Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Matematica

ARTIFICIAL NEURAL NETWORKS FOR CLASSIFICATION OF EMG DATA IN HAND MYOELECTRIC CONTROL

Tesi di Laurea in Analisi Matematica

Relatore:
Chiar.ma Prof.ssa
GIOVANNA CITTI

Presentata da:
VIVIANA TARULLO

Correlatori:
Prof.
DAVIDE BARBIERI
Ing.
EMANUELE GRUPPIONI
Dott.ssa
NOEMI MONTOBBIO

Sessione Unica
Anno Accademico 2018/2019

Introduction

This thesis studies the state-of-the-art in myoelectric control of active hand prostheses for people with trans-radial amputation using pattern recognition and machine learning techniques.
Myoelectric control systems are based both on the phantom limb, which is the impression of being able to move an amputated limb, and on the technique of electromyography (EMG), which records electrical potentials generated in neuromuscular activation. In prosthetic control, EMG signals are collected by surface sensors able to directly activate electrical motors of the prosthetic device. This procedure is not invasive, and combined with pattern recognition techniques it leads to a natural control due to the corrispondence between movements of the prosthesis and phantom limb gestures.
Our work is supported by Centro Protesi INAIL in Vigorso di Budrio (BO). We studied the control system developed by INAIL consisting in acquiring EMG signals from amputee subjects and using pattern recognition methods for the classification of acquired signals, associating them with specific gestures and consequently commanding the prosthesis. Our work consisted in improving classification methods used in the learning phase. In particular, we proposed a classifier based on a neural network as a valid alternative to the INAIL one-versus-all approach to multiclass classification.

The thesis is structured as follows:

- In Chapter 1 we describe the biomedical background introducing the electromyography tecnique and giving an overwiew on the functioning

of upper limb prostheses. The focus is on myoelectric control and on the structure of a pattern recognition-based system;

- In Chapter 2 we describe some classical classification methods present in literature. In particular we intoduce Logistic regression (LR), that is a linear and binary supervised classification algorithm whose aim is to calculate a class membership probability, used in multiclass classification following the one-versus-all approach, and the Softmax classifier, that is the extension of LR to multiclass classification. Finally we describe the structure of Artificial Neural Networks (ANN), from the Perceptron that is the first neural network described by algorithms to the Multilayer Perceptron and Convolutional Neural Networks (CNN);

- In Chapter 3 we show the results of our analysis on classifiers. First we describe the setting of the esperiment and the structure of the Matlab code to build each classifier, then we discuss classification performances obtained with our code tested on data sets acquired by INAIL patients.

The results obtained with the algorithm proposed here considerably improve the performances obtained with the one previously used by INAIL.

Introduzione

Questa tesi studia lo stato dell'arte riguardo al controllo mioelettrico delle protesi attive per la mano destinate a persone con amputazione trans-radiale, utilizzando tecniche di pattern recognition e machine learning.
I sistemi di controllo mioelettrico si basano sia sull'arto fantasma, ovvero l'impressione di poter muovere un arto amputato, sia sulla tecnica dell'elettromiografia (EMG), che registra i potenziali elettrici generati nell'attivazione neuromuscolare. Nel controllo protesico, i segnali EMG sono raccolti da sensori di superficie in grado di attivare direttamente i motori elettrici del dispositivo protesico. Questa procedura è non invasiva, e associata a tecniche di pattern recognition porta a un controllo naturale dovuto alla corrispondenza tra i movimenti della protesi e i gesti compiuti dall'arto fantasma.
Il nostro lavoro è supportato dal Centro Protesi INAIL di Vigorso di Budrio (BO). Noi abbiamo studiato il sistema di controllo sviluppato in INAIL che consiste nell'acquisizione dei segnali EMG da soggetti con amputazione e nell'utilizzo di metodi di pattern recognition per classificare i segnali acquisiti, associandoli a gesti specifici e comandando conseguentemente la protesi. Il nostro lavoro è consistito nel migliorare i metodi di classificazione utlizzati nella fase di apprendimento. In particolare abbiamo proposto un classificatore basato su una rete neurale come valida alternativa all'approccio alla classificazione multiclasse del tipo one-versus-all adottato in INAIL.

La tesi è strutturata come segue:

- Nel Capitolo 1 descriviamo il background biomedico introducendo la

tecnica dell'elettromiografia e dando una panoramica del funzionamento delle protesi di arto superiore. L'attenzione è focalizzata sul controllo mioelettrico e sulla struttura di un sistema basato sulla tecnica di pattern recognition;

- Nel Capitolo 2 descriviamo alcuni classici metodi di classificazione presenti in letteratura. In particolare introduciamo la Logistic Regression (LR), un algoritmo di classificazione lineare e binario il cui obiettivo è di calcolare la probabilità di appartenenza a una classe, utilizzato nella classificazione multiclasse seguendo l'approccio one-versus-all, e il classificatore Softmax, ovvero l'estensione della LR alla classificazione multiclasse. Infine descriviamo la struttura delle Reti Neurali Artificiali (ANN) dal Perceptron, che è la prima rete neurale descritta da algoritmi, al Multilayer Perceptron e alle Reti Neurali Convoluzionali (CNN);

- Nel Capitolo 3 proponiamo i risultati della nostra analisi sui classificatori. Inizialmente descriviamo l'impostazione dell'esperimento e la struttura del codice Matlab per costruire ogni classificatore, in seguito analizziamo le performances di classificazione ottenute con il nostro codice testate su data set acquisiti da pazienti INAIL.

I risultati ottenuti con l'algoritmo qui proposto migliorano notevolmente le performances ottenute con quello precedentemente utilizzato in INAIL.

Contents

Introduction **i**

Introduzione **iii**

1 EMG signal and pattern recognition **1**

1.1 sEMG 1

1.2 Myoelectric prostheses 2

1.3 Pattern recognition 2

1.3.1 EMG data acquisition 4

1.3.2 Feature extraction 4

1.3.3 Classification 4

2 Classification methods **7**

2.1 Logistic Regression 9

2.2 Softmax 12

2.3 Model optimization 15

2.3.1 Method of gradient descent 16

2.3.2 Variations to GD method 17

2.4 Artificial Neural Networks 19

2.4.1 Model of a neuron 20

2.4.2 ANN structure 22

2.4.3 Rosenblatt's Perceptron 24

2.4.4 Multilayer Perceptron 27

2.4.5 Convolutional Neural Networks 36

3 Experiments and results **41**
3.1 Experiment setup 42
3.2 Non-linear Logistic Regression classifier 44
3.2.1 Scaling and subsampling 44
3.2.2 Models of features 47
3.2.3 Structure of NLR classifier 49
3.2.4 Training process 52
3.2.5 Decision thresholds 56
3.2.6 Network evaluation 57
3.2.7 Test on GS and voting 58
3.3 Multilayer Feedforward ANN 60
3.3.1 Scaling and subsampling 61
3.3.2 Structure of the multiclass classifier 62
3.3.3 Training process 65
3.3.4 Test on GS and voting 66
3.4 CNN 67
3.4.1 Reorganization of the data set 68
3.4.2 Structure of the CNN 69
3.4.3 Training process and evaluation 70
3.5 Results and discussion 72
3.6 Conclusions 86

Bibliography **89**

Ringraziamenti **91**

Chapter 1

EMG signal and pattern recognition

In this chapter we present EMG signals and describe their use in problems of hand prosthesis control, adopting the experimental setup implemented by Centro Protesi INAIL (BO).
The main references for this chapter are [2], [3], [7], [8], [9], [10].

1.1 sEMG

The key technique of this analysis is surface electromyography (sEMG). The EMG signal measures the electrical activation of the muscular fibres generated during the voluntary contraction of a muscle, called muscle action potentials.
The most accurate tecnique for EMG signal acquisition is to use implantable sensing electrodes, to be placed directly near the muscular fibers; however, they are invasive and need surgery. Therefore, surface electrodes can be used to carry out the measurement: when EMG electrodes are placed on the skin surface they measure all the action potentials of the fibres underlying the electrode. These sensors are made by metal plates connected to the inputs of

a differential amplifier that can sense the action potential of muscular cells and amplify the signal.
With respect to implantable electrodes, surface sensors suffer from lack of performance due to the noise mainly caused by motion artefacts, electrical equipment and floating ground noise. Minimizing this noise during the acquisition of the signal is crucial.

1.2 Myoelectric prostheses

In general, prosthetic devices for amputee patients can be divided in passive and active ones. Passive prostheses do not support any of the hand functionalities (as for example cosmetic ones, a prosthetic option similar in appearance to the non-affected arm or hand that provides just a simple help in balancing and carrying), while active prostheses are externally powered and perform advanced grasping and control functionalities.
Upper limb myoelectric prostheses are active prostheses that use small electrical motors to provide power to the active joints. These motors are directly activated by the amputee by means of sEMG input signals, collected by sensors properly placed and then processed by a programmable electronic circuit that carries out the control strategy. In this way, prostheses are controlled via surface electromyography by patients themselves who take advantage of contractions of what remains of muscles (corresponding to the real movements) to assume allowed hand postures.

1.3 Pattern recognition

Myoelectric control, i.e. feed-forward control of prostheses using surface EMG, is in use since the 1960s to control (externally powered) upper-limb prostheses by amputees mostly due to its relatively low cost and non-invasiveness. However, controlling via sEMG has often been quite non-

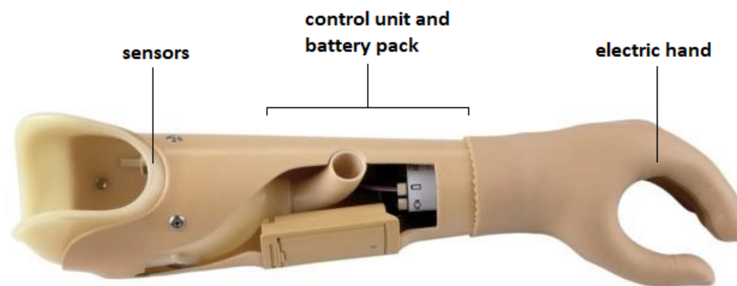

Figure 1.1: Myoelectric prosthesis

natural, meaning that the amputee had to learn to associate muscle remnants contractions to unrelated postures of the prosthetic device. Since remaining muscles continue to work after the amputation, it would be natural to let the amputee command a grasp posture just by performing the corresponding action with the “phantom limb”, i.e. the part of the limb that has been amputated but that the amputee still feels to have, which includes the hand: indeed, [7] shows that amputees can still produce distinct and stable EMG signals several years after the operation.
Pattern recognition and machine learning techniques applied to sEMG currently represent a good compromise between invasiveness and natural prosthesis controllability.
A pattern recognition-based system is tipically structured in three main steps:

- EMG signal acquisition;
- feature extraction and identification of the information content of the signal;
- classification, i.e. assigning the extracted features to the class (gesture) they most probably belong to, after a training phase during which the system learns the way of linking myoelectric patterns to the postures.

1.3.1 EMG data acquisition

During the acquisition of EMG signals the most important goal is reducing noise to get a good classification performance afterwards.
The bandwidth of the signals stays within 2 kHz, while the amplitude is 20 mV (-10 to $+10$) depending on the diameter of the muscles and on the distance from the sensing elements. To provide significant values of muscular activation, Ottobock 13E200 sensors are used: these are pre-amplified electrodes that amplify and filter EMG signals to reach an output range of 0–5 V, with a bandwidth of 90–450 Hz. To minimize the misplacement of the EMG electrodes among different acquisition sessions, the sensors are equidistantly placed on a silicone adjustable bracelet to locate about 5 cm below the elbow, around the stump. The data are collected using a purpose-built software acquisition system and USB-transmitted to the PC.

1.3.2 Feature extraction

The process of extracting the main components of signals typically occurs after the segmentation of collected data during time, fixing a time window length depending on the ability of patients. Examples of commonly used time domain features are mean (M), variance (V), root mean square (RMS), slope sign change (SSC), Willison amplitude (WA), simple square integral (SSI) and waveform length (WL) (for more details see [8]).
In addition to feature extraction, a dimensionality reduction can be applied to EMG data making use of Principal Component Analysis technique (PCA), i.e. orthogonally projecting the data onto a lower dimensional linear space such that the variance of the projected data is maximized ([4, Section 12.1]).

1.3.3 Classification

Pattern recognition techniques applied to myoelectric control are based on supervised machine learning classification algorithms. Therefore an initial

training phase is needed, during which the system learns to associate collected EMG signals to the correct gesture label. Afterwards, the trained system is set to recognize postures.

Classification accuracy varies according to different classifiers but mostly depends on the ability of the patients to associate commanded actions with corresponding phantom limb movements.

In the next chapter an excursus about classification algorithms focusing on Artificial Neural Networks (ANN) will be presented.

Chapter 2

Classification methods

Machine learning (ML) can be defined as the sub-domain of Artificial Intelligence that allows computer systems to perform a specific task (as for example classification) with no need of an explicit model, but relying on inference. Learning procedures can be categorized as follows:

i) *supervised learning*, that relies on the availability of a dataset of labeled samples made of input signals with the corresponding target response in order to realize a specific input-output mapping by minimizing a proper cost function;

ii) *unsupervised learning*, which employs a set of inputs without any corresponding target values, that consists in the self-organized process of determining correspondences within the data according to distribution, visualization or similarities of samples;

iii) *reinforcement learining*, based on the continuing interactions between a learning system and the response of its environment in order to maximize a reward.

In particular, supervised ML problems can be categorized into *regression* and *classification* problems: in a regression problem the desired output consists of one or more continuous variables while in a classification problem the aim is to predict results among a discrete output set corresponding to

different categories.

In this work the focus is on supervised ML multiclass classification methods.
In the case of classification involving $K > 2$ classes, there are two possible approaches: one consists in combining multiple binary classifiers in order to build a multiclass classifier, another is to construct a unique classifier whose output is a vector of probabilities. In the first case, K separated binary classifiers are trained respectively using the data from every class C_k, with $k = 1, \ldots, K$, as the positive samples and the data from the remaining $K - 1$ classes as the negative samples (this is known as the *one-versus-all* approach). In the second case, a class label for training is represented by a vector $\mathbf{t} = (t_1, \ldots, t_K)$ of length K such that if the class is C_j, then all elements t_k of $\mathbf{t}$ are 0 except element t_j, which takes the value 1. Here the value of t_k is interpreted as the probability that the sample class is C_k, and data are classified according to these probability values.

In this chapter we present some classical classification methods. We start from Logistic Regression for a one-vs-all approach (with reference to [17], [6]), then we describe Softmax classifier for multiclass classification (with reference to [18]) and Artificial Neural Networks. In particular, we focus on Rosenblatt's Perceptron, Multilayer Perceptron and Convolutional Neural Networks (with reference to [13], [11, Chapter 9]). The description of these methods is given in order to understand the structure of classifiers that will be used in our experiment and examined in the next chapter, since Logistic Regression with the one-versus-all approach is the method currently used in INAIL to approach the problem of the recognition of myoelectric patterns, while our new proposals are a Softmax classifier built on an Artificial Neural Network and a Convolutional Neural Network.

In general, the main references for the topics in this chapter are [4], [12] and [5].

2.1 Logistic Regression

Logistic Regression (LR) is a linear and binary supervised classification algorithm whose aim is to calculate the class membership probability.
Given a dataset $\{(x(n), t(n))\}_{n=1}^N$ of N samples, where $x(n) \in \mathbb{R}^m$ are the input variables, or *features*, and $t(n) \in \{0, 1\}$ are the target variables, we can write the probabilities that the class is $t(n) = 1$ and $t(n) = 0$ for all $n = 1, \ldots, N$ respectively as

$$P(t(n) = 1|x(n), \theta) = \sigma(\theta^T x(n) + \theta_0) \tag{2.1}$$

$$P(t(n) = 0|x(n), \theta) = 1 - \sigma(\theta^T x(n) + \theta_0) \tag{2.2}$$

where $\theta \in \mathbb{R}^m$ is the vector of classification parameters, or *weights*, $\theta_0 \in \mathbb{R}$ is the *bias* term and σ is the *logistic sigmoid* function that maps its input to an output between 0 and 1, defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{2.3}$$

We observe that denoting $z(n) = \theta^T x(n) + \theta_0$, the parameters θ transform each input sample $x(n)$ into an input $z(n)$ to the logistic function, and the output of the model $y(n) = \sigma(z(n)) = P(t(n) = 1|z(n))$ can be interpreted as the probability that input $z(n)$ belongs to 1-class.

The Logistic Regression model will be optimized by maximizing the likelihood that a given set of weights θ may predict the correct class of each input sample.
Denoting the likelihood function with $\mathcal{L}(\theta|t, z)$, the maximization can be written as

$$\underset{\theta}{\operatorname{argmax}}\, \mathcal{L}(\theta|t, z) = \underset{\theta}{\operatorname{argmax}} \prod_{n=1}^{N} P(t(n), z(n)|\theta)$$

where $P(t(n), z(n)|\theta)$ is the joint probability of generating $t(n)$ and $z(n)$ given the parameters θ. We can rewrite this probability as

$$P(t(n), z(n)|\theta) = P(t(n)|z(n), \theta)P(z(n)|\theta) = P(t(n)|z(n), \theta).$$

Since the variable $t(n)$ can assume 0 or 1 values, we can see the set of $t(n)$ for $n = 1, \dots, N$ as independent Bernoulli variables. In general, for a Bernoulli variable $S \sim Bernoulli_{\mathrm{p}}$ the following formula holds:

$$P(S = k) = p^k (1-p)^{1-k} \quad \text{for } k = 0, 1.$$

Therefore, fixed a set of weights θ we have

$$\begin{aligned} P(t(n)|z(n)) &= P(t(n) = 1|z(n))^{t(n)} \cdot (1 - P(t(n) = 1|z(n)))^{1-t(n)} \\ &= y(n)^{t(n)} \cdot (1 - y(n))^{1-t(n)}. \end{aligned}$$

Now, since the logaritmic function is a monotone increasing function, the set of weights that maximizes the likelihood function will be the same as the one that maximizes its logarithm. The advantage of using log-likelihood maximization is the prevention of numerical underflow due to low probabilities. Then, according to previous observations, for a fixed θ we obtain

$$\begin{aligned} \log \mathcal{L}(\theta|t, z) &= \log \prod_{n=1}^{N} y(n)^{t(n)} \cdot (1 - y(n))^{1-t(n)} \\ &= \sum_{n=1}^{N} t(n) \log(y(n)) + (1 - t(n)) \log(1 - y(n)). \end{aligned}$$

Remarking that

$$\max_{\theta} \ \log \mathcal{L}(\theta|t, z) = \min_{\theta}(-\log \mathcal{L}(\theta|t, z)), \tag{2.4}$$

we can introduce an error function $\xi(t, y)$, known as the *cross-entropy loss function*, defined as follows:

$$\begin{aligned} \xi(t, y) &= -\log \mathcal{L}(\theta|t, z) \\ &= -\sum_{n=1}^{N} [t(n) \log(y(n)) + (1 - t(n)) \log(1 - y(n))] \\ &= -\sum_{n=1}^{N} [t(n) \log(\sigma(z(n))) + (1 - t(n)) \log(1 - \sigma(z(n)))]. \end{aligned}$$

Pointing out the behaviour of this error function, we can understand that it works well as a loss function for Logistic Regression. In fact, denoting for all $n = 1, \dots, N$

$$\xi(t(n), y(n)) = \begin{cases} -\log(y(n)) & \text{if } t(n) = 1 \\ -\log(1 - y(n)) & \text{if } t(n) = 0 \end{cases}$$

it's clear that when $t(n) = 1$ the loss is 0 if $y(n) = 1$ and goes to infinity as $y(n) \to 0$, and in case of $t(n) = 0$ the loss is 0 if $y(n) = 0$ and goes to infinity as $y(n) \to 1$. So the contribute of every sample to the loss function grows when the probability to predict the correct class goes to 0, and finding the set of weights θ that minimize the cross-entropy loss function makes sense. Another reason to use the cross-entropy function in LR model is that it is a convex loss function, and this results in a convex optimization problem in which a global minimum exists.

We finally show how to calculate the derivative of the term $\xi(t(n), y(n))$ in the cross-entropy loss for the logistic function, which is crucial in gradient based optimization techiques (described further in Section 2.3). Firstly, it is necessary to calculate the derivative of the output $y(n)$ of the logistic function σ with respect to the input $z(n)$:

$$\frac{\partial y(n)}{\partial z(n)} = \frac{\partial \sigma(z(n))}{\partial z(n)} = \frac{-1}{(1 + e^{-z(n)})^2} \cdot e^{-z(n)} \cdot (-1) = \frac{1}{1 + e^{-z(n)}} \cdot \frac{e^{-z(n)}}{1 + e^{-z(n)}}.$$

Since $1 - \sigma(z(n)) = 1 - \frac{1}{1+e^{-z(n)}} = \frac{e^{-z(n)}}{1+e^{-z(n)}}$, we obtain

$$\frac{\partial y(n)}{\partial z(n)} = \sigma(z(n))(1 - \sigma(z(n))) = y(n)(1 - y(n)). \tag{2.5}$$

Now we can calculate the derivative of the term $\xi(t(n), y(n))$ with respect to the input $z(n)$. By the chain rule for derivatives we have

$$\frac{\partial \xi(t(n), y(n))}{\partial z(n)} = \frac{\partial \xi(t(n), y(n))}{\partial y(n)} \frac{\partial y(n)}{\partial z(n)}.$$

The second derivative in the right hand side is given by (2.5), so we are going to calculate $\frac{\partial \xi(t(n),y(n))}{\partial y(n)}$:

$$\begin{aligned}\frac{\partial \xi(t(n), y(n))}{\partial y(n)} &= \frac{\partial(-t(n)\log(y(n)) - (1-t(n))\log(1-y(n)))}{\partial y(n)} \\ &= \frac{-t(n)}{y(n)} + \frac{1-t(n)}{1-y(n)} = \frac{y(n)-t(n)}{y(n)(1-y(n))}.\end{aligned}$$

Therefore we finally have

$$\frac{\partial \xi(t(n), y(n))}{\partial z(n)} = \frac{y(n)-t(n)}{y(n)(1-y(n))} y(n)(1-y(n)) = y(n) - t(n).$$

Logistic Regression as a Non-linear Classifier

Linearity in the Logistic Regression model is given by the formula

$$z(n) = \theta^T x(n) + \theta_0$$

that determines a linear correlation between features and classification parameters. There is the possibility to extend classification by LR algorithm to the non-linear case: it requires the creation of additional features to obtain non-linear terms in the previous equation. A way to obtain new features is combining the starting ones $x(1), x(2), \dots, x(N)$ by multiplications. For example, we have new *polynomial features* starting from initial features high till the chosen non-linearity degree and considering all the multiplications between the possible permutations of these new features without exceeding in non-linearity degree (for example $x(1), ..., x(N), x(1)x(2), x(1)x(3), \dots, x(1)^2$, $..., x(N)^2, x(1)x(2)x(3), \dots, x(1)^3, ..., x(N)^3$ if the maximum degree of non-linearity is 3). Therefore the dimension of the weights vector changes according to the selected non-linearity degree.

2.2 Softmax

Softmax classification can be considered the extension of Logistic Regression to multiclass classification.

Given a dataset $\{(x(n), t(n))\}_{n=1}^{N}$ of N samples, where $x(n) \in \mathbb{R}^m$ are the features, now we consider K distinct categories thus $t(n) \in \{1, \ldots, K\}$ are the target variables.
To generalize the logistic sigmoid function to output a multiclass categorical probability distribution we introduce the *softmax function* ς. This function takes a K-dimensional input vector $\mathbf{z} = (z_1, \ldots, z_K)$ and outputs a K-dimensional vector $\mathbf{y} = (y_1, \ldots, y_K)$ whose components are real values between 0 and 1 defined as follows:

$$y_k = \varsigma(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } k = 1, \ldots, K. \tag{2.6}$$

Here the denominator makes sure that the softmax function output satisfies $\sum_{k=1}^{K} y_k = 1$, so we may correctly interpret components y_k as probabilities.
While in LR model we have a vector $\theta \in \mathbb{R}^m$ of classification parameters and a real bias term, now the set of weights consists in a matrix $\Theta \in \mathbb{R}^{K \times m}$, where K is the total number of classes and m is the features dimension, and the bias term is a vector $\theta_0 \in \mathbb{R}^K$. So we can observe, in analogy with the LR classification, that denoting $\mathbf{z}(n) = \Theta x(n) + \theta_0$, the parameters Θ transform each input sample $x(n)$ into an input $\mathbf{z}(n) \in \mathbb{R}^K$ to the softmax function.
We can write the probabilities that the class is $t(n) = k$ for $k = 1, \ldots, K$ given an input $\mathbf{z}(n)$ for $n = 1, \ldots, N$ as

$$\begin{bmatrix} P(t(n) = 1|\mathbf{z}(n)) \\ \vdots \\ P(t(n) = K|\mathbf{z}(n)) \end{bmatrix} = \begin{bmatrix} \varsigma(\mathbf{z}(n))_1 \\ \vdots \\ \varsigma(\mathbf{z}(n))_K \end{bmatrix} = \frac{1}{\sum_{k=1}^{K} e^{z_k(n)}} \begin{bmatrix} e^{z_1(n)} \\ \vdots \\ e^{z_K(n)} \end{bmatrix} \tag{2.7}$$

where $P(t(n) = k|\mathbf{z}(n))$ is the probability that the class is k given the input $\mathbf{z}(n)$.

The Softmax model will be optimized by maximizing the likelihood that a given set of weights Θ may predict the correct class of each input sample. For the following study, the class label for samples $x(n)$ is represented by the vector $\mathbf{t}(n) = (t_1(n), \ldots, t_K(n))$ of length K such that if the class is j, then

all elements $t_k(n)$ of $\mathbf{t}(n)$ are 0 except element $t_j(n)$, which takes the value 1.

Denoting the likelihood function with $\mathcal{L}(\Theta|\mathbf{t},\mathbf{z})$, the maximization can be written as

$$\underset{\Theta}{\operatorname{argmax}}\, \mathcal{L}(\Theta|\mathbf{t},\mathbf{z}) = \underset{\Theta}{\operatorname{argmax}} \prod_{n=1}^{N} P(\mathbf{t}(n),\mathbf{z}(n)|\Theta)$$

where $P(\mathbf{t}(n),\mathbf{z}(n)|\Theta)$ is the joint probability of generating $\mathbf{t}(n)$ and $\mathbf{z}(n)$ given the parameters Θ. Since

$$P(\mathbf{t}(n),\mathbf{z}(n)|\Theta) = P(\mathbf{t}(n)|\mathbf{z}(n),\Theta)P(\mathbf{z}(n)|\Theta) = P(\mathbf{t}(n)|\mathbf{z}(n),\Theta)$$

and in the variable $\mathbf{t}(n)$ only one class can be activated, fixed a matrix of weights Θ we have

$$P(\mathbf{t}(n)|\mathbf{z}(n)) = \prod_{k=1}^{K} P(t_k(n)|\mathbf{z}(n))^{t_k(n)} = \prod_{k=1}^{K} (\varsigma(\mathbf{z}(n))_k)^{t_k(n)} = \prod_{k=1}^{K} (y_k(n))^{t_k(n)}.$$

where $\varsigma(\mathbf{z}(n))_k = y_k(n)$ is the k-th component of the softmax function output given the input $\mathbf{z}(n)$.

As observed during the derivation of the loss function for the logistic function, the parameters Θ that maximize the likelihood are the same that minimize the negative log-likelihood (see (2.4)). So introducing the *cross-entropy loss function* for the softmax function

$$\xi(\mathbf{t},\mathbf{y}) = -\log \mathcal{L}(\Theta|\mathbf{t},\mathbf{z}) = -\log \prod_{n=1}^{N}\prod_{k=1}^{K} (y_k(n))^{t_k(n)} = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_k(n)\log(y_k(n)) \tag{2.8}$$

and remarking that $t_k(n)$ is 1 if and only if n-th sample belongs to class k and $y_k(n)$ is the output probability that n-th sample belongs to class k, the set of weights Θ that optimize the Softmax model is calculated minimizing the cross-entropy loss function.

Finally, if we write the cross-entropy loss for the softmax function given by (2.8) as $\xi(\mathbf{t},\mathbf{y}) = \sum_{n=1}^{N} \xi(\mathbf{t}(n),\mathbf{y}(n))$, we show how to calculate the derivative

of the term $\xi(\mathbf{t}(n), \mathbf{y}(n))$ as we did for LR model.
As before, we need to calculate the derivative of the output $\mathbf{y}(n)$ of the softmax function with respect to its input $\mathbf{z}(n)$. Defining $\Sigma_K = \sum_{d=1}^{K} e^{z_d(n)}$ so that $y_k(n) = \frac{e^{z_k(n)}}{\Sigma_K}$, we have the following two cases:

i) if $k = j$:
$\frac{\partial y_j(n)}{\partial z_j(n)} = \frac{e^{z_j(n)}\Sigma_K - e^{z_j(n)}e^{z_j(n)}}{\Sigma_K^2} = \frac{e^{z_j(n)}}{\Sigma_K}(1 - \frac{e^{z_j(n)}}{\Sigma_K}) = y_j(n)(1 - y_j(n))$;

ii) if $k \neq j$:
$\frac{\partial y_k(n)}{\partial z_j(n)} = \frac{0 - e^{z_k(n)}e^{z_j(n)}}{\Sigma_K^2} = -\frac{e^{z_k(n)}}{\Sigma_K}\frac{e^{z_j(n)}}{\Sigma_K} = -y_k(n)y_j(n)$.

Now we can calculate the derivative of $\xi(\mathbf{t}(n), \mathbf{y}(n))$ with respect to the input $\mathbf{z}(n)$ as:

$$
\begin{aligned}
\frac{\partial \xi}{\partial z_j(n)} &= -\sum_{k=1}^{K} \frac{\partial (t_k(n)\log(y_k(n)))}{\partial z_j(n)} = -\sum_{k=1}^{K} t_k(n)\frac{\partial \log(y_k(n))}{\partial z_j(n)} \\
&= -\sum_{k=1}^{K} t_k(n)\frac{1}{y_k(n)}\frac{\partial y_k(n)}{\partial z_j(n)} = -\frac{t_j(n)}{y_j(n)}\frac{\partial y_j(n)}{\partial z_j(n)} - \sum_{k\neq j}\frac{t_k(n)}{y_k(n)}\frac{\partial y_k(n)}{\partial z_j(n)} \\
&\overset{\text{i),ii)}}{=} -\frac{t_j(n)}{y_j(n)}y_j(n)(1 - y_j(n)) - \sum_{k\neq j}\frac{t_k(n)}{y_k(n)}(-y_k(n)y_j(n)) \\
&= -t_j(n) + t_j(n)y_j(n) + \sum_{k\neq j} t_k(n)y_j(n) \\
&= -t_j(n) + y_j(n)\left(\sum_{k=1}^{K} t_k(n)\right) = -t_j(n) + y_j(n).
\end{aligned}
$$

2.3 Model optimization

For both LR model and Softmax model, we found that the optimum set of parameters (weights) to predict the correct class for each sample is calculated by minimizing the respective cross-entropy loss. In the following section we will show the iterative algorithm to reach this minimum.

In general, consider a loss function $J(\theta)$ that is a continuously differentiable function of the unknown weigths $\theta = (\theta_1, \dots, \theta_M)$. The output function

$J(\theta)$ is a real number and represents a sort of measure of how well parameters behave according to the problem. The aim is to find an optimal solution θ^* that satisfies

$$J(\theta^*) \leq J(\theta), \tag{2.9}$$

so we need to solve the following *unconstrained-optimization problem*:

Minimize the loss function $J(\theta)$ with respect to the weight vector θ.

A necessary condition for optimality is

$$\nabla J(\theta^*) = \mathbf{0}$$

where $\nabla = \left[\frac{\partial}{\partial \theta_1}, \frac{\partial}{\partial \theta_2}, \dots, \frac{\partial}{\partial \theta_M}\right]^T$ is the gradient operator, therefore $\nabla J(\theta) = \left[\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \dots, \frac{\partial J}{\partial \theta_M}\right]^T$.
A class of unconstrained-optimization problems is based on the idea of local *iterative descent*.

2.3.1 Method of gradient descent

The method of gradient descent is a general minimization method, based on the following theorem, with reference to [1].

Theorem 2.3.1. *Let $J \in C^2(\mathbb{R}^M, \mathbb{R})$. Suppose that J is convex, bounded from below and satisfies the condition $J(\theta) \to +\infty$ if $|\theta| \to +\infty$. Consider the Cauchy problem*

$$\begin{cases} \theta'(t) = -\nabla J(\theta(t)) \\ \theta(t_0) = \theta_0. \end{cases} \tag{2.10}$$

Then the solution satisfies $\theta(t) \to \theta_1$ with θ_1 minimum point of J.

Remark 1. Theorem 2.3.1 admits generalisations under the hypothesis of convexity. Without this condition, the method could stop in presence of local minimum points or critical points where the functional assumes a lower value with respect to the initial one.

The gradient descent method (GD) is an algorithm implementing the solution of the Cauchy problem (2.10) in order to find the minimum. The direction of the gradient, $\nabla J(\theta)$, is called the direction of *steepest descent.* Denoting with

$$g = \nabla J(\theta),$$

the gradient descent algorithm is formally described by the discretization of the condition (2.10)

$$\theta^{(n+1)} = \theta^{(n)} - \eta g^{(n)} \tag{2.11}$$

where η is a positive constant called *learning rate* and $g^{(n)}$ is the gradient vector evaluated at the point $\theta^{(n)}$. Therefore, the algorithm applies the *correction*

$$\Delta\theta^{(n)} = \theta^{(n+1)} - \theta^{(n)} = -\eta g^{(n)}. \tag{2.12}$$

To show that the gradient descent algorithm satisfies the unconstrained-optimization condition for iterative descent, we simply note that

$$(J(\theta(t)))' = \nabla J(\theta(t))\theta'(t) \overset{(2.10)}{=} -||\nabla J(\theta(t))||^2.$$

The same property is preserved by the discretization.
We observe that the setting of the learining rate η is crucial in terms of convergence behaviour: if η is too small the response of the algorithm is overdamped, i.e. the iterative descent is overly slow, while if η is too large the response is underdamped, i.e. the trajectory of $\theta^{(n)}$ follows an oscillatory path and the algorithm risks to become unstable.

2.3.2 Variations to GD method

A valid reference for this section is [19].

Stochastic gradient descent

In simple GD, the loss function J is defined with respect to a set of training data (for example see the cross-entropy loss for the softmax function (2.8)), and so each iteration of the algorithm requires the entire data set to

be processed in order to evaluate ∇J (techniques like this that use the whole data set at once are named *batch* methods).
By contrast, there is an on-line version of gradient descent, known as *stochastic gradient descent*, that evaluates the gradient and updates the parameters using a different subset of the training data, called mini-batch, for each iteration. In this case the loss function results from maximum likelihood for a set of independent observations, and it has the form

$$J(\theta) = \sum_{l=1}^{L} J_l(\theta).$$

The algorithm is formally described by the following formulation:

$$\theta^{(n+1)} = \theta^{(n)} - \eta \nabla J_l(\theta^{(n)}). \tag{2.13}$$

Stochastic GD is useful in practice for working with large data sets, and the update of weights made by using a mini-batch can be interpreted as a noisy estimate of the weights update that would result from using the entire data set.

Stochastic gradient descent with momentum

The stochastic gradient descent algorithm may oscillate along the path of steepest descent reaching the optimum. A way to reduce this oscillation consists in the addition of a new term to the parameter update, called *momentum* term.
The *stochastic gradient descent with momentum* (SGDM) update is

$$\theta^{(n+1)} = \theta^{(n)} - \eta \nabla J_l(\theta^{(n)}) + \gamma(\theta^{(n)} - \theta^{(n-1)}), \tag{2.14}$$

where γ takes account of the contribution of the previous gradient step to the current iteration.

2.4 Artificial Neural Networks

The term Artificial Neural Networks (ANN), more commonly "neural networks", takes its origins from the search for mathematical representations of information processing in biological systems on the basis of the awareness that the human brain computes in a completely different, more complex and faster way from digital computers.
To understand the ANN structure we are going to give a schematic description of the human nervous system (see Figure 2.1):

- the brain, represented by the *neural net*, is the center of the system which receives information, understands it and makes decisions consequently;

- the *receptors* convert stimuli into electrical impulses that transmit information to the neural net;

- the *effectors* convert electrical impulses generated by the neural net into proper responses that represent system outputs.

We observe that left-right transmission is named *forward* transmission, while the transmission in the opposite verse denotes the presence of *feedback* in the system.

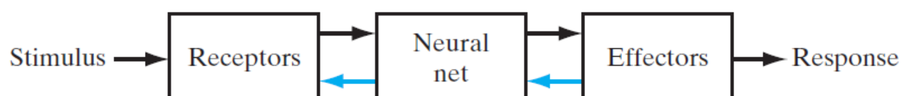

Figure 2.1: Diagram representation of nervous system

The capability of the nervous system to adapt to its surrounding environment, known as *plasticity*, is essential for information processing through brain structural constituents, i.e. neurons. For this reason an ANN is made up of *artificial neurons* and it can be described as a machine designed to model the way in which the brain performs particular tasks.

In summary we can give the following definition of Artificial Neural Network [12]:

> *An artificial neural network is a massively parallel distributed processor made up of simple processing units that has a natural propensity of storing knowledge from experience and make it available for use. It is similar to the brain for two aspects:*
>
> 1. *Knowledge is acquired by the network from its environment by means of a learning process;*
> 2. *Interconnections between neurons, known as synaptic weights, are used to store the acquired knowledge.*

2.4.1 Model of a neuron

In the ANN structure, a neuron functions as an information-processing unit. Now we present a mathematical model of a neuron, which constitutes the basis for the design of a large family of neural networks. The model is made of three basic elements:

1. A set of *synapses*, connecting links each characterized by a *weight*. At the input synapse j connected to the neuron k, the input signal x_j is multiplied by the synaptic weight w_{kj};
2. An operator for *summing* the input signals weighted by the respective synaptic weights of the neuron;
3. An *activation function* for limiting the amplitude range of the neuron output, which is typically set to the interval $[0, 1]$ or alternatively $[-1, 1]$.

An externally applied *bias* term b_k is also included in the model, which determines the increasing or lowering of the activation function input according to its sign. A schematic depiction of the model is given in Figure 2.2.

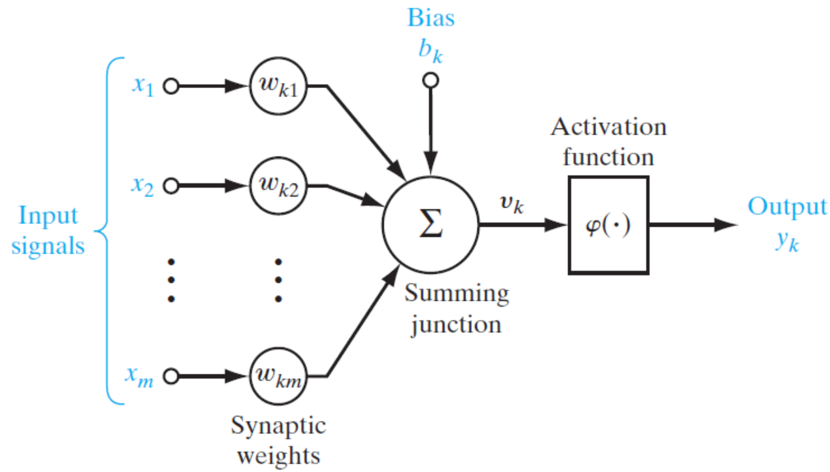

Figure 2.2: Diagram of the model of a neuron, labeled k.

In mathematical terms, the following equations characterize the neuron k:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{2.15}$$

$$v_k = u_k + b_k \tag{2.16}$$

$$y_k = \varphi(v_k) \tag{2.17}$$

where $x_1, x_2, \ldots, x_m$ are the input signals, $w_{k1}, w_{k2}, \ldots, w_{km}$ are the respective synaptic weights, u_k is the linear combiner output from the sum of weighted inputs, b_k is the bias, v_k is the input to the activation function $\varphi(\cdot)$ and y_k is the output of the neuron.

Regarding the activation function φ, we can identify three different types:

1. *Hard limiter function*, given by the formula

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0. \end{cases} \tag{2.18}$$

   Therefore φ sets the output of a neuron to 1 when the input signal exceeds a fixed threshold (that in (2.18) is 0), otherwise it sets the

output to zero. The hard limiter activation function takes discrete values that can be $\{0, 1\}$, as in (2.18), or alternatively $\{-1, 1\}$ and it is not differentiable;

2. *ReLU function*, that stands for rectified linear unit function, given by the formula
$$\varphi(v) = \max(0, v). \tag{2.19}$$
It is a piecewise linear function that will output the input directly if is positive, otherwise it will output zero. Unlike the hard limiter function, the ReLU function takes continous values and it is not differentiable in 0;

3. *Sigmoid function*, a common example of which is the logistic function
$$\varphi(v) = \frac{1}{1 + \exp(-av)}$$
where a is the slope parameter. By contrast to the previous activation functions, the sigmoid activation function can assume a continous range of values between 0 and 1 and it is differentiable in every point.

2.4.2 ANN structure

In general, we can distinguish three different classes of Artificial Neural Network architectures:

- *Single-Layer Feedforward Network* (see Figure 2.3, left). Generally, a *layered* network is made of neurons organized in several layers. The simplest structure of a layered network consists in an *input layer* with no computation performed projected onto an *output layer* of neurons, but not vice versa (this aspect characterizes the network as a *feedforward* one). Such a network is named *single-layer network* referring to the output layer of computation neurons.

- *Multilayer Feedforward Network* (see Figure 2.3, center). This type of network is characterized by the presence of one ore more *hidden layers*

between the input layer and the output layer. Its computation neurons, called *hidden neurons*, have the function to operate between the external input and the network output to extract higher-order information. Typically, the neurons in each hidden layer have as their inputs the output of the preceding layer. Intuitively, the more hidden layers the network has, the more synaptic connections are present, the more accurate the response is due to more connectivity.

- *Recurrent Network* (see Figure 2.3, right). A recurrent neural network is different from a feedforward network for the presence of a *feedback loop*, that is the situation where neurons feed their output back to the input of its own and/or of all other neurons. Therefore, there are connections from a layer to the previous layers but also from a layer to itself. Feedback loops improve ANN performance in terms on learning capability also involving the use of unit-time delay elements (denoted in Figure 2.3 with z^{-1}) which act in a non-linear dynamic way.

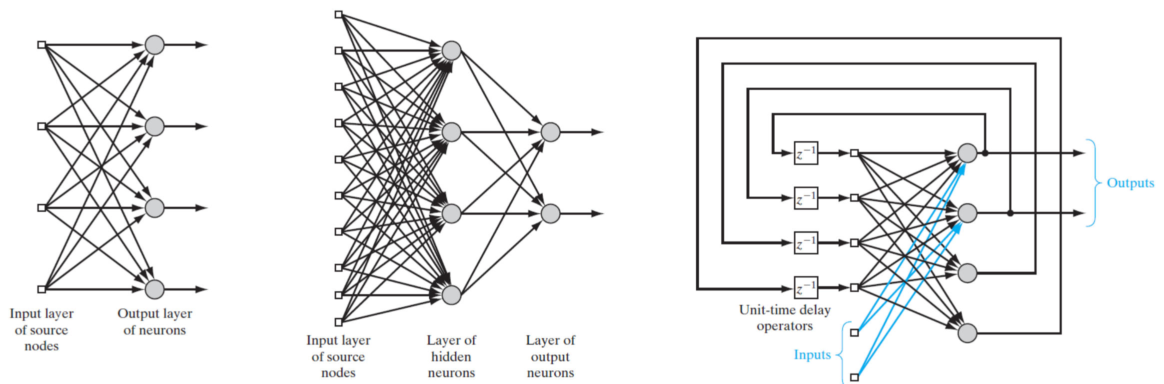

Figure 2.3: Examples of ANN structure diagrams: on the left a single-layer feedforward network, in the center a multilayer feedforward network with one hidden layer of 4 neurons, on the right a recurrent network.

2.4.3 Rosenblatt's Perceptron

The Perceptron is the first neural network described by algorithms, invented in 1958 by the American psychologist Rosenblatt. It was studied to be used in the classification of *linearly separable* patterns, i.e. patterns that live in decision regions separated by a hyperplane.
The Perceptron structure consists of a single-layer feedforward network with one neuron: it computes a combination of weighted inputs to apply to a hard limiter type activation function in order to perform classification with two classes $\mathcal{C}_1$ and $\mathcal{C}_2$. Denoting with $x_1, x_2, \ldots, x_m$ the input signals, $w_1, w_2, \ldots, w_m$ the synaptic weights and b the bias, the input to the hard limiter of the neuron is

$$v = \sum_{i=1}^{m} w_i x_i + b. \tag{2.20}$$

The network assigns in a correct way the set of inputs $x_1, x_2, \ldots, x_m$ to the class $\mathcal{C}_1$ if the hard limiter output is $+1$ and to the class $\mathcal{C}_2$ if the output is -1.
The *separation hyperplane* between the two decision regions in the m-dimensional signal space is defined by the equation

$$\sum_{i=1}^{m} w_i x_i + b = 0, \tag{2.21}$$

and synaptic weights are to be adapted during the *perceptron convergence algorithm* to perform the proper space division.

Before the presentation of the algorithm we point out some aspects concerning the notation:

- We define the n-th input vector as the $(m+1)$-dimensional vector

$$x(n) = [+1, x_1(n), x_2(n), \ldots, x_m(n)]^T;$$

- We define the weight vector at the n-th time step in the algorithm application as the $(m+1)$-dimensional vector

$$w(n) = [b, w_1(n), w_2(n), \ldots, w_m(n)]^T.$$

Observe that n refers to the sample number in the entire data set for the input x, while it denotes the iteration number for w.
In this way the output of the linear combination of weighted input (sum of the bias included) is

$$v(n) = \sum_{i=0}^{m} w_i(n)x_i(n) = w^T(n)x(n) \tag{2.22}$$

where $x_0(n) = +1$ and $w_0(n)$ represents the bias b, and the separation hyperplane between classes $\mathcal{C}_1$ and $\mathcal{C}_2$ is given by the formula

$$w^T x = 0 \tag{2.23}$$

such that the class $\mathcal{C}_1$ corresponds to the subspace $w^T x > 0$ and the class $\mathcal{C}_2$ to the subspace $w^T x \leq 0$.

The Perceptron algorithm

The goal of the Perceptron algorithm is to find a weight vector w such that every input vector belongs to the correct subspace of the m-dimensional input space (i.e. $w^T x > 0$ or $w^T x \leq 0$). The algorithm applies the following correcting rule:

> *If the n-th sample $x(n)$ is correctly classified at iteration n by the current weight vector $w(n)$, there is no update, that is*
>
> $$w(n+1) = w(n) \;\text{ if } w^T(n)x(n) > 0 \text{ and } x(n) \in \text{class } \mathcal{C}_1 \tag{2.24}$$
> $$w(n+1) = w(n) \;\text{ if } w^T(n)x(n) \leq 0 \text{ and } x(n) \in \text{class } \mathcal{C}_2. \tag{2.25}$$
>
> *Otherwise,*
>
> $$w(n+1) = w(n) - \eta(n)x(n) \;\text{ if } w^T(n)x(n) > 0 \text{ and } x(n) \in \text{class } \mathcal{C}_2 \tag{2.26}$$
> $$w(n+1) = w(n) + \eta(n)x(n) \;\text{ if } w^T(n)x(n) \leq 0 \text{ and } x(n) \in \text{class } \mathcal{C}_1 \tag{2.27}$$
>
> *where $\eta(n) > 0$ is the learning-rate parameter.*

In the following part we present a convergence result known as the *fixed-increment perceptron convergence theorem*, that refers to the case of constant learning-rate parameter $0 < \eta \leq 1$:

Theorem 2.4.1 (Fixed-increment Perceptron Covergence). *Let H_1 be the subspace of training vectors that belongs to class $\mathcal{C}_1$ and let H_2 be the the subspace of training vectors that belongs to class $\mathcal{C}_2$. Let H_1 and H_2 be linearly separable. Then the perceptron converges in a finite number of iterations.*

Proof. We prove the Theorem for $\eta = 1$ since a value of $\eta \in (0, 1)$ implies a rescaling so it doesn't intervene in separability.
Consider the initial condition $w(0) = 0$ and suppose that $w^T(n)x(n) \leq 0$ for $n = 1, 2, \ldots$ and the input vector $x(n)$ belongs to the class $\mathcal{C}_1$. We are in the case of (2.27) with $\eta(n) = 1$, therefore $w(n+1) = w(n) + x(n)$ for $x(n)$ belonging to the class $\mathcal{C}_1$. Proceeding iteratively backwards (remembering that $w(0) = 0$) we obtain

$$w(n+1) = x(1) + x(2) + \cdots + x(n). \tag{2.28}$$

Now define α as the positive integer

$$\alpha = \min_{x(n) \in H_1} w_0^T x(n)$$

where w_0 is the weight vector for which $w_0^T x(n) > 0$ for samples $x(1), \ldots, x(n)$ which belongs to class $\mathcal{C}_1$ (w_0 exists since classes $\mathcal{C}_1$ and $\mathcal{C}_2$ are linearly separable for hypothesis). Multiplying (2.28) for w_0^T we have

$$w_0^T w(n+1) = w_0^T x(1) + \cdots + w_0^T x(n) \geq n\alpha,$$

and applying the Cauchy-Schwarz inequality to the squared left hand side of this equation we obtain

$$\|w_0\|^2 \|w(n+1)\|^2 \geq [w_0^T w(n+1)]^2 \geq n^2 \alpha^2. \tag{2.29}$$

Now we deduce another inequality writing the correction rule as $w(k+1) = w(k) + x(k)$ for $k = 1, \ldots, n$ and $x(k)$ belonging to class $\mathcal{C}_1$. Calculating the

Euclidean norm we have

$$\|w(k+1)\|^2 = \|w(k)+x(k)\|^2 = \|w(k)\|^2 + \|x(k)\|^2 + 2w^T(k)x(k)$$
$$\leq \|w(k)\|^2 + \|x(k)\|^2.$$

Then defining β as the positive constant

$$\beta = \max_{x(k)\in H_1} \|x(k)\|^2$$

and summing the last inequality over k, we obtain

$$\|w(n+1)\|^2 \leq \sum_{k=1}^{n} \|x(k)\|^2 \leq n\beta. \tag{2.30}$$

Combining inequalities (2.29) and (2.30) we find out that to have both of them satisfied, n have to be smaller than some value $n_{\max}$ given by

$$\frac{n_{\max}^2 \alpha^2}{\|w_0\|^2} = n_{\max}\beta \Longrightarrow n_{\max} = \frac{\beta\|w_0\|^2}{\alpha^2}.$$

$n_{\max}$ is the maximum number of iterations that the algorithm performs to reach convergence. □

2.4.4 Multilayer Perceptron

In the last section we studied Rosenblatt's Perceptron that works well with two classes of linearly separable data, based on updating synaptic weights of one single neuron. Now we present a neural network structure, called *Multilayer Perceptron*, which overcomes the limitations of the perceptron algorithm including one or more hidden layers that work as *feature detectors*. In this case hidden neurons gradually acquire the most important characteristics of training data computing a non-linear transformation of input samples, which results in a better separation of distinct classes.
The training of Multilayer Perceptron (i.e. the process of modifying synaptic weights properly) can be divided in two phases:

1. the *forward phase* during which, fixed some synaptic weights, the input signal is propagated through the newtork layers to calculate the output. In this phase we can identify the *function* signal type, that is a signal that comes in at the input to come out at the output;

2. the *backward phase*, during which we can identify the *error* signal type that originates at an output neuron by the comparison between the output and the desired response and is propagated backwards through layers. In this phase synaptic weights are modified according to the information given by error signals.

Therefore we note that each neuron of a Multilayer Perceptron perform two types of computations, the first for the function signal that consists in a non-linear combination of weighted inputs, the second for the error signal that involves the derivatives of the network function in order to use gradient descent optimization methods and decide in which direction synaptic weights have to be modified.

The back-propagation algorithm

The *back-propagation algorithm* provides a computationally efficient method for the training of multilayer perceptrons. We are going to present the algorithm in the simplest case of the Mean Squared Error loss function, given by (2.32). Changing the loss function e.g. with a cross-entropy loss, the derivatives involved have to be calculated properly but the steps of the computation remain the same.
To introduce some notations, considering the set $\{(x(n), t(n))\}_{n=1}^{N}$ of training samples, we indicate with $y_j(n)$ the function signal produced as output of neuron j given the input $x(n)$. The corresponding error signal that originates at the output of neuron j is denoted with

$$e_j(n) = t_j(n) - y_j(n) \tag{2.31}$$

where $t_j(n)$ is the j-th component of the target-response vector $t(n)$, and the quantity

$$\mathcal{E}_j(n) = \frac{1}{2} e_j^2(n) \tag{2.32}$$

is defined as the *instantaneous error energy* of neuron j. Consequently, the *total instantaneous error energy* of the network is defined by

$$\mathcal{E}(n) = \sum_{j \in C} \mathcal{E}_j(n) \tag{2.33}$$

where C is the set of all the neurons of the output layer, and considering the entire training set of N samples, we define the *error energy averaged over the training sample* as

$$\mathcal{E}_{\mathrm{av}}(N) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{E}(n) = \frac{1}{2N} \sum_{n=1}^{N} \sum_{j \in C} e_j^2(n). \tag{2.34}$$

As explained in the presentation of the mathematical model of a neuron, we remind that the input to the activation function of hidden neuron j, given the input $x(n)$, is

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{2.35}$$

where m is the total number of inputs and w_{ji} is the synaptic weight that connect neuron j with neuron i (we observe that the addition of the bias b_j is equal to fix an initial input $y_0 = +1$ and the synaptic weight $w_{j0} = b_j$). Therefore, the function signal $y_j(n)$ that is the output of neuron j is

$$y_j(n) = \varphi_j(v_j(n)). \tag{2.36}$$

As a supervised-learning process, multilayer perceptron training is based on *error correction.* The performance measure is a cost function, depending on the synaptic weights of the network, which can be visualized as an *error surface* in the weights space. Every vector of synaptic weights calculated during the training process represents a point on the error surface, and to improve classification performance this point has to move in direction of a minimum point of the error surface. To do this, the optimization method

used is the GD method (see Section 2.3.1) that evolves in the direction of steepest descent of the error function.
In the batch method of learning, the cost function is defined by the averaged error energy $\mathcal{E}_{\text{av}}(N)$ (see (2.34)) and the synaptic weights are updated after processing the entire training set, while in the on-line method (which we are going to see in detail) the cost function is the istantaneous error energy $\mathcal{E}(n)$ (see (2.33)) and weights are corrected sample-by-sample.

In the case of on-line learning, the back-propagation algorithm applies a correction Δw_{ji} to the weight $w_{ji}(n)$ proportional to the partial derivative $\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}$, which indicates the right direction of search for weights in order to minimize the error energy. Applying the chain rule for partial derivatives, we obtain that

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}. \tag{2.37}$$

Partial derivatives that intervene in (2.37) can be calculated as follow:

1. From (2.33) we have
$$\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} = e_j(n); \tag{2.38}$$
2. From (2.31) we have
$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1; \tag{2.39}$$
3. From (2.36) we have
$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi'_j(v_j(n)); \tag{2.40}$$
4. From (2.35) we have
$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n). \tag{2.41}$$

Therefore substituting these results in (2.37) we obtain

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi'_j(v_j(n))y_i(n) \tag{2.42}$$

and the correction $\Delta w_{ji}(n)$ can be defined as

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}, \tag{2.43}$$

where η is the learning-rate parameter of the back-propagation algorithm. Observe that from the negative sign in (2.43) we recognize the direction of steepest descent for the error energy $\mathcal{E}(n)$.
Substituting (2.42) in (2.43) we obtain

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) \tag{2.44}$$

where $\delta_j(n)$ is the *local gradient* of the output neuron j defined by

$$\delta_j(n) = \frac{\partial \mathcal{E}(n)}{\partial v_j(n)} \tag{2.45}$$

$$= \frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} = e_j(n) \varphi_j'(v_j(n)). \tag{2.46}$$

We found that the error signal $e_j(n)$ that originates at the output of neuron j is a term involved in the correction formula. So we have to distinguish two cases: in the first case neuron j is a neuron of the output layer, then its output is comparable with a desired response, while in the second case neuron j is a hidden node, then the corresponding error signal has to be determined recursively in terms of the error signals of all other neurons connected to neuron j.

- *Case 1.* If neuron j is an output node, its output $y_j(n)$ is known as network output, so we can use (2.31) and calculate the local gradient $\delta_j(n)$ with formula (2.46), then the correction $\Delta w_{ji}(n)$ by using (2.44);

- *Case 2.* If neuron j is a hidden node, we have to redefine the local gradient $\delta_j(n)$ for hidden neuron j by the formula

$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \overset{(2.40)}{=} -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \varphi_j'(v_j(n)). \tag{2.47}$$

  To calculate the derivative $\frac{\partial \mathcal{E}(n)}{\partial y_j(n)}$, we first rewrite the error energy as $\mathcal{E}(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n)$ to avoid confusing neuron index (now the index j

refers to a hidden neuron while the index k refers to output neurons), so we have

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_{k \in C} e_k(n) \frac{\partial e_k(n)}{\partial y_j(n)} \stackrel{\text{(chain rule)}}{=} \sum_{k \in C} e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}. \quad (2.48)$$

Now for a hidden neuron results that

$$e_k(n) = t_k(n) - y_k(n) = t_k(n) - \varphi_k(v_k(n)) \Longrightarrow \frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'(v_k(n)), \quad (2.49)$$

and from (2.35) we obtain that

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n). \quad (2.50)$$

Combining last equations in (2.48) we get

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = -\sum_{k \in C} e_k(n) \varphi_k'(v_k(n)) w_{kj}(n) \stackrel{(2.46)}{=} -\sum_{k \in C} \delta_k(n) w_{kj}(n), \quad (2.51)$$

and we finally have from (2.47) that the *back-propagation formula* for the local gradient of hidden neuron j is

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_{k \in C} \delta_k(n) w_{kj}(n). \quad (2.52)$$

The correction Δw_{ji} is then computed using the back-propagation formula in (2.44).

Activation function

During the computation of the correction Δw_{ji} we found that the derivative $\varphi'(\cdot)$ of the activation function intervenes in the local gradient formula, so we need a differentiable activation function. We present two forms of continuously differentiable non-linear activation functions commonly used in multilayer perceptrons, which can be included in the category of Sigmoid activation functions (identified in Section 2.4.1):

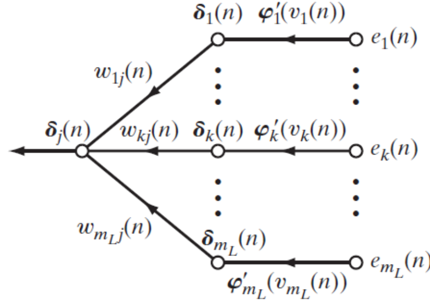

Figure 2.4: Graph to visualize how the back-propagation formula works.

1. *Logistic Function*, already mentioned in Section 2.4.1, defined by

$$\varphi_j(v_j(n)) = \frac{1}{1+\exp(-av_j(n))}, \quad a > 0$$

where a is a positive parameter and $v_j(n)$ has the form (2.35). This activation function generates an output $y_j(n) \in [0,1]$, and its derivative is

$$\varphi_j'(v_j(n)) = \frac{a\exp(-av_j(n))}{[1+\exp(-av_j(n))]^2} \overset{y_j(n)=\varphi(v_j(n))}{=} ay_j(n)\big(1-y_j(n)\big).$$

If neuron j is an output node and $o_j(n)$ is the j-th component of the output vector of the multilayer perceptron, then $y_j(n) = o_j(n)$ and the local gradient is

$$\delta_j(n) = e_j(n)\varphi_j'(v_j(n)) = a\big(t_j(n) - o_j(n)\big)o_j(n)\big(1-o_j(n)\big).$$

If neuron j is a hidden node, the local gradient is

$$\delta_j(n) = \varphi_j'(v_j(n))\sum_k \delta_k(n)w_{kj}(n) = ay_j(n)\big(1-y_j(n)\big)\sum_k \delta_k(n)w_{kj}(n).$$

2. *Hyperbolic tangent function*, defined by

$$\varphi_j(v_j(n)) = a\tanh(bv_j(n))$$

where a, b are positive constants. It generates an output $y_j(n) \in [-1,1]$ and its derivative is

$$\varphi_j'(v_j(n)) = ab\big(1-\tanh^2(bv_j(n)\big) = \frac{b}{a}\big(a+y_j(n)\big)\big(a-y_j(n)\big).$$

If neuron j is an output node, the local gradient is

$$\delta_j(n) = \frac{b}{a}(t_j(n) - o_j(n))\big(a + o_j(n)\big)\big(a - o_j(n)\big).$$

On the other side, if neuron j is a hidden node we have

$$\delta_j(n) = \frac{b}{a}\big(a + y_j(n)\big)\big(a - y_j(n)\big)\sum_k \delta_k(n) w_{kj}(n).$$

Stopping criteria

Generally, the back-propagation algorithm cannot be shown to converge. Therefore there are some stopping criteria for the algorithm that derive from some properties of the optimization problem at the base of the method. Denoting with w^* a local or global minimum of the error surface, we remind that a necessary condition to be a minimum, denoting with $g(w)$ the gradient vector of the cost function, is

$$\nabla g(w^*) = \mathbf{0}.$$

Hence a stopping criteria is formulated as follow ([12, Section 4.4]):

> *The back-propagation algorithm is considered to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold.*

A second stopping criteria is based on the cost function that is stationary at the minimum point $w = w^*$:

> *The back-propagation algorithm is considered to have converged when the absolute rate of change in the error function per iteration is sufficiently small.*

We observe that these stopping criteria are supported by theory, so during computation long learning times as well as early termination of the algorithm may occur.

In conclusion, the back-propagation procedure for on-line learning in multilayer perceptrons can be summarized as follows:

Back-propagation algorithm

---

1. Given a training sample $\{(x(n), t(n))\}_{n=1}^{N}$, fix parameters and thresholds needed and initialize synaptic weights and biases;

2. Apply an input vector to the network: let the training sample be $(x(n), t(n))$, so the input vector is $x(n)$. Then forward propagate layer by layer computing for every neuron j in layer l, for $l = 1, \dots, L$, the activation function input

$$v_j^{(l)}(n) = \sum_i w_{ji}^{(l)}(n) y_i^{(l-1)}(n)$$

where $w_{ji}^{(l)}(n)$ is the synaptic weight from neuron i in layer $l - 1$ to neuron j in layer l and $y_i^{(l-1)}$ is the output signal of neuron i in layer $l - 1$, and then the output signal of neuron j in layer l

$$y_j^{(l)}(n) = \varphi_j(v_j^{(l)}(n)).$$

Finally compute the error signal $e_j(n) = t_j(n) - o_j(n)$.

3. Compute for every output neuron j in layer L the local gradient

$$\delta_j^{(L)}(n) = e_j(n)\varphi_j'(v_j^{(L)}(n))$$

and then backpropagate to obtain the local gradient for each hidden neuron by the formula

$$\delta_j^{(l)}(n) = \varphi_j'(v_j^{(l)}(n)) \sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n)$$

for neuron j in hidden layer l.

4. Modify the synaptic weights according to the correction formula

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \eta \delta_j^{(l)} y_i^{(l-1)}(n)$$

for every l, for every i, j;

5. Verify the chosen stopping criterion: if the stopping condition is not satisfied, back to point 2 with a new training sample.

2.4.5 Convolutional Neural Networks

Convolutional Neural Networks (CNN) represent a particular class of neural networks suitable for pattern classification. Their structure is made up of neurons characterized by learnable weights and biases as well as the multilayer perceptron, but they are specialized for processing data with a known grid-like topology. Examples are time-series data viewed as a 1-D grid of samples taken in different time instants, or image data viewed as a 2-D grid of pixels. So basically CNNs take account of the *arrangement* of input data in time or in space to improve classification.
We can summarize the process of supervised learning in a CNN in the following points:

1. *Feature extraction*: each neuron takes its inputs only from a small subregion in the previous layer known as *local receptive field*. This fact forces the neuron itself to a *local* feature extraction;

2. *Feature mapping*: neuron units in a convolutional layer are organized into plans, each of wich is called *feature map*, and each unit in a feature map is constrained to share the same weight values. In this contest, a set of weights is known as a *filter*. During this phase, each neuron combines its local input with its set of weights by mean of a *convolution* (from which the network takes its name): if $(w_{ij})_{i=1,\dots,F_1, j=i,\dots,F_2}$ is the filter and $(x_{ij})_{i=1,\dots,F_1, j=i,\dots,F_2}$ is the input, the convolution is ([11, Chapter 9, Section 9.1])

$$\sum_{i=1}^{F_1} \sum_{j=1}^{F_2} x_{i+m,j+n} w_{ij}$$

where indeces m, n individuate the local input region;

3. *Subsampling*: each convolutional layer is followed by a computational layer whose task is to refine the outcome of feature mapping. In this layer there is a plane of neuron units for every feature map in the convolutional layer, and each unit takes inputs from a small receptive field in the corresponding feature map to perform subsampling.

In light of this, the positive aspects of using a CNN are the reduction in the number of weights parameters thanks to the sharing and the possibility to take advantage of data evolution in time/space to acquire information from a local research of features (nearby samples will be more strongly correlated than more distant ones).

CNN Layers

As we described above, the structure of a Convolutional Neural Network includes several layers. In detail, these layers are a Convolutional Layer, a Pooling Layer and a Fully-Connected Layer. We present the steps of the computation in the case of image data as inputs since in the next chapter we need samples in an image format to be the input of the built CNN classifier (see Section 3.4).

The *Convolutional Layer* (Conv Layer) does most of the computation. In case of 2-D inputs (i.e. images), its synaptic weights consists in K filters of size $F_1 \times F_2$ (which corresponds to the receptive field size of Conv Layer neurons). Each filter slides across the input and neurons belonging to the current feature map compute the convolution with its input region, that is essentially the dot product between the entries, and then they add the bias.

Three hyperparameters control the size of the output, that are depth, stride and zero-padding:

- the *depth* of the output corresponds to the number of filters we would like to use, i.e. K;

- the *stride* refers to how filters slide across the input. With an image input, if the stride is 1 the filter moves pixel-by-pixel, while if the stride inscreases to S the filters jump S pixels at a time. The larger the stride, the smaller the output spatially (i.e. not considering the depth). We denote with S_1 the vertical stride size and with S_2 the horizontal stride size;

- sometimes it may be convenient to add rows or columns of zeros around the input border, and the *zero-padding* controls this number of rows and columns. We denote with P_1 the padding applied to the top and bottom of the input and with P_2 the padding applied to the left and right.

Therefore, given an input image of size $W_1 \times W_2$ and fixed the number of filters K of dimension $F_1 \times F_2$, the stride $S_1 \times S_2$ and the zero-padding $P_1 \times P_2$, the output size of the Conv Layer is $H_1 \times H_2 \times D$ where

$$H_1 = \frac{W_1 - F_1 + 2P_1}{S_1} + 1 \tag{2.53}$$

$$H_2 = \frac{W_2 - F_2 + 2P_2}{S_2} + 1 \tag{2.54}$$

$$D = K \quad \text{(the depth of the ouput).} \tag{2.55}$$

With filter sharing, the total number of weights is $(F_1 \cdot F_2) \cdot K$ and the number of biases is K (one per filter).

The layer following the Covolutional is a *Pooling layer*, that has the function of a subsampling layer. In fact it reduces the spatial size of the Conv ouput, which becomes its input, and as a consequence the amount of computation in the network. The most common operation to subsample in a Pooling layer is the maximum operation, which behaves similarly to the convolution in a Conv layer: fixed a spatial extension the maximum operation slides across the input and selects its value locally.

Therefore, given an input of size $W_1 \times W_2 \times D$ and fixed a spatial extension $F_1 \times F_2$ and a stride $S_1 \times S_2$, the output size of the Pooling layer is $H_1 \times H_2 \times D$

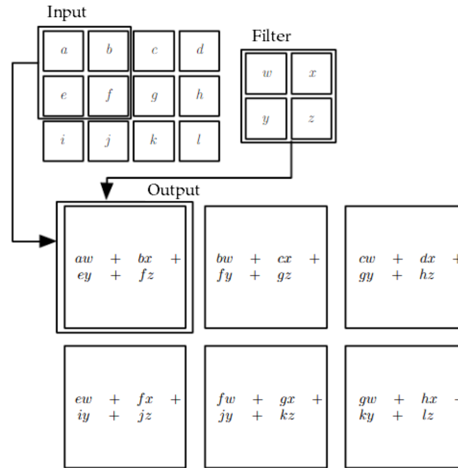

Figure 2.5: Example of convolution of an input of size 3×4 with a 2×2 filter, fixed stride 1×1 and no padding. The following step is the addition of the bias to every output component.

(the depth remains the same) where

$$H_1 = \frac{W_1 - F_1}{S_1} + 1 \tag{2.56}$$

$$H_2 = \frac{W_2 - F_2}{S_2} + 1 \tag{2.57}$$

Going to the example of Figure 2.5, if the output of Conv layer of spatial dimension 2×3 is pooled with spatial extension 1×2 and stride 1×1 we obtain a Pooling output of spatial size 2×2.

Finally, the Pooling layer is followed by a *Fully-connected layer*, every single neuron of which is connected with all the input nodes and compute the "regular" weighted sum described in (2.15),(2.16) as the input to an appropriate activation function φ.

Chapter 3

Experiments and results

In this chapter we present the result of our study based on the attempt to improve myoelectric pattern recognition using different classification methods.

The starting point of the research is the approach used in INAIL to classify EMG signals acquired by amputee patients. The adopted algorithm is the binary Non-linear Logistic Regression (NLR) with a one-vs-all classification: by this way more than one binary classifier is separately trained according to the respective target class (i.e. gesture), and then combined to obtain a multiclass classification. Intuitively, this procedure can be globalized using a unique classifier which works simultaneously on the entire set of gestures. Therefore, our aim is to make the learning process more global and then verify if the new procedure returns acceptable results.

In Section 3.1 we present the EMG data acquisition protocol, specifying the way in which data are collected and gestures to classify. In Section 3.2 the NLR algorithm used in INAIL is shown with focus on how every single part of the process is implemented, while in Section 3.3 and 3.4 we present our new proposals of multiclass classifiers that are a Softmax Classifier built on an Artificial Neural Network and a Convolutional Neural Network. Finally the classification results obtained from NLR and ANNs are compared on the same data and discussed in Section 3.5.

The documentation consulted for Matlab implementation is given by [14] and [15].

3.1 Experiment setup

The data sets of EMG signals used for our classification analysis were collected in INAIL according to the following data acquisition protocol.
The participants in the experiment were twenty people with trans-radial amputation, aged between 18 and 65, free of known muscular and/or neurological diseases. These subjects were already experienced in myoelectric control of prosthetic hands, thus they were trained in this type of recording sessions. The six used EMG sensors, i.e. Ottobock 13E200=50 active sensors whose characteristics were explained in Chapter 1, were placed on the subjects' forearm using a silicone bracelet. The number of the sensors to be used was fixed at six because this is the highest possible number to place into the prosthesis in order to mantain its structural integrity (for the study on sensor reduction see [8]). Sensors were equidistantly placed in the bracelet on the circumference of the stump, about 5 cm below the elbow, and their right position above muscles were identified by manual inspection of the stump. The data was collected using a software built by INAIL on LabView platform able to sample the six sEMG signals at 1 kHz frequency and convert them from analog to digital signals.
During the acquisition, each subject was sitting in front of a PC monitor where one of the following five selected hand gestures was randomly shown:

1. *Rest*, relaxed hand;

2. *Spherical*, hand with all finger closed;

3. *Pinch*, hand with thumb and finger touching;

4. *Platform*, hand completely open and stretched;

5. *Point*, hand with all fingers closed except for the index pointing.

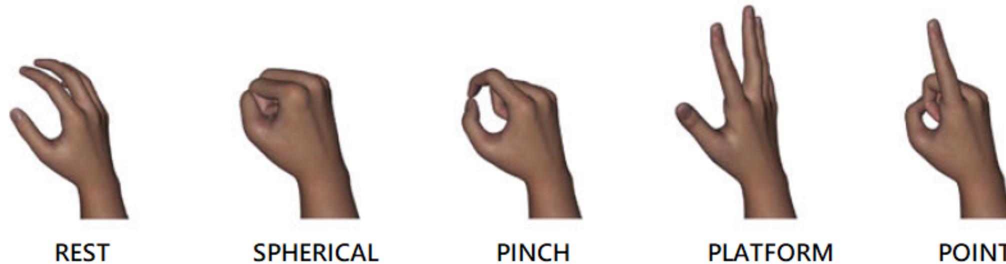

Figure 3.1: The five selected hand gestures

Participants were instructed to reproduce steady state the displayed gesture with their phantom limb. Therefore, fixed a recording time window of 2 seconds, once the signals become stable the sampling session started and continued for 2 seconds obtaining from each sensor 2032 samples. Every single acquisition started from *Rest* position, and after having recorded the current gesture signals the subjects were asked to return to *Rest* position.

During a single recording session each gesture was randomly repeated $nr = 10$ times, then the final data set corresponding to a single subject acquisition consists in a matrix of size $(2032 \cdot nr \cdot ng) \times (1 + ns)$ where $ng = 5$ is the number of gestures and $ns = 6$ is the number of sensors. The first column of the matrix contains the labels of acquired samples, which vary between $1, 2, 3, 4, 5$ that correspond to rest, spherical, pinch, platform and point gestures respectively. The i-th column, for $i = 2, \ldots, 1 + ns$, contains the acquisitions of the entire session collected by sensor $i - 1$. For a better understanding we observe, reading the samples matrix by rows, that every row corresponds to a sample vector made by current gesture label followed by the six sensor acquisition for that gesture at the same time.

After collecting EMG signals, the final step is to transmit samples to the PC to train the classification system.

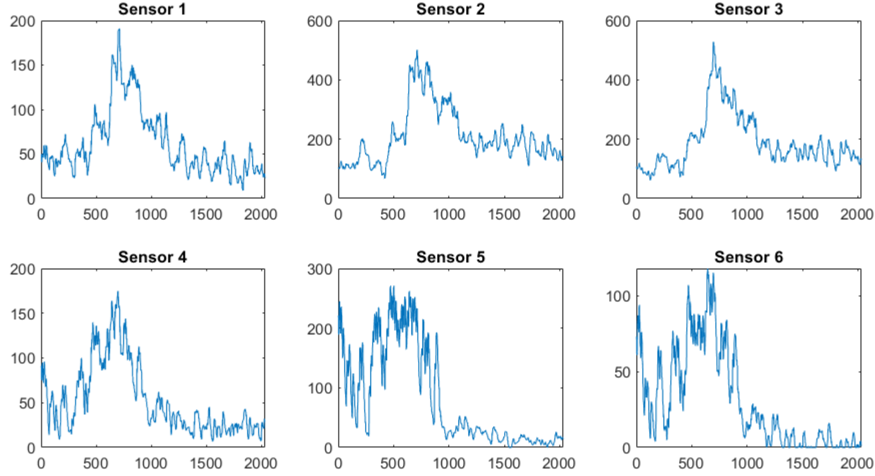

Figure 3.2: Plot of EMG signals acquired by the six different sensors at the same time during the reproduction of a *Pinch* gesture (3).

3.2 Non-linear Logistic Regression classifier

After collecting EMG signals from the twenty amputee patients, the following step is to train the classification system for every subject. The classifier chosen by INAIL to recognize the five selected gestures is a Non-linear Logistic Regression classifier (NLR). In this section we describe in detail the code developed to implement NLR algorithm in Matlab.

3.2.1 Scaling and subsampling

The starting point of the implementation is to identify the input data set, that is the matrix collected during the single subject recording session, exactly a $2032 \cdot 10 \cdot 5 = 101600 \times 7$ matrix. Denoting with $x_1, \ldots, x_6$ our features (i.e. the signal value acquired by sensors $1, \ldots, 6$) and with t the target class/gesture, every matrix has the form

$$
\begin{pmatrix}
t(1) & x_1(1) & x_2(1) & \dots & x_6(1) \\
t(2) & x_1(2) & x_2(2) & \dots & x_6(2) \\
\vdots & & & & \\
t(m) & x_1(m) & x_2(m) & \dots & x_6(m)
\end{pmatrix} \tag{3.1}
$$

where $m = 101600$ is the total number of training samples. In order to obtain a real-time classification with the fastest response, no feature extraction is performed from the acquired signals, so the sEMG signal is used directly as input for the classifier. The only operation made on input signals is the *scaling* per feature, i.e. subtracting the mean to each signal and dividing the result by the range. In terms of code, if we call `TrainingSet` the matrix (3.1), we have

```
for i=2,...,7
      Media(i)=mean(TrainingSet(:,i));
      Range(i)=max(TrainingSet(:,i))-min(TrainingSet(:,i));
      TrainingSet(:,i)=(TrainingSet(:,i)-Media(i))/Range(i);
end
```

In this way we obtain for each time step (i) a six-element vector $x(i)$ of scaled EMG signals, acquired by the six sensors, that belongs to the class $t(i)$. The vector $x(i)$ will be used as input for the classifier.

The next step consists in dividing the entire data set in order to train the classifier and test the result afterwards on different samples.
The division strategy relies on a *three ways data split* approach, i.e. the set of m samples $x(i)$ is divided into three subsets:

1. the *Training Set* (TR), which contains 60% of the data, is used to train the supervised classification algorithm by minimizing the specific cost function;

2. the *Cross Validation Set* (CV), containing 20% of the data, is used to evaluate performance of configurations selected during training in order to find out the best model and avoid overfitting, that is the situation in which the classification parameters set after training fit the training data too well to obtain a good generalization;

3. the *Test Set* (TS), that contains the remaining 20% of the data, is used to evaluate performance in terms of generalization, i.e. how the classifier behaves with respect to input samples different from the training set ones.

In addition to the three ways data split, another set is created before the training. This derives from the fact that for each participant to the recording session we acquire a data set of size 101600×7, which compose a large scale-data set. Consequently, the time needed to complete training and model optimization would be very long using all the samples. Therefore, *downsampling* is applied at the beginning to select the part of the dataset to train, cross-validate and test the classifier. To not lose information, the data not selected by downsampling compose a new set of samples, called *Generalization Set* (GS), that will be used as a second test set to evaluate the classifier generalization ability.

In the code, the downsampling process is controlled by the integer variable `s` that represents the sampling step. For example, if we fix `s` to 5 (one in five) we obtain a GS containing 80% of the data, a TR containing 12% of the data, a CV containing 4% of the data and a TS containing the remaining 4%. Therefore during implementation we have

```
fixed s
for i=1:s:m
    selectedData = [selectedData;TrainingSet(i,:)];
    while i+s does not exceed m
        GS = [GS; TrainingSet(i+1:i+s-1,:)];
    when i+s exceeds m
        GS=[GS;TrainingSet(i+1:m,:)];
end
```
(3.2)

Once the data for training, cross-validating and testing have been selected, the three ways data split can be done on this new dataset (`selectedData` in the code). At first, samples $x(i)$ in `selectedData` matrix (i.e. its rows) are randomly shuffled and divided in TR, CV and TS with a proportion in percentage of 60 - 20 - 20. Then in order to mantain a right proportion of samples belonging to different class/gestures in each of the three subsets, a treshold in percentage is fixed: if the percentage of sample per class in each of the set TR, CV and TS does not exceed the threshold, then samples in `selectedData` have to be reshuffled until the achievement of the equilibrium. Once the TR, CV and TS sets have been filled, the system is ready for the beginning of the leaning process.

3.2.2 Models of features

As explained in Section 2.1, in order to extend the LR classification to the non-linear case, the algorithm by INAIL creates additional features combining the initial ones according to a fixed non-linearity degree.
In our study features are combined in two different ways to create two distinct models:

1. *Exponential model* of degree d, obtained adding to the initial features $x_1, \ldots, x_6$ new features corresponding to the initial ones high till the indicated maximum degree;

2. *Multinomial model* of degree d, obtained adding to the initial features all the monomials of maximum degree d obtained from the multiplication of $x_1, \ldots, x_6$ with their exponentiations.

To give an example, if we set the non-linearity degree d to 2, the features of the exponential model are $x_1, \ldots, x_6, x_1^2, \ldots, x_6^2$ while the features of the multinomial model are $x_1, \ldots, x_6, x_1x_2, x_1x_3, \ldots, x_5x_6, x_1^2, \ldots, x_6^2$.

In the Matlab code, the maximum degree of non-linearity of the model is given by the variable `dmax` fixed at the beginning. To crate the additional features, two auxiliary functions are defined respectively to compute exponential and multinomial features, `mapping_exp(X,d)` and `mapping_multinomial(X,d)`. Each of the function has the matrix of data selected by subsampling and the grade of the model d as inputs, and returns a matrix in which the initial six columns are followed by new columns corresponding to the new additional features generated by multiplying values of columns 1 to 6 as the feature requires.

To determine the optimal model, hence the optimum degree of non-linearity for the classification, all models will be tested during the training of NLR classifier depending on the value of `dmax`, so `mapping_exp` and `mapping_multinomial` works to calculate all features up to `dmax` degree:

```
fixed dmax:
    - consider the selected data without the first column of gesture labels
    and store the labels in a separate vector:
        X1=selectedData(:,2:end);
        t=selectedData(:,1);
    - compute matrix of additional features:
        X2=mapping_exp (X1,2);
        X3=mapping_multinomial(X1,2);
        X4=mapping_exp(X1,3);
        X5=mapping_multinomial(X1,3);
        ⋮
        X*=mapping_exp(X1,dmax);
        X**=mapping_multinomial(X1,dmax);
end
```

(3.3)

The rows of the matrices `X1`,`X2`,... will be the input vectors to the NLR classifier.

3.2.3 Structure of NLR classifier

Once the organization of the dataset is completed, we are ready to describe the tools necessary for the implementation of the NLR classifier.
Here we introduce a Matlab toolbox that provides a framework for designing and implementing Artificial Neural Networks with algorithms and pretrained models, that is the *Deep Learning Toolbox* [14],[15]. Thanks to its functions, it is possible to create and train a customized neural network for regression and classification problems as well as for the recognition of patterns in images. In our case, this toolbox allows to produce a neural network that works in the same way as a NLR classifier, exactly a single-layer feedforward network with the logistic sigmoid function as activation function, whose inputs are vectors of features (i.e. the rows of matrices `X1`,`X2`,... created during

feature mapping) and whose ouput is the estimated class/gesture label of the input.
The starting command in the code is

```
net=network;
```

that returns a network type variable which contains a new neural network with no inputs, layers or outputs. The next step consists in defining the architecture properties, i.e. setting the following values:

- `numInputs` indicates the number of inputs, so its possible values are 0 or a positive integer;

- `numLayers` indicates the number of layers, so its possible values are 0 or a positive integer;

- `biasConnect` is a `numLayers`×1 Boolean vector which contains 1 in position i if layer i has a bias;

- `inputConnect` is a `numLayers`×`numInputs` Boolean matrix which contains 1 in position (i, j) if layer i has a weight coming from input j;

- `outputConnect` is a 1×`numLayers` Boolean vector which contains 1 in position i if the network has an output from layer i.

Therefore, to create the single-layer feedforward network we need, all the five architecture parameters described above are set to 1.
Then function properties have to be defined, that are:

- `transferFcn`, which defines the activation function for each layer used to calculate the layer output during training and simulation;

- `initFcn`, that defines which of the layer initialization functions are used to initialize weights and biases for every layer.

In our case, the transfer function selected for the single layer is the Log-sigmoid transfer function

```
logsig(n)=1/(1+exp(-n))
```

while the chosen initialization function is the Nguyen-Widrow function (for more details see [16]).

In relation to the *training process*, two more function properties have to be set:

- `performFcn`, which defines the function used to measure the network performance during training, that corresponds to the cost function to minimize;
- `trainFcn` which determines the optimization method, i.e. the correction to apply to synaptic weights in order to find out the ones that minimize the loss function set in `performFcn`.

As performance function the selected one is the Mean Squared Error (MSE)

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(t(n) - y(n))^2$$

where N is the number of samples used for training, $t(n)$ is the gesture label of sample n and $y(n)$ is the network output, i.e. the estimated label of sample n, while the optimization method used is the *Resilient Backpropagation* (Rprop). This training method differs from the Gradient Descent method in the fact that the direction of weights update is determined only by the sign of partial derivatives of the cost function with respect to the weights, not by the magnitude. The size of the weight change is determined by a separate update value set a priori (for more details see [15]).

Finally, the last set of values to fix refers to the *division of the data set* according to the three ways data split. They are the following:

- `divideFcn` defines the data division function to be used when the network is trained;
- `divideParam` defines parameters and values of the current data division function. In particular, the three main parameters to set are `trainRatio`, `valRatio` and `testRatio` to estabilish the proportion of data to fill TR, CV and TS sets.

Since in our implementation the organization of the data set has already been done ad hoc by sumbsampling, shuffling and multipling, obtaining as result matrices in (3.3), the data division function is set to `'divideblock'` that considers the totality of samples and makes the division in three blocks mantaining the order in the matrix. Then, according to the 60 - 20 - 20 proportion in percentage for the division in TR, CV and TS, we set

`trainRatio=0.6`, `valRatio=0.2`, `testRatio=0.2`.

In summary, the code to set all the network commands has this form:

```
net=network;
net.numInputs=1; net.numLayers=1;
net.biasConnect=1; net.inputConnect=1; net.outputConnect=1;
net.layers{1}.initFcn='initnw';
net.layers{1}.transferFcn='logsig';
net.performFcn='mse';
net.trainFcn='trainrp';
net.divideFcn='divideblock';
net.divideParam.trainRatio=0.6;
net.divideParam.valRatio=0.2;
net.divideParam.testRatio=0.2;
```

3.2.4 Training process

At this point of the implementation, we remember that the NLR classification algorithm is used in a one-versus-all approach. In our classification

problem where classes/gestures are $ng = 5$, this implies that five different NLR classifiers have to be trained and then combined. To do this, a *for* loop runs through the five classes and computes the training process class-by-class.

Now we consider one iteration of the *for* loop, identifying the gesture to classify with the variable `class`, which can assume values from 1 to 5.
The first step consists in modifying labels of training samples, that are the components of vector `t` obtained in (3.3), to distinguish `class` labels from not-`class` labels:

```
for i=1:length(t)
    if t(i)==class
       t(i)=1;
    else
       t(i)=0;
end
```
(3.4)

In this way we make our data set suitable for a binary classification problem whose aim is to recognize `class` gesture.
Subsequently, in order to select the feature mapping which gives the best classification result, each feature model have to be tested during the training process. We saw that for each feature model we have a different matrix of training samples, then input vectors of different size for the classifier. For this reason we create as many neural networks by the command `network`, with properties explained in the previous part, as feature models are to train each of the network with its respective samples and compare results.

Neural networks are trained using the command `train(net,X,t)`, where the network-type variable `net` contains the structure of the network to train, the matrix `X` contains samples $x(i)$ per row and `t` is the 0-1 vector of gesture labels, and the training options are those set in the network structure. With this command the way in which training is implemented is *batch mode*, i.e.

in each iteration all the inputs in the training set are applied to the network before the weights are updated.
According to the network setting we selected in Section 3.2.3 to build the NLR classifier, the steps performed in each iteration of the training are the following:

1. each sample $x(i)$ belonging to TR set passes trough the net to generate an output, which corresponds to the output of the Logistic Sigmoid activation function;

2. the MSE is computed, then the Rprop algorithm correction, and current weights are updated;

3. with the new set of weights, each sample $x(i)$ belonging to CV set passes through the net to generate the output, then MSE is computed over the CV set. The same step is also computed on TS set;

4. stopping criteria are checked.

In relation to step 4 the criteria used to stop the network training are:

- minimum magnitude of the gradient of performance reached;

- maximum number of validation checks reached, i.e. the number of successive iterations where the validation performance fails to decrease;

- maximum training time reached;

- maximum number of training iterations reached;

- minimum performance value reached.

These limits have their default value, but can also be adjusted by setting the appropriate training parameters.

Once all the feature models have been tested on the corresponding network, we select the best model comparing the network performances.

The command `train` returns the two outputs `[net,tr]`, where `net` is the network object whose structure is the same as the initial network but with the optimum synaptic weight and `tr` is a structure containing the training record. In particular `tr.best_perf`, `tr.best_vperf` and `tr.best_tperf` contain respectively the minimum train/validation/test performance value over iterations.

In this study each of the neural networks crated for the feature model selection is trained for `Ntrain` = 10 times to better generalize the process since each initialization of the network results in different weights and biases and might produce different solutions. The best training session is selected according to the best validation performance: the value of `tr.best_vperf` is compared between all the others, and the best session is chosen in correspondence of the minimum.
The best feature model is selected in the same way with regard to the best validation performance.
Summarizing the training process in terms of Matlab code, we have the following lines:

```
Xmod={X1,X2,...,X*,X**};   % feature models
networks={net1,net2,...,net*,net**};
Nets={}; PerfCV=[ ]; NETS={}; PERFCV=[ ];
for i=1:size(networks,2)
   net=networks{i};
   X=Xmod{i};
   for j=1:Ntrain
      [nettemp,tr]=train(net,X,t);   % t defined in (3.4)
      Nets{j}=nettemp;
      PerfCV(j)=tr.best_vperf;
```

```
        net=init(net);    % the network is re-initialized
    end
    best=find(PerfCV==min(PerfCV));
    NETS{i}=Nets{best};
    PERFCV(i)=PerfCV(best);
end
d=find(PERFCV==min(PERFCV));    % index of the best model in Xmod
```

In this way we find out that for `class` gesture the best model of features is `Xmod{d}` and the best NLR classifier is built on `NETS{d}`. We remember that `NETS{d}` from an input $x(i)$ returns an output $y(i)$ that is the probability that the sample $x(i)$ belongs to the current gesture `class`.

3.2.5 Decision thresholds

The step following the selection of the optimum model of features consists in fixing the *decision thresholds* which determine the predicted gesture label for each sample as a function of the probability output of the NLR classifier. The purpose of INAIL is obtaining a classification that is as certain as possible in the sense that when the system gives a response, the probability that the estimated label is the right label must be high. For this reason the decision thresholds in probability are fixed as high as possible for every gesture compatibly with classification accuracy.
First of all we fix a set of thresholds to test, whose values vary between 0.2 and 0.99. For each of these values denoted with th, `NETS{d}` is tested on the CV set obtained from the division of samples in `Xmod{d}`. From the probability vector output y we obtain the 0-1 vector of estimated labels setting to 1 the components $y(i)$ that satisfy $y(i) \geq th$ and to 0 the others. To evaluate performance in terms of accuracy the $F1score$ is used, defined as follows. If t is the known class vector and y is the estimated class vector, the relative confusion matrix is

	$(t(i) = 1)$	$(t(i) = 0)$
$(y(i) = 1)$	nP	nFP
$(y(i) = 0)$	nFN	nN

(3.5)

where nP is the number of true positive, nN the number of true negative, nFP the number of false positive and nFN the number of false negative. From this matrix we extract

$$PR = \frac{nP}{nP + nFP} \tag{3.6}$$

$$RE = \frac{nP}{nP + nFN} \tag{3.7}$$

where PR is called *Precision* and indicates the ability of the classifier not to label as positive a sample that is negative, while RE is called *Recall* and indicates the ability of the classifier to find all the positive samples. Combining these two quantities we define the $F1score$ in percentage as

$$F1score = 2 \cdot \frac{PR \cdot RE}{PR + RE} \cdot 100. \tag{3.8}$$

The higher the percentage, the more accurate the performance.
Therefore, going back to decision thresholds, the $F1score$ is calculated for the performances obtained from each fixed threshold. The optimal decision threshold will be the one that maximizes the $F1score$.

3.2.6 Network evaluation

Once the best model of feature has been selected and the decision threshold has been chosen, the performance of each of the five resulting NLR classifiers has to be evaluated. This step is done computing the following evaluation parameters on the test set TS:

- *Precision*;

- *Recall*;

- *F1score*;

- confusion matrix.

At the end of networks evaluation, for each of the five NLR classifiers we have obtained:

1. the typology of the best model of features and its number of features, bias feature included;

2. the optimum synaptic weights determined by the training process;

3. the optimal decision threshold value;

4. *Precision*, *Recall*, $F1score$ and confusion matrix calculated on the test set TS.

3.2.7 Test on GS and voting

At this point in the study we have determined the most performing NLR classifier separately for each gesture on the basis of information provided by samples in `selectedData`. The next step consists in combining the binary classifiers to obtain a multiclass classification and testing the performance on the set of samples we called *Generalization Set* (GS), defined in (3.2). Firstly, the labels of GS samples are stored in a separate variable, as we did for `selectedData`, and the bias feature $x_0 = 1$ is added to the data set:

```
t=GS(:,1);
X=GS(:,2:end);
X=[ones(length(t),1),GS];
```

Afterwards the function `HypMulticlass` is specially created for multiclass classification. This function takes as input the matrix of samples `X`, the vector of known gesture labels `t`, the vector `D` whose component `D`(i) identifies the feature model chosen for gesture i for $i = 1, \ldots, 5$, the structure `Theta`

containing the five optimum vectors of synaptic weights and the vector `Th` of the five optimal decision threshold values, and computes for each sample (each row) of `X` the corresponding class value. As output it returns a matrix `Yhyp` in which the first column contains known class labels, the second column contains the estimated class labels and the remaining five columns contain the probabilities in percentage that each input $x(i)$ belongs to the output class.
To do multiclass classification, the function rebuilds for each gesture i the matrix of features `Xf` according to the best model encoded in `D`(i) and computes the networks outputs by the function evaluation

```
1/(1+exp(-Theta{i}*Xf(j,:)'))
```

for $i = 1, \ldots, 5$, $j = 1, \ldots,$ `size(X,1)`. Then for each sample of `X` it compares the ouput probabilities of belonging to class i with the relative decision threshold `Th`(i). In this step there is the possibility of *abstention* in the following sense: considering the sample $x(i)$, if none of the five probabilities exceeds its decision threshold, then the estimated class label is set to zero (i.e. the multiclass classifier does not choose).

Once obtained the matrix `Yhyp`, the procedure known as *voting* is implemented. This step adds a sort of stability to the classification process in view of the online classification to do once the prosthesis will be worn by the patient and used in everyday life.
The key point of voting is that, while the data set `selectedData` was shuffled before training and evaluation, this time samples in the matrix `GS` are placed in order of acquisition during the EMG recording session. In fact it consists in selecting an amplitude `nVote` intended for the voting interval and dividing the first two columns of `Yhyp` (that contains known and estimated labels *in order of acquisition*) in blocks according to this amplitude. Then the estimated class that occurrs the most in each of the voting blocks becomes the estimated class of each single sample of the block.

After voting we obtain a `Yhyp` matrix with the same structure but with the column of estimated class labels that has been modified, and on the basis of these new values the performance of the multiclass classifier is evaluated by computing:

1. the confusion matrix `M`, whose size is 5×5 and contains the number of true/false positives/negatives with respect to each gesture;

2. the $F1score$ value for each gesture i, given by formulas

```
Pr=M(i,i)/sum(M(i,:));
Re=M(i,i)/sum(M(:,i));
F1score=2 · (Pr · Re)/(Pr+Re) · 100;
```

   hence the mean of the five $F1score$ values;

3. the percentage of abstention.

3.3 Multilayer Feedforward ANN

To reach the target of a more global classification, we propose the use of a single multiclass classifier as an alternative to the one-vs-all classification approach. Instead of considering five distinct binary classification problems by the use of the NLR algorithm, this time each sample is considered with its initial label i for $i = 1, \dots, 5$ and it is given as input to the new classifier which returns a vector with five components corresponding to the probabilities that the sample belongs to each class.

To obtain this kind of classification result we build a Softmax Classifier on a Multilayer Feedforward ANN, including a non-linear transformation of input samples. In the same way as with the NLR classifier, we take advantage of the Matlab *Deep Learning Toolbox* functions to select and divide our data properly, create and train our customized ANN and evaluate performance.

In particular, we use the neural network pattern recognition app `nprtool`. As in the previous section, now we describe every passage of the code necessary to implement the multiclass classification following this new approach.

3.3.1 Scaling and subsampling

The initial setting of the problem is the same as described for NLR classification in Section 3.2.1. EMG signals acquired during the recording session are organized in the `TrainingSet` matrix (3.1), and the *scaling* per feature is the unique operation made on signals (no feature extraction is performed). Afterwards, the first data set division is made according to the fixed sampling step `s` to select samples for the GS set, and the remaining data are intended to be divided following the *three ways data split* approach to train and test the classifier.
The code for data division is equal to the one described in (3.2).
We point out that in this new setting, the shuffle of selected data made to obtain a balanced three ways data split is not done in this phase of the implementation. The network parameter `divideFcn` will be set properly to run the shuffle automatically before training.

In order to face up the multiclass problem with a single global classifier, we have to modify labels of samples from $1, 2, 3, 4, 5$ to 0-1 vectors of length 5 such that if the sample label is i, then the new label vector has all 0 elements except for element i that is 1. So the new label is $(1, 0, 0, 0, 0)$ for samples belonging to gesture 1, $(0, 1, 0, 0, 0)$ for those belonging to gesture 2 and so on. A function named `convertT` is specially created to modify labels structure: it takes as input a vector of N labels and returns a $5 \times N$ matrix whose columns are the new 0-1 vectors labels.

3.3.2 Structure of the multiclass classifier

The command used to create the suitable network structure for our multiclass problem is `patternnet`, which set the network parameters to have a two-layer feedforward network with a sigmoid transfer function in the hidden layer and a softmax transfer function in the output layer.
The number of neurons in the hidden layer is set in the integer variable `hiddenLayerSize`, which can be arbitrarily chosen by the user. We observe that the more this number increases, the more synaptic weights will be in the network.
By the command

```
net=patternnet(hiddenLayerSize)
```

the tool creates a network that detects its number of input neurons according to the length of input samples (i.e. the features number) and its number of output neurons according to the number of distinct categories. Each output neuron represents a category, so when an input vector of the appropriate category is applied to the network, the corresponding neuron should produce a 1, and the other neurons should output a 0.
Going back to the description of architecture parameters in Section 3.2.3, for the network variable `net` created with `patternnet` we have the following setting:

- `numInputs`=1, `numLayers`=2;
- `biasConnect`=[1;1] (i.e. both hidden and output layers have a bias);
- `inputConnect`=$[1;0]$ (it indicates the presence of weights going to the first layer from the input layer, and the absence of weights going to the second layer from the input layer);
- `layerConnect`=$[0\ \ 0;1\ \ 0]$ (set when `numLayers` > 1, it indicates the presence of weights only going to the second layer from the first layer);

- `outputConnect`=[0 1] (it indicates the presence of the network output from the second layer).

In terms of function properties of the network we have to distinguish the two layers, whose structures are stored separately and shown by the command `net.layers{i}` for $i = 1, 2$. In light of this we can identify the following set parameters:

- `net.layers{1}.initFcn='initnw'`, `net.layers{2}.initFcn='initnw'` (weights and biases of both layers are initialized according to the Nguyen-Widrow initialization algorithm in the same way as the previous network);

- `net.layers{1}.transferFcn='tansig'` (the hidden layer activation function is the Hyperbolic Tangent Sigmoid function

```
tansig(n)=2/(1+exp(-2*n))-1
```

  whose output range is $[-1, 1]$. It is numerically equivalent to the Hyperbolic Tangent function in the sense that the numerical difference between the ouputs is very small, but in Matlab it runs faster);

- `net.layers{2}.transferFcn='softmax'` (the output layer activation function is the Softmax function defined in (2.6) and wrote in Matlab, given an input vector v, as

```
softmax(v)=exp(v)/sum(exp(v))
```

).

Going to the function properties related to the *training process*, we have:

- `net.performFcn='crossentropy'` (the selected performance function is the cross-entropy loss for the Softmax function defined in (2.8) and wrote in Matlab as

```
ce=-t.*log(y);
perf = sum(ce(:))/numel(ce);
```

where `ce` is the cross-entropy for each pair (y, t) of output-target vectors and `perf` is the aggregate cross-entropy performance normalized with respect to the number of samples).

- `net.trainFcn='trainscg'` (the optimization method used during training is the *scaled conjugate-gradient method* (SCG), which differs from the GD method for the direction in which synaptic weights are updated, known as the *conjugate-direction*. For more details see [12, Chapter 4, Section 4.16]).

Finally, the parameters values referring to the *division of the data set* are the following:

- `net.divideParam.trainRatio`=0.6, `net.divideParam.valRatio`=0.2, `net.divideParam.testRatio`=0.2 (to mantain the 60 - 20 - 20 proportion in percentage for the division in TR, CV and TS);

- `net.divideFcn='dividerand'` (with this setting the sets TR, CV and TS are filled after randomly shuffling the totality of samples in the matrix, in contrast to the `'divideblock'` setting).

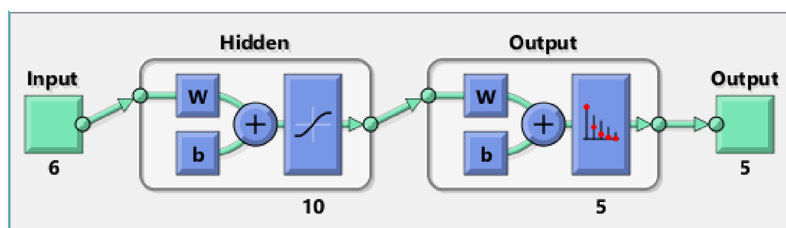

Figure 3.3: Diagram of the two-layer feedforward network created by the command `patternnet(10)`, where 6 is the number of features and 5 the number of classes.

3.3.3 Training process

Once the customized network has been created with `patternnet` and set all the parameters to obtain a Softmax model, the network training starts.
The command used to train is `train(net,X,T)` again, but this time the network-type variable `net` contains the structure of the network given by `patternet` and the training options, `X=selectedData(:,2:end)'` is the matrix of data and contains the six element samples $x(i)$ per columns, while `T` is the matrix obtained by `convertT` whose columns are the respective vector labels of samples.
The training is implemented in a batch mode, and the steps performed in each iteration are the following:

1. each sample $x(i)$ belonging to TR set passes trough the hidden layer of the network to generate an output vector, whose components correspond to the outputs of the Hyperbolic Tangent Sigmoid function for each hidden neuron;

2. the output of the hidden layer becomes the input for the output layer, so it passes through the new layer to generate the network output vector given by the Softmax activation function;

3. the cross-entropy loss is computed, then error signals are backpropagated to calculate weights corrections according to the SCG method and weights are updated;

4. with the new set of weights, each sample $x(i)$ belonging to CV set passes through the net to generate the output, then the cross-entropy is computed over the CV set. The same step is also computed on TS set;

5. stopping criteria are checked, which are the same five described in Section 3.2.4 for NLR.

To better generalize the process of training, the network is trained for `Ntrain` = 10 times on the same data since each initialization of the network results in different weights and biases. As for NLR, the best training session is selected according to the best validation performance among the 10, stored in `tr.best_vperf` for each training.

3.3.4 Test on GS and voting

During the training session of the network, the best setting of synaptic weights has been determined. Consequently, the network can be tested on samples of GS.
Firstly, samples to evaluate the network are separated from their labels always in the same way, setting `Xgs=GS(:,2:end)'`, `tgs=GS(:,1)'` and modifying the structure of labels with the function `convertT` to obtain the matrix `Tgs` with 0-1 vector labels as columns. Then we use an option of Deep Learning toolbox to simulate the network on new samples. If we name `netbest` the network whose structure contains the optimum synaptic weights determined by the training, the command

```
Ygs=netbest(Xgs)
```

takes each column of the matrix `Xgs` (i.e. each sample of GS set), calculates the output of the network whose structure and weights are given by `netbest` and stores the output vectors in the columns of `Ygs`. Therefore the column i of `Ygs` is a five element vector whose components correspond to the probability that the i-th sample in GS belongs to each of the five classes (component j refers to class j for $j = 1, \dots, 5$). To select the highest probability among the five for each sample, i.e. to determine the estimated class, the Matlab command `vec2ind` is used. In general, `vec2ind` takes as input an $N \times M$ matrix V and returns a $1 \times M$ vector of indices indicating the position of the largest element in each column of V. In our case, `vec2ind(Ygs)` returns a vector of indices which correspond to the estimated classes for each sample since the largest element per column identifies the highest probability per

sample.

Once obtained the vector of estimated labels, the procedure of *voting* is implemented in the same way as for NLR testing session (see Section 3.2.4). Samples of GS in order of acquisition are divided in blocks according to the amplitude given by the variable `nVote`, and the estimated class of samples of each block is adjusted on the basis of the most occurring estimated class of the block.

After voting, we obtain the final vector of estimated class labels for GS. Then the performance of the classifier is evaluated by computing the confusion matrix, the $F1Score$ for each class, the mean $F1Score$ given by the mean of the five $F1Score$ values and the percentage of abstention determined by voting.

3.4 CNN

The idea of testing a convolutional network on the EMG data sets comes from the procedure of acquisition of signals and the structure of samples in the initial matrix. Describing the experiment setup in Section 3.1 we said that during the execution of a gesture, the number of samples recorded from each sensor is 2032 in a time window of 2 seconds, and these samples are stored in the data matrix in order of acquisition. Therefore to determine the gesture associated with a single sample, it can be useful to acquire information from samples recorded in a temporal range centered in the instant of acquisition of the sample in hand. Features of signals “close in time” are similar in a good acquisition since signals are recorded during the steady state of the movement, so there isn't any evolution.
As for the previous classifiers, in this section we describe the code implemented in Matlab to build a CNN, which has a different setting with respect to the previous networks.

3.4.1 Reorganization of the data set

As for NLR and Softmax classifiers, the first step consists in the *scaling* per feature of signals in the matrix `TrainingSet`, defined in (3.1). Afterwards, a temporal amplitude is fixed in order to divide scaled samples of `TrainingSet` in different blocks in order to be analyzed by the filters of the CNN and provide “local” information. The blocks can be sequential, i.e. where a block ends the next one starts, or overlapping, i.e. the final samples of a block are in common with the following block, and they are selected so as to contain samples with the same label. The block amplitude is determined by the variable `lblock` fixed at the beginning.
Once the raws of `TrainingSet` matrix have been divided, each of the blocks is saved separately as a MAT-file in a folder named *cnn* on the basis of the label of its samples. The folder *cnn* has five subfolders named 1, 2, 3, 4 and 5, one for each class, and the blocks are stored in the proper subfolder.
The code for the division in sequential blocks and the saving, setting `lblock` = 16, is the following:

```
for i=1:16:size(TrainingSet,1)
   c=TrainingSet(i,1);
   filename=strcat('cnn/',num2str(c),'/sample',num2str(i),'.mat');
   signaldata=TrainingSet(i:i+16-1,2:end);
   save(filename,'signaldata');
end
```

The code shows that each block has a size of `lblock` × 6 and it is saved as *sampleX.mat* where X corresponds to its identification number.

The following step consists in reading the data using the Matlab function `imageDatastore`, which allow to manage the blocks as a collection of image files with the appropriate format to be the inputs of the CNN. With the command

```
location='cnn';
imds=imageDatastore(location,'FileExtensions','.mat', ...
          'IncludeSubfolders',1,'LabelSource','foldernames');
```

the blocks in the folder *cnn* are saved as images labeled in relation to the subfolder in which they are stored. Then they are divided in training data and validation data with a proportion in percentage of 80 - 20 to form the two sets `imdsTrain` and `imdsValidation`.

3.4.2 Structure of the CNN

To define the CNN architecture for our classification problem, we have to specify the layers of the network and set proper parameters for each layer. Firstly we fix some variables about the classification problem, that are

`numFeatures=6`, `numClasses=5`,

then we set the parameters for the convolution with the following variables:

- `filtersize` and `numFilters` indicate respectively the size (height and width) and the quantity of filter used;

- `Padding` indicates the size of padding to apply to input borders vertically and horizontally;

- `poolSize` indicate the dimensions of the pooling regions in height and width;

- `Stride` indicates the step size for traversing the input vertically and horizontally.

Now we can specify the layers of the network with the following command:

```
layers=[ ...
    imageInputLayer([lblock numFeatures 1],'Normalization','none')
    convolution2dLayer(filterSize,numFilters,'Padding',Padding)
    reluLayer
    maxPooling2dLayer(poolSize,'Stride',Stride)
    fullyConnectedLayer(numClasses)
    softmaxLayer
    classificationLayer ];
```

The first layer initializes each block as an image input for the network. The Convolutional layer calculates the convolution between filters and the block regions, then the ReLU layer and the Pooling layer reduce the dimensionality of the Conv layer output. Finally the Fully-connected, the Softmax and the Classification layers adapt the network output to the number of classes and perform classification.

3.4.3 Training process and evaluation

As for the previous classifiers described, training options have to be set before the session.
The optimization method used during training to update synaptic weights is the Stochastic Gradient Descent with Momentum (SGDM), described in Section 2.3.2. For this reason we have to choose the size of the mini-batch and the maximum number of epochs, i.e. the maximum number of full passes through the entire data set during the training session. These two aspects are controlled setting the variables `miniBatchSize` and `maxEpochs`.
Therefore the set of training options is created as follows.

```
options=trainingOptions('sgdm', ...
   'MaxEpochs',maxEpochs, ...
   'MiniBatchSize',miniBatchSize, ...
   'ValidationData',imdsValidation, ...
   'Shuffle','every-epoch', ...
   'Plots','training-progress');
```

The training process starts with the command

`net=trainNetwork(imdsTrain,layers,options)`.

Having specified `'training-progress'` as the `'Plots'` value in `trainingOptions`, the process is monitored thanks to a graphic which displays training metrics at every iteration, where with iteration we intend the step taken in the gradient descent algorithm towards minimizing the cross-entropy loss function using a mini-batch. The training metrics shown are:

- Training accuracy, that is classification accuracy on each individual mini-batch in terms of the right responses given by the network;
- Validation accouracy, that is classification accuracy on the entire validation set with a default frequency of validation of 50 iterations;
- Training loss and validation loss, that are the values of the cross-entropy loss on each mini-batch and on the validation set respectively.

Once training is completed, `trainNetwork` returns the trained network `net`. To evaluate the classification performance after training, the CNN is tested on validation data. The labels of `imdsValidation` samples are predicted using the command

`YPred=classify(net,imdsValidation)`

where `YPred` is the vector of estimated labels, and the final validation accuracy is given by the fraction of labels that the network predicts correctly on `imdsValidation` set.

3.5 Results and discussion

In this section the classification results obtained by using the three classifiers described in Sections 3.2, 3.3, 3.4 are presented. For brevity of notation we will mention the classifiers respectively with the abbreviations NLR, ANN and CNN.
To better understand the results we present step by step the procedure followed to set up the analysis for the first two data sets, but it is the same for each of the twenty data sets.
The parameters taken into account to evaluate performances are the following:

1. *Classification accuracy*, given by the comparison between estimated gesture labels and known gesture labels of EMG samples;

2. *Execution time*, obtained by timing the training process of the classifier from the launch of the code to the generation of the output (the training is performed offline, and this parameter allows to check if the time spent is reasonable);

3. *Memory storage*, encoded in the total number of synaptic weights of the classifier (it is an important aspect since the optimum weights determined during the offline procedure have to be uploaded and stored into the microcontroller embedded in the prosthetic device in order to perform online classification, and the available memory is limited);

4. *Computational burden*, depending on how many and which operations are necessary to do classification after training (similarly to the memory storage, it is crucial for the online classification: if the operations required to recognize signals are too complex, the time required is long and the microcontroller won't give a real time response).

The first step consists in launching the code implemented in INAIL performing the organization of the data set and the research of the best feature

model in order to obtain the best NLR classification performance. The down-sampling step `s` is set to 100 (i.e. samples are divided at first in `selectedData` and `GS` with a proportion in percentage of 1 - 99, then the five binary NLR classifiers are trained with 1% of the entire data set), the maximum degree of non-linearity `dmax` is 3 and the amplitude of blocks `nVote` for the voting procedure is 50. After computing the output, the evaluation parameters such as the $F1Score$ per class, the percentage of abstention and the execution time are stored. Then the competitor classifier ANN is built setting the variable `hiddenLayerSize` according to the total number of synaptic weights resulting as optimum for NLR multiclass classification. The formula which links the number of synaptic weights of ANN (bias included) with the number of neurons in its hidden layer is

$$\#weights = (\texttt{hiddenLayerSize} \cdot 6) + \texttt{hiddenLayerSize} \\ + (5 \cdot \texttt{hiddenLayerSize}) + 5$$

where 6 is the number of features $x_1, \ldots, x_6$ and 5 is the number of gestures. Using this formula we set the number of hidden neurons such that the resulting number of synaptic weights of ANN is the nearest integer to the obtained number of synaptic weights of NLR. In this way none of NLR and ANN has an advantage in terms of memory, and the fact that INAIL code is implemented taking into account the limitations in memory storage guarantees not to exceed in storable parameters. No feature mapping is done before ANN training session since non-linearity is given by the sigmoid activation function, then the inputs for the classifier are six elements vectors. After the creation of the ANN structure, the network is trained with the same set of samples used for NLR training (the same `selectedData` matrix made of 1% of the total) and tested on the same GS.

At the end of the process the same evaluation parameters are saved, that are $F1Score$ per class, abstention percentage and execution time, and compared with the NLR ones. In this first comparison it is important to specify that NLR performs its optimal classification unlike ANN since for ANN initial

features are not combined to obtain additional features and the number of hidden units is constrained by NLR performance.
We report this comparison result on the first two data sets in the following tables:

DATASET 1

s=100, 1016 samples to train	**NLR**	**ANN**
#*weights*	131	125 (# hidden neurons=10)
$F1Score$ class 1 (%)	97.43	95.37
$F1Score$ class 2 (%)	91.58	85.36
$F1Score$ class 3 (%)	85.04	75.72
$F1Score$ class 4 (%)	95.39	92.72
$F1Score$ class 5 (%)	92.26	91.95
$F1Score$ med (%)	92.34	88.23
Abstention (%)	6.07	0.64

Comparison time: 33 s

DATASET 2

s=100, 1016 samples to train	**NLR**	**ANN**
#*weights*	148	149 (# hidden neurons=12)
$F1Score$ class 1 (%)	100	100
$F1Score$ class 2 (%)	96.24	96.97
$F1Score$ class 3 (%)	95.45	96.41
$F1Score$ class 4 (%)	88.88	89.49
$F1Score$ class 5 (%)	86.80	88.38
$F1Score$ med (%)	93.47	94.25
Abstention (%)	0.74	0.34

Comparison time: 31 s

As first we observe that the total procedure of NLR and ANN classification is rather fast since the global time spent is little more than 30 seconds. According to the classification accuracy, while for the second data set

the ANN $F1Score$ for each class is slightly increased compared to the NLR $F1Score$, as also shown by the mean $F1score$, for the first data set the performance of ANN gets worse for each class, with a worsening of more than 4% on average. This result can be justified by the fact that ANN works simultaneously on the five different classes of signals to identify characteristics of each gesture which make it distinguishable from the other four, unlike NLR which works separately on each gesture. Therefore ANN needs a larger number of samples for the training in order to acquire more accurate information.

In light of the previous considerations, a second comparison between NLR and ANN is made modifying the INAIL code setting the downsampling step `s` to 2, i.e. dividing the data set in two blocks with a proportion in percentage of 50 - 50 and training the classifiers with 50% of the entire data set. As in the previous test, the sets TR, CV, TS and GS for NLR and ANN contain the same samples respectively and the number of hidden neurons for ANN is chosen not to have advantages in terms of memory. Making this change in the setting we expect a better accuracy for ANN classification, but we know from the performance analysis done by INAIL on NLR classifier that increasing the percentage of data to be used for training the execution time increases considerably.
The results of the second comparison are reported below for data sets 1 and 2, so we can analyze the evolution with respect to the previous test.

DATASET 1

s=2, 50800 samples to train	**NLR**	**ANN**
#weights	270	269 (# hidden neurons=22)
$F1Score$ class 1 (%)	98.55	98.55
$F1Score$ class 2 (%)	96.75	98.28
$F1Score$ class 3 (%)	93.69	96.46
$F1Score$ class 4 (%)	97.29	98.02
$F1Score$ class 5 (%)	98.76	98.77
$F1Score$ med (%)	97.01	98.02
Abstention (%)	1.96	0
Time	600 s	341 s

DATASET 2

s=2, 50800 samples to train	**NLR**	**ANN**
#weights	235	233 (# hidden neurons=19)
$F1Score$ class 1 (%)	100	100
$F1Score$ class 2 (%)	98.52	99.01
$F1Score$ class 3 (%)	98.52	99.01
$F1Score$ class 4 (%)	93.10	93.88
$F1Score$ class 5 (%)	92.96	93.73
$F1Score$ med (%)	96.62	97.13
Abstention (%)	0.29	0.39
Time	571 s	191 s

Several observations can be made on this new tables with respect to the previous results. The number of synaptic weights and the execution time is increased for both data sets 1 and 2, consistently with the increase of the number of samples to analyze which leads more time to pass throug the data and more memory to store information. Moreover, the mean $F1Score$ for both NLR and ANN is increased, so classification is more accurate if classifiers are trained with more samples. The significant result of this second test is that

for both data sets and for each class the ANN $F1Score$ becomes higher than the NLR $F1Score$, with an execution time that exceedes 10 minutes for NLR with respect to ANN which takes half NLR time for data set 1 and one-third of NLR time for data set 2. Hence with more data to train the classifiers, ANN performs a better classification with respect to NLR spending less time.

Given the evident improvement in ANN classification accuracy (in particular data set 1 shows a 10% increase in mean $F1Score$ from the training with 1% of the data to the training with 50% of the data), a third test is done on ANN classifier using 80% of the data to perform the training of the network. Setting the downsampling step `s` to 5 the entire data set is divided in two subsets with a proportion in percentage of 80 - 20. The `selectedData` matrix is made of the 80% while `GS` is filled with the 20%. According to the hidden layer size of ANN, the number of hidden neurons is set so that the total number of synaptic weights is as near as possible to the optimum number of parameters given by the NLR classification when `s` = 100 (i.e. INAIL setting of downsampling step). With this setting we want to test, using a number of synaptic weights on par with NLR on its best but increasing the number of samples to train the classifier, if the ANN classification accuracy increases further mantaining an accettable execution time, which does not happen training NLR with an ever more number of data. For the voting procedure, we keep the blocks amplitude `nVote` at 50.
Results are shown again for data sets 1 and 2.

DATASET 1

s=5, 81280 samples to train	**ANN**
#weights	125 (# hidden neurons=10)
$F1Score$ class 1 (%)	98.18
$F1Score$ class 2 (%)	99.37
$F1Score$ class 3 (%)	95.59
$F1Score$ class 4 (%)	97.00
$F1Score$ class 5 (%)	98.76
$F1Score$ med (%)	97.78
Abstention (%)	0
Time	283 s

DATASET 2

s=5, 81280 samples to train	**ANN**
#weights	149 (# hidden neurons=12)
$F1Score$ class 1 (%)	99.38
$F1Score$ class 2 (%)	99.37
$F1Score$ class 3 (%)	100
$F1Score$ class 4 (%)	97.04
$F1Score$ class 5 (%)	96.81
$F1Score$ med (%)	98.52
Abstention (%)	0.24
Time	211 s

What is evident comparing these new results with tables given by the first test is that the mean $F1Score$ training ANN with the 80% of data has improved with respect to NLR and ANN peformances with the 1% of training data both for data set 1 and 2 mantaining the execution time under five minutes. For data set 2 the classification accuracy had already improved slightly with $\texttt{s} = 100$ from NLR ($F1Score$ med equal to 93.47%) to ANN ($F1Score$ med equal to 94.25%), and now with $\texttt{s} = 5$ ANN gets still better. For data

set 1 after the performance worsening with $\texttt{s} = 100$ (from a mean $F1Score$ of 92.34% with NLR to a mean $F1Score$ of 88.23% with ANN) now ANN shows an improvement reaching the 97,78% of mean $F1score$.
It is interesting the evolution of the $F1Score$ of class 3 for data set 1: while with the 1% of training data it passes from 85% with NLR to 75% with ANN, increasing the number of training samples it reaches the 95% with ANN. This case shows how crucial is the amount of available information during the training session of a multiclass classifier and how much it can affect the recognition of each gesture pattern.

After testing ANN classification performances it is the turn of CNN. Our aim is to take advantage of the temporal order of samples given by the acquisition procedure to improve classification. The parameters on the network are set as follows: $\texttt{lblock} = 16$, $\texttt{numFilters} = 10$, $\texttt{filterSize} = [5\ 6]$, $\texttt{Padding} = [2\ 0]$, $\texttt{MaxPooling} = [4\ 1]$, $\texttt{Stride} = [1\ 1]$, $\texttt{miniBatchSize} = 254$, $\texttt{MaxEpochs} = 50$. Considering overlapping blocks with a proportion of 80% to train and 20% to validate, from each data set results 8080 training blocks and 2020 validating blocks. The CNN is trained for five times to better generalize the performance due to the fact that initial weights and biases change for every training and samples to fill each individual mini-batch are different per epoch. The classification results are given in terms of validation accuracy and total number of weights, get by the formula

$$\begin{aligned}\#weights = {} & \texttt{filterSize}(1) \cdot \texttt{filterSize}(2) \cdot \texttt{numFilters} + \texttt{numFilters} \\ & + 5 \cdot (H_1 \cdot \texttt{numFilters}) + 5\end{aligned}$$

where H_1 is the first spatial dimension of the Pooling layer output, calculated in (2.56), and 5 is the number of gestures.
The results of CNN from data sets 1 and 2 are reported in the following tables.

CNN: DATASET 1 – 2

DATASET 1	*Validation accuracy*
Training 1	88.56 %
Training 2	89.36 %
Training 3	88.47 %
Training 4	89.60 %
Training 5	90.25 %

weights = 965

DATASET 2	*Validation accuracy*
Training 1	91.88 %
Training 2	92.67 %
Training 3	92.72 %
Training 4	92.62 %
Training 5	92.43 %

weights = 965

With respect to the performance shown by ANN when $\texttt{s} = 5$, in this test the accuracy for both data set 1 and 2 is lower in spite of the considerable number of synaptic weights (which is seven times greater than ANN). A possible explanation of this fact is that in our experiment we are recognizing gesture types from EMG signals acquired during the steady state of the movement. Therefore signals acquired by EMG sensors are constant during the acquisition time window, although their values are different depending on muscles activated by the gesture. For this reason the filters of CNN, which pass through the blocks to catch signal variations in order to distinguish gestures, do not capture significant changes then the classification do not improve as expected. In addition, we remember that the operations to do during the simulation of the network have to be reproduced to perform online classification and give a real time response, and with the convolution this is not the case due to the complexity of execution.
For all these considerations we focus our study on the improvement in classification combined with limited execution time given by ANN.

To verify that the improvement in accuracy and time obtained using ANN trained with 80% of the samples is consistent, the comparison between NLR and ANN when $\texttt{s} = 100$ followed by the training of ANN setting $\texttt{s}$ to 5 is done on the other 18 data sets (i.e. the first and the third test did on data sets 1 and 2 with the same setting), from data set 3 to data set 20.

We report below the tables with the results relative to each data set in terms of number of weights, mean $F1score$, percentage of abstention and execution time.

DATASET 3	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	162	85.85	1.09	30 s
ANN, s=100	161 (#*hn*=13)	81.35	0.50	
ANN, s=5	161 (#*hn*=13)	92.02	0.49	192 s

DATASET 4	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	157	98.59	0.60	31 s
ANN, s=100	161 (#*hn*=13)	98.15	0.10	
ANN, s=5	161 (#*hn*=13)	99.76	0	202 s

DATASET 5	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	197	86.84	2.69	38 s
ANN, s=100	197 (#*hn*=16)	84.47	0.80	
ANN, s=5	197 (#*hn*=16)	92.90	0.98	334 s

DATASET 6	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	183	94.40	2.74	38 s
ANN, s=100	185 (#*hn*=15)	91.80	0.20	
ANN, s=5	185 (#*hn*=15)	98.52	0	255 s

DATASET 7	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	244	99.10	0.94	35 s
ANN, s=100	245 (#*hn*=20)	98.31	0	
ANN, s=5	245 (#*hn*=20)	99.75	0	219 s

DATASET 8	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	177	93.15	1.34	33 s
ANN, s=100	173 (#*hn*=14)	92.24	0.30	
ANN, s=5	173 (#*hn*=14)	95.27	0	170 s

DATASET 9	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	218	84.74	4.23	33 s
ANN, s=100	221 (#*hn*=18)	84.05	1.29	
ANN, s=5	221 (#*hn*=18)	94.29	0.49	243 s

DATASET 10	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	244	97.87	1.59	35 s
ANN, s=100	245 (#*hn*=20)	98.15	0.20	
ANN, s=5	245 (#*hn*=20)	99.51	0	325 s

DATASET 11	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	188	98.46	3.18	34 s
ANN, s=100	185 (#*hn*=15)	97.10	0.25	
ANN, s=5	185 (#*hn*=15)	99.50	0	216 s

DATASET 12	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	157	82.00	5.92	41 s
ANN, s=100	161 (#*hn*=13)	77.27	0.80	
ANN, s=5	161 (#*hn*=13)	89.10	1.72	412 s

DATASET 13	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	244	87.18	4.68	41 s
ANN, s=100	245 (#*hn*=20)	84.90	1.04	
ANN, s=5	245 (#*hn*=20)	95.57	0.24	734 s

DATASET 14	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	183	98.64	0.65	44 s
ANN, s=100	185 (#*hn*=15)	98.75	0.10	
ANN, s=5	185 (#*hn*=15)	100	0	341 s

DATASET 15	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	116	75.42	3.88	40 s
ANN, s=100	113 (#*hn*=9)	74.53	0.50	
ANN, s=5	113 (#*hn*=9)	82.39	1.47	261 s

DATASET 16	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	166	90.11	2.59	41 s
ANN, s=100	161 (#*hn*=13)	89.46	0.65	
ANN, s=5	161 (#*hn*=13)	94.82	0	258 s

DATASET 17	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	194	79.89	10.14	35 s
ANN, s=100	197 (#*hn*=16)	75.00	0.45	
ANN, s=5	197 (#*hn*=16)	87.99	0.74	402 s

DATASET 18	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	157	88.55	1.59	40 s
ANN, s=100	161 (#*hn*=13)	88.43	0.44	
ANN, s=5	161 (#*hn*=13)	96.32	0	437 s

DATASET 19	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	270	75.69	10.85	42 s
ANN, s=100	269 (#*hn*=22)	71.12	0.84	
ANN, s=5	269 (#*hn*=22)	90.99	1.23	612 s

DATASET 20	*#weights*	*F1Score (%)*	*Abstention (%)*	*Time*
NLR, s=100	235	97.37	1.29	30 s
ANN, s=100	233 (#*hn*=19)	96.00	0.20	
ANN, s=5	233 (#*hn*=19)	98.77	0	302 s

To get an overview of the results, a statistic is done considering also performances obtained by data sets 1 and 2.
For every data set from 1 to 20 we obtain an improvement in classification accuracy from the best NLR performance with $\mathtt{s} = 100$ to the ANN performance with $\mathtt{s} = 5$ and a reduction of the abstention at the same time. The mean $F1Score$ from NLR to ANN (with $\mathtt{s} = 5$) increases by 5.20% on average, from a minimum improvement of 0.65% given by data set 7 to a maximum improvement of 15.3% given by data set 19, with a percentage of abstention that decreases by 2,96% on average, from a minimum decrease of 0.5% to a maximum decrease of 9.62% (in 11 of the 20 data sets the abstention with ANN reaches 0%).
The number of synaptic weights varies from 113 (corrsponding to 9 hidden neurons for ANN) to 269 (corresponding to 22 hidden neurons for ANN).
The time spent on average by ANN for the classification process with 80% of data to train is 320 seconds (i.e. a little over 5 minutes), from a minimum time of 192 seconds to a maximum of 734 seconds.

Going into details of new results, we observe that the ones obtained by each of the data sets from 3 to 20 have the same trend as the first two data sets previously studied. Given the same memory storage (i.e. the same number of weights), training the classifiers with 1% of the data returns a worse performance of ANN with respect to NLR. The only exception is given by data sets 10 and 14 for which ANN already works better than NLR, as we observed for data set 2. About these cases we can say that they have in common high $F1Score$ values even exceeding 98%, and these numbers suggest that samples belonging to each gesture are so well distinguishable that the 1% of the data to train the classifier is enough to acquire information.

Increasing the number of training data to 80% of the total, ANN fills the previous gap giving a very positive response for every dataset from 3 to 20 mantaining a time significantly short, result not achievable by NLR.
An example that properly shows all the mentioned improvements is given by data set 19. The classification with 1% of the data to train returns 75.69% of $F1Score$ with NLR versus 71.12% with ANN per 270 synaptic weights, the highest number recorded in the study. It is evident that ANN does not take advantage of the available memory. In fact increasing the quantity of sample for training ANN reaches a 90.99% of $F1Score$ in 612 seconds, which is one of the longest execution times but accetable for an improvement of more than 15%. We also point out that from an abstension of 10.85% with NLR we obtain 1.23% with ANN, so overall ANN respondes more times and giving the right label.

To complete the study, a final test is done on ANN in order to try to further optimize classification performances.
As explained during the description of the analysis set up, we set the number of hidden neurons of ANN on the basis of the optimum number of synaptic weights given by the optimization procedure implemented by INAIL for NLR. This fact represents a constraint on ANN performance. For this reason, on the basis of $F1Score$ values obtained in the last test, the data sets with more margin of improvement are selected among the 20, that are 3, 5, 12, 15, 17, 19. For each of the six selected data sets the test is the following. ANN classifier is built varying `hiddenLayerSize` value from 9 to 22 (that is the maximum to not exceed in memory), and for each value the classification process starts training the network with 80% of the samples and evaluating it with the remaining 20%. At the end $F1Score$ values corresponding to each layer size are compared to verify if from the best performance we have any improvement with respect to results reported in the previous tables.
The best performances for each selected data set are the following:

Dataset 3) $F1Score$ of 92.15% with 15 neurons;

Dataset 5) $F1Score$ of 93.48% with 19 neurons;
Dataset 12) $F1Score$ of 89.72% with 22 neurons;
Dataset 15) $F1Score$ of 83.20% with 16 neurons;
Dataset 17) $F1Score$ of 86.85% with 20 neurons;
Dataset 19) $F1Score$ of 89.18% with 22 neurons;

Making the comparison with the tables corresponding to these data sets, we can conclude that this test does not show significant improvements in classification accuracy since data sets 3, 5, 12 and 15 shows an increase of just 1% while data sets 17 and 19 give a slightly lower value of $F1Score$ (this is due to the arbitrariness of initial weights and biases which lead to different classification results for every training).

The final consideration is according to the computational burden of operations necessary for the ANN classification, which is definitely comparable with the NLR one. For each sample two matrix-vector multiplications are computed, one per ANN layer, and we have two function evaluations that are the hyperbolic tangent sigmoid function and the softmax function. They require the use of the exponential function such as NLR classification with the logistic function, therefore there are not complexity problems.
Further studies on the execution time of online ANN classification are in progress in INAIL in order to have an exact measurement and compare it with the NLR performance online.

3.6 Conclusions

In this final chapter we explained how NLR, ANN and CNN classification algorithms are implemented in a Matlab code, then the comparative analysis was carried out testing and evaluating performances on 20 data sets of 5 hand gestures classes composed of the samples recorded from 20 people with trans-radial amputation, using 6 sEMG sensors. We point out that each of the 20 data sets presented different characteristics depending on the ability

of the amputee patients, but the results obtained by the analysis showed the same trend. Our aim was to globalize classification process with respect to the NLR model presented by INAIL which follows a one-vs-all approach, and multilayer neural networks represented a valid alternative working simultaneously on samples of the 5 gesture classes. While CNN presented a stucture too complex for the embedded system of the prosthetic device, ANN was more consistent with the requirements of the study that were memory storage, computational burden, execution time and classification accuracy. In fact for each data set, with the same classification memory, the performance of ANN with a training set made of 80% of the totality of samples showed a significant improvement with respect to NLR performance including the optimization of the model, with an execution time that remained limited for ANN increasing the number of training samples to acquire more information, which is not the case for NLR. Finally, the decrease in the number of abstentions combined with the improvement in classification accuracy of ANN showed a more frequent response of the classifier with the correct label of the input sample.

Bibliography

[1] Ambrosetti A., Malchiodi A., *Nonlinear Analysis and Semilinear Elliptic Problems*, Cambridge University Press, 2007.

[2] Benatti S., Farella E., Gruppioni E., Benini L., *Analysis of Robust Implementation of an EMG Pattern Recognition Based Control*, 2014

[3] Benatti S., Milosevic B., Farella E., Gruppioni E., Benini L., *A Prosthetic Hand Body Area Controller Based on Efficient Pattern Recognition Control Strategies*, Sensors, 2017

[4] Bishop C. M., *Pattern Recognition and Machine Learning*, Springer, 2006

[5] Bonacorsi D., Lecture notes of *Applied Machine Learning* course, University of Bologna, a.y. 2018/2019

[6] Casella G., Berger R. L., *Statistical Inference*, Second Edition, Duxbury, 2001, Chapter 12, Section 12.3

[7] Castellini C., Gruppioni E., Davalli A., Sandini G., *Fine detection of grasp force and posture by amputees via surface electromyography*, Journal of Physiology - Paris, 2009

[8] Costantini G., Saggio G., Quitadamo L., Casali D., Gruppioni E., *Sensor reduction on EMG-based hand gesture classification*, 2014

[9] Dellacasa Bellingegni A., *Pattern Recognition Systems for Myoelectric Control of Biomechatronic Prosthetic Hands*, 2017

[10] Dellacasa Bellingegni A., Emanuele Gruppioni E., Colazzo G., Davalli A., Sacchetti R., Guglielmelli E., Zollo L., *NLR, MLP, SVM, and LDA: a comparative analysis on EMG data from people with trans-radial amputation*, Journal of NeuroEngineering and Rehabilitation, 2017

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

[12] Haykin S., *Neural Networks and Learning Machines*, Third Edition, Pearson, 2008

[13] Johnson J., Karpathy A., Lecture notes of *Convolutional Neural Networks for Visual Recognition* course, Stanford University, `http://cs231n.github.io/`

[14] Mathworks, *Deep Learning Toolbox Getting Started Guide*, `http://www.mathworks.com`

[15] Mathworks, *Deep Learning Toolbox User's Guide*, `http://www.mathworks.com`

[16] Pavelka A., Procházka A., *Algorithms for the initialization of neural network weights*, 2004

[17] Roelants P., Notes on Machine Learning: *Logistic classification with cross-entropy*, `http://peterroelants.github.io/posts/cross-entropy-logistic/`

[18] Roelants P., Notes on Machine Learning: *Softmax classification with cross-entropy*, `http://peterroelants.github.io/posts/cross-entropy-softmax/`

[19] Ruder S., *An overview of gradient descent optimization algorithms*, 2016.

Ringraziamenti

Arrivato il fatidico e tanto atteso momento della fine del mio percorso di studi, è ora di dire Grazie a chi in questi anni ha contribuito al raggiungimento di questo traguardo.

Per iniziare ringrazio la Professoressa Citti per avermi dato la possibilità di svolgere questo lavoro e avermi seguita passo passo durante questi mesi. Allo stesso modo sento di dover ringraziare Davide e Noemi, Davide per essere stato sempre presente anche da “lontano”, e Noemi per non essersi mai tirata indietro nonostante gli impegni dovuti all’imminente conclusione anche del suo percorso. Ringrazio poi Emanuele per avermi avvicinata a un mondo che non conoscevo, quello del controllo protesico, poichè proprio durante il mio tirocinio ho avuto la possibilità di applicare le mie conoscenze matematiche a qualcosa di reale per avere poi un riscontro nel quotidiano.

Il Grazie più grande va però a chi mi è stato vicino in questi anni e mi ha supportato nei momenti più bui.
A tutta la mia famiglia, che oltre a supportarmi deve da una vita sopportarmi, dico Grazie per avermi sempre incoraggiata e sostenuta.
Grazie a mio padre per tutti i suoi preziosi insegnamenti, che mi portano a riflettere sui miei errori e a rivedere le mie posizioni. So che a lui potrà non sembrare, ma custodisco ognuno dei suoi consigli perchè per me sono i più importanti. Allo stesso modo Grazie a mia madre, a volte ci basta un’occhiata per capirci e ormai ho imparato a interpretare i suoi silenzi e i

suoi sguardi, e questo legame per me significa tanto. Quindi babbo e mamma Grazie per tutte le dritte che mi date, per le parole di conforto sempre pronte e per essere sempre così premurosi.
A mio fratello, che mi fa sciogliere quando osserva in silenzio i miei sproloqui e se la ride sotto i baffi, dico Grazie perchè la determinazione e la caparbietà che lo contraddistinguono sono da esempio per me. Sappi Danilo che se hai bisogno di una spalla io ci sono sempre.
A Lorenzo, perchè per me ormai è di famiglia, dico Grazie per essere un fidanzato, amico, confidente, una spalla modello. So che per qualunque cosa lui c'è, come c'è sempre stato, è il mio punto di riferimento e per me averlo vicino è importante. Un Grazie immenso perchè, nonostante tu abbia conosciuto anche la me peggiore, non te ne sei mai andato.

Un grande Grazie va a tutti i miei amici per avermi fatto passare i migliori momenti. Una menzione speciale va a Bea e Luci, le mie socie, perchè anche se spesso lontane non ci siamo mai perse, e so che posso contare su di loro. Non posso poi dimenticarmi di Villa Begatto storica con Gaia, Giulia e Marta. Con voi questo percorso è iniziato e ci tengo a ringraziarvi per quello che è stato, per esserci fatte forza a vicenda e per avermi fatto sentire un pò più a casa a Bologna.

Il supporto di tutti Voi per me ha fatto e farà la differenza, quindi ringrazio tutti ancora una volta per aver contribuito, ognuno a suo modo, al raggiungimento di questo traguardo. Siete speciali!