np.hstack([F_pw[rep], F_pn[rep], F_ul[rep]])
```

```
            # REFERENCE SIGNAL
            # normalize REF to 0-1 and flip
            R_pw = 1 - (F_pw_ref[rep] - np.min(F_pw_ref[rep])) / np.ptp(F_pw_ref[rep])
            R_pn = 1 - (F_pn_ref[rep] - np.min(F_pn_ref[rep])) / np.ptp(F_pn_ref[rep])
            R_ul = 1 - (F_ul_ref[rep] - np.min(F_ul_ref[rep])) / np.ptp(F_ul_ref[rep])
            R = labeler.merge_1_signal(R_pw, R_pn, R_ul)

            # NORMALIZE X, T and R
            Xn = (X - np.min(X)) / np.ptp(X)
            Tn = (T - np.min(T)) / np.ptp(T)
            Rn = (R - np.min(R)) / np.ptp(R)

            data_dict[rep] = {"X": copy.deepcopy(X), "T": copy.deepcopy(T), "R":
copy.deepcopy(R), "Xn": copy.deepcopy(Xn), "Tn": copy.deepcopy(Tn), "Rn":
copy.deepcopy(Rn)}

            if plot: self.plot(X, T, R, Xn, Tn, Rn)
        return data_dict

    def plot(self, X, T, R, Xn, Tn, Rn):
        fig, axs = plt.subplots(4, 2, figsize=(15, 8))

        axs[0, 0].set_title("Not normalized")
        axs[0, 1].set_title("Normalized")

        axs[0, 0].plot(X.T)

        axs[1, 0].plot(T[0, :].T)
        axs[1, 0].plot(R[0, :].T)

        axs[2, 0].plot(T[1, :].T)
        axs[2, 0].plot(R[1, :].T)

        axs[3, 0].plot(T[2, :].T)
        axs[3, 0].plot(R[2, :].T)
```

```
        axs[0, 1].plot(Xn.T)

        axs[1, 1].plot(Tn[0, :].T)
        axs[1, 1].plot(Rn[0, :].T)

        axs[2, 1].plot(Tn[1, :].T)
        axs[2, 1].plot(Rn[1, :].T)

        axs[3, 1].plot(Tn[2, :].T)
        axs[3, 1].plot(Rn[2, :].T)

        axs[3, 0].set_xlabel("Number of samples")
        axs[3, 1].set_xlabel("Number of samples")
        axs[0, 0].set_ylabel("Myoelectric Reading (mV)")
        axs[1, 0].set_ylabel("Neural Drive Label\n(Power Grasp)")
        axs[2, 0].set_ylabel("Neural Drive Label\n(Pinch Grasp)")
        axs[3, 0].set_ylabel("Neural Drive Label\n(Ulnar Grasp)")

        plt.tight_layout()
        plt.show()


if __name__ == "__main__":

    PLOT = True
    SEED = 1
    N_REP_NMF = 10
    ALPHA = "ar10"
    DATA_PATH = "datasets/data_6"

    np.random.seed(SEED)
    random.seed(SEED)

    OUTPUT_PATH = "datasets/dataset_{}".format(ALPHA)
    labeler = Labeler(data_path=DATA_PATH, alpha_type=ALPHA, diff_signal=True,
n_rep_nmf=N_REP_NMF)
```

```
os.makedirs(OUTPUT_PATH, exist_ok=True)
for subj_id in range(1, 6):
    print("***** subj {} *****".format(subj_id))

    data = labeler.run_subj(subj_id, plot=PLOT)

    with open(f'{OUTPUT_PATH}/subj{subj_id}.pickle', 'wb') as handle:
        pickle.dump(data, handle)
```

**Models Python script** (Feedforward and LSTM algorithms implementation)

```
import torch

class NN(torch.nn.Module):
    # Simple feedforward neural network.

    def __init__(self, input_dim=8, hidden_units=32, out_dim=3, num_layers=3):
        super().__init__()
        self.in_dim = input_dim
        self.mid_dim = hidden_units
        self.out_dim = out_dim
        self.num_layers = num_layers

        self.layers = torch.nn.ModuleList([])
        self.layers.extend([torch.nn.Linear(self.in_dim, self.mid_dim), torch.nn.ReLU()])
        for _ in range(self.num_layers - 2):
            self.layers.extend([torch.nn.Linear(self.mid_dim, self.mid_dim), torch.nn.ReLU()])
        self.layers.extend([torch.nn.Linear(self.mid_dim, self.out_dim), torch.nn.Sigmoid()])

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x
```

```
class LSTM(torch.nn.Module):

    # Simple LSTM network.
    def __init__(self, input_dim=8, hidden_units=32, out_dim=3, num_layers=1):
        super().__init__()
        self.input_dim = input_dim
        self.hidden_units = hidden_units
        self.num_layers = num_layers

        self.lstm = torch.nn.LSTM(
            input_size=input_dim, hidden_size=hidden_units, batch_first=True,
num_layers=self.num_layers)
        self.linear = torch.nn.Linear(in_features=self.hidden_units, out_features=out_dim)

    def forward(self, x):
        batch_size = x.shape[0]
        h0 = torch.zeros(self.num_layers, batch_size, self.hidden_units).requires_grad_()
        c0 = torch.zeros(self.num_layers, batch_size, self.hidden_units).requires_grad_()

        _, (hn, _) = self.lstm(x, (h0, c0))
        out = self.linear(hn[0])

        return out
```

**Train Python script** (Feedforward and LSTM NNs training):

```
import numpy as np
from dataset import EMGDataSeq, EMGData, DataPreProcessing
from models import LSTM, NN
import torch, random
from torch.utils.data import DataLoader
from tqdm import tqdm
```

```
def set_seeds(seed):
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    random.seed(seed)


def train_model(data_loader, model, loss_function, optimizer):
    num_batches = len(data_loader)
    total_loss = 0
    model.train()

    for x, y in data_loader:
        output = model(x)
        loss = loss_function(output, y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    avg_loss = total_loss / num_batches
    print(f"Train loss: {avg_loss}")

    return avg_loss


def val_model(data_loader, model, loss_function):
    num_batches = len(data_loader)
    total_loss = 0

    model.eval()
    with torch.no_grad():
```

```
        for x, y in data_loader:
            output = model(x)
            total_loss += loss_function(output, y).item()

    avg_loss = total_loss / num_batches
    print(f"Test loss: {avg_loss}")

    return avg_loss


if __name__ == "__main__":
    base_config = dict(
        batch_size=32,
        lr=1e-3,
        hidden_units=32,
        seed=0,
        iters=15000,
        device="cpu",
        downsample_rate=10,
        dataset_name="dataset_ar10",
        subject=1,
        network_type="lstm",  # lstm or nn
    )

    set_seeds(base_config["seed"])

    pickle_path = "{}/subj{}.pickle".format(base_config["dataset_name"],
base_config["subject"])

    dataset = DataPreProcessing(pickle_path=pickle_path)
    counter = 0
    for data in dataset.cross_validation_out():
        train_data, val_data, train_label, val_label = data

        if base_config["network_type"] == "lstm":
            dataset_train = EMGDataSeq(np.hstack(train_data), np.hstack(train_label),
```

```
rate=base_config["downsample_rate"])
        dataset_val = EMGDataSeq(np.array(val_data), np.array(val_label),
rate=base_config["downsample_rate"])
    elif base_config["network_type"] == "nn":
        dataset_train = EMGData(np.hstack(train_data), np.hstack(train_label),
rate=base_config["downsample_rate"])
        dataset_val = EMGData(np.array(val_data), np.array(val_label),
rate=base_config["downsample_rate"])
    else:
        raise NotImplementedError

    # DATASET
    loader_train = DataLoader(dataset_train, batch_size=base_config["batch_size"],
num_workers=0, shuffle=False)
    loader_val = DataLoader(dataset_val, batch_size=base_config["batch_size"],
num_workers=0, shuffle=False)

    print("train set: ", len(dataset_train))
    print("val set: ", len(dataset_val))


    if base_config["network_type"] == "lstm":
        model = LSTM(hidden_units=base_config["hidden_units"], num_layers=1)
    elif base_config["network_type"] == "nn":
        model = NN(hidden_units=base_config["hidden_units"], num_layers=3)
    else:
        raise NotImplementedError

    loss_function = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=base_config["lr"])

    epochs = base_config["iters"] // (len(dataset_train) // base_config["batch_size"])

    log_dict = {}
    for epoch in tqdm(range(epochs)):
        loss_train = train_model(loader_train, model, loss_function, optimizer)
```

```
        loss_val = val_model(loader_val, model, loss_function)

        log_dict[epoch] = {"train": loss_train, "val": loss_val}

    state = dict(base_config)
    state["model"] = model.state_dict()
    state["log"] = log_dict
    torch.save(state, f"model_{counter}_{base_config['network_type']}.pt")
    counter += 1
```

### 8.3 Code implementation of NN models' performance evaluation

**Predict Python script** (Feedforward and LSTM NNs' performance evaluation)

```
import numpy as np
from models import LSTM, NN
import torch
from torch.utils.data import DataLoader
from dataset import EMGDataSeq, EMGData, DataPreProcessing
import matplotlib.pyplot as plt

@torch.no_grad()
def test_model(data_loader, model):
    pred_list = []
    label_list = []
    for x, y in data_loader:
        output = model(x)
        pred_list.append(output.detach().numpy())
        label_list.append(y.detach().numpy())

    return np.vstack(pred_list), np.vstack(label_list)


if __name__ == "__main__":
```

```
num_models = 6  #Number of trained models
#Validation loss computation for Feedforward models
val_losses_nn = []
for i in range(num_models):
    checkpoint = f"model_{i}_nn.pt"
    state = torch.load(checkpoint)
    log_dict = state["log"]

    # MODEL
    model = NN(hidden_units=state["hidden_units"], num_layers=3)
    model.load_state_dict(state["model"])
    model.eval()

    # DATASET
    pickle_path = "{}/subj{}.pickle".format(state["dataset_name"], state["subject"])

    dataset = DataPreProcessing(pickle_path=pickle_path)
    dataset_cv_out = dataset.cross_validation_out()
    train_data, val_data, train_label, val_label= next(dataset_cv_out)

    dataset_train = EMGData(np.hstack(train_data), np.hstack(train_label),
rate=state["downsample_rate"])
    dataset_val = EMGData(np.array(val_data), np.array(val_label),
rate=state["downsample_rate"])

    loader_val = DataLoader(dataset_val, batch_size=1, num_workers=0, shuffle=False)

    pred, label = test_model(loader_val, model)

    pred = pred.squeeze()
    label = label.squeeze()

    val_loss = np.sqrt(np.mean((pred - label) ** 2))
    val_losses_nn.append(val_loss)

plt.figure() #Algorithm performance: training vs validation of 6th trained NN model
```

```
train_loss = [[k, v["train"]] for k, v in log_dict.items()]
val_loss = [[k, v["val"]] for k, v in log_dict.items()]
train_loss = np.array(train_loss)
val_loss = np.array(val_loss)
plt.plot(train_loss[:, 0], train_loss[:, 1], label="train")
plt.plot(val_loss[:, 0], val_loss[:, 1], label="val")
plt.xlabel("Number of epochs")
plt.ylabel("Root-Mean-Square Error")
plt.title("Training vs Validation performance for Feedforward NN")
plt.legend()
plt.tight_layout()

fig, axs = plt.subplots(3, 1)  # Actual vs Predicted labels of 6th trained NN model
axs[0].plot(pred[:, 0], label="pred")
axs[0].plot(label[:, 0], label="label")

axs[2].plot(pred[:, 1], label="pred")
axs[2].plot(label[:, 1], label="label")

axs[4].plot(pred[:, 2], label="pred")
axs[4].plot(label[:, 2], label="label")

axs[0].set_ylim = (0, 1)
axs[2].set_ylim = (0, 1)
axs[4].set_ylim = (0, 1)

axs[4].set_xlabel("Number of samples")
axs[2].set_ylabel("Normalized Activation Values")
fig.suptitle("Actual vs Predicted labels for Feedforward NN")

plt.legend()
plt.tight_layout()

# Validation loss computation for LSTM models
val_losses_lstm = []
for i in range(num_models):
```

```
        checkpoint = f"model_{i}_lstm.pt"
        state = torch.load(checkpoint)
        log_dict = state["log"]

        # MODEL
        model = LSTM(hidden_units=state["hidden_units"], num_layers=1)
        model.load_state_dict(state["model"])
        model.eval()

        # DATASET
        pickle_path = "{}/subj{}.pickle".format(state["dataset_name"], state["subject"])

        dataset = DataPreProcessing(pickle_path=pickle_path)
        dataset_cv_out = dataset.cross_validation_out()
        train_data, val_data, train_label, val_label = next(dataset_cv_out)

        dataset_train = EMGDataSeq(np.hstack(train_data), np.hstack(train_label),
rate=state["downsample_rate"])
        dataset_val = EMGDataSeq(np.array(val_data), np.array(val_label),
rate=state["downsample_rate"])

        loader_val = DataLoader(dataset_val, batch_size=1, num_workers=0, shuffle=False)

        pred, label = test_model(loader_val, model)

        pred = pred.squeeze()
        label = label.squeeze()

        val_loss = np.sqrt(np.mean((pred - label) ** 2))
        val_losses_lstm.append(val_loss)

    plt.figure()  # Algorithm performance: training vs validation of 6th trained LSTM model
    train_loss = [[k, v["train"]] for k, v in log_dict.items()]
    val_loss = [[k, v["val"]] for k, v in log_dict.items()]
    train_loss = np.array(train_loss)
    val_loss = np.array(val_loss)
```

```
    plt.plot(train_loss[:, 0], train_loss[:, 1], label="train")
    plt.plot(val_loss[:, 0], val_loss[:, 1], label="val")
    plt.xlabel("Number of epochs")
    plt.ylabel("Root-Mean-Square Error")
    plt.title("Training vs Validation performance for LSTM NN")
    plt.legend()
    plt.tight_layout()

    fig, axs = plt.subplots(3, 1)  # Actual vs Predicted labels of 6th trained LSTM model
    axs[0].plot(pred[:, 0], label="pred")
    axs[0].plot(label[:, 0], label="label")

    axs[2].plot(pred[:, 1], label="pred")
    axs[2].plot(label[:, 1], label="label")

    axs[4].plot(pred[:, 2], label="pred")
    axs[4].plot(label[:, 2], label="label")

    axs[0].set_ylim = (0, 1)
    axs[2].set_ylim = (0, 1)
    axs[4].set_ylim = (0, 1)

    axs[4].set_xlabel("Number of samples")
    axs[2].set_ylabel("Normalized Activation Values")
    fig.suptitle("Actual vs Predicted labels for LSTM NN")

    plt.legend()
    plt.tight_layout()

    #Barplot for comparing Feedforward and LSTM performance

    fig, ax = plt.subplots()
    ax.bar(0, np.mean(val_losses_nn), yerr=np.std(val_losses_nn), color='blue', alpha=0.5,
capsize=4, label='Feedforward')
    ax.bar(1, np.mean(val_losses_lstm), yerr=np.std(val_losses_lstm), color='orange',
alpha=0.5, capsize=4, label='LSTM')
```

```
ax.set_xticks([0, 1])
ax.set_xticklabels(['Feedforward', 'LSTM'])
ax.set_ylabel('Average RMSE Loss')
legend=False
plt.title("Feedforward vs LSTM algorithms' prediction performance")

plt.tight_layout()
plt.show()
```

In summary, this script loads the checkpoints corresponding to the different trained models, infers on a validation dataset and performs a graphical evaluation of the algorithms performance, first individually and afterwards a general comparison of both ML algorithms' predictions. The script works in the following way:

First, the necessary libraries and modules in order to perform the model's evaluation are imported, such as **numpy** for numerical operations [11], **LSTM** and **NN** from the models module, **torch** for deep learning tools [18], **DataLoader** from **torch.utils.data** to create data loaders for the datasets and **matplotlib.pyplot** for graph charting [12].

Then, the function **test_model** is defined. It accepts **data_loader** and **model** as input and does inference on the data given by the data loader using the model. It loops over the data loader, runs the model on the input data **x**, appends the predictions (**output**) and labels (**y**) to separate lists, and eventually returns the forecasts and labels as NumPy arrays using **np.vstack** to vertically stack the lists.

Afterwards, inside the main block, a "for loop" is first defined for performing the evaluation of the trained Feedforward NNs by computing the validation loss resulting from testing the models on a validation dataset. This cannot be done without first importing the checkpoint file which is used for loading the chosen model's state that comprises the data about the trained model.

The dictionary **log_dict** is then given the "**log**" key value from the loaded state, which contains the training and validation loss values obtained when the models were trained for evaluating the degree of accuracy achievement during this process. Furthermore, the number of hidden units and number of layers collected from the loaded state are used to create the respective model classes. Then, **model.load_state_dict** is used to load the model's state dictionary from the loaded state and **model.eval()** is used to switch the model to evaluation mode, which inhibits gradient calculation and triggers evaluation-specific activities.

Moreover, the dataset is created by loading the pickle file supplied by **pickle_path** and creating a **dataset** object using the **DataPreProcessing** class for performing the outer cross-validation process the same way as in the training script. Once the datasets and data loader are built, the **loader_val** and **model** are sent to the **test_model** function for the predictions and labels to be generated and set to the **pred** and **label** variables, respectively. This allows to compute the RMSE validation loss for each available trained model, which are all then stored in the **val_losses_nn** variable so that they can be later used for the algorithms' performance comparison.

Once the validation loss computation loop has finished, the training and validation losses from the last model evaluated are plotted, as well as the actual and predicted labels, so that a graphical representation of its performance is generated.

The same validation loop is performed for the LSTM trained models, generating the vector called **val_losses_lstm** that contains the RMSE validation losses of the different evaluated models, as well as the equivalent plots that were just explained for the Feedforward algorithm.

Finally, a bar plot is generated with the average value of the mentioned vectors containing the RMSE losses for both algorithms, as well as a confidence interval resulting form calculating the standard deviation between the computed losses of the different trained models, so that a fair comparison can be performed between both Feedforward and LSTM algorithms (all graphical results are shown in the results subsections).