UNIVERSITÀ DI PISA

MSc in Computer Engineering

Intelligent Systems

Project Report

TEAM MEMBERS:

Fabrizio Lanzillo

Federico Montini

Riccardo Sagramoni

Academic Year 2021/2022

Table of Contents

1. Introduction

The aim of this project is to design and develop several intelligent systems, which will analyze the timeseries of collected physiological signals in order to distinguish various characteristics.

We carried out all the points of the specifications, in particular:

- 3.1 section:
    - Design and develop **two multi-layer perceptron (MLPs)** artificial neural networks **that estimate**, respectively **the mean and standard deviation of the ecg** of a person (XX) during the three activities (YYYY), based on sensor data stored in the sXX_YYYY_timeseries file. In order to train the system to predict these values, we had to extract the features from the timeseries, perform a **data augmentation** and **select the most significant** ones **via** the **sequentialfs**.
    - Design and train two **radial basis function (RBF)** networks that do the same thing of the two MLPs.
- 3.2 section: design and develop **a multi-layer perceptron** which classifies a person's activity among 'sit', 'walk' and 'run'.
- 3.3 section: design and develop a **fuzzy inference system to classify a person's activity** using the k most relevant features ($k \leq 5$) in the set of features used to train the classifier developed in section 3.2.
- 4.1 section: the aim of this part of the project **is the same as in Section 3.1**, with the exception of **using a convolutional neural network (CNN) instead of an MLP**. The value to be estimated (mean/standard deviation) is the one that achieved the worst performance using the MLP in section 3.
- 4.2 part: Design and develop a **recurrent neural network (RNN) that predicts one value of a person's ecg**, based on part or all the signals in the dataset. The RNN takes these signals at time steps ($t - k, \dots, t$) as input along with the corresponding ecg values, and returns the ecg value at time step $t + 1$
- 4.3 part: design and develop **an RNN which predicts a set of consecutive future values** of a person's ecg.

2. Dataset

This chapter describes **how** the data were **processed** in order **to make the predictions** that will be described in the following chapters.

The data are stored in CSV format files.

For **each subject we have 3 recordings**, each **divided into 2 CSV files**, **one** containing **physiological signals** (**timeseries** files), and the **other** containing related **target values** (**target** files).
Each recording is related to a different physical activity.
Thus, we have 66 CSV files of timeseries type and 66 of target type, for a total of 132 CSV files.

Below the name and structure of the 2 CSV files of a single recording:

- ***sXX_YYYY_timeseries***;
- ***sXX_YYYY_targets;***

'***s***' stands for subject, '***XX***' stands for the subject number (1-22), and '***YYYY***' stands for an activity among '***sit***', '***walk***', and '***run***'.

Each file ***sXX_YYYY_timeseries*** is organized in columns. The first column contains a timestamp. Each of the other columns contains a time series corresponding to a physiological signal. Signals are as follows:

- ***pleth_1***: red wavelength PPG from the distal phalanx (first segment) of the left index finger palmar side (sampling rate 500 Hz)
- ***pleth_2***: infrared wavelength PPG from the distal phalanx (first segment) of the left index finger palmar side (sampling rate 500 Hz)
- ***pleth_3:*** green wavelength PPG from the distal phalanx (first segment) of the left index finger palmar side (sampling rate 500 Hz)
- ***pleth_4***: red wavelength PPG from the proximal phalanx (base segment) of the left index finger palmar side (sampling rate 500 Hz)
- ***pleth_5***: infrared wavelength PPG from the proximal phalanx (base segment) of the left index finger palmar side (sampling rate 500 Hz)
- ***pleth_6***: green wavelength PPG from the proximal phalanx (base segment) of the left index finger palmar side (sampling rate 500 Hz)
- ***lc_1***: load cell proximal phalanx (first segment) PPG sensor attachment pressure (sampling rate 80Hz)

- ***lc_2***: load cell (base segment) PPG sensor attachment pressure (sampling rate 80Hz)
- ***temp_1***: distal phalanx (first segment) PPG sensor temperature (°C, sampling rate 10Hz)
- ***temp_2***: proximal phalanx (base segment) PPG sensor temperature in (°C, sampling rate 10 Hz)
- ***temp_3***: ambient temperature (°C, sampling rate 500 Hz)

Each file **sXX_YYYY_targets** contains the target time series. The first column contains a timestamp. The other contains the target time series:

- ***ecg***: 3-channel ECG sampled at 500 Hz

The creation process of the dataset consisted in 3 main steps:

1. **Feature Matrix Creation**
2. **Data Augmentation**
3. **Feature Selection**

2.1 Feature Matrix Creation

All data from each recording were imported into MATLAB.

All timeseries type files were scanned.

Within each of these files, **11 features were extracted from** each of the **physiological signals.**

These features were extracted in **both time and frequency domains** through the ***extract_feature*** function.

The extracted features are:

TIME DOMAIN	FREQUENCY DOMAIN
Minimum	Mean
Maximum	Median
Mean	Occupied Bandwidth
Median	
Variance	
Kurtosis	
Skewness	
Interquantile range	

We **tried extracting** these features in **three different ways** from individual signals, in order to see which technique produced better results.

In particular, in the **first place** we **extracted** the features **from the entire signal**, thus extracting 11 features from each signal, for a total of 121 features.

Then we chose to **partition the number of samples**, from the individual signal **into 4 contiguous windows**, and in doing so we were able to extract 44 features from each signal, for a total of 484 features.

Finally, we used the **technique of windows again** here, but by arranging them **in an overlapped** instead of contiguous **way**. we partitioned the signal **into 7 windows**, and by doing so for each signal we were able to obtain 77 features for a total of 847 features.

We compared these techniques during the feature selection (Section 2.3) and we saw that **we obtained better results**, through criterion value comparison, **using the third technique**.

Using the third technique, we had a discrete number of features for each recording, which is 847 features.

Among these features, some might have been redundant, so we chose to eliminate them through an **analysis of their correlation**.

Thus, as we can see from this code snippet, **we wrote a function that checks the correlation between the features in the matrix**. If one or more **features** are found to be correlated **with a value greater than 0.90** then only one is kept and the remaining ones **are discarded**.

```
1.  function [FEATURES_without_correlated_columns] = remove_correlated_features(FEATURES)
2.      correlation_coef = corrcoef(FEATURES);
3.      [correlated_features, ~] = find( tril( (abs(correlation_coef) > 0.9), -1 ) );
4.      correlated_features = unique(sort(correlated_features));
5.
6.      FEATURES_without_correlated_columns = FEATURES;
7.      FEATURES_without_correlated_columns(:, correlated_features) = [];
8.
```

This reduced the number of features from 847 to 277.

Finally, to complete the creation of the matrix, **we performed a matrix normalization operation**. This operation transforms the data, such that the values are distributed on an even scale.
This helps the training algorithm to obtain more accurate results, by preventing high variations affecting the rest of the data in a negative way.
All sample data for a given feature **were normalized between 0 and 1**. This process was repeated for all features.

In addition, such as for the timeseries type files, all target files were also scanned, and **two target vectors were created**, where **in one we have the mean ECG** value of each recording and **in the other** we have its **standard deviation**.

2.2 Data Augmentation

The dataset obtained in the previous step consists of only *66 samples*, which are not enough to properly train the neural networks of Task 3. In order to obtain more samples, we executed a **data augmentation** process of the dataset.

The data augmentation was performed using an **autoencoder**, i.e. a neural network which is trained to replicate its input at its output. Each input sample generates *50 new samples*, so that the final dataset will be composed of **3366 samples** (enough for the aim of this projects).

The autoencoder was configured in the following way:

- Encoder/decoder functions are linear (*satlin* for encoding and *purelin* for decoding) to reduce distortion from original sample.
- The desired proportion of training examples a neuron reacts to is set to **100%** and he sparsity regularization was set to **zero**, so that all the hidden neurons learn how to replicate the input features.

```
function data = data_augmentation(input)

% Constants
hidden_layer_size = round(size(input, 2) / 4);  % = 69
samples_to_generate = constants.autoenc_samples_to_generate; % = 50

% Prepare input data
input = input';
data = zeros(size(input, 1), size(input, 2) * samples_to_generate);
data(:, 1:size(input, 2)) = input;

% Train autoencode
for i=1:samples_to_generate
    autoenc = trainAutoencoder(input,...
                               hidden_layer_size,... Hidden layer size
                               'EncoderTransferFunction', 'satlin', ...
                               'DecoderTransferFunction', 'purelin', ...
                               'L2WeightRegularization', 0.001, ...
                               'SparsityRegularization', 0, ...
                               'SparsityProportion', 1, ...
                               'ShowProgressWindow', false ...
                               );

    output_autoenc = predict(autoenc, input);
    fprintf('Error of step %i: %f\n', i, mse(output_autoenc, input));
    start_index = size(input, 2) * i + 1;
    end_index = size(input, 2) * (i + 1);
    data(:, start_index : end_index) = output_autoenc;

end

% Prepare output
data = data';

end
```

2.3 Feature Selection

To identify the most relevant features to predict the output, **we** chose to **test the predictive power of each feature.**

The technique we used is the **Sequential Feature Selection**. This technique is already implemented in MATLAB through ***sequentialfs***, using an MLP as the criterion function as we can see in the snippet code below.

We chose the best 10 features as the maximum number of features. This was **repeated for both the network** that has to predict the mean value of the ECG and the network that has to predict the std of the ECG.

```
1. function performance = sequentialfs_criterion (input_train, target_train, input_test,
   target_test)
2.
3.     input_train = input_train';
4.     target_train = target_train';
5.     input_test = input_test';
6.     target_test = target_test';
7.
8.     % Create a fitnet network
9.     net = fitnet(constants.sequentialfs_hidden_layer_size);
10.    net.divideParam.trainRatio = 1;
11.    net.divideParam.testRatio = 0;
12.    net.divideParam.valRatio = 0;
13.    net.trainFcn = 'trainbr';
14.    net.trainParam.showWindow = 0; % Disable GUI
15.
16.    % Train network
17.    net = train(net, input_train, target_train);
18.
19.    % Test network
20.    y_test = net(input_test);
21.    performance = perform(net, target_test, y_test);
22.
23.end
```

The results achieved are different for the two different networks and are:

MLP Estimating MEAN ECG		MLP Estimating STD ECG	
FEATURE #	**CRITERION VALUE**	**FEATURE #**	**CRITERION VALUE**
234	11.6561	216	4175.92
186	8.79622	249	2980.41
138	6.05457	267	2240.34
199	5.31239	251	1791.45
71	4.05988	186	1441.29
149	3.52051	214	1195.2
210	3.09568	220	1040
93	2.86499	148	918.089
156	2.39744	48	772.537
244	2.27067	104	660.217

3. Artificial Neural Networks

3.1 Estimating ECG using Neural Networks

In this section we will discuss the design and implementation of the models required in section 3.1 of the specification, which include:

- The creation of a neural network that can estimate the mean value and standard deviation of ECG based on the provided dataset
- The creation of two Radial Basis Functions (RBFs) that can estimate the mean value and standard deviation like in the previous point.

3.1.1 Multi-Layer Perceptron Neural Networks

In order to find the best architecture for the neural network, we decided to use the **fitnet** to predict both the mean value and the standard deviation.

Is important to notice that **every experiment has been conducted on augmented data** to increase the amount of data on which the model gain the knowledge needed to correctly predict the targets, since the provided amount of data was not enough to train the net and obtain good results.

To test the model, we split the dataset into two ways:

1. For the algorithms that use a validation set:
    a. 70% for training
    b. 15% for validation
    c. 15% for testing
2. For the algorithms that does not need a validation set:
    a. 85% for training
    b. 15% for testing

To get the best possible performance from the network, we tried tuning by changing the training algorithm and the number of neurons. Some of the results obtained testing the algorithms are:

TRAINING ALGORITHM	RESULTS FOR MEAN VALUE PREDICTION	RESULTS FOR STD PREDICTION
trainlm	0.96879	**0.98364**
trainbr	**0.98428**	0.96333
trainbfg	0.88427	0.92017
trainrp	0.96754	0.91937
trainscg	0.96402	0.88534
traincgb	0.97351	0.93188

Which ended up returning best results using **'trainbr' for the mean value** prediction and **'trainlm' for the std** prediction.

Finally, we tested some values that the hidden layer could take. In particular, we tested values in the range [30, 70] with a step of 5. We concluded that **the optimal** number of neurons **to predict the std is 60 while for the mean value is 50**.

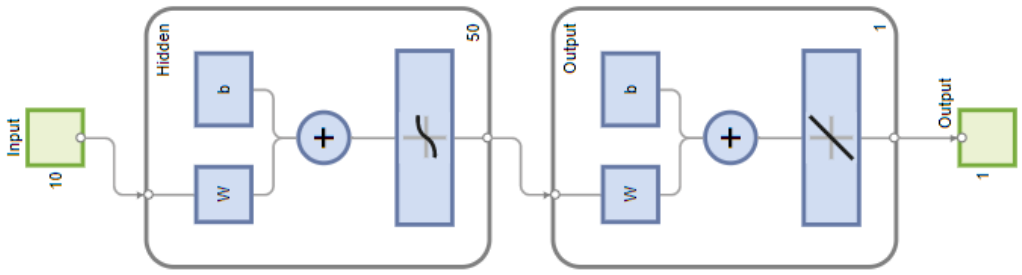

Figure 1: Structure of the net used to predict the ECG mean value

And the results are showed in figure 2 and 3 respectively for mean value and standard deviation:

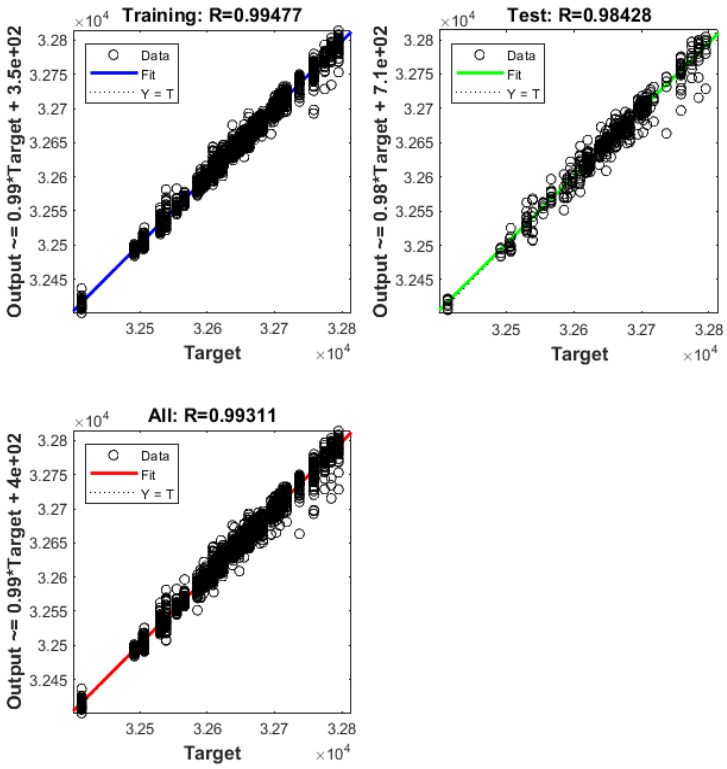

Figure 3: Results of the mean ECG value

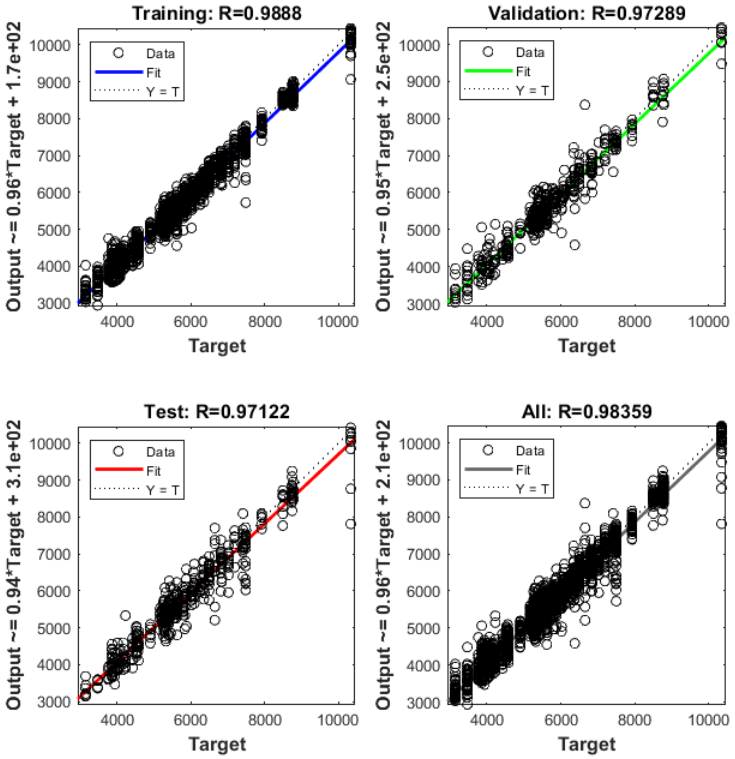

Figure 2: Results of the std ECG value

3.1.2 Radial Basis Function networks

The last step that concerns the 3.1 section of the specifics is the design of two **Radial Basis Function networks**, which are expected to estimate the mean and the standard deviation of the ECG of a person, just like the two MLPs in 3.1.1 section.

We tried four different approaches, during the design of the two RBFNs:

- Exact RBFN (`newrbe`), i.e. a RBF network with zero errors on the design vectors.
- RBFN (newrb), i.e. iteratively generates a RBFN by adding *radbas* neurons until the mean squared error falls below a given goal or the max number of neurons is met.
- Generalized Regression Neural Network (`newgrnn`).
- **RBFN (`newrb`) trained with Bayesian Regularization (`trainbr`)**.

The fourth approach (generate a RBFN network with **`newrb`** command and then train it with **`trainbr`** algorithm) yielded the best performance, since allowed a finer tuning with less neurons. This characteristic, together with the Bayesian regularization, will prevent the network from overfitting.

For both networks, we decided to set the **max number of *radbas* neurons** to **30**, since it tuned out to be a good tradeoff point for avoiding overfitting and allowing the network to learn the characteristics of dataset.

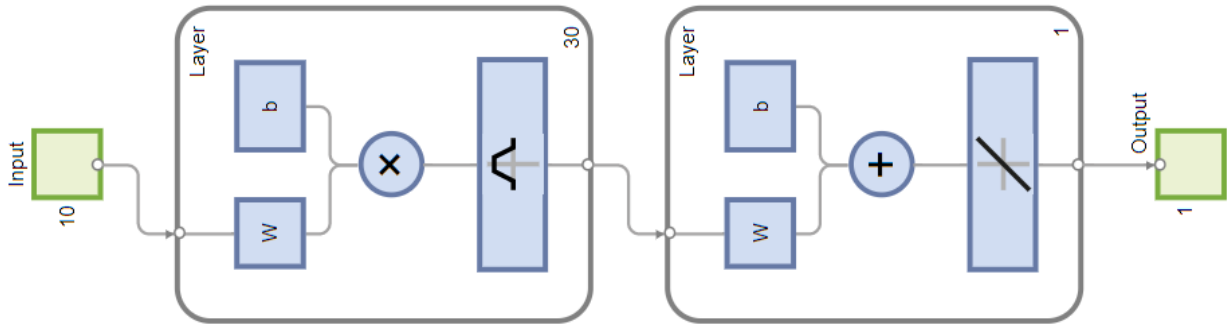

Figure 4: RBF network's architecture

We then tuned the **spread** value for both networks, by trying different value between *the minimum and the maximum* distance between points in the input space.

MEAN			STD		
SPREAD	**R-VALUE AFTER NEWRB**	**R-VALUE AFTER TRAINBR**	**SPREAD**	**R-VALUE AFTER NEWRB**	**R-VALUE AFTER TRAINBR**
0.3	0.80306	0.95291	0.25	0.81697	0.95919
0.4	0.83271	0.97239	0.4	0.88806	0.97839
0.45	**0.78172**	**0.97684**	0.5	0.90471	0.97906
0.5	0.73785	0.96903	**0.6**	**0.90803**	**0.97991**
			0.65	0.90090	0.97641
			0.7	0.90067	0.97731

Thus, the optimal spread values are:

- **0.45** to estimate the mean value of the ecg;
- **0.6** to estimate the standard deviation of the ecg.

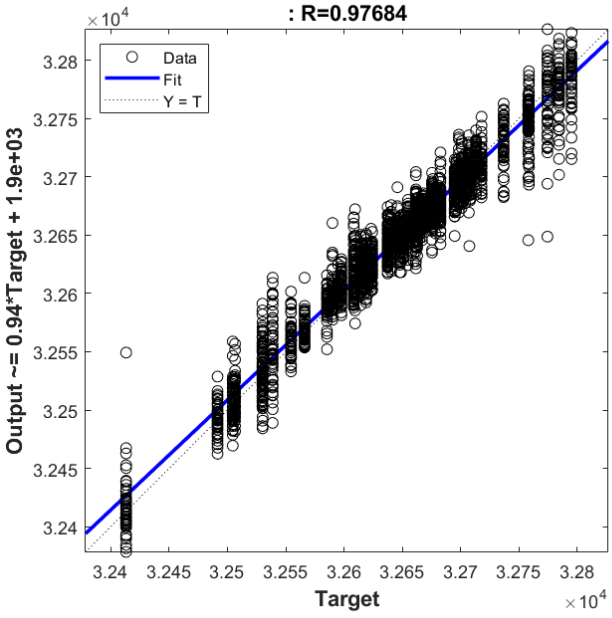

Figure 5: Regression plot of the estimation of the mean

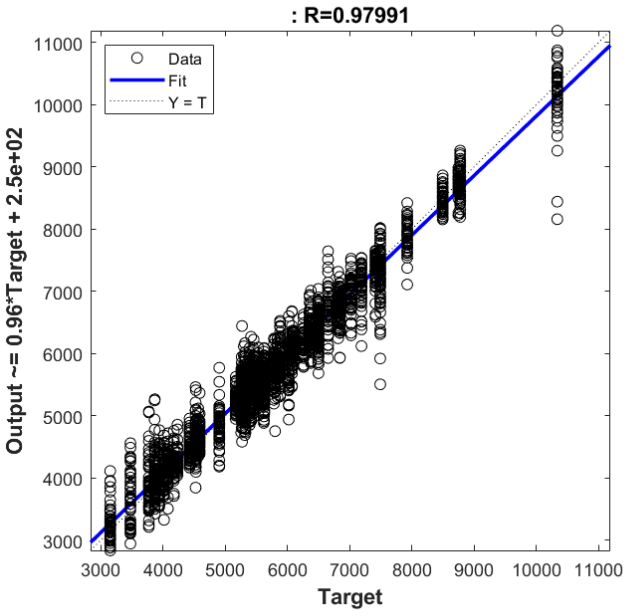

Figure 6: Regression plot for the estimation of the standard deviation

3.2 Determining a person's activity using an MLP neural network

In this section we face the problem of the **design and the development** of a **multi-layer perceptron** (MLP) that **can classify the activity**, which the person was performing during the recordings.

We used ***patternnet*** to generate a pattern recognition network to predict the activity of the subject.

As we mentioned in Section 3.1.1, again each **experiment was conducted on augmented data** to increase the number of data and have an **improvement in performance**.

We chose to provide as **input** to the network a feature matrix composed of **19 features** that represent the union of those selected in section 2 during feature selection, and **3366 samples** which are the result of data augmentation.

As a **target**, we created a vector containing the **class labels representing the different activities** performed by the subjects. Then we **brought the target vector into a vector form**.

The **3 classes** are distinguished as follows:

Three-Class Classifier		
ACTIVITY	**CLASS LABEL**	**TARGET**
SIT	1	100
WALK	2	010
RUN	3	001

After several tests, we observed that the **best results** were achieved by **this neural network**:

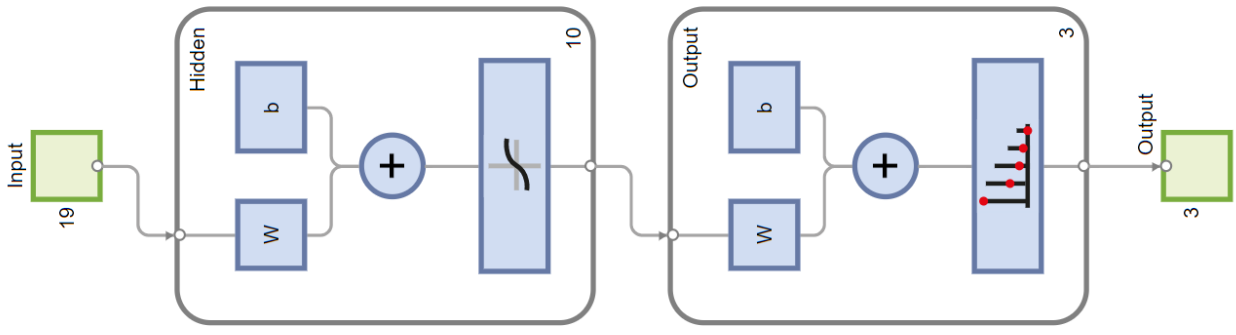

Figure 7: Network Architecture

In this neural network we used the **Bayesian regularization backpropagation** as **Training Function**.

With a **number of neurons**, in the **hidden layer** equal to **10**.

We **divided the data for training, validation and testing in this way**, as we can see in the snippet code, noting that Bayesian regularization does not require a validation dataset:

```
net.divideFcn = 'dividerand';  % Divide data randomly
    net.divideMode = 'sample';  % Divide up every sample
    net.divideParam.trainRatio = 70/100;
    net.divideParam.valRatio = 0/100;
    net.divideParam.testRatio = 30/100;

```

This neural network has led to these results:

Figure 8: Confusion Matrix

To reach these results, as we anticipated earlier, we performed several tests.

These tests **mainly focused on 3 aspects**:

- The **number of hidden neurons**;
- The **training function**;
- The **partitioning of the data** for Training, Validation, Testing

For the first aspect, we **ranged the number of hidden neurons** in the **interval between 9 and 12**, following heuristic guidelines, which recommended starting with a number of neurons equal to about the half of the input size, until reaching 2/3 of the input size.
We found the **best performance with a number of hidden neurons equal to 10**.

For the **second and third aspects**, we **compared 3 different types of training functions**, **changing** the **partition of the data** for each training function **until we reached the optimum**.

The results are collected in the table below, and we can see how the best algorithm is **trainbr**, by comparing the results obtained.

Percentage Correct Predictions				
TRAINING FUNCTION	**TRAINING**	**VALIDATION**	**TEST**	**ALL**
Bayesian Regularization	**100%**	**NaN**	**98.8%**	**99.6%**
Levenberg-Marquardt	99.4%	98.6%	99.2%	99.3%
Scaled Conjugate Gradient	98.4%	98.4%	97.6%	98.3%

3.3 Fuzzy Inference System

This part aimed to design and to develop a fuzzy inference system to classify a person's activity into *sit*, *walk* or *run* classes.

The goal was reached by studying the distribution of the **three most relevant features** in the set of features used to train the MLP classifier. The features were chosen by executing sequentialfs procedure on a MLP classification network (`patternet` with five hidden neurons).

Sequentialfs Results	
FEATURE #	**CRITERION VALUE**
3	0.000844791
12	0.000721586
9	0.0006458

We built two separate FIS:

- A **Mamdani fuzzy system**, built by hand with the *Fuzzy Logic Design* graphical tool.
- A **TSK fuzzy system**, automatically generated with a *grid-partitioning* approach (`genfis`) and tuned with an *adaptive neuro-fuzzy algorithm* (`tunefis`).

3.3.1 Mamdani Fuzzy System

We started by studying the distributions of the features through their histograms, which we can see in the following picture:

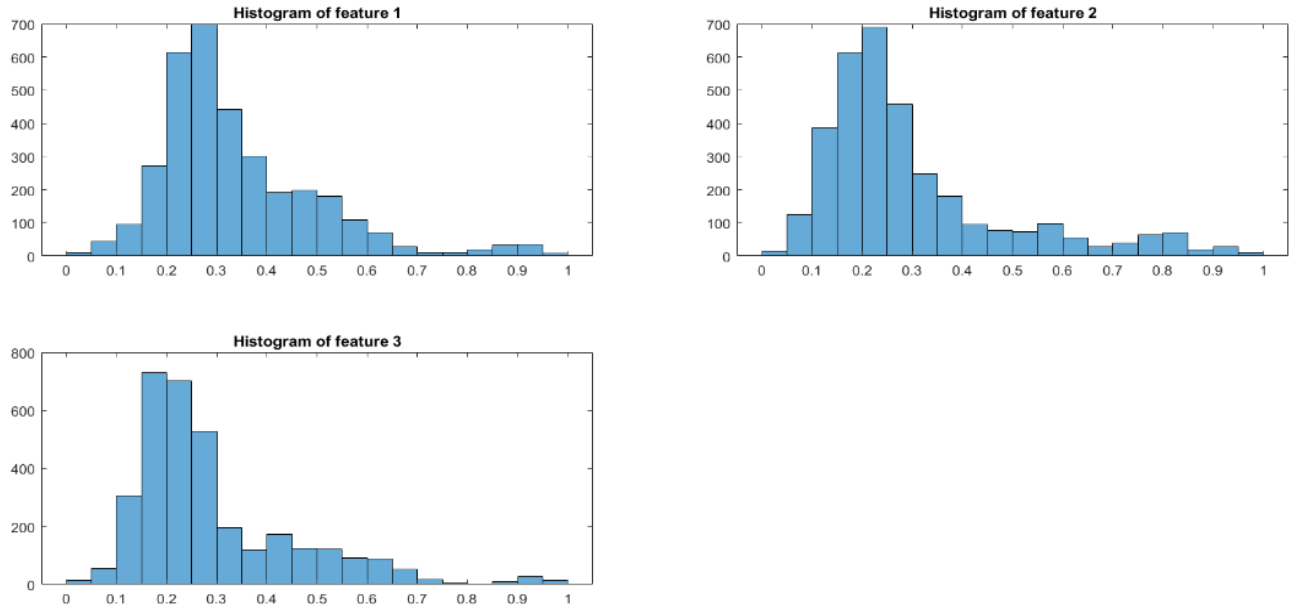

Figure 9: Distribution of the three features for Mamdani FIS

These distributions have been used to *model the linguistic variables* of the three features. Since the distributions are skewed, we divide the feature space into **four zones**, depending on the position relative to the peak and values assumed inside the zone:

- *low* (left zone, with small values)
- *medium* (zone around the peak)
- *medium-high* (zone to the immediate right of the peak, with average values)
- *high* (far right zone, with small values)

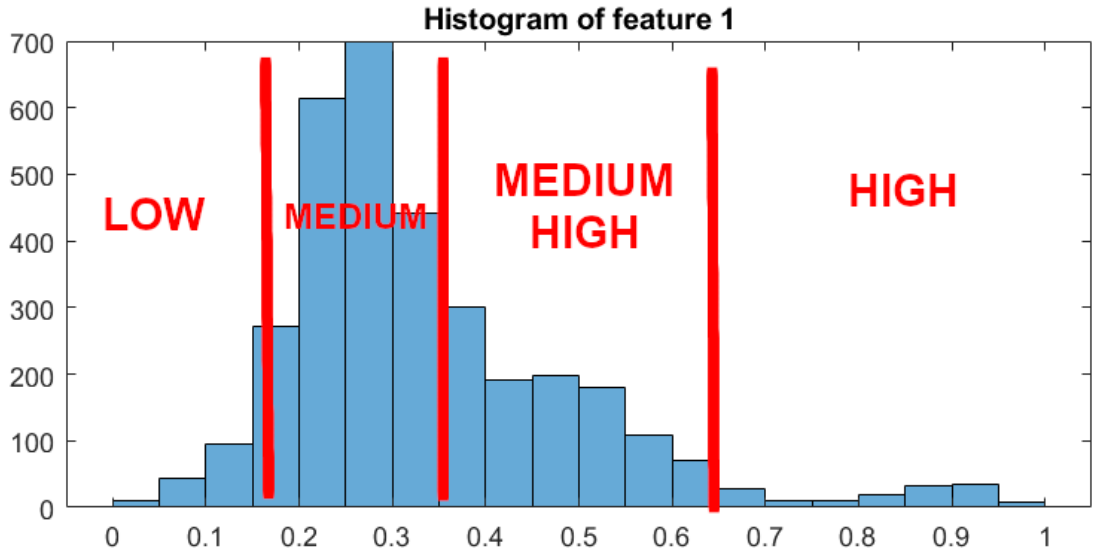

Figure 10: Division of Feature 1 interval into four zone

We then generated **four membership functions** as sigmoidal function (**`sigmf`**) for "low" and "high" zones and **`dsigmf`** for "medium" and "medium-high". These functions were chosen because of their *smoothness* (so that the zones' borders could overlap) and their *high customizability* (they are asymmetric and have more parameters than a generalized bell function). That allowed us to better calibrate the fuzzy rules.

We modeled the membership function by fitting the plateau section inside the selected zone and by calibrating the exponential part so that it would overlap with the contiguous zones, accordingly to the distribution of the feature.

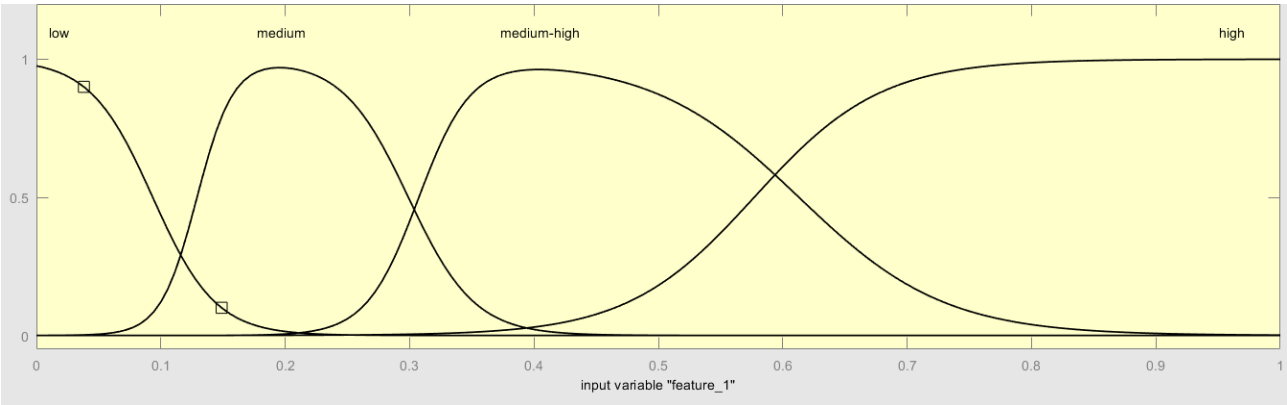

Figure 11: Membership functions of feature 1

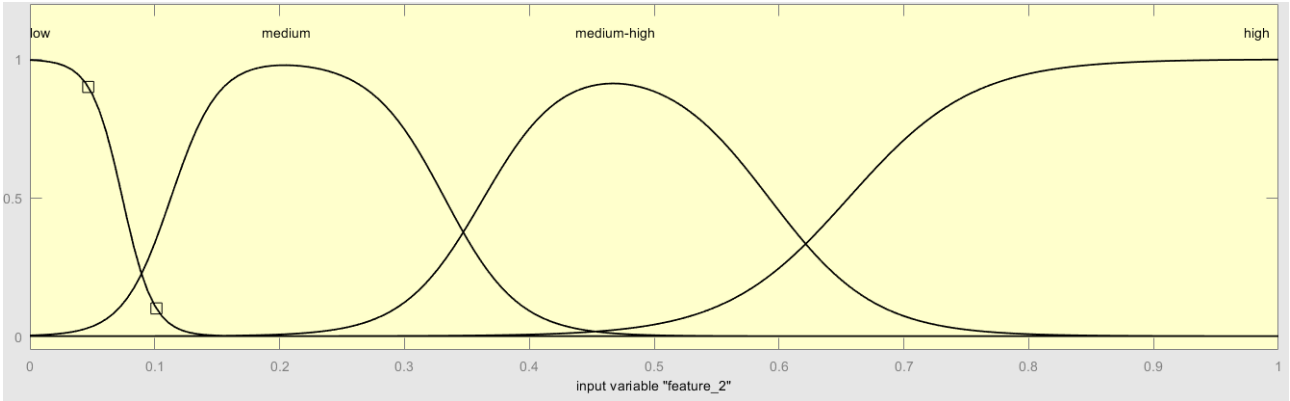

Figure 12: Membership functions of feature 2

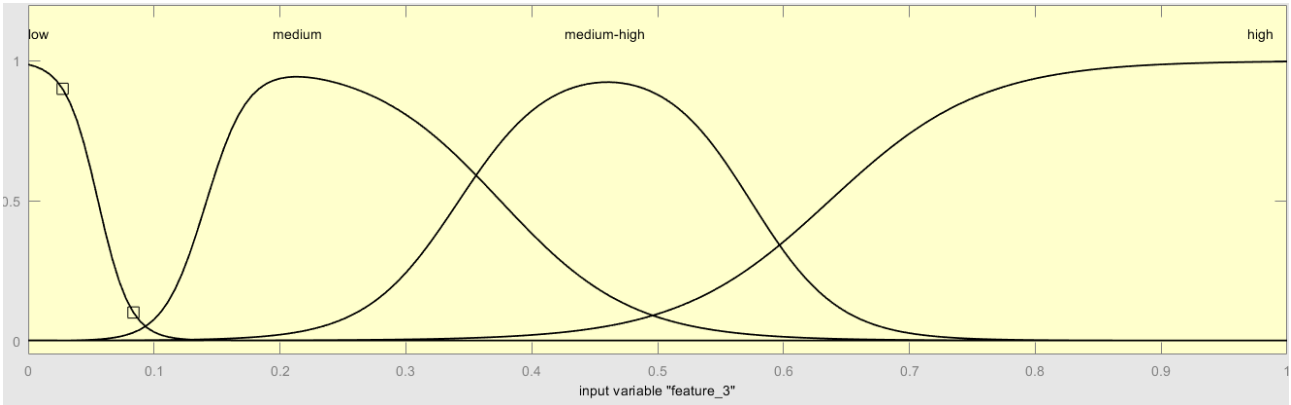

Figure 13: Membership functions of feature 3

For the output membership functions, three generalized bell functions (*sit, walk, run*) were adopted due to their smoothness and symmetry.

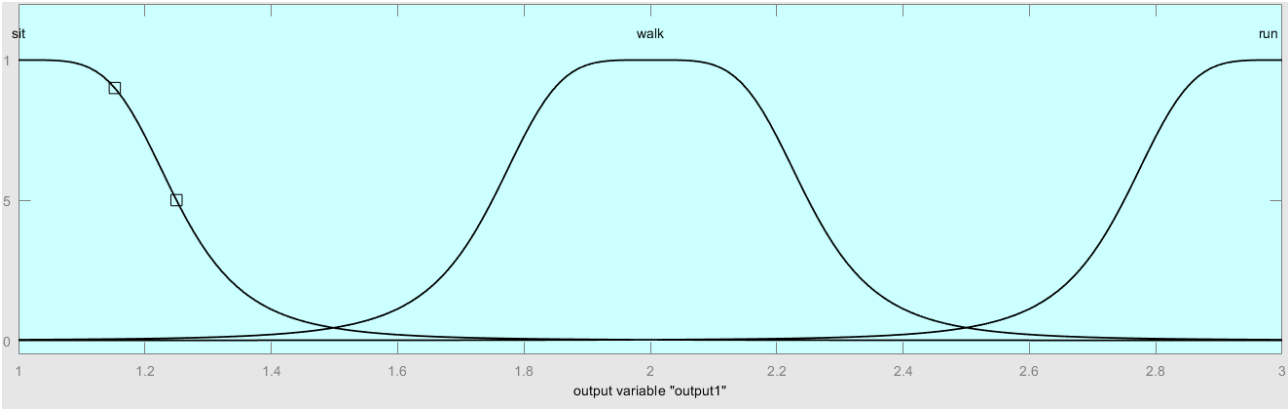

Figure 14: Membership function of the output variable
SIT = 1, WALK = 2, RUN = 3

Finally, to write the fuzzy rules we analyzed the distribution of the three features for each person's activity.

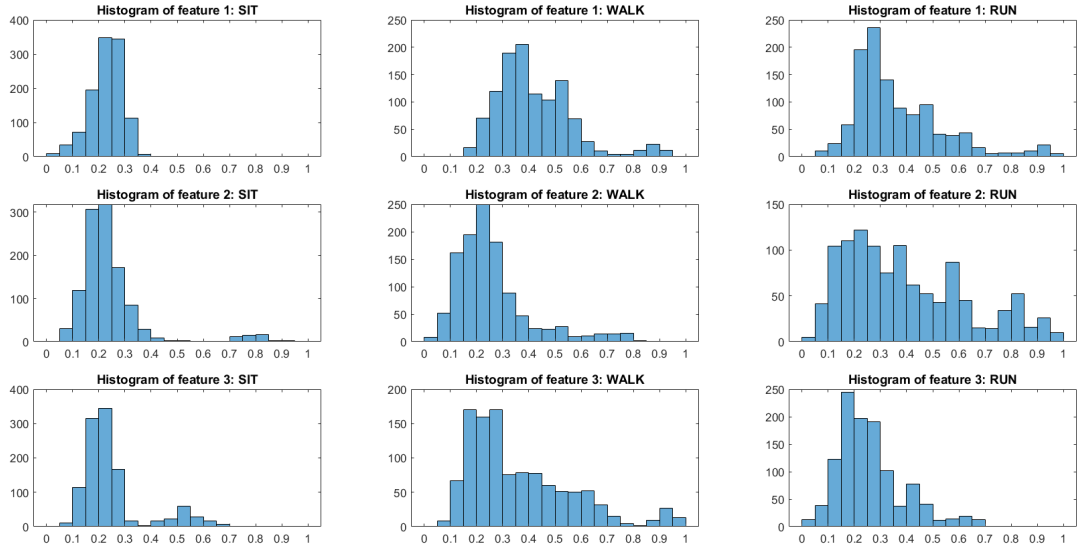

Figure 15: Distribution of features for each activity class

Since the distributions are quite similar to each other, **16 rules** with different weights were necessary to achieve an acceptable accuracy rate.

1. If (feature_1 is medium) and (feature_2 is medium) and (feature_3 is medium) then (output1 is sit) (0.8)
2. If (feature_1 is medium) and (feature_2 is medium) and (feature_3 is medium-high) then (output1 is sit) (0.5)
3. If (feature_1 is medium) and (feature_2 is high) and (feature_3 is medium) then (output1 is sit) (0.5)
4. If (feature_1 is medium) and (feature_2 is high) and (feature_3 is medium-high) then (output1 is sit) (0.5)
5. If (feature_1 is medium) and (feature_2 is medium) and (feature_3 is medium) then (output1 is walk) (0.5)
6. If (feature_1 is medium) and (feature_2 is medium) and (feature_3 is high) then (output1 is walk) (1)
7. If (feature_1 is medium-high) and (feature_2 is medium) and (feature_3 is medium) then (output1 is walk) (1)
8. If (feature_1 is medium-high) and (feature_2 is medium) and (feature_3 is medium-high) then (output1 is walk) (0.8)
9. If (feature_1 is medium-high) and (feature_2 is medium) and (feature_3 is high) then (output1 is walk) (1)
10. If (feature_1 is high) and (feature_2 is not high) then (output1 is walk) (0.5)
11. If (feature_2 is high) and (feature_3 is medium) then (output1 is run) (1)
12. If (feature_1 is medium) and (feature_3 is medium-high) then (output1 is run) (0.5)
13. If (feature_1 is medium) and (feature_2 is medium) and (feature_3 is medium-high) then (output1 is run) (1)
14. If (feature_1 is medium) and (feature_2 is medium-high) and (feature_3 is medium) then (output1 is run) (1)
15. If (feature_1 is medium-high) and (feature_2 is medium-high) and (feature_3 is medium) then (output1 is run) (1)
16. If (feature_1 is high) and (feature_2 is medium-high) and (feature_3 is medium) then (output1 is run) (1)

Figure 16: Fuzzy rules

Regarding the configuration of the fuzzy inference system (and, or, implication, aggregation, defuzzification methods), we adopted the standard methods for a Mamdani system (since they yielded the best performance), except for the **defuzzification** method, which was set to ***Largest of Maximum*** (lom).

In fact, the centroid and bisector methods tended to favor the *walk* class at the expense of the *sit* and *run* classes, since it is positioned in the center of the output space and the input features weren't splittable enough). On the other hand, the three methods which exploits the maximum value (MOM, LOM and SOM) were better at classifying the given samples. The chose the LOM because it yielded the best performance.

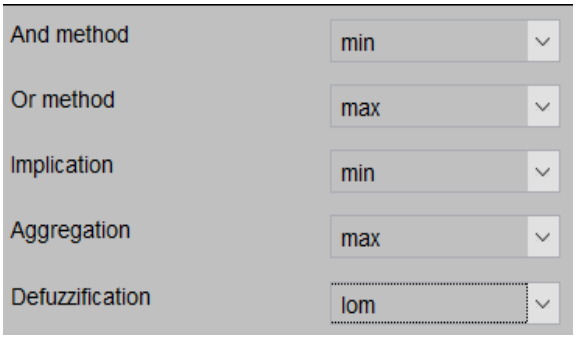

Figure 17: FIS configuration

METHOD	SIT	WALK	RUN	AVERAGE
Centroid	0 %	95.544 %	24.153 %	39.899 %
Bisector	0 %	94.029 %	40.107 %	44.712 %
MOM	69.875 %	77.718 %	58.378 %	68.657 %
LOM	**69.519 %**	**76.560 %**	**62.567 %**	**69.548 %**
SOM	70.143 %	77.451 %	57.754 %	68.449 %

In order to classify the sample, the output of defuzzication process was **rounded to the closest integer**.

In the following picture, we show the final FIS seen from the “Rule Viewer” tool.

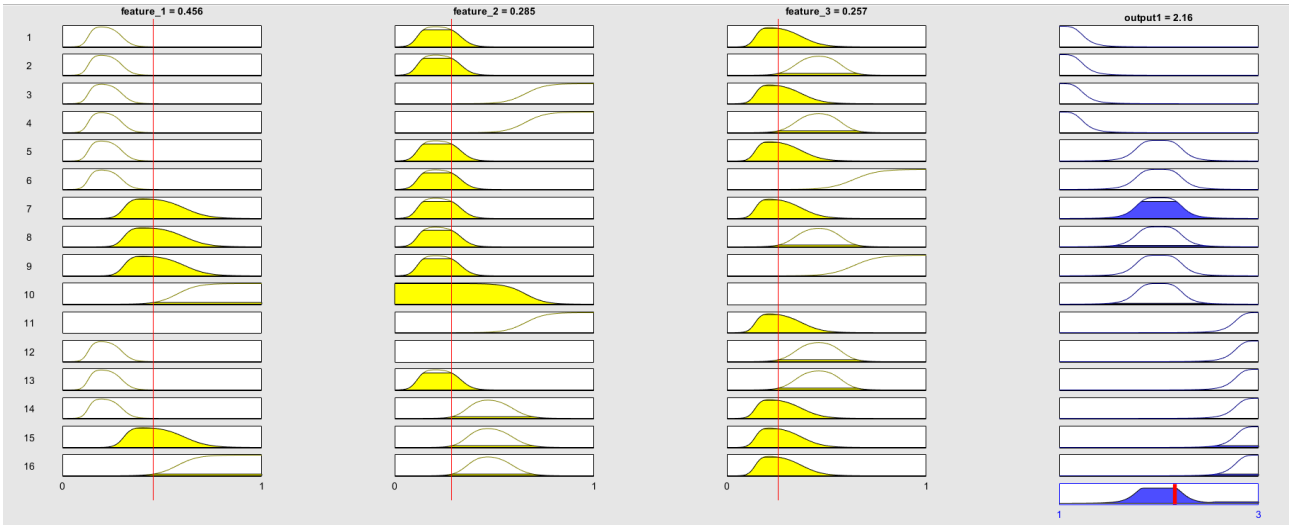

Figure 18: Rule viewer

3.3.2 TSK fuzzy system

Besides designing a Mamdani fuzzy system by hand through the *Fuzzy Logic Designer* toolbox, we tried another approach: automatically generate a FIS from data (`genfis` function) and tune it with an adaptive neuro-fuzzy algorithm (`tunefis` function).

```
% Generate FIS
genfis_options = genfisOptions('GridPartition', ...
                               NumMembershipFunctions = 5);

fis_in = genfis(x, y, genfis_options);

% Tune FIS
[in, out, ~] = getTunableSettings(fis_in);
tunefis_options = tunefisOptions('Method','anfis');
tunefis_options.MethodOptions.EpochNumber = 50;
fis_out = tunefis(fis_in, [in; out], x, y, tunefis_options);

% Test FIS
test_fis(fis_out, x', y');
```

Figure 19 - MATLAB code: generate and tune TSK fuzzy inference system

First, we generate a completed fuzzy system with input/output membership functions and rule set. We have compared different clustering methods, in order to select the best one:

METHOD	SIT	WALK	RUN	AVERAGE
GridPartitioning (4 MF)	72.727 %	87.166 %	64.528 %	74.807 %
GridPartitioning (5 MF)	**79.679 %**	**91.711 %**	**74.777 %**	**82.056 %**
SubtractiveClustering	41.890 %	89.305 %	39.305 %	56.833 %
FCMClustering	54.011 %	85.294 %	38.414 %	59.239 %

As we can see, the *Grid Partitioning* method with **5 input membership functions** yields the best results, far superior to the performance achieved in the previous section.

4 Deep Neural Networks

In this section we will discuss the design and implementation of the models required in section 4 of the specification, which include:

- The creation of a **convolutional neural network** that is able to estimate the **standard deviation** of the ECG from the dataset provided to us. We chose to estimate the standard deviation because it is the one that returned lower accuracies via the MLP
- The creation of a **recurrent neural network** that is able to **predicts one value of a person's ECG**, based on dataset signals
- The creation of a recurrent neural network that is able to **predict a set of consecutive future values** of a person's ECG based on dataset signals

4.1 Improve ECG estimation using a convolutional neural network

In this section we worked again on the estimation of ECG values, but we used a **Convolutional Neural Network**. We designed a CNN which estimates the **standard** deviation of the ECG signal, since the corresponding MLP in section 3.1.1 achieved the worst performance.
To improve the performance, we executed a **data augmentation** process on the dataset, i.e. we **divided** each **biophysical signal into windows of 5000 time steps**. All the windows were grouped and given as input to the network for training.

To find the best CNN architecture, which would provide us the best results, we proceeded through **a few main steps** that brought consistent improvements at each step to the network and **led us to the final neural network**.

These main steps involved the modification of the network itself, its layers, and the values of its hyperparameters and they are:

- **Modification of the number of convolutional layers**
- **Modification of the number and size of filters**
- **Change of the training algorithm**
- **Change of data normalization algorithm**

Before proceeding to the analysis of each step, we need to describe the base architecture of the neural network on which we applied our adjustments to find the best achievable network.

4.1.1 Starting Architecture

The **base block** of our Convolutional Neural Network architecture is composed by:

- A **1-D convolution layer**, which applies sliding convolutional filters to 1-D input.
- A **batch normalization layer**, to speed up training of the convolutional neural network, reduce the sensitivity to network initialization and prevent the exploding gradient problem. We chose a **`batchNormalizationLayer`** instead of a **`layerNormalizationLayer`** because it was more suitable for our problem according to MATLAB documentation.
- A **leaky ReLU layer**, to **avoid the vanishing gradient problem**.
- A **1-D max pooling layer**, which performs downsampling by dividing the input into 1-D pooling regions, then computing the maximum of each region.

The blocks are connected in a cascade fashion. The **last block is connected to**:

- A **1-D global average pooling layer**, which performs downsampling by outputting the average of the time dimension of the input.
- A **fully connected layer** with a **leaky ReLU** activation function (hidden layer of the MLP).
- A **fully connected layer** with a **regression** activation layer (output layer of the MLP).

All of these blocks and connection can be observed in this code snippet:

```
%% Define the layers for the net
% This gives the structure of the convolutional neural net
numFilters = 16;

layers = [
    sequenceInputLayer(11)

    convolution1dLayer(7, numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    leakyReluLayer
    maxPooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    convolution1dLayer(5, 2 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    leakyReluLayer
    maxPooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    globalAveragePooling1dLayer

    fullyConnectedLayer(100)
    leakyReluLayer
    fullyConnectedLayer(1)

    regressionLayer
];

```

The base architecture configuration has been chosen by trial-and-error, in particular the type of the local and global pooling layers (max or average)

The Options for the training of our neural network are described in this code snippet:

```
options = trainingOptions('adam', ...
    ...
    MaxEpochs = 30, ...
    MiniBatchSize = 80, ...
    Shuffle = 'every-epoch' , ...
    ...
    InitialLearnRate = 0.01, ...
    LearnRateSchedule = 'piecewise', ...
    LearnRateDropPeriod = 10, ...
    LearnRateDropFactor = 0.1, ...
    L2Regularization = 0.01, ...
    ...
    ValidationData =  {XTest TTest}, ...
    ValidationFrequency = 30, ...
    ...
    ExecutionEnvironment = 'auto', ...
    Plots = 'training-progress', ...
    Verbose = 1, ...
    VerboseFrequency = 1 ...
);

```

Using this neural network and those training options, we obtained an **R-value of 0.43094 for validation and of 0.48817 for training**.

We took these values as a starting point and modified the network and tuned its values to achieve better performance

4.1.2 Modification of the number of Convolutional Layers

As a first step to improve network performance, we **increased the number of base blocks** of our CNN **from 2 to 4**, keeping the **number and size of filters identical**, and **not changing** the training **options**.

The structure of the new network can be seen in this code snippet:

```
numFilters = 16;

layers = [
    sequenceInputLayer(11)

    convolution1dLayer(7, numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    leakyReluLayer
    maxPooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    convolution1dLayer(5, 2 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    leakyReluLayer
    maxPooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    convolution1dLayer(5, 3 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    leakyReluLayer
    maxPooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    convolution1dLayer(3, 4 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    leakyReluLayer
    maxPooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    globalAveragePooling1dLayer

    fullyConnectedLayer(100)
    leakyReluLayer

    fullyConnectedLayer(1)

    regressionLayer
];

```

Using this neural network, we obtained an **R-value of 0.59142 for validation and of 0.78859 for training**.

4.1.3 Modification of the number and size of filters

Starting from the results previously obtained, we tried to **increase both the number of filters and the size of the filters for each convolutional layer**.

We also **added an additional base block**, which brought our neural network **to** have **5 convolutional layers.**

Another important change was **the addition of a Dropout Layer**. This Layer has the function of setting input elements to zero with a given probability, with the goal **to preventing overfitting.**

Also on this occasion we did not introduce consistent changes to the training options.

In the following code snippet, we can see the changes made to the network.
In particular we can observe the new number of filters and to their size for each layer, as well as the probability value of the dropout layer.

```
numFilters = 64;

layers = [
    sequenceInputLayer(11)

    convolution1dLayer(15, numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    leakyReluLayer
    maxPooling1dLayer(4, 'Stride', 4, 'Padding', 'same')


    convolution1dLayer(11, 2 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    leakyReluLayer
    maxPooling1dLayer(4, 'Stride', 4, 'Padding', 'same')


    convolution1dLayer(9, 3 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    leakyReluLayer
    maxPooling1dLayer(4, 'Stride', 4, 'Padding', 'same')


    convolution1dLayer(9, 4 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    leakyReluLayer
    maxPooling1dLayer(4, 'Stride', 4, 'Padding', 'same')


    convolution1dLayer(9, 5 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    leakyReluLayer
    maxPooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    globalAveragePooling1dLayer

    fullyConnectedLayer(100)

    dropoutLayer(0.3)

    leakyReluLayer
    fullyConnectedLayer(1)

    regressionLayer
];

```

Using this new neural network, we obtained an **R-value of 0.70129 for validation and of 0.88496 for training**.

Since we **had achieved a rather complex architecture** so far, we considered to modify other elements or values to see if we could improve performances.
Thus, we proceeded with our improvement process by **attempting to change** values in the **training options** and the **data generation**.

4.1.4 Change of the training algorithm

We tried to change the training algorithm type and consequently some network parameters to better optimize the behavior of different algorithms.
These changes were made in order to see if there would be an increase in performances.

However, **we saw that the other algorithms were performing worse**, as we can see from the following table.

R-value Per Training Algorithm		
ALGORITHM	**VALIDATION**	**TRAINING**
Adam	**0.70129**	**0.88496**
Rmsprop	0.70044	0.8593
Sgdm	0.33359	0.45831

Therefore, **we kept as training algorithm** "***Adam".***

4.1.5 Change of Data Normalization Algorithm

Having reached this point after fine-tuning both the architecture and the training options of the network, we tried to **change** some elements during **the data generation**, data, that is given as input to the network.

In particular, **we removed some particularly noisy recordings** such as the number 13, we also **removed some outliers**, but more importantly we **attempted** to **change the data normalization algorithm**, that has been used up to that point, **"zscore".**

We tried different algorithms and compared them, discovering as we can see from the following table that the best algorithm turns out to be ***"scale".***

R-value Per Data Normalization Algorithm		
ALGORITHM	**VALIDATION**	**TRAINING**
Scale (best)	**0.9522**	**0.95074**
Range	0.82644	0.91057
Norm	0.79378	0.83378
Medianiqr	0.73783	0.88704
Center	0.71153	0.81802
Zscore (current)	0.70345	0.87897

In this **plot** we can see the **performance trend** of **CNN during** the different **steps described** in the **previous sections.**

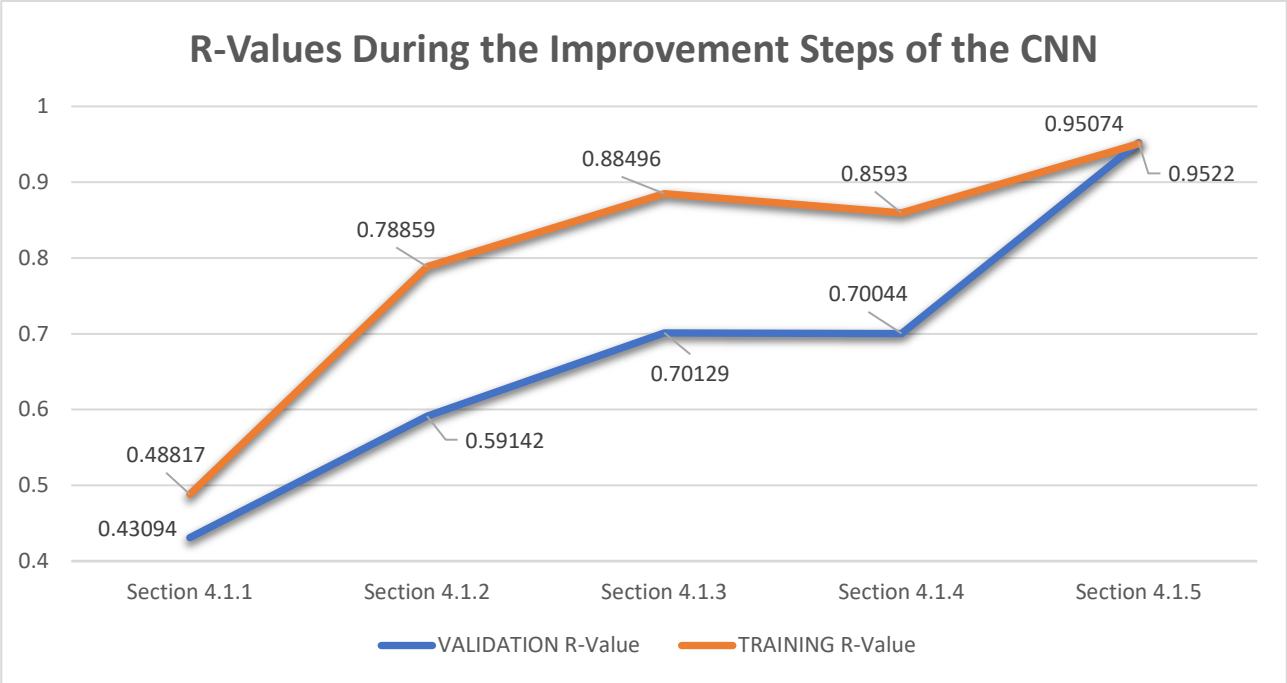

Figure 20: Performance Improvement of the CNN during the steps

Thus, the best **convolutional neural network** is the one that **have the architecture** described in section 4.1.3, "***adam***" as training algorithm (section 4.1.4), the training options defined in section 4.1.1 and "***scale***" as normalization algorithm (section 4.1.5). Here we can see the results:

- Training R-Value = 0.95074 (fig. 21)
- Validation R-Value = 0.9522 (fig. 22)

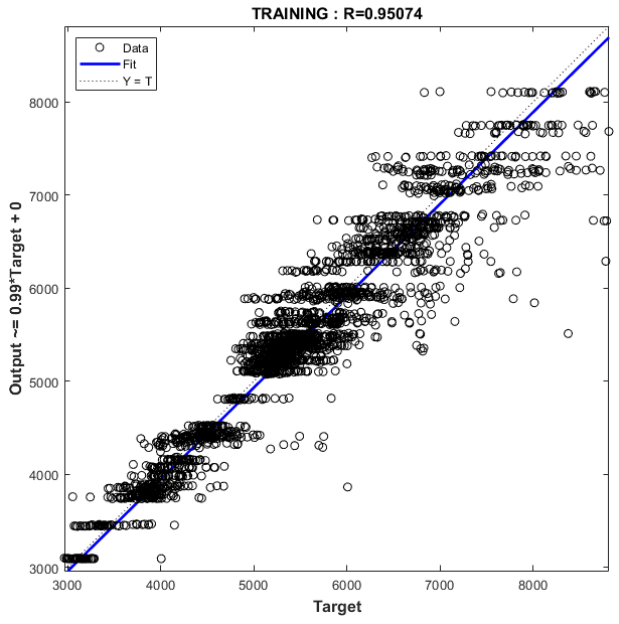

Figure 21: Training Regression Plot

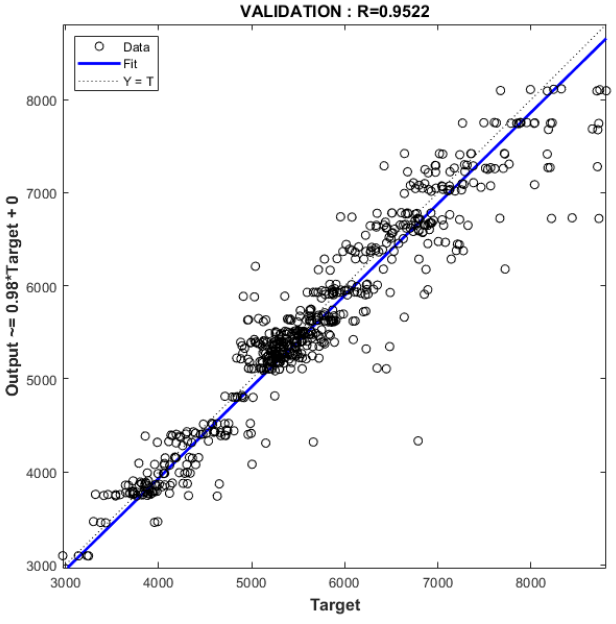

Figure 22: Validation Regression Plot

4.1.6 Additional improvements to the neural network

Although we had achieved some very good results, we still **attempted to perform other tests** to try **to maximize the performances**.

What we did was to **find the best combination** of **convolutional layer**, **number and size of filters**, **output size** of the **fully connected layers** and **probability value** of the **dropout layer**.

All these tests can be seen and compared in the table below.

Additional Tests to Maximize Performance of the CNN							
TEST #	**NUMBER CONV. LAYERS**	**NUMBER FILTERS**	**FILTER SIZE FOR CONV. LAYER**	**OUTPUT SIZE FULLY CONN. LAYER**	**PROB. DROPOUT LAYER**	**VALIDATION R-VALUE**	**TRAINING R-VALUE**
1	**4**	**64**	**[15 11 9 9]**	**[100 1]**	**0.3**	**0.95331**	**0.95102**
2	4	32	[15 11 9 9]	[100 1]	0.3	0.95196	0.95123
3	4	32	[15 11 9 9]	[50 1]	0.2	0.95226	0.95109
4	3	64	[15 11 9]	[80 1]	0.3	0.95327	0.95065
5	3	50	[7 5 5]	[80 1]	0.2	0.95272	0.95107
6	4	50	[7 5 5 5]	[100 1]	0.3	0.95246	0.95123

As can be seen from the table, **test number 1** turns out to be **the best among all the additional tests**, but it **also** turns out to be **better than the neural network** found earlier **in section** [4.1.5](#).

So **we** actually **managed to find** a **better performing** convolutional neural **network**.

In fact, if we compare it with the previous neural network, we see that **by decreasing the number of Convolutional Layers from 5 to 4**, **we** were able to **improve both** the R-Value of **Validation** by increasing it from 0.9522 **to 0.95331**, and the R-Value of **Training** by increasing it from 0.95074 **to 0.95102.**

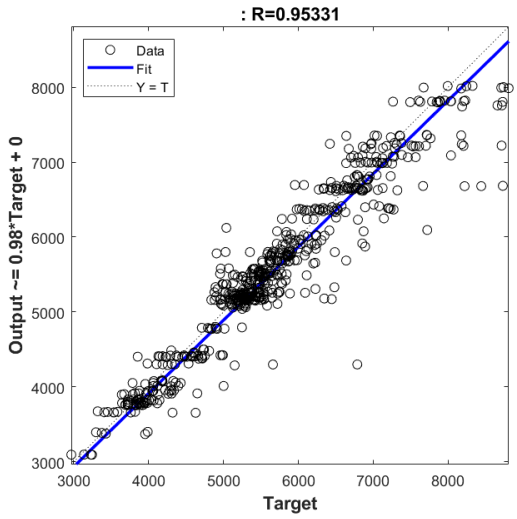

Figure 23: Training Regression Plot

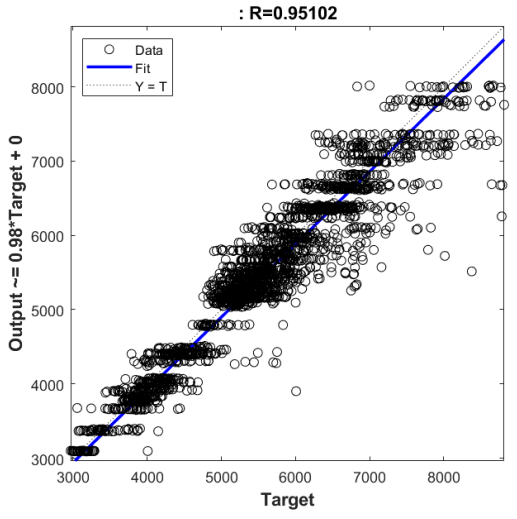

Figure 24: Validation Regression Plot

4.2 Predict ECG values using Recurrent Neural Networks (RNN)

As explained in the specification, the neural network receives as input one or more signals through the values they take in **various time intervals [t – k, t]**, where **t denotes the starting time** of the interval (window) and **k corresponds to the width of the window**.

Our first goal is to find the data normalization algorithm that will give us the best possible results. For this, we have done various experiments that led to the following results:

Algorithm	Results
'z-score'	RMSE on validation set: 0.057256
'scale'	RMSE on validation set: 0.147988
'range'	**RMSE on validation set: 0.023397**
'center'	RMSE on validation set: 5793.277832
'medianiqr'	RMSE on validation set: 0.078807

So 'range' turns out to be the optimal algorithm. Note that it does not turn out the 'norm' algorithm this is because it equalized the data by reducing the variation almost completely.

We then tuned **the window size**, we performed some tests by varying the window size in the range [10, 40] and keeping the number of neurons in the LSTM layer at 64. The results were the following:

Window size	Results
30	RMSE on validation set: 0.025443
40	RMSE on validation set: 0.024287
50	**RMSE on validation set: 0.022642**
60	RMSE on validation set: 0.025523

We also tuned the number of neurons in the LSTM layer. We performed tests by varying the neurons in the range [40, 64] in steps of 6. **The best results were obtained with the number of neurons set to 52**.

To conclude, we tested the training algorithms:

- ‘adam’: RMSE on validation set: 0.027281
- **‘rmsprop’: RMSE on validation set: 0.016576**
- ‘sgdm’: RMSE on validation set: 0.062173

The final network is as follows:

```
% Define LSTM Network Architecture
layers = [
    sequenceInputLayer(num_channels)
    lstmLayer(52, 'OutputMode', 'last')
    fullyConnectedLayer(num_channels)
    regressionLayer
];


% Specify Training Options
options = trainingOptions('rmsprop', ...
    MaxEpochs = 20, ...
    MiniBatchSize = 3000, ...
    Shuffle = 'every-epoch', ...
    ...
    ValidationData = {XTest TTest}, ...
    ValidationFrequency = 50, ...
    OutputNetwork = 'best-validation-loss', ...
    ...
    LearnRateSchedule = 'piecewise', ...
    LearnRateDropFactor = 0.1, ...
    LearnRateDropPeriod = 10, ...
    ...
    SequencePaddingDirection = 'left', ...
    ...
    Plots = 'training-progress', ...
    Verbose = 1, ...
    VerboseFrequency = 5, ...
    ExecutionEnvironment = 'auto' ...
);


```

It’s important to notice that the variable 'num_channels', which represents the number of signals given as input to the network, has been set to the value '1'. This is because, by running various tests, we concluded that the values that ecg itself takes before the value to be predicted are sufficient for the network to return a fairly accurate value.

In the following figure we can observe a portion of the predicted values compared with the expected values:

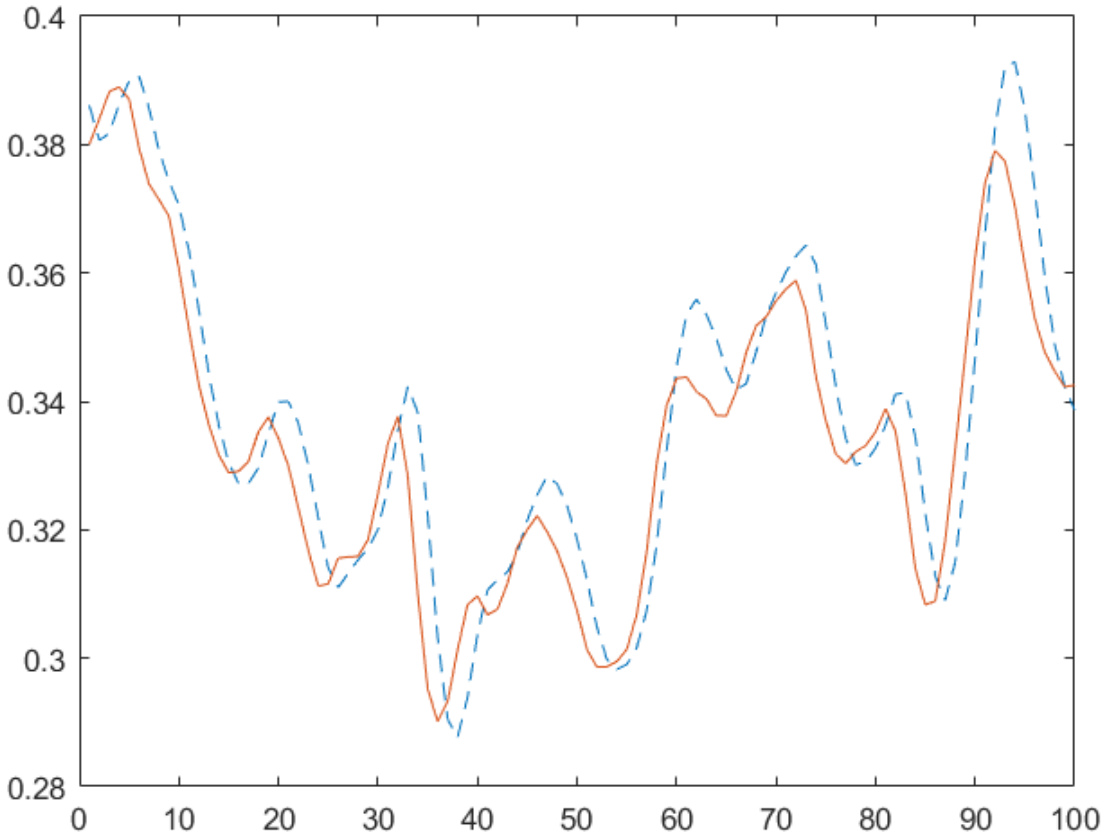

Figure 25: Results RNN one-step forecasting

Where the dotted line indicates the predicted values and the solid line the expected values.

4.3 Multi-Step Forecasting of ECG values

Like in the previous paragraph the goal is to create a recurrent neural network that, instead of predicting a single value, given a window is able to predict a range of values in the future. To do this **we give as input to the network for step t+1 what it produced as output at step t** (closed loop).

Before proceeding to model the network, we tested the various normalization algorithms using a 5 timestep prediction and obtained the following results:

Algorithm	Results
'z-score'	[0.0814, 0.1022, 0.1449, 0.1894, 0.2285]
'scale'	[0.0727, 0.1040, 0.1508, 0.1966, 0.2382]
'range'	**[0.0121, 0.0195, 0.0274, 0.0348, 0.0413]**
'center'	[3.7234, 3.6552, 3.7006, 3.6826, 3.6661]
'medianiqr'	[0.1226, 0.1496, 0.2095, 0.2716, 0.3252]

So 'range' turns out to be the optimal algorithm. Note that it does not turn out the 'norm' algorithm this is because it equalized the data by reducing the variation almost completely.

Note that the 5 results represent the average RMSE on each predicted step.

The second step was to **set the window value**, again we performed some tests by **varying the window size in the range [10, 40]** and **keeping** the **number of neurons** in the LSTM layer at **64**. The results were the following:

Number of time steps	Results
10	[0.0118, 0.0196, 0.0291, 0.0396, 0.0506]
20	[0.0122, 0.0198, 0.0277, 0.0353, 0.0419]
30	**[0.0121, 0.0192, 0.0264, 0.0333, 0.0392]**
40	[0.0122, 0.0198, 0.0278, 0.0353, 0.0419]

We got that **the best value turns out to be 30**.

We also took care of tuning the number of neurons in the LSTM layer. We performed tests by varying the neurons in the range [52, 100] in steps of 12. **The best results were obtained with the number of neurons set to 64**.

To conclude, we tested the training algorithms:

- 'adam': [0.0141 0.0210 0.0280 0.0344 0.0400]
- **'rmsprop': [0.0115 0.0186 0.0260 0.0330 0.0389]**
- 'sgdm': [0.0261 0.0327 0.0387 0.0438 0.0482]

So, in the end, we got the network structured as follows:

```
% Define LSTM Network Architecture
layers = [
    sequenceInputLayer(num_channels)
    lstmLayer(64, 'OutputMode', 'last')
    fullyConnectedLayer(num_channels)
    regressionLayer
];


% Specify Training Options
options = trainingOptions('rmsprop', ...
    MaxEpochs = 20, ...
    MiniBatchSize = 1500, ...
    Shuffle = 'every-epoch', ...
    ...
    ValidationData = {XTest TTest}, ...
    ValidationFrequency = 50, ...
    OutputNetwork = 'best-validation-loss', ...
    ...
    LearnRateSchedule = 'piecewise', ...
    LearnRateDropFactor = 0.1, ...
    LearnRateDropPeriod = 10, ...
    ...
    SequencePaddingDirection = 'left', ...
    ...
    Plots = 'training-progress', ...
    Verbose = 1, ...
    VerboseFrequency = 5, ...
    ExecutionEnvironment = 'auto' ...
);
```

In order to make a correct estimation of the goodness of the results, it is necessary to assess its influence on the expected output. Observing the RMSE and the expected output, it can be seen that **the error is always at least an order of magnitude smaller than the output**.

One example of the obtained results is:

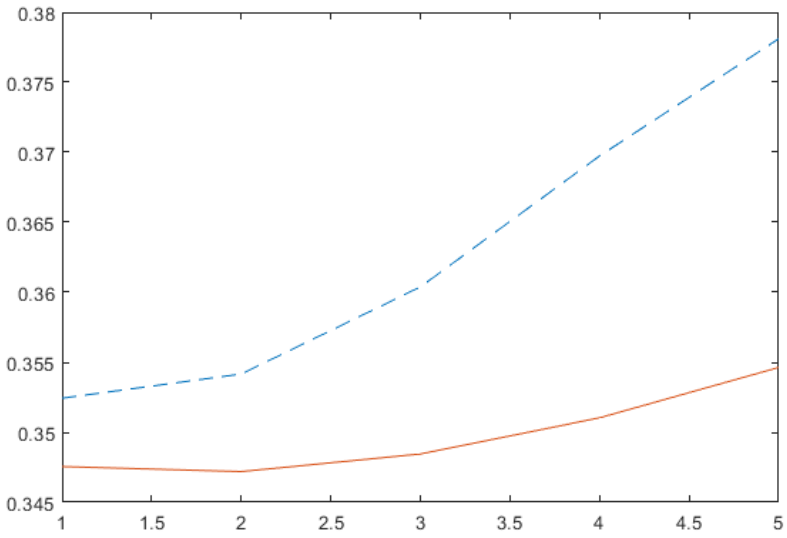

Figure 26: Prediction example

In the example, it can be seen that the error grows as the number of predicted steps increases. This makes perfect sense as the errors made in the previous steps will be added to those of subsequent steps, worsening the accuracy of the prediction.

In any case, the error in the five steps is always an order of magnitude smaller. The distribution of errors is as follows:

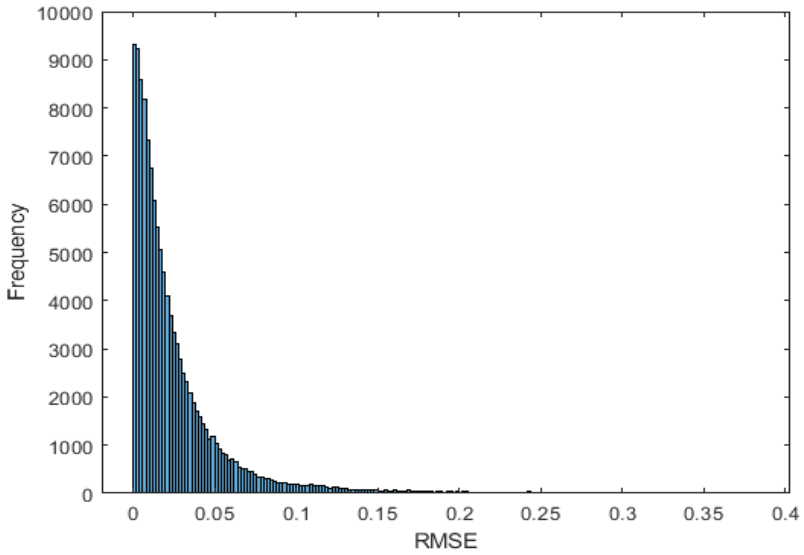

Figure 27: Average RMSE distribution