Machine Learning Engineer Nanodegree
Capstone Project

Jonatan Dellagostin
July 6th, 2018

I. Definition

Project Overview

Surface electromyography (sEMG) is the temporal and spatial superposition of faint bioelectrical signals generated by the muscle nerve cells during muscle contraction [1]. It is collected and recorded through skin surface electrodes. Compared to conventional EMG signal acquisition that requires inserting a needle electrode into muscle tissue, the sEMG signal has the advantages of being noninvasive and providing the convenience of collection [2]. Besides, it is known [3] that information extracted from intramuscular MES (myoelectric signal) results in the same classification accuracy compared to information extracted from surface MES.

Knowing the daily life of hand amputees can be extremely difficult compared to what it was before the amputation, the research involving hand prosthetics is of great importance. The development of open source state-of-the-art algorithms in hand movement classification is crucial in this field, parallel with the development of open-source easy-to-print 3d robotic hands, both contributing to the development of cheaper and more accessible prosthesis to amputees, since commercial devices are too expensive and unaffordable for most of them.

Problem Statement

Amputees that lack some part of the arm, specially the hand, suffer from not being able to perform simple and crucial task in everyday life. A robotic hand is a proposed solution in the field of prosthetics. In a real hand, electrical impulses sent by the brain travel through the nerves until reaching the arm muscles innervations, which are then activated and produces the muscular activity responsible for the hand movements.

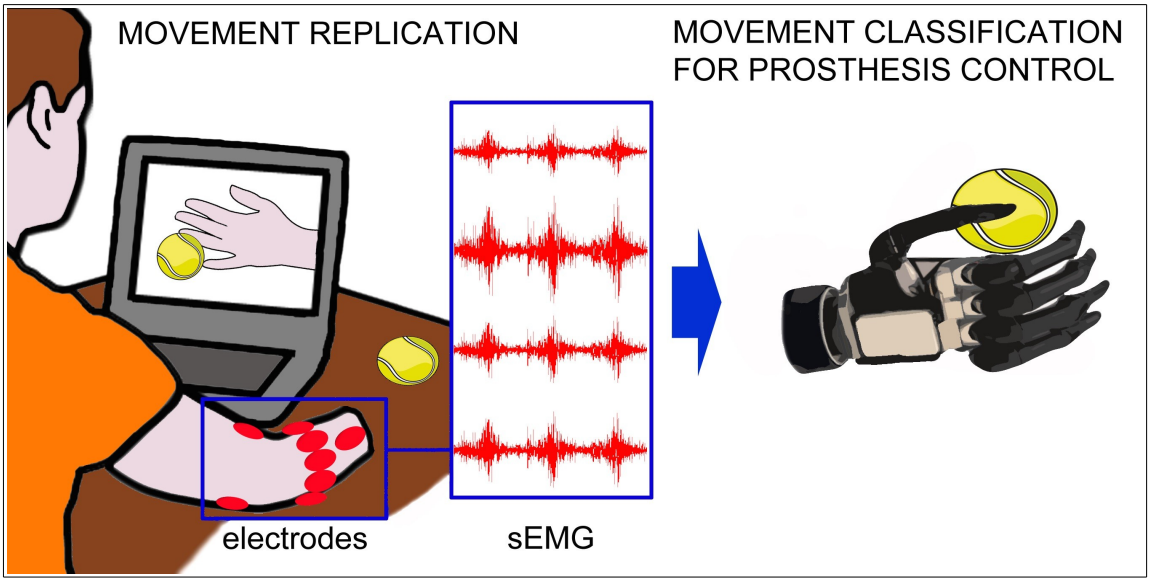

Figure 1: Illustration of prosthesis training and operation (source: http://ninapro.hevs.ch)

A common and established approach in the literature to mimic this behavior is to capture the arm muscle's electric signals through sEMG techniques, preprocess them in a computer, apply some classification algorithm to map these signals into a class from a pre-determined finite set of hand motions, and send to the robotic hand the appropriate electrical signals in order to perform the required movement. Our problem then, in the present work, comprises of finding and training an appropriate classifier for the task of predicting the output class (hand movement) given the input data collected (preprocessed sequences of electrical signals converted to the digital world).

Metrics

For the datasets used in this project, the classes are balanced by definition (the acquisition phase took the same number of examples for each class), the classification accuracy (number of correct classifications/ number of trials) seems a reasonable metric to be used when comparing the performance of different algorithms (in fact, this is the usual metric found in the literature).

Nevertheless, some other metrics will be reported along, like the TA/CL (test accuracy / chance level (classification accuracy / number of classes)) and the training and classification average times (given a certain CPU or GPU used). Since the model training could be done offline (after the data acquisition phase), but the classification couldn't, the classification average time is of great importance: a delay of more than 300 ms between the muscular activation an the prosthesis response can be noted by the patient and degrade the usability of the device.

II. Analysis

Data Exploration

A research into the field led to the first successful initiative to establish an open sEMG database, the NinaPro - Non-Invasive Adaptive Hand Prosthetics. The goal of this project is to develop a family of algorithms able to significantly augment the dexterity, and reduce the training time, for sEMG controlled prosthesis. The project is thoroughly described in [4].

By testing researchers findings on a very large collection of data, this project pave the way for a new generation of prosthetic hands. The Data Acquisition is done with the goal of developing a reproducible protocol to acquire large data sets for healthy patients performing certain movements and amputated patients also making complex movements, while analyzing and assessing the data as they become available. The data acquisition includes collecting signal data calibrating the sensors to limit the noise in data. Relevant clinical data is acquired at the same time such as age, gender, height, weight and for amputated patients also the exact place of the amputation and the time between amputation and tests performed.

From its beginning (2014) until the present day, the NinaPro has collected seven databases, which one comprising of data collected in a uniform way (Ninapro protocol) from several healthy and disabled patients, while performing a variety of hand movements. Each database collection and posterior analysis was performed by a group of researchers and is thoroughly described in academic papers. All of the databases are available online in a dedicated website: http://ninapro.hevs.ch/.

The present work will focus on the database number 1 to 3 (http://ninapro.hevs.ch/data1, http://ninapro.hevs.ch/data2 and http://ninapro.hevs.ch/datah3). The specific data we are interested in is the raw sEMG time-series samples collected from all individuals in these databases, along with the associated labeled vector, containing the movement corresponding to each sEMG sample.

From these databases, the more important are database 1 (healthy patients, sensors acquisition frequency of 100 Hz) and database 3 (disabled patients, sensors acquisition frequency of 2kHz), since it seems more logical comparing the proposed model performance against both healthy and disabled patients. Due to limited computational resources, the analysis of database 2 (healthy

patients, sensors acquisition frequency of 2 kHz) will be considered optional and of secondary importance.

Lets see in more depth the characteristics of each database:

Database 1:

The first Ninapro database includes data acquired with the acquisition protocol described in [18] and [17]. The database data consists of 10 repetitions of 52 different movements of 27 intact subjects.

The subjects had to repeat several movement represented by movies that are shown on the screen of a laptop.

The experiment is divided in three exercises:

1. Basic movements of the fingers
2. Isometric, isotonic hand configurations and basic wrist movements
3. Grasping and functional movements

The sEMG data are acquired using 10 Otto Bock MyoBock 13E200 electrodes (at a rate of 100 Hz), while kinematic data are acquired using a Cyberglove 2 data glove.

Figure 2: Illustration the 52 movements in database 1 (source: http://ninapro.hevs.ch)

Datasets

For each exercise, for each subject, the database contains one matlab file with synchronized variables.

The variables included in the matlab files are:

- subject: subject number
- exercise: exercise number
- acc (36 columns): three-axes accelerometers of the 12 electrodes
- emg (12 columns): sEMG signal of the 12 electrodes
- glove (22 columns): uncalibrated signal from the 22 sensors of the cyberglove
- inclin (2 columns): signal from the 2 axes inclinometer positioned on the wrist
- stimulus (1 column): the movement repeated by the subject.
- restimulus (1 column): again the movement repeated by the subject. In this case the duration of the movement label is refined a-posteriori in order to correspond to the real movement
- repetition (1 column): repetition of the stimulus
- rerepetition (1 column): repetition of restimulus

Database 3:
The third Ninapro database is thoroughly described in [17].
The experiment is divided in three exercises:
1. Basic movements of the fingers and of the wrist
2. Grasping and functional movements
3. Force patterns
In the first two exercises, the subjects naturally try to repeat several movements represented by movies that are shown on the screen of a laptop.
In the last exercise the subjects have to think to press combinations of fingers with an increasing force on a custom made device called Finger Force Linear Sensor.
The muscular activity is gathered using 12 active double–differential wireless electrodes from a Delsys Trigno Wireless EMG system. The electrodes are positioned as shown in the figure: eight electrodes are equally spaced around the forearm in correspondence to the radio humeral joint; two electrodes are placed on the main activity spots of the flexor digitorum and of the extensor digitorum as described in; two electrodes are placed on the main activity spots of the biceps and of the triceps. The described locations have been chosen in order to combine a dense sampling approach with a precise anatomical positioning strategy.
The two electrodes placed on the main activity spots of the flexor digitorum and of the extensor digitorum were not used for subject 6 and 7 due to space reasons.

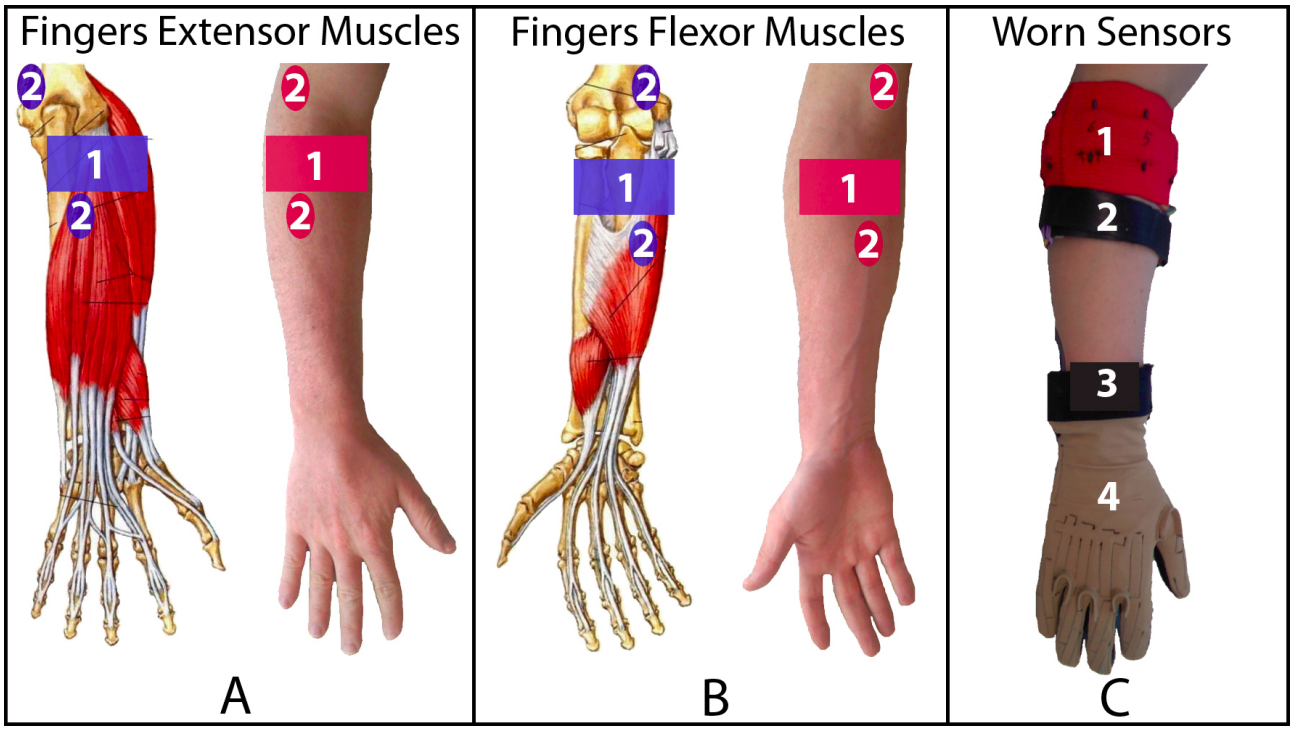

Figure 3: Positions of electrodes and glove in the arm (source: http://ninapro.hevs.ch)

The electrodes were fixed on the forearm using their standard adhesive bands. Moreover, a hypoallergenic elastic latex–free band was placed around the electrodes to keep them fixed during the acquisition. The sEMG signals are sampled at a rate of 2 kHz.

During the acquisition, the subjects were asked to repeat the movements with the right hand. Each movement repetition lasted 5 seconds and was followed by 3 seconds of rest. The protocol includes 6 repetitions of 49 different movements (plus rest) performed by 11 trans-radial amputated subjects. The movements were selected from the hand taxonomy as well as from hand robotics literature.

The number of movements is reduced in subjects 1(39 plus rest), 3(48 plus rest), and 10(43 plus rest) due to experimental causes, e.g. stress. The number of repetitions in subject 1 was originally 10 and has been reduced to 6 during post-processing in order to avoid classification

biases. This subject was the first subject being recorded with the acquisition protocol and determined the choice of reducing the number of repetitions to 6.

Figure 4: Illustration the 49 movements in database 3 (source: http://ninapro.hevs.ch)

Datasets

For each exercise, for each subject, the database contains one matlab file with synchronized variables.

The variables included in the matlab files are:

- subject: subject number
- exercise: exercise number
- acc (36 columns): three-axes accelerometers of the 12 electrodes
- emg (12 columns): sEMG signal of the 12 electrodes
- glove (22 columns): uncalibrated signal from the 22 sensors of the cyberglove
- inclin (2 columns): signal from the 2 axes inclinometer positioned on the wrist
- stimulus (1 column): the movement repeated by the subject.
- restimulus (1 column): again the movement repeated by the subject. In this case the duration of the movement label is refined a-posteriori in order to correspond to the real movement
- repetition (1 column): repetition of the stimulus
- rerepetition (1 column): repetition of restimulus
- force (6 columns): force recorded during the third exercise
- forcecal (2 x 6 values): the force sensors calibration values, corresponding to the minimal and the maximal force.

Exploratory Visualization

The sEMG data for each one of the subjects is comprised of a long time-series sequence of alternating muscular high activity (movement) and muscular low activity (rest). Thus, it is expected

from the data segments of 5 seconds of relative high amplitude electric potential measurements (movement) followed by 3 seconds of lower amplitude electric potential (rest). This behavior is illustrated in the following figure, showing a window of time from subject 10, database 3.

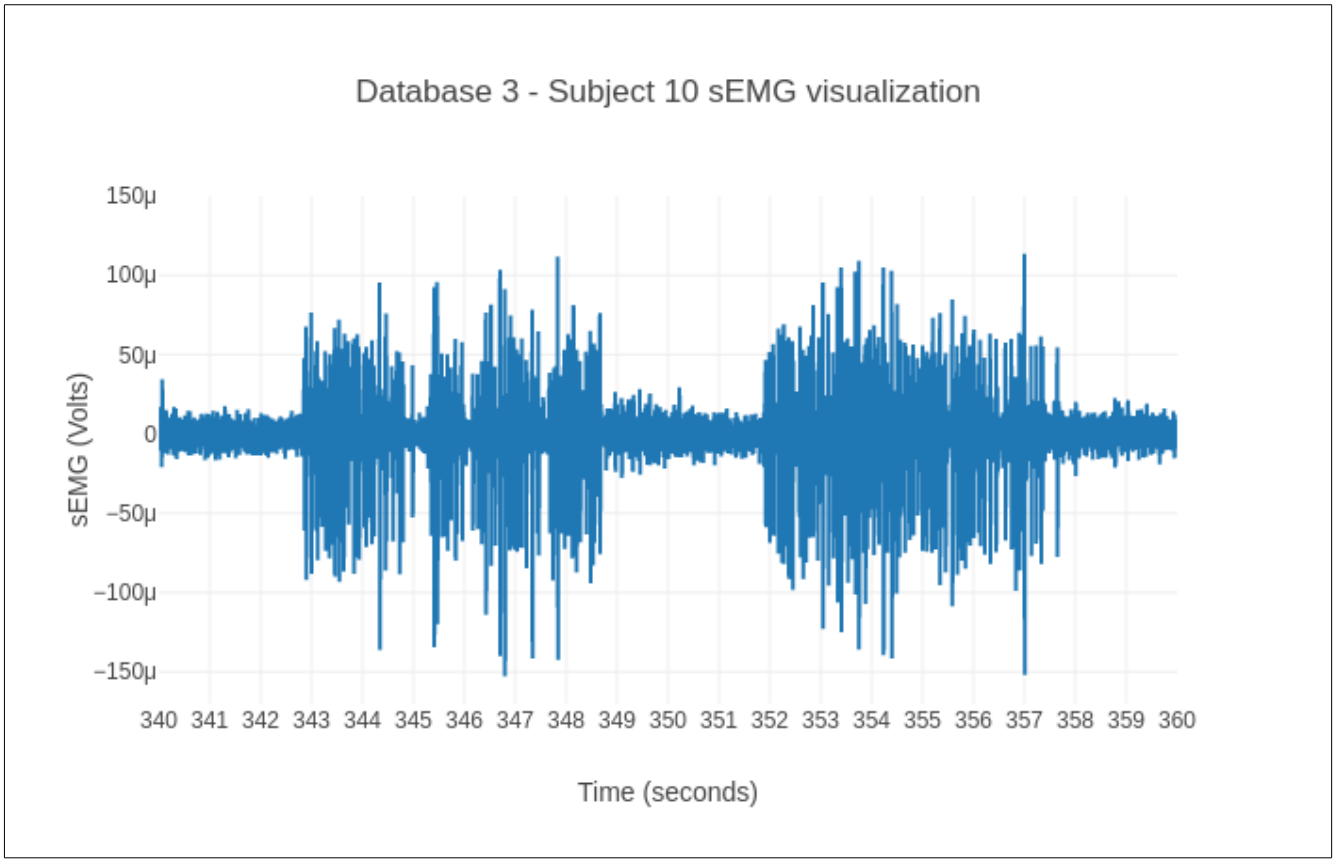

Figure 5: window of time of sEMG amplitude - subject 3, database 3

The sEMG frequency decomposition is supposed to be spread typically from 0 to 500 Hz (a bandwidth of 500 Hz), which, by the Nyquist Theorem, requires at least a sampling rate of 2x500=1kH for perfect reconstruction of the signal representation. In the case of database 1 (sampling rate of 100Hz), this will be not possible, resulting in a loss of information. That is not the case for databases 2 and 3 (both use a sampling rate of 2kHz, 2 times more than necessary).

A characteristic to be noted is that the amplitude of the electric signals will vary from subject to subject, and it is expected to be significantly less in amputated subjects than in healthy subjects. This behavior can be seen in the following figure, that shows the power spectral density of 4 subjects (2 healthy and 2 amputated).

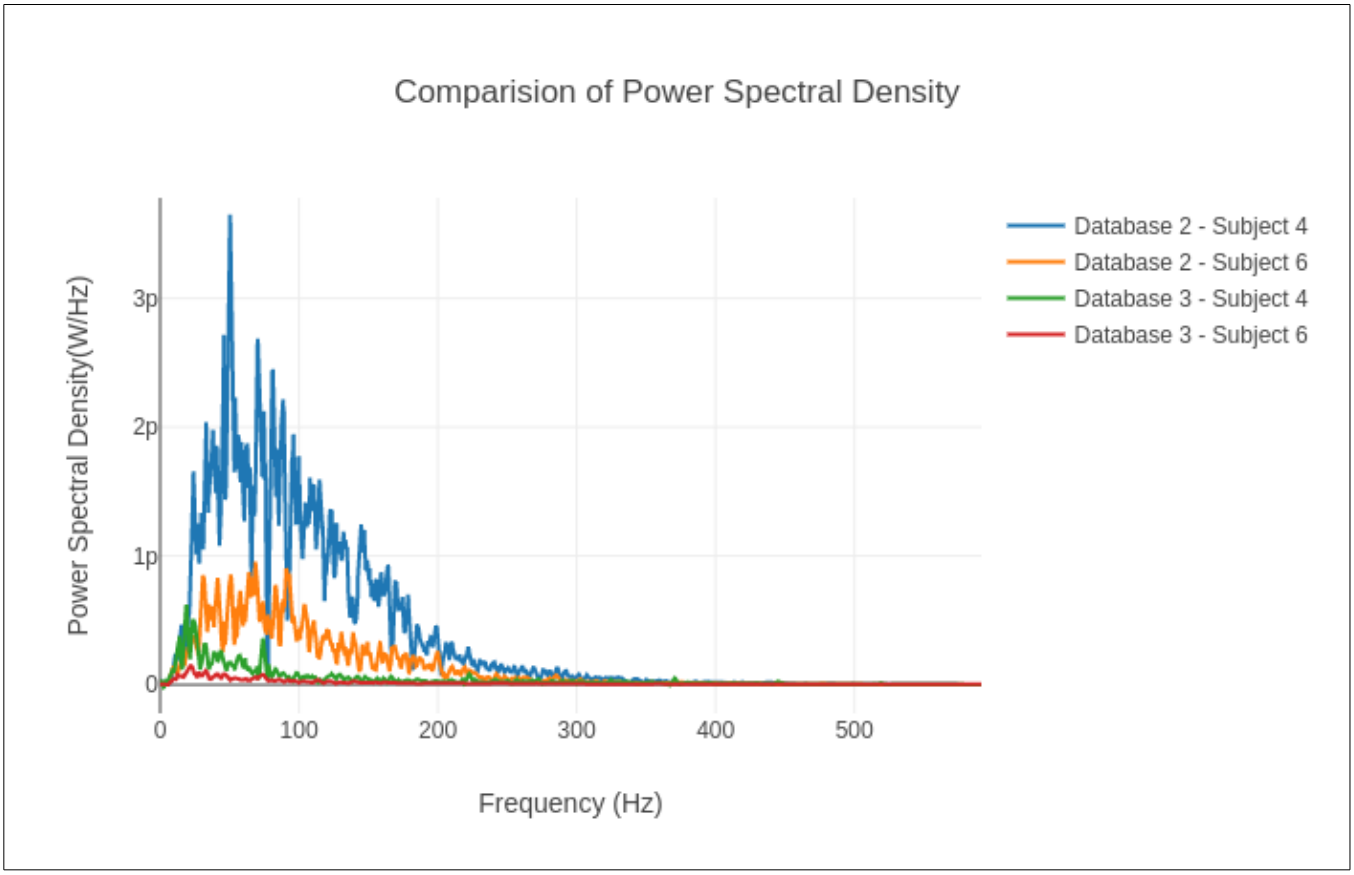

Figure 6: Comparision of 4 subjects sEMG power spectral density

Algorithms and Techniques

Usually, researchers use classifiers renowned for a long time for being successful in the task of accurately predicting hand movements from sEMG, mostly Random Forests and Support Vector Machines, using hand-crafted or engineered features.

However, it's well known that in the last years deep learning based models are surpassing in several fields the performance of other algorithms. These approaches, like in Convolutional Neural Networks, induce some bias into the models about the nature of the inputs, and overcome the burden of feature engineering by learning the features itself.

Since the inputs are basically time-series events, its seems appropriate the combination of Convolutional Neural Network layers (for learning the input features from raw data, instead of hand-craft them, as is currently common practice in the sEMG field) followed by Recurrent Neural Network layers, more particularly Long-Short Term Memory Neural Network (LSTM) layers (since LSTM is already known for its overwhelming capacity of dealing with time-series sequences), hoping it will improve the classification performance when compared to other already used models.

Data extraction

The input data is downloaded from Ninapro web site and separated by database, specifically databases one and three. Although there are other variables included in the datasets, the focus is only in the sEMG signals. These raw sEMG signals, segmented in time windows, will be used as input, and the corresponding label (hand movement) will be used as the output class, in order to train the classifier.

Model Implementation

Convolutional neural networks (CNNs) are a type of DNN (deep neural network) with the ability to act as feature extractors, stacking several convolutional operators to create a hierarchy of

progressively more abstract features. Such models are able to learn multiple layers of feature hierarchies automatically (also called “representation learning”). Long-short-term memory recurrent (LSTMs) neural networks are recurrent networks that include a memory to model temporal dependencies in time series problems.

The combination of CNNs and LSTMs in a unified framework has already offered state-of-the-art results in the speech recognition domain, where modeling temporal information is required. This kind of architecture is able to capture time dependencies on features extracted by convolutional operations.

This project uses a very interesting Deep Neural Network Architecture, described extensively in [4]. The model, called DeepConvLSTM, is a deep learning framework composed of convolutional and LSTM recurrent layers, and is perfectly suitable for this project needs, since it is capable of automatically learning feature representations along the Convolutional layers (automating feature extraction from raw sensor data instead of using a time consuming heuristic process of engineering the features) and modeling the temporal dependencies between their activation (using the LSTM units in the recurrent layers). Finally, the model has a softmax layer that outputs the predicted class probability.

The input to the network consists of a data sequence. The sequence is a short time series extracted from the sensor data (sEMG) using a sliding window approach composed of several sensor channels. The number of sensor channels is denoted as D. Within that sequence, all channels have the same number of samples $\boldsymbol{S^l}$. The length of feature maps $\boldsymbol{S^l}$ varies in different convolutional layers. The convolution is only computed where the input and the kernel fully overlap. Thus, the length of a feature map is defined by:

$$\boldsymbol{S^{(l+1)} = S^l - P^l + 1}$$

where $\boldsymbol{P^l}$ is the length of kernels in layer $\boldsymbol{l}$. The length of the kernels is the same for every convolutional layer, being defined as $\boldsymbol{P^l = 5,\ \forall l = 2, \ldots, 5}$.

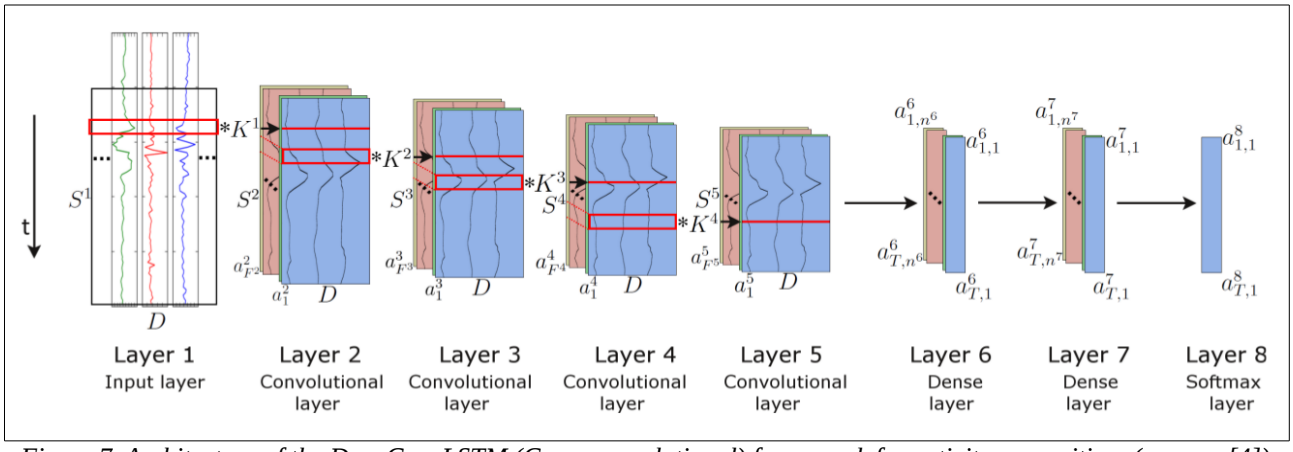

Figure 7. Architecture of the DeepConvLSTM (Conv, convolutional) framework for activity recognition. (source: [4])

In the figure above, from the left, the signals coming from the sEMG sensors are processed by four convolutional layers, which allow learning features from the data. Two dense layers then perform a non-linear transformation, which yields the classification outcome with a softmax logistic regression output layer on the right. Input at Layer 1 corresponds to sensor data of size $\boldsymbol{D \times S^1}$, where $\boldsymbol{D}$ denotes the number of sensor channels and $\boldsymbol{S^l}$ the length of features maps in layer $\boldsymbol{l}$. Layers 2–5 are convolutional layers. $\boldsymbol{S^l}$ denotes the kernels in layer $\boldsymbol{l}$ (depicted as red squares). $\boldsymbol{F^l}$ denotes the number of feature maps in layer $\boldsymbol{l}$. In convolutional layers, $\boldsymbol{a^l_i}$ denotes the activation that defines the feature map $\boldsymbol{i}$ in layer $\boldsymbol{l}$. Layers 6 and 7 are dense layers. In dense layers, $\boldsymbol{a^l_{t,i}}$ denotes the activation of the unit $\boldsymbol{i}$ in hidden layer $\boldsymbol{l}$ at time $\boldsymbol{t}$. The time axis is vertical.

Convolutional layers process the input only along the axis representing time. The number of sensor channels is the same for every feature map in all layers. These convolutional layers employ rectified linear units (ReLUs) to compute the feature maps. Layers 6 and 7 are recurrent dense

layers. Recurrent dense layers adapt their internal state after each time step. The activation of the recurrent units is computed using the hyperbolic tangent function. The output of the model is obtained from a softmax layer (a dense layer with a softmax activation function), yielding a class probability distribution for every single time step t.

A shorthand description of this model would be: ***C(64)*** => ***C(64)*** => ***C(64)*** => ***C(64)*** => ***R(128)*** => ***R(64)*** => ***Sm***, where ***C*** (F^l) q denotes a convolutional layer ***l*** with F^l feature maps, ***R***(n^l) a recurrent LSTM layer with n^l cells and ***Sm*** a softmax classifier.

The model is trained in a fully-supervised way, backpropagating the gradients from the softmax layer through to the convolutional layers. The network parameters are optimized by minimizing the cross-entropy loss function using mini-batch gradient descent with the RMSProp update rule.

For the sake of efficiency, when training and testing, data are segmented on mini-batches of a size of 100 data samples. Using this configuration, an accumulated gradient for the parameters is computed after every mini-batch. Both models are trained with a learning rate of 0.001 and a decay factor of ρ = 0.9. Weights are randomly orthogonally initialized. We introduce a drop-out operator on the inputs of every dense layer, as a form of regularization. This operator sets the activation of randomly-selected units during training to zero with probability p = 0.5.

The length of the time window is 200 ms, with a step size of 100 ms. The number of instances (segments) obtained after using this sliding window configuration will be approximately 8s (each movement duration plus rest)/100ms = 80. Each segment is composed by synchronized sEMG signals captured from D sensors, at a rate of 100 Hz(database 1), or 2 kHz (database 3). The result is an input matrix of dimensions D x 20 (database 1), or D x 400 (database 3). The class associated with each segment corresponds to the hand movement that has been observed during that interval (0.2 seconds).

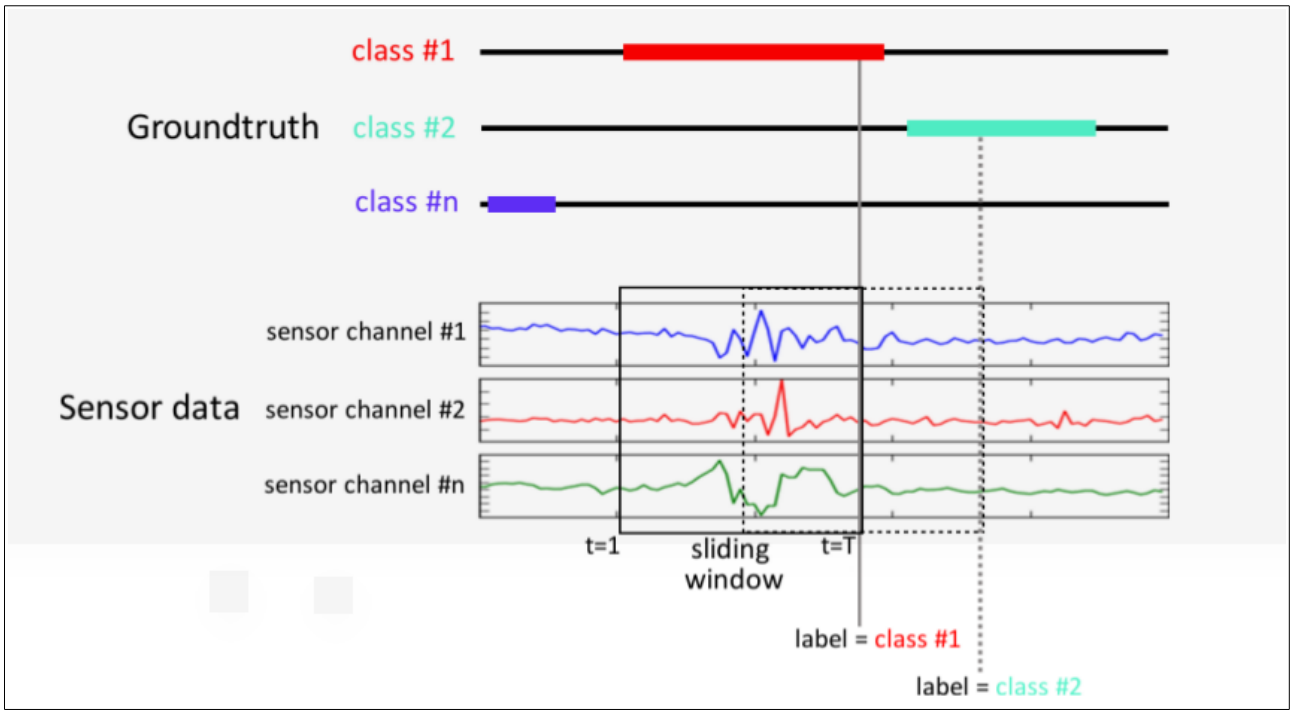

Figure 8: Each step of the sliding window T corresponds to a class (source: [4])

DeepConvLSTM outputs a class probability distribution for every single time step t in the sequence (i.e., the 200-ms window of the sEMG sensor signal). However, the interest is in the class probability distribution once DeepConvLSTM has observed the entire 200-ms sequence.

Since the memory of LSTM units tends to become progressively more informed as a function of the number of samples they have seen, DeepConvLSTM returns the class probability distribution only at the last time step T, when the full sequence has been observed. Thus, at the time

of each sample of the original sensor signal, DeepConvLSTM provides a class probability distribution inferred from processing a 200-ms extract of the sEMG sensor signal prior to that time, as illustrated in Figure 6.

This approach of only using the last time step T is encouraged by the results of a previous work [5]. Their investigation demonstrates that misclassifications occur primarily during the movement onset (initial phase) and offset (final phase). The explanation for this phenomenon is that the sEMG signals are not yet (or not anymore) sufficiently discriminative in these transitory phases between movement and rest. Thus, it is reasonable to suppose the LSTM units will overcome this issue by captioning these temporal dynamics of the sEMG signals.

Benchmark

The proposed DeepConvLstm model will be benchmarked against the models used and described in [4]. In [4], the databases 1 to 3 from NinaPro were used as input, and a Convolutional Neural Network architecture was chosen as a model for the classifier. This paper compares its own results with earlier works too.

For 50 classes (hand movements), the classifier obtained an average classification accuracy of 66.59 ± 6.40% on database 1, 60.27 ± 7.7% on database 2 and 38.09 ± 14.29% on database 3 (composed only by data collected from amputees – classification accuracy is expected to be lower than the previous databases, composed only by able subjects).

Earlier works cited in the paper got better performance, though; the best classical classification methods were: on database 1, Random Forests with all features, with an average classification accuracy of 75.32 ± 5.69%; on database 2, Random Forests with all features with an average classification accuracy of 75.27% ± 7.89%; and in database 3, a Support Vector Machine with all features, with an average classification accuracy of 46.27% ± 7.89%.

III. Methodology

Data Preprocessing

The Ninapro data is downloaded from Dryad, an online repository for research data (https://datadryad.org/resource/doi:10.5061/dryad.1k84r). Each subject for each database has an associated zip file, accounting for a total of 78 files. The zip compacted files have 3 *.mat (matlab format) files in it, each one corresponding to data gathered from a different set of movements (already described in the ***Data Exploration*** section).

Once the zip file is downloaded and its files extracted, the sEMG content for each *.mat file and its corresponding classification label (movement associated) is obtained and gathered together. Then, for every subject, there are thousand of examples, where each one is an input vector (sEMG data from all sensors) and its output labeled vector (containing the class or movement associated).

The data is split into training/test sets, in a proportion of 7:3 (database1) and 8:2(database 3). The training set is then normalized, rearranged into time steps for feeding the LSTM layers and split again into training/validation sets in a proportion of 9:1, being ready for training the neural network.

Implementation

The model already described and all the auxiliary scripts are implemented using python (version 3.5 for CPU and 3.6 for GPU) as the programming language, Keras (version 2.2) as the deep learning framework, using Tensorflow (version 1.8) library as the backend. Keras was chosen as the high-level framework because of its simple and concise interface, allowing the user implement a deep learning model within a few lines of code. Besides, cloud providers, as AWS

(Amazon Web Services), provide clean and ready to work powerfull GPU instances for trainining Deep Learning models, powered by conda environments, using Keras.

All processes and results described in this session are easily reproducible, by means of modules specially written for this purpose, along with jupyter notebooks and helpfull scripts. For this, it's enough to clone the repository https://github.com/jonDel/emg_mc.git and follow the steps in the README.md file. Extraction and data pre-processing are performed using nina_helper library (source: https://github.com/Lif3line/nina_helper_package_mk2). This library was forked from github (source: https://github.com/jonDel/nina_helper_package_mk2) and modified in order to support NinaPro database 3 (the original version only supports databases 1 and 2). This library provides methods for extracting all the necessary information from the matlab files (*.mat) into tensors, as well as for splitting the data into test/training sets, normalizing the training set and reorganizing the input raw sEMG samples into time steps for feeding the LSTM layers.

The training process is carried out for every one of the subjects from database one (27) and database 3 (11). At the end of each training session, the model is evaluated against the training set and the resulting validation accuracy is recorded into log files (one for database 1 and other for database 3 results). After all training sessions of every dataset from each database, the mean and standard deviation is calculated and used for benchmarking.

Refinement

Some problems arose in the process of training the model, mostly caused by unlucky hiperparameter choosing. These problems are summarized bellow.

First, it was notice that, although the training curve continued improving through time, the validation curve did not. It randomly oscilated around a mean line, showing signs that the model was badly overfitting and not capturing the variance of the model; instead, it was just memorizing the input data. After some hiperparameters tweaking, it was noticed that the initial batch size of 100 was causing this issue, maybe due to the relative low number of input samples when compared to the high capacity of the model. The batch size was reduced to 64, 32 and finally 16 samples, which solved the problem, allowing the model to generalize. However, there is no free-lunch: a lower batch size resulted in a much longer training session (Note: batch sizes lower than 16 could make the training time impractible in a CPU).

Second, a discrepancy was noticed between the validation accuracy and the training accuracy computed after the end of a training session. This was probably due to a low proportion of training/validation samples (9:1), when compared to training/test (7:3). The proportion was changed to 6.67:3.33, leading the validation accuracy very close to the test accuracy. Again, the no free-lunch theory applies: more validation samples means less training samples, hence less data to train the model.

Third, certain elements where lacking when training, as progress' visualization, weights' recovering after a crash and training's early stopping of stalled sessions. This need was overcome by applying the techniques that follows:

- The technique of model checkpoint (saving the weights during training process), using the maximum value of validation set accuracy as a parameter, i.e, at the end of each iteration (epoch), the validation accuracy is taken, and the model weights at that time are saved on disk only if the accuracy outperforms all other iterations. Thus, if the training process is interrupted for some reason, when the training script runs again, it checks in a predefined folder for previous trained weights and loads the best of them into the model before trainining. This step is crucial and saves a lot of time in the case of interruption, since there is no need to run again the training process from scratch.
- The technique of early stopping, so there was no need to run the training process for all the predefined number of epochs, when it reachs a point where clearly no more improvements are made. A total number of 150 epochs was defined, and if the training got stuck (no more improvements in validation accuracy) for more than 20 epochs since the last best accuracy value, it

was stopped. In addition, before the training beginning, if it finds and loads the weights file saved from a previous session, the iteration when it was saved was recovered and used. Lets say a training session saves its last best weight at epoch 40, and for some reason crashes. Then, when the session is started again, the script loads the best weight into the model, and run again the training, but not for 150 epochs; now it runs for 150 - 40 = 110 epochs, saving time and providing consistency among diferent training sessions.

- A Keras callback to a Tensorboard (https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard) session was attached to the model, so that training and validation curves, along with its respective loss curves, could be monitorated at training time, through a browser. This technique proved very usefull, particulary when running the training sessions in the cloud.

Fourth, after running the trainining sessions for all database one subjects, facing the results (which will be analised and discussed in the IV session of this report) led to the thought that maybe these results could be improved by an architectural change. How? At that time, the LSTM cells were only allowed to remember the samples of one time window (100 ms), when the total duration of a movement was 5 seconds plus 3 seconds of rest. Then, the cells were not being fed with a sequence containing all the dynamics of the beggining and the end of each movement. The proposed solution was to extend the number of input time-steps in order to cover a much greater time window of 1.25 x 8 = 10 seconds (100 times greater than the original). Notice that this increase in input dimensions of course significantly increased the training time.

Fifth, when running the trainining sessions for database 3 subjects, it ran into a system memory problem: the data acquistion rate was 2kHz, which means 20x more samples than database one and consequently 20x more memory needed to store the input tensors for the model. It could not fit in a desktop memory of 8 GB. The solution was to choose a GPU cloud instance to train the model. An AWS Deep Learning instance was used, with a Volta V-100 Nvidia GPU attached, powered by conda virtual environments already configured for Keras and Tensorflow.

Sixth, even when running the model for database 3 in a powerfull GPU as V-100 for several hours, not a single session of one subject training was finalized. This behavior brought surprise and disapponted, since was expected the GPU to be much more powerful and fast than the desktop CPU (i5 4 cores). On the opposite, it even performed a bit worse than the CPU. What was happening? A research led to an explanation: RNNs in particular performs badly on GPUs, since it can't be parallelized in time because the hidden layer's previous output is required to be known at each current input in the forward pass. So no hope? Yes, there are hope somewhere, and it is called CuDNNLSTM, a Keras LSTM cell specially designed for running on Nvidia CUDA GPUs. The code was changed to support both GPU and CPU LSTM cells automatically, and when ran, the training session showed a fantastic improvement (more than 30 times faster).

Seven, even when running the code in the cloud GPU instance with the CuDNNLSTM cells, the estimated total training time for all subjects of database 3 turns out to be economically unaffordable. So, the solution was to subsample the datasets to 1/16 of the original size, at the expense of getting only 1/16 samples to feed the model.

IV. Results

Model Evaluation and Validation

Given the considerations and corrections suggested and implemented in the last session, the final model is basically still the same as the one idealised in the **Algorithms and Techniques** session, regarding some hiperparameters changing and input data subsampling (database 3). The following changes were made: batch size, decreased from 100 to 16; the size of the sliding window, increased from 100ms to 10s; and a subsampling scheme was applied only to database 3, in a proportion of 1:16, i.e, at each group of 16 samples, 15 was discarded and only the one remaining was used.

Although this model constitutes the final one, preliminary trainings were made with smaller sliding window sizes in order to compare the model performance in different scenarios. By doing so, it was possible to prove the model performs better when given more samples to the LSTM layers to remember, i.e., the bigger the sliding window size, the better the performance (until the model reachs a limit window size from where no more significant improvements are perceived) and consequently the greater the training time.

Database 1 subjects were trained using sliding windows of 0.2 second (20 time steps) and 2 seconds (200 time steps). Due to limited resources of time, for a sliding window of 10 seconds (1000 time steps), the training was performed only for subject 1 (since a noticeable improvement were made from an increase in window size of 2s to 10s, it is reasonable to suppose a similar gain is expected if the training is extended to the other subjects from database 1). All sessions ran in a desktop computer, with a 4 core CPU (Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz), and 8 GB of RAM.

The table that follows contains the test set accuracy for all subjects, the epoch where the best result was obtained, the Test Accuracy/Chance Level ratio (TA/CL; a ratio of 1 is expected when the results are no better than pure chance; a bigger ratio indicates the model captures some input variance rather than mere randomness), along with the mean and standard deviation of each one of them.

Subject number	**Test accuracy (20 timesteps)**	**TA/CL (20 timesteps)**	**Test accuracy (200 timesteps)**	**TA/CL (200 timesteps)**	**Epochs (20 timesteps)**	**Epochs (200 timesteps)**
1	0.716680	37.984040	0.806368	42.737504	17	57
2	0.737429	39.083737	0.795925	42.184025	42	86
3	0.695032	36.836696	0.745698	39.521994	29	44
4	0.626994	33.230682	0.660562	35.009786	33	47
5	0.675799	35.817347	0.739318	39.183854	94	42
6	0.645522	34.212666	0.695786	36.876658	11	9
7	0.748971	39.695463	0.767191	40.661123	22	48
8	0.731541	38.771673	0.799908	42.395124	49	46
9	0.703405	37.280465	0.793612	42.061436	14	63
10	0.693525	36.756825	0.778467	41.258751	34	55
11	0.702032	37.207696	0.745035	39.486855	36	77
12	0.703305	37.275165	0.769662	40.792086	45	34
13	0.729927	38.686131	0.767524	40.6787720	38	60
14	0.683526	36.226878	0.769445	40.780585	60	65
15	0.702466	37.230698	0.706869	37.464057	47	22
16	0.717505	38.027765	0.781934	41.442502	25	88
17	0.739481	39.192493	0.785408	41.626624	46	42
18	0.644295	34.147635	0.642381	34.046193	72	15
19	0.681540	36.121620	0.717729	38.039637	19	63
20	0.706396	37.438988	0.798440	42.317320	35	37

Subject number	Test accuracy (20 timesteps)	TA/CL (20 timesteps)	Test accuracy (200 timesteps)	TA/CL (200 timesteps)	Epochs (20 timesteps)	Epochs (200 timesteps)
21	0.776243	41.140879	0.836223	44.319819	41	58
22	0.689955	36.567615	0.802321	42.523013	77	38
23	0.657660	34.855980	0.669453	35.481009	28	26
24	0.755642	40.049026	0.781508	41.419924	53	67
25	0.734835	38.946255	0.769137	40.764261	33	24
26	0.775681	41.111093	0.814809	43.184877	22	99
27	0.643924	34.127972	0.698854	37.039262	37	59
Mean	0.704419	37.334203	0.757021	40.122113	39.22	50.78
Standard Deviation	0.039678	2.102931	0.050215	2.661389	19.48	22.13

Table 1: model results gathered from all subjects in database 1

The following table informs the total time required for training all subjects from database 1, for sliding windows of 0.2 seconds (20 time steps) and 2 seconds (200 time steps).

Number of time steps	Total training time (days)
20	3.14
200	12.01

Table 2: training time of database 1 for time steps 20 and 200

The next table compares the test accuracy and total running time for **subject 1** only, using sliding windows of 0.2 seconds (20 time steps), 2 seconds (200 time steps) and 10 seconds (1000 time steps).

Number of time steps	Test accuracy	Training time (hours)	Best result's epoch
20	0.716680	1.28	17
200	0.806368	10.93	57
1000	0.830781	46.01	44

Table 3: training time of subject 1, database 1, for time steps 20, 200 and 1000

Database 3 subjects were trained using sliding windows of 0.2 second (25 time steps) and 10 seconds (1250 time steps), and an input subsampling rate of 1/16. An experiment with a sliding window of 24 seconds (3000 time steps) was performed only for subject 2 (due to limited economic resources). The 25 time steps sessions ran in a desktop computer, with a 4 core CPU (Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz), and 8 GB of RAM, while the 1250 and 3000 time steps sessions ran on a NVIDIA® Tesla® V100 GPU (Volta Architecture), 16B of RAM, in a rent per hour Deep Learning instance in AWS.

The table that follows presents the same structure as table number *, but for diferent time steps and for database 3.

Subject number	Test accuracy (25 timesteps)	TA/CL	Test accuracy (1250 timesteps)	TA/CL	Epochs (25 timesteps)	Epochs (1250 timesteps)
1	0.481201	19.248040	0.596119	23.844760	40	40
2	0.486963	24.348150	0.618542	30.927100	53	61
3	0.626149	31.307450	0.694411	34.720550	29	71
4	0.313824	15.691200	0.524122	26.206100	10	61
5	0.368629	18.431450	0.384526	19.226300	33	25
6	0.292106	14.605300	0.380437	19.021850	31	61
7	0.292592	14.629600	0.279037	13.951850	5	38
8	0.473183	23.659150	0.544933	27.246650	78	67
9	0.547991	27.399550	0.666587	33.329350	58	70
10	0.287800	12.375400	0.329661	14.175423	25	44
11	0.540861	27.043050	0.673598	33.679900	15	68
Mean	0.428300	20.794395	0.517452	27.424961	34.272727	55.09
Standard Deviation	0.121559	6.286815	0.149564	0.149564	21.913881	15.61

Table 4: model results gathered from all subjects in database 3

The following table informs the total time required for training all subjects from database 3, for sliding windows of 0.2 seconds (25 time steps) and 10 seconds (1250 time steps).

Number of time steps	Total training time (hours)	Hardware
25	24.1	Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz
1250	13.25	NVIDIA® Tesla® V100 GPU

Table 5: training time of database 3 for time steps 25(CPU) and 1250(GPU)

The next table compares the test accuracy and total running time for **subject 2** only, using sliding windows of 0.2 seconds (25 time steps), 10 seconds (1250 time steps) and 24 seconds (3000 time steps).

Number of time steps	Test accuracy	Training time (minutes)	Best result's epoch	Hardware
25	0.486963	55	53	Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz
1250	0.618542	69	61	NVIDIA® Tesla® V100 GPU
3000	0.638971	135	49	NVIDIA® Tesla® V100 GPU

Table 6: training time of subject 2, database 3, for time steps 25(CPU), 1250(GPU) and 3000(GPU)

Justification

Two diferent results from previous works were cited for comparison with the results from the present work: one using a Convolutional Neural Network (CNN), and other using Support Vector Machine (SVM, for database 3) and Random Forests (for database 1).

The following table compares both previous results with the ones from this work, for database 1.

Model	**Test accuracy mean (%)**	**Standard deviation (%)**	**Number of classes**	**TA/CL**
CNN	66.59	6.40	50	33.30
Random Forests	75.32	5.69	50	37.66
DeepConvLstm	75.70	5.02	53	40.12

Table 7: benchmark of previous best models and DeepConvLstm, on database 1

The **DeepConvLstm** model clearly outperformed the other models in database 1, as we can see. Although it only performed 0.38% better than Random Forests, it is to be noted that the number of classes used for DeepConvLstm is 53, 3 more than the other models that used 50; so, it is more than reasonable to suppose that if DeepConvLstm used only 50 classes, the resulting accuracy would be greater. This is more visible by looking at the TA/CL ratio, which takes into account the number of classes used; it shows a notable improvement of 6.53% in the performance when compared to the Random Forests model.

The following table compares both previous results with the ones from this work, for database 3.

Model	**Test accuracy mean (%)**	**Standard deviation (%)**	**Number of classes**	**TA/CL**
CNN	38.09	14.29	50	19.05
SVM	46.27	7.89	50	23.14
DeepConvLstm	51.75	14.96	50	25.88

Table 8: benchmark of previous best models and DeepConvLstm, on database 3

The **DeepConvLstm** model visibly outperformed the other models in database 3. It performed 5.47% better than the previous best result, from the SVM model, a remarkable increase in accuracy.

V. Conclusion

Free-Form Visualization

As discussed before, the greater the number of time steps of the inputs (i.e., more information of the past for the LSTM layers to learn from), the greater is the performance. This characteristic can be visualized in the following figure, taken from Tensorboard:

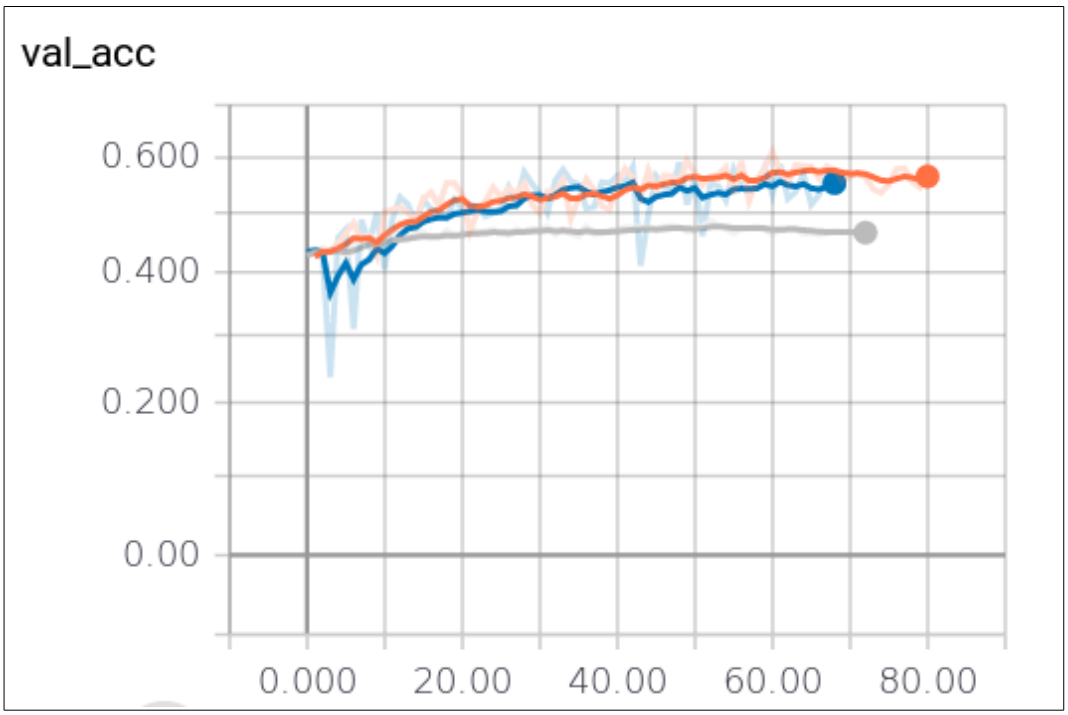

Figure 9: validation accuracy curves during training for 25(gray), 1250(orange) and 3000(blue) time steps, subject 2, database 3. X axis is the epoch number and Y axis the validation accuracy (source: Tensorboard)

Clearly, the validation accuracy increases greatly from changing the time steps number from 25 to 1250 samples. However, the increase is not that clear when changing it from 1250 to 3000 samples. Maybe the training accuracy curves can help, as depicted in the following figure:

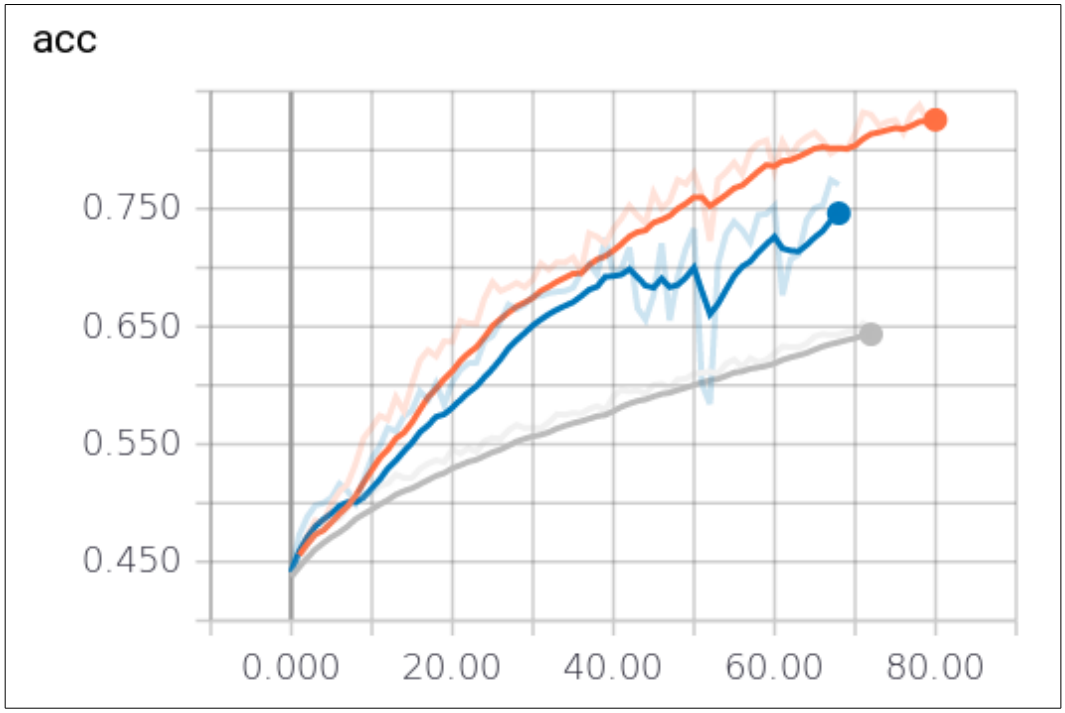

Figure 10: training accuracy curves during training for 25(gray), 1250(orange) and 3000(blue) time steps, subject 2, database 3. X axis is the epoch number and Y axis the training accuracy (source: Tensorboard)

Neither curve shows a clear sign of overfitting: it seems as, giving more epochs and increasing the patience (number of epochs to wait, after the last optimal weight saved, before considering the training staled and stop it), the models could reach better results (getting near the theoretical optimal point where the validation and training curves crosses).

Besides, looking at the curves' shape and steepness, it is reasonable to expect that the 1250 and 3000 ones have more room to improve, with the 3000 showing signs of surpassing the 1250 one in the future (greater steepness). However, the cost of having 3000 time steps (more than double the 1250 one) seems not to pay off, due to very limited gains in accuracy.

Reflection

The solution proposed in this work to solve the problem of hand movement classification consists briefly in the following steps:

- Download all the datasets (subject data) from each database (databases 1 and 3);
- Extract the relevant data from these datasets (sEMG and classification label);
- Format the input data into time steps and subsample when necessary (database 3);
- Split the input data into training/test sets;
- Normalize the training set and split it into training/validation sets;
- Train the model using the validation set as metric to monitor the training progress;
- Save all the bests weights for reproducing the results later;
- Gather the results for benchmarking.

There were challenging tasks along the development of the project. One was how to prepare the datasets from database 3 to be fed as input data to the model. The library used for this purpose, for database 1, had to be understood and extended in order to be able to deal with database 3 too (it only has support for databases 1 and 2), which is not uniform (some subjects had less classes or less electrodes applied during acquistion time). Other challenge was how to deal with the memory and processing limitations of the available desktop computer, which led to learn how to deploy a Deep Learning model in the cloud and how a limited budget impacts on the development of a project.

The most interesting part of the project was the fact that the model, as hoped, really learned automatically the features from raw data. This aspect is crucial, since data preprocessing (like Discrete Fourier Transform) has to be done online and demands a lot from CPU, impacting negativelly in the cost or the time needed for classifing the inputs. In fact, it's intriguing why previous efforts using CNNs, like [16], hand-crafted the features before feeding the CNN layers instead of simply feeding the raw data, saving time and effort by leaving the CNN itself do the job of learning the features.

As a whole, the final results of this project are very satisfying. The assumptions and insights about the problem and the data proved to be reasonable, needing only minor modifications on the model proposed, in order to achieve great results, which surpasses the previous best works on the field of hand movement classification using sEMG.

Improvement

The process of implementing the model and the dificulties faced along brought a lot of ideas and thoughts of possible lines of improvement. First, the datasets used have few samples. Considering an entire movement (5 seconds of movement plus 3 seconds of rest) as effectively a sample, a 10 or 6 repetition of each movement means too little meaningful data to feed a deep learning model. A suggestion would be to perform more repetitions when collecting the data from subjects, perhaps diminishing the number of classes, giving more rest time for them, or dividing the aquisition in different sessions, in order to not cause pain or injure the patients. An alternative strategy would be generating artificial input data, using some data augmenting scheme, like adding gaussian noise to real samples or creating new ones using GANs (generative adversial networks).

Given the necessary computational power and time, a more precise hiperparameter tweaking could be done, using a grid search. This measure could greatly benefit the performance of the model (as it was seen in the **Model Evaluation and Validation** session, when the batch size was decreased from 100 to 16).

Another suggestion would be to try architectural changes in the model, like adding or removing LSTM or CNN layers, or resizing the existing layers. It could be the case where a smaller model achieves similar performance to the present model, while providing faster training sessions.

Ideally, a sampling rate of 1KHz would be enough to capture almost all the muscular activity bandwidth (mostly in the range of 0-500 Hz). However, as it was seen, with only 0.1 (database 1) or 0.125 (database 3) of the ideal sampling rate, the model was able to meaningfully learn, what leads to wonder how much frequency information the model really needs, in order to be able to learn the data structure. It is probably the case where it reaches a point where increasing the sampling rate would only results in minimal and not very relevant gains. So, it would be interesting to test incremental sampling rates in order to discover the optimal one, considering the trade-off between test accuracy/training time.

The same logic applies to the number of time steps fed into the model. From the ***Results*** session, it is obvious that an increase in the number of time steps, i.e, feeding more memory into the LSTM layers, results in an increase in test accuracy. The question to be solved is: what is the optimal number of time-steps, considering the trade-off already mentioned?

Another question to be asked is if transfer learning techniques can be applied. In an affirmative case, a larger session of data gathering could be performed, using data from diferent subjects, in order to learn more general features from data, rather than subject-specific features. Then, new training sessions could be performed faster, by reusing the first layers weights (mostly the CNN layers), leaving for the training te task of learning only the dynamical part of data (mostly the LSTM and softmax layer) proper of each subject.

And last but not least, the input data could be extend, in the case of database 3, to include more variables, like accelerometer data, since it is known for other works [12] to help improving the accuracy of the model.

References

1. G Staude, W Wolf, Objective motor response onset detection in surface myoelectric signals. Med. Eng. Phys. 21, 449–467 (1999)
2. Yansheng Wu, Shili Liang , Ling Zhang, Zongqian Chai, Chunlei Cao and Shuangwei Wang. Gesture recognition method based on a single-channel sEMG envelope signal, https://link.springer.com/content/pdf/10.1186/s13638-018-1046-0.pdf
3. Hargrove, Levi & Englehart, Kevin & Hudgins, Bernard. (2007). A Comparison of Surface and Intramuscular Myoelectric Signal Classification. IEEE transactions on bio-medical engineering. 54. 847-53. 10.1109/TBME.2006.889192.
4. Atzori M, Gijsberts A, Castellini C, Caputo B, Mittaz Hager A, Elsig S, Giatsidis G, Bassetto F, Müller H (2014) Electromyography data for non-invasive naturally controlled robotic hand prostheses. Scientific Data 1:140053. https://doi.org/10.1038/sdata.2014.53
5. Ordóñez F.J., Roggen, D., 2016. Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition. Sensors 16, 115
6. Allard, Ulysse Côté et al. “Deep Learning for Electromyographic Hand Gesture Signal Classification by Leveraging Transfer Learning.”*CoRR* abs/1801.07756 (2018)
7. Boyali, Ali. (2015). Spectral Collaborative Representation based Classification for Hand Gestures recognition on Electromyography Signals. Biomedical Signal Processing and Control. 24. 10.1016/j.bspc.2015.09.001.
8. M. Romaszewski, P. Glomb, and P. Gawron. Natural hand gestures for human identification in a human-computer interface. In Image Processing Theory, Tools and Applications (IPTA), 2014 4th International Conference on, pages 1–6, Oct 2014.
9. Cisotto, Giulia & Michieli, Umberto & Badia, Leonardo. (2017). A coherence study on EEG and EMG signals.

10. Francesca Palermo, Matteo Cognolato, Arjan Gijsberts, Henning Müller, Barbara Caputo, Manfredo Atzori:Repeatability of grasp recognition for robotic hand prosthesis control based on sEMG data. ICORR 2017: 1154-1159
11. Khushaba, Rami & Al-Timemy, Ali & Kodagoda, Sarath & Nazarpour, Kianoush. (2016). Combined Influence of Forearm Orientation and Muscular Contraction on EMG Pattern Recognition. Expert Systems with Applications. 61. 10.1016/j.eswa.2016.05.031.
12. F. Palermo, M. Cognolato, A. Gijsberts, H. Muller, B. Caputo, and ¨ M. Atzori, “Repeatability of grasp recognition for robotic hand prosthesis control based on sEMG data,” in Rehabilitation Robotics (ICORR), 2017 International Conference on. IEEE, 2017, pp. 1154–1159.
13. Pizzolato S. Comparison of six electromyography acquisition setups on hand movement classification tasks. PLoS One. 2017; 12(10): e0186132.
14. Cognolato, Matteo & Atzori, Manfredo & Caputo, Barbara & Brugger, Peter & Müller, Henning. (2016). From NinaPro to MeganePro towards the natural control of myoelectric prosthetic hands,.
15. Atzori, M., and Müller, H. (2015). Control capabilities of myoelectric robotic prostheses by hand amputees: a scientific research and market overview. Front. Syst. Neurosci. 9:162. doi: 10.3389/fnsys.2015.00162
16. Manfredo Atzori, Matteo Cognolato, Henning Müller: Deep Learning with Convolutional Neural Networks Applied to Electromyography Data: A Resource for the Classification of Movements for Prosthetic Hands. Front. Neurorobot. 2016 (2016)
17. Atzori, M., Gijsberts, A., Castellini, C., Caputo, B., Hager, A.M., Elsig, S., Giatsidis, G., Bassetto, F., & Müller, H. (2014). Electromyography data for non-invasive naturally-controlled robotic hand prostheses. Scientific data.
18. Atzori, M., Gijsberts, A., Kuzborskij, I., Elsig, S., Hager, A.M., Deriaz, O., Castellini, C., Muller, H., & Caputo, B. (2015). Characterization of a Benchmark Database for Myoelectric Movement Classification. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 23, 73-83.