UNIVERSITÀ DI PISA

MSc Computer Engineering
Intelligent Systems

Project Report

Federica Perrone

Veronica Torraca

A.Y. 2022-2023

Summary

1. Introduction 2
2. Data preprocessing 3
   2.1. Handle missing data 3
   2.2. Features extraction 3
   2.3. Remove outliers 3
   2.4. Normalization 4
   2.5. Data augmentation 4
   2.6. Feature selection 4
3. Artificial Neural Networks 6
   3.1. Estimating ECG using neural networks 6
      3.1.1. Multi-Layer Perceptron Neural Networks 6
      3.1.2. Radial Basis Function networks 8
   3.2. Determining a person's activity using neural networks 11
   3.3. Fuzzy inference system 14
      3.3.1. Mamdani Fuzzy System 14
      3.3.2. TSK Fuzzy System 19
4. Deep Neural Network 20
   4.1. Improve ECG estimation using a Convolutional Neural Network (CNN) 20
      4.1.1. Starting Architecture 20
      4.1.2. Modification of the number of convolutional layers and the number and size of filters 22
      4.1.3. Addition of dropout layer 23
      4.1.4. Change of the training algorithm 24
      4.1.5. Final architecture 24
   4.2. Predict ECG values using Recurrent Neural Networks (RNN) 26

1. Introduction

The goal of this project is to design and develop several intelligent systems, to estimate and classify some characteristics and activities by analyzing time series of physiological signals.

In particular, following the specifications provided, we have to:

a. Design and develop two multi-layer perceptron (**MLPs**) artificial neural networks that estimate, respectively, the **mean and standard deviation** of ECG of a person (XX) during the three activities (YYYY), based on sensor data stored in the **sXX_YYYY_timeseries** file. The MLPs take as input a set of features extracted from the sensor data and return the value of the two features extracted from the ECG time series
b. Design and train two radial basis function (**RBF**) networks that do the same thing as MLPs developed in previous section *a*
c. Design and develop a multi-layer perceptron (**MLP**) that **classifies** a person's **activity** among 'sit', 'walk' and 'run', taking as input the set of features used to train the network developed in the first section and finding the best architecture for the MLP
d. Design and develop a fuzzy inference system (**FIS**) to **classify** a person's activity using the k most relevant features ($k \leq 5$) in the set of features used to train the MLP classifier developed in the previous section *c*
e. As in section *a*, design and develop one convolutional neural network (**CNN**) to estimate the feature (mean or standard deviation) that achieved the worst performance using the MLP developed in the first section
f. Design and develop a recurrent neural network (**RNN**) that predict 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 $t + 1$

At each point, the goal is to find the best architecture for the neural network being developed.

2. Data preprocessing

The data available must be preprocessed before it can be used in the neural network development.

A total of **22 participants** were involved in the data collection while performing **3 different activities** ('sit', 'walk', 'run'). The dataset contains **66 recordings**. A recording contains **11 time series**, each representing a physiological signal for more than 40,000 heartbeats.

So there are 66 time series files and 66 target files, for each different activity, for a total of **132 files**.

2.1. Handle missing data

All data from each recording were imported into MATLAB.

The first thing to do was to manage the **missing values**. We decided to replace the missing values with the **mean** of each signal.

2.2. Features extraction

A total of **11 features** were extracted from each of the collected signal.

These features were extracted in both **time** and **frequency domains**. The extracted features are:

- **Time domain**:
  - Minimum
  - Maximum
  - Mean
  - Median
  - Variance
  - Kurtosis
  - Skewness
  - Interquartile range
- **Frequency domain**:
  - Mean
  - Median
  - Occupied bandwidth

To extract the features, we used the **overlapped windows technique**: we divided the signal into **9 windows**, and in this way we were able to obtain 99 features for each signal (9 windows * 11 features for each window), for a total of **1089 features** (99 features * 11 signals).

2.3. Remove outliers

After extracting the features using the above procedure, we decided to **remove any outliers**, using the MATLAB function *rmoutliers*. As stated in the documentation, by default, an outlier is a value that is more than three scaled median absolute deviations (MAD) away from the median.

Outlier removal was repeated for both the network predicting the mean value of the ECG and the network predicting the standard deviation of the ECG.

In both cases a single outlier was removed, resulting in **65 samples**.

2.4. Normalization

Finally, to complete the creation of the matrix, we performed a **matrix normalization operation**.

This operation transformed the data, so that the values were distributed on an even scale.

This helped the training algorithm to produce more accurate results, by preventing large variations from negatively affecting the rest of the data.

All sample data for a given feature were **normalized between 0 and 1**. This process was repeated for all features.

Min-Max normalization was used to normalize the data.

2.5. Data augmentation

The dataset obtained in the previous step consists of **only 65 samples**, which is not enough to train the neural networks properly. In order to obtain more samples, we performed a **data augmentation process** on the dataset.

Data augmentation was performed using an **auto-encoder**, i.e. a neural network trained to replicate its input at its output. Each input sample generated **50 new samples**, resulting in a final dataset of **3315 samples** (65 samples without outliers * 50 new samples + 65 original samples), sufficient for the purposes of this project.

2.6. Feature selection

To identify the most relevant features for the prediction of the output, we decided to test the predictive power of each feature.

The technique used is the **Sequential Feature Selection**. This technique is already implemented in MATLAB through *sequentialfs*, using an **MLP** as the **criterion function**, as suggested in the project specifications.

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

Before selecting the 10 best features, we decided to **remove the redundant features** by **analyzing their correlation**. Therefore, if a **correlation** was found between features with a **value greater than 0.90**, only one was retained and the others were **discarded.**

This procedure reduced the number of features **from 1089 to 336** for the network predicting ECG **mean value**, and **from 1089 to 333** for the network predicting ECG **standard deviation**.

By eliminating the redundant features, we were able to run *sequentialfs* with a shorter run time.

The results obtained with the *sequentialfs* are different for the two different networks and are shown below in Table 1 and Table 2.

MLP estimating mean ECG	
FEATURE #	**CRITERION VALUE**
242	7.73892
307	4.96606
292	2.59933
7	1.3222
251	0.800629
223	0.496943
299	0.29095
42	0.179573
100	0.121633
324	0.0858967

Table 1 - Sequentialfs results for mean

MLP estimating std ECG	
FEATURE #	**CRITERION VALUE**
259	2744.22
77	1575.4
179	918.25
115	594.777
151	418.958
53	295.212
207	235.011
184	149.06
291	84.439
257	48.9529

Table 2 - Sequentialfs results for std

3. Artificial Neural Networks

3.1. Estimating ECG using neural networks

As previously introduced, the first point of the project specification required the design and implementation of neural network model for the estimation of some characteristic ECG values, more in detail:

- The creation of two **MLP neural networks** capable of estimating the **mean value** and the **standard deviation** of the ECG, respectively, based on the given dataset
- The creation of two **Radial Basis Function Networks** (**RBFNs**) capable of estimating the **mean value** and the **standard deviation**, as in the previous point

3.1.1. Multi-Layer Perceptron Neural Networks

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.

To get the best possible performance from the network, we tried tuning by changing the **training algorithm** and the **number of neurons.**

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

1. For the algorithms that **use a validation set** (all except *trainbr*):
    a. 70% for training
    b. 15% for validation
    c. 15% for testing
2. For the algorithms that does **not need a validation set** (only *trainbr*):
    a. 85% for training
    b. 15% for testing

To get the best possible performance, we first tried **tuning** by changing the **training algorithm**, with a **fixed number of hidden neurons** equal to **20**. The results are shown in Table 3.

TRAINING ALGORITHM	RESULTS FOR MEAN PREDICTION	RESULTS FOR STD PREDICTION
trainbr	**0,999469**	**0,999104**
trainlm	0,998501	0,99873
trainbfg	0,907793	0,802842
trainrp	0,990908	0,976038
trainscg	0,934241	0,881221

Table 3 - Results training algorithm tuning

The **best training algorithm** was found to be ***trainbr*** for both **mean** and **standard deviation** prediction.

Finally, we tested some values that the hidden layer could take. In particular, we tested values in the range **[20, 60]** with a **step of 10**. We concluded that the **optimal number of neurons** to predict both the mean value and the standard deviation was **50** (Figure 1 - Structure of the net used to predict mean and std of ecg).

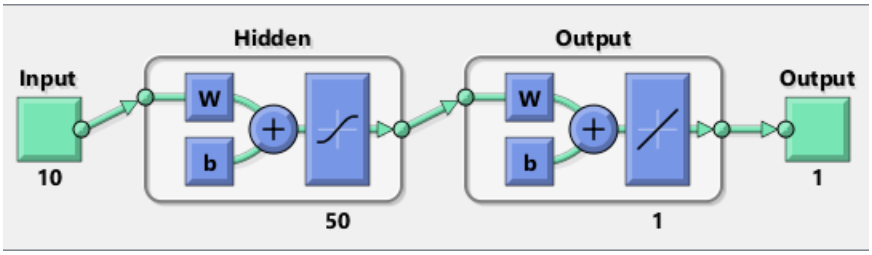

Figure 1 - Structure of the net used to predict mean and std of ecg

In Figure 2, the results for the prediction of the **mean value** are shown.

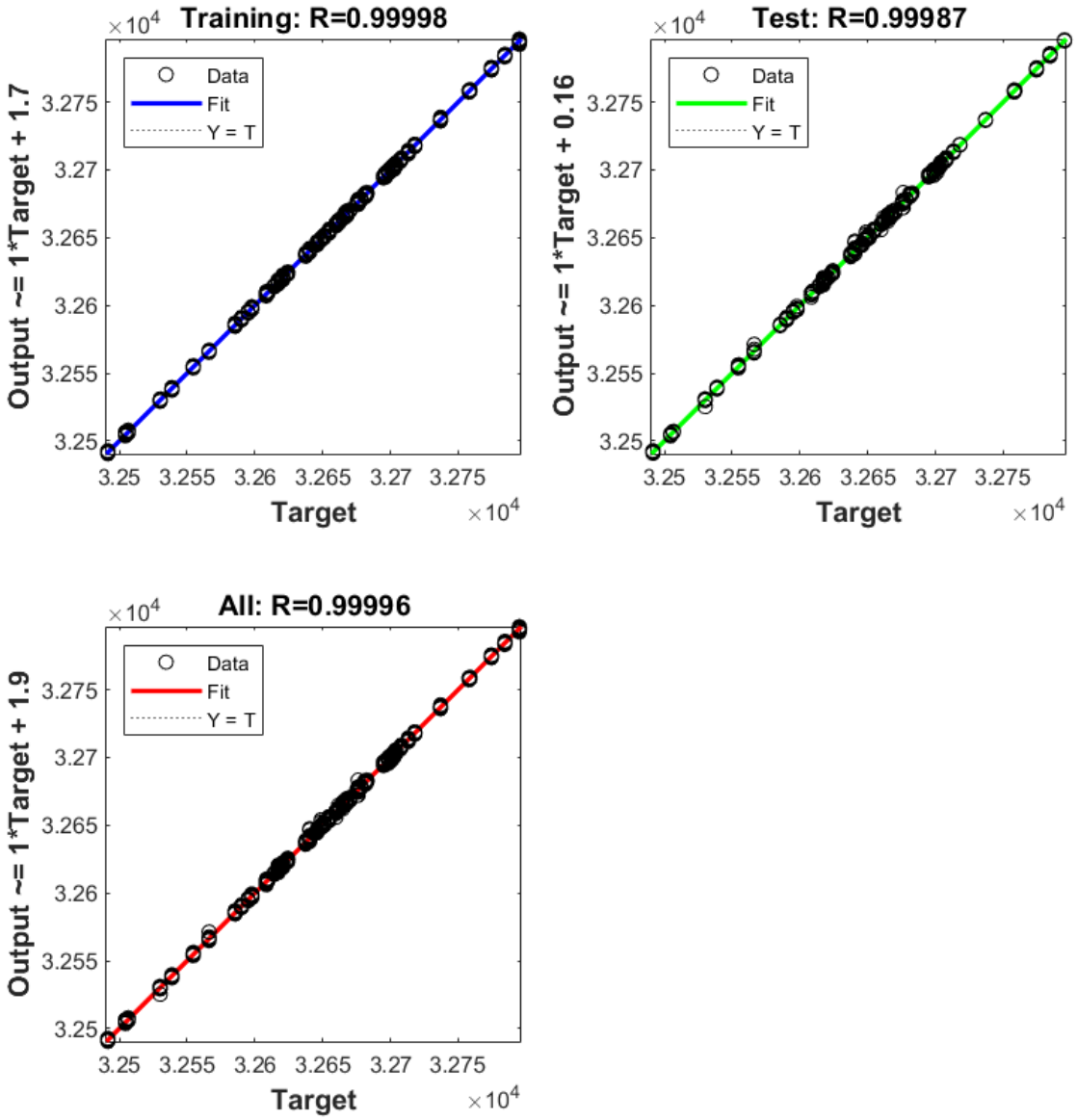

Figure 2 - Results for mean value prediction

Figure 3 shows the results for the **standard deviation** prediction.

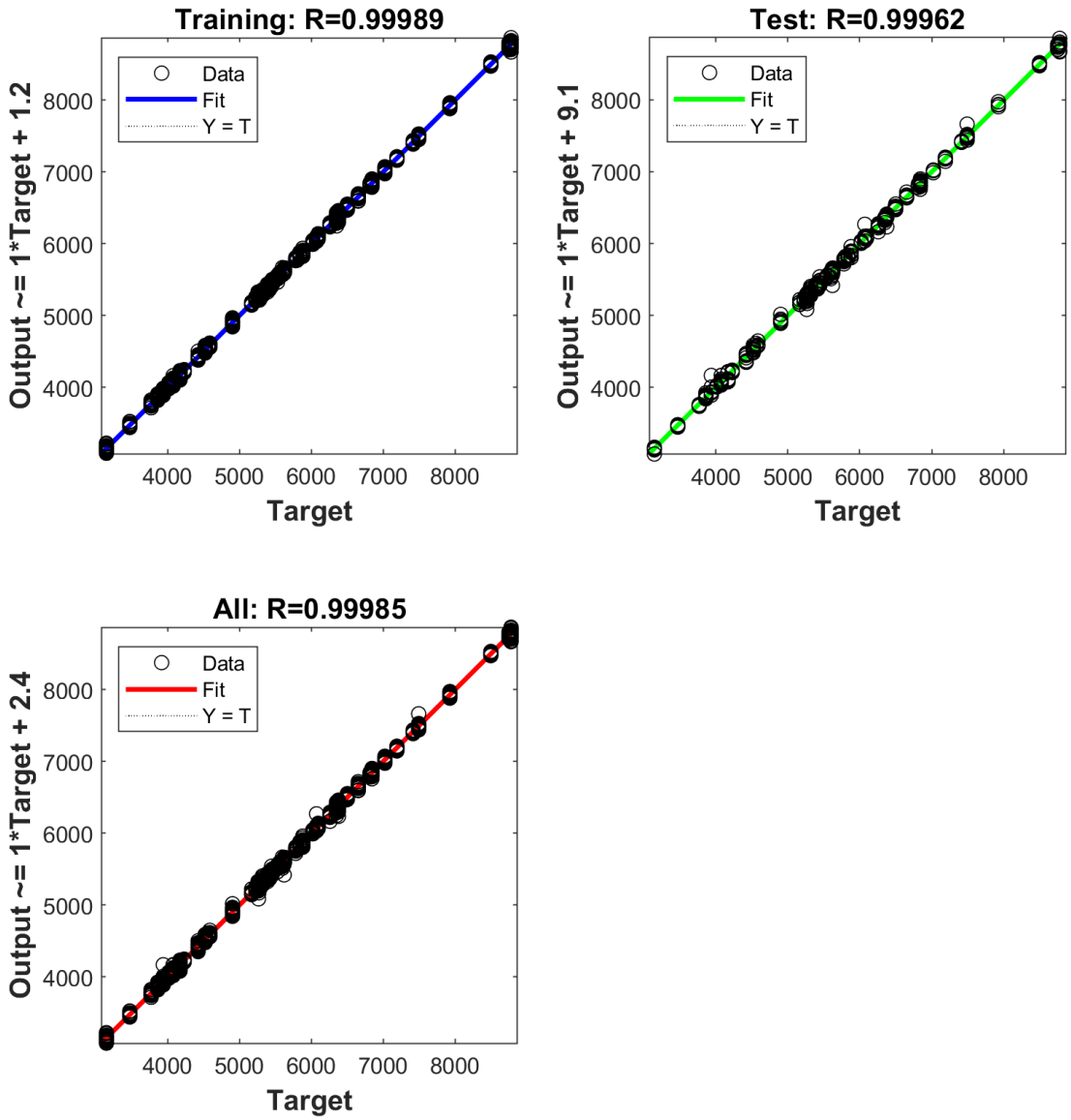

Figure 3 - Results for std prediction

3.1.2. Radial Basis Function networks

The final step in section 3.1 of the project specification was to design **two Radial Basis Function networks** to estimate the **mean** and the **standard deviation** of a person's ECG.

For the design of the two RBFNs, we tried four different approaches:

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

The fourth approach (generating a RBFN network with the ***newrb*** command and then training it with the ***trainbr*** algorithm) gave the best performance, for both predictions, i.e. the **mean** and the **standard deviation**.

To get the best possible performance, we first tried **tuning** by changing the *Spread* value, with a **fixed maximum number of *radbas* neurons** equal to **20**. We tuned the **Spread value** for both networks by trying different values **between the minimum and the maximum distance between points in the input space.** The results of the spread tuning for mean value prediction and std prediction are shown in Table 4 and Table 5, respectively.

Mean		
SPREAD	**R-VALUE AFTER NEWRB**	**R-VALUE AFTER TRAINBR**
0,2	0,56774	0,97782
0,4	0,59234	0,99672
0,6	0,61225	0,9985
0,8	**0,56151**	**0,99852**
1	0,61023	0,99568
1,2	0,62498	0,99683
1,4	0,62258	0,94654
1,6	0,61323	0,9425

Table 4 - Results spread tuning for mean prediction

Std		
SPREAD	**R-VALUE AFTER NEWRB**	**R-VALUE AFTER TRAINBR**
0,2	0,74344	0,98508
0,4	0,74232	0,99678
0,6	**0,71948**	**0,99756**
0,8	0,70166	0,99269
1	0,69367	0,99084
1,2	0,73007	0,97354
1,4	0,69807	0,98554
1,6	0,70354	0,88981

Table 5 - Results spread tuning for std prediction

The optimal spread values are therefore:

- **0,8** to estimate the mean value of the ECG
- **0,6** to estimate the standard deviation of the ECG

Finally, we tested some values that the **maximum number of *radbas* neurons** could take. In particular, we tested values in the range **[20, 60]** with an **increment of 10**. The results of the max number of neurons tested for mean value prediction and std prediction are in Table 6 and Table 7, respectively.

Mean		
MAX NEURONS	**R-VALUE AFTER NEWRB**	**R-VALUE AFTER TRAINBR**
20	**0,56151**	**0,99852**
30	0,69575	0,99762

40	0,78619	0,99738
50	0,87577	0,99791
60	0,94227	0,99838

Table 6 - Results max number of neurons tuning for mean prediction

Std		
MAX NEURONS	**R-VALUE AFTER NEWRB**	**R-VALUE AFTER TRAINBR**
20	**0.71948**	**0.99756**
30	0.84814	0.99648
40	0.93533	0.99605
50	0.97309	0.9969
60	0.98304	0.99754

Table 7 - Results max number of neurons tuning for std prediction

We concluded that the **optimal max number of *radbas* neurons** to predict both mean and standard deviation is **20** (Figure 4 - RBFN architecture to predict both mean and std). The results for the mean and standard deviation predictions are shown in Figure 5 and Figure 6, respectively.

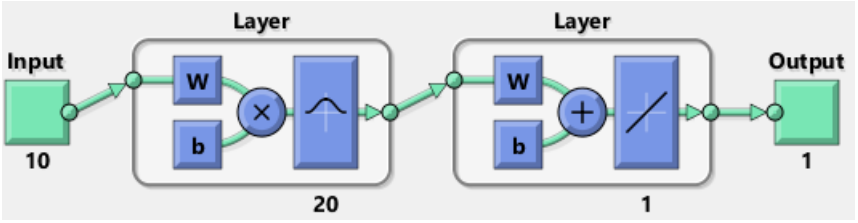

Figure 4 - RBFN architecture to predict both mean and std

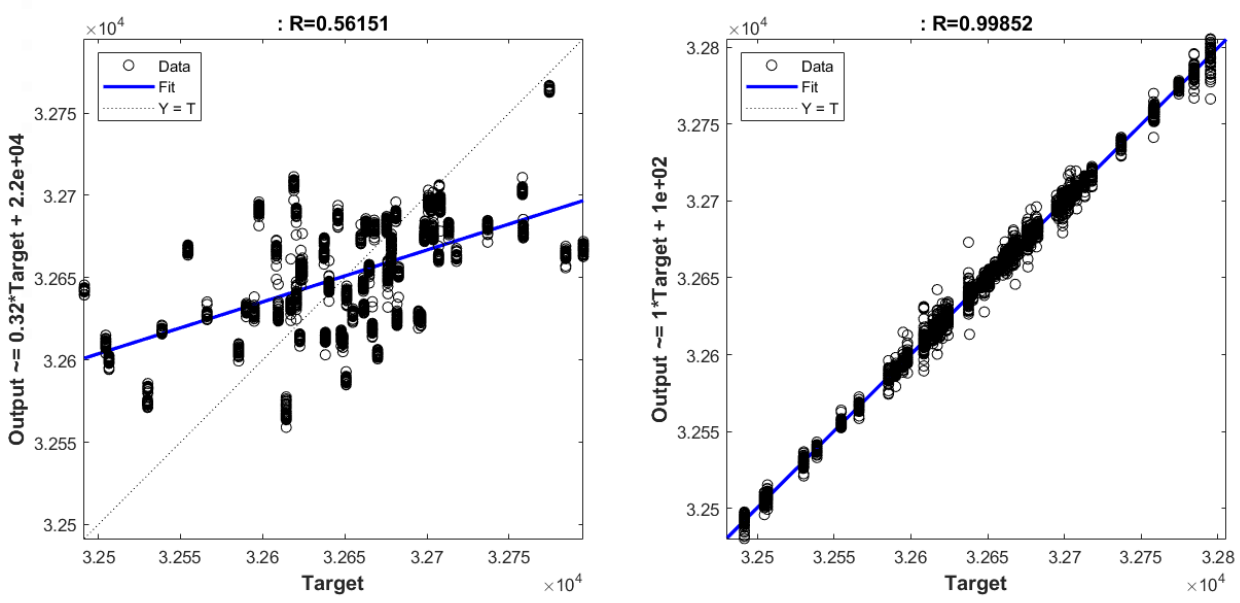

Figure 5 - Results for mean prediction before and after trainbr

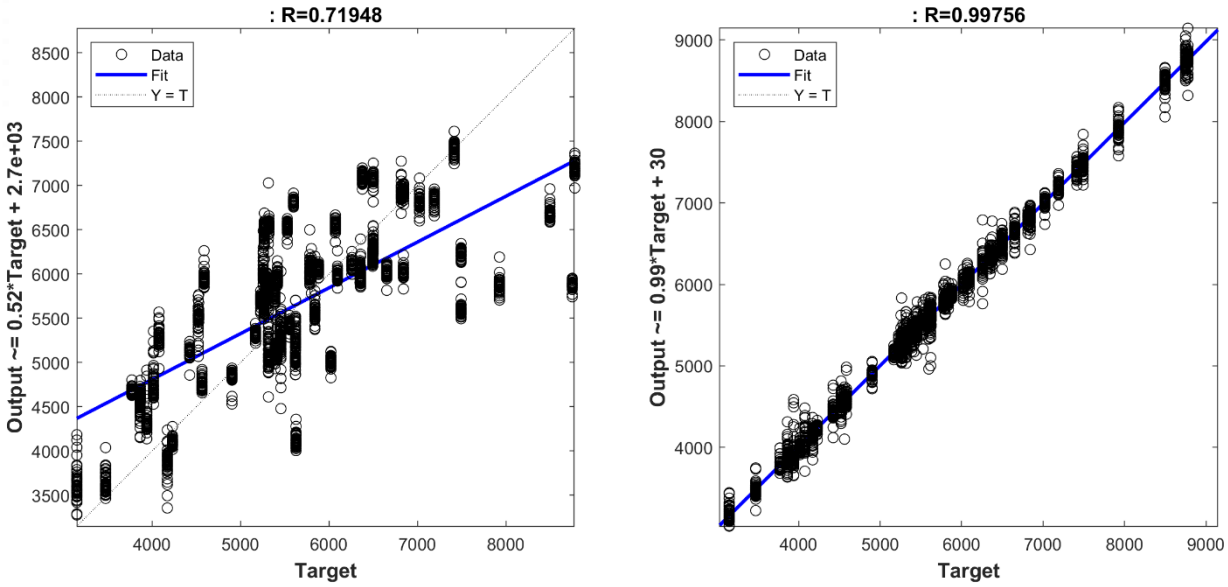

Figure 6 - Results for std prediction before and after trainbr

3.2. Determining a person's activity using neural networks

The next step in the project requirements was to design and develop a **multi-layer perceptron** (**MLP**) that would classify the **activity** a person was performing during the recordings.

We used ***patternnet*** to create a pattern recognition network to predict the activity of the subject being tested.

As required by the project specifications, the set of features to be used is the union of those selected and used to train the networks developed in the previous sections. Thus, we have a feature matrix composed of **20 features** and **3366 samples** (66[1] samples * 50 new samples + 66 original samples), which are the result of the data augmentation procedure.

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

The distinction between the 3 classes is shown in Table 8.

Three-class classifier		
ACTIVITY	**CLASS LABEL**	**TARGET**
Sit	1	100
Walk	2	010
Run	3	001

Table 8 - Class Target

[1] To develop and design the MLP that classifies a person's activity, we decided not to perform outlier removal.

To get the best possible performance from the classifier, we tried tuning by changing the **training algorithm** and the **number of hidden neurons.**

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

1. For the algorithms that **use a validation set** (all except *trainbr*):
    a. 70% for training
    b. 15% for validation
    c. 15% for testing
2. For the algorithms that do **not require a validation set** (*trainbr* only):
    a. 70% for training
    b. 30% for testing

To achieve optimal performance, our initial attempt involved adjusting the **training algorithm**, while keeping the **number of hidden neurons fixed at 10**. The results are shown in Table 9.

Percentage Correct Prediction				
TRAINING FUNCTION	**TRAINING**	**VALIDATION**	**TEST**	**ALL**
trainbr	**93,7%**	**NaN**	**94,6%**	**93,9%**
trainlm	92,4%	93,1%	91,5%	92,4%
trainscg	89,6%	87,9%	90,1%	89,4%

Table 9 - Results tuning training algorithm

Thus, we found the best performance when using ***trainbr*** the as training algorithm.

We followed heuristic guidelines and selected the number of **hidden neurons** within the range of **9 to 14**. The guidelines suggested starting with half the input size and increasing to two-thirds of the same input size. The results are shown in Table 10.

Percentage Correct Prediction				
HIDDEN NEURONS #	**TRAINING**	**VALIDATION**	**TEST**	**ALL**
9	95,6	NaN	95	95,5
10	93,7	NaN	94,6	93,9
11	**95,4**	**NaN**	**95,6**	**95,5**
12	95,4	NaN	95,6	95,5
13	93,8	NaN	94,3	93,9
14	91,3	NaN	90	90,9

Table 10 - Results tuning number of hidden neurons

We found the best performance with a **11 hidden neurons** (Figure 7 - Classifier Architecture). The classifier produced to the results illustrated in Figure 8.

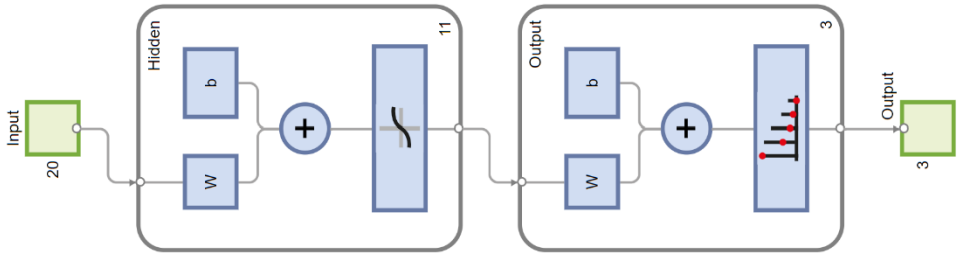

Figure 7 - Classifier Architecture

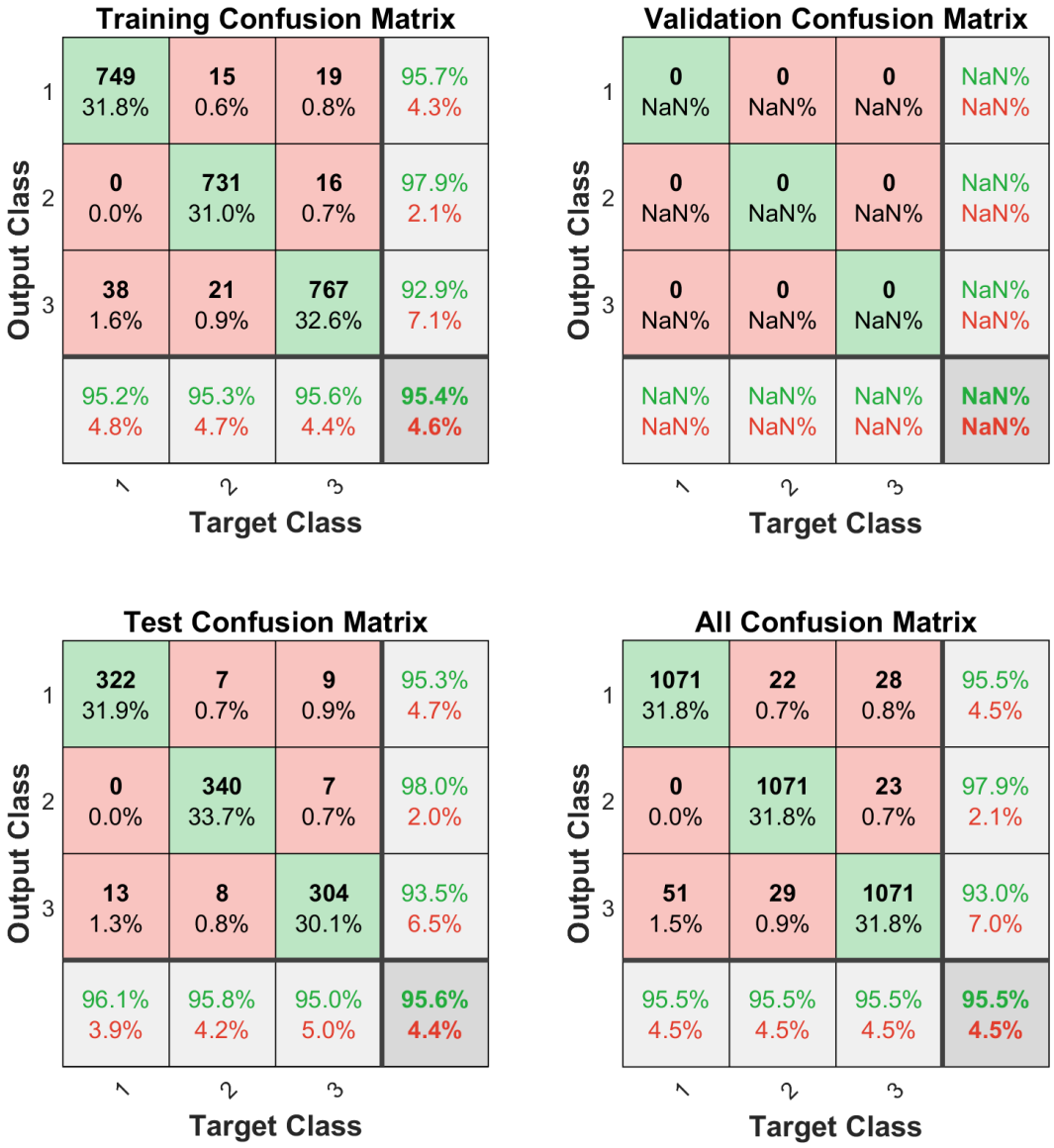

Figure 8 - Confusion Matrix

3.3. Fuzzy inference system

This section describes the design and development of a **fuzzy inference system** for **classifying** the **activity** performed by a person, distinguishing between the classes sit, walk or run.

As per the project specifications, we utilized the **3 most relevant features** from the set of features used to train the classifier. The features were selected by performing the *sequentialfs* procedure on an **MLP classification network** (*patternet*). The *sequentialfs* results are displayed in Table 11.

***Sequentialfs* Results**	
Feature #	**Criterion value**
20	0,00041691
8	0,000328819
17	0,000279825

Table 11 - Sequentialfs results

We developed two distinct FIS:

- A **Mamdani fuzzy system**, built manually employing the **Fuzzy Logic Design graphical tool**
- A **TSK fuzzy system**, generated automatically using a **grid-partitioning** approach (*genfis*) and fine-tuned through the use of an **adaptive neuro-fuzzy algorithm** (*tunefis*)

3.3.1. Mamdani Fuzzy System

We began our analysis by examining the distributions of the features through their histograms, as shown in Figure 9. These distributions have been utilized to model the linguistic variables of the three features. As the distributions exhibit skewness, we partitioned the feature space into multiple zones, determined by their relative position to the peak, as illustrated in Figure 10.

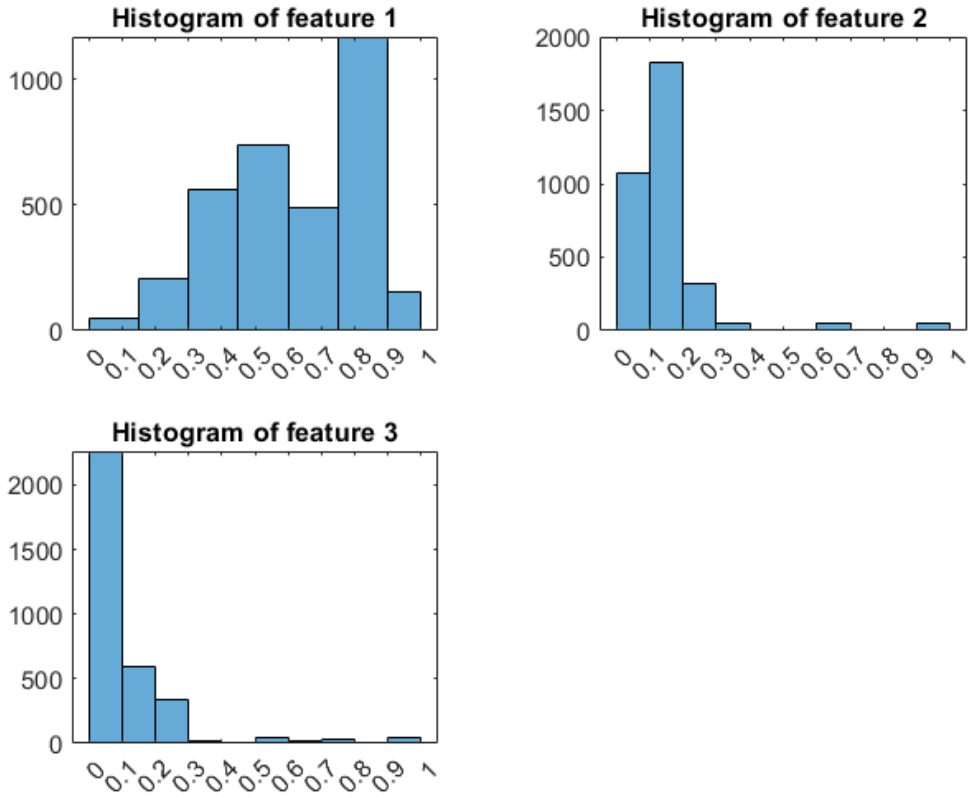

Figure 9 - Features' distribution

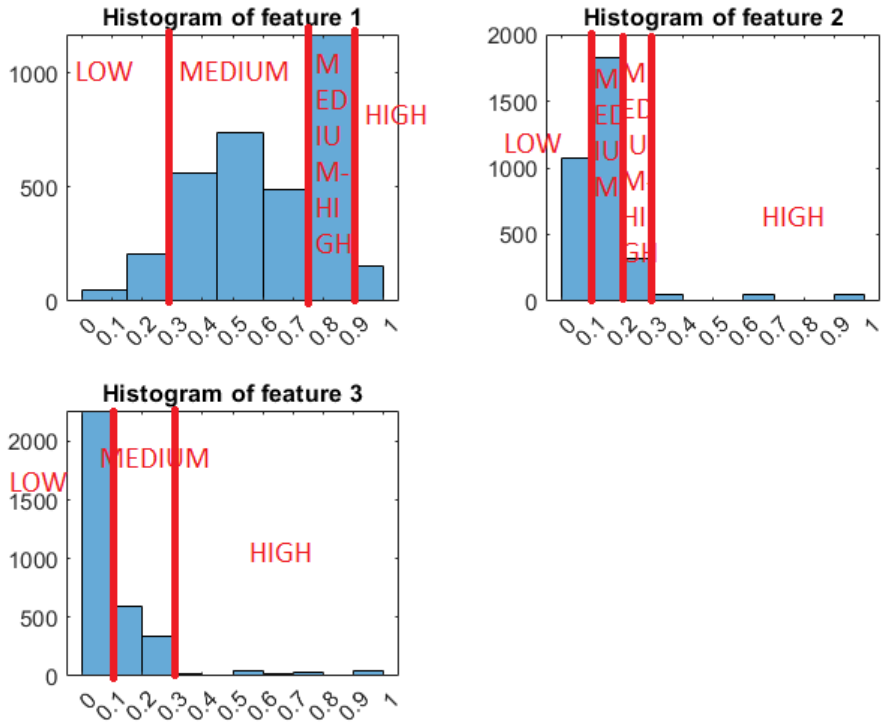

Figure 10 - Division of features' interval

Then, we created the membership functions for each feature, as shown in Figure 11, Figure 12 and Figure 13, along with the membership function for the output variable, as shown in Figure 14.

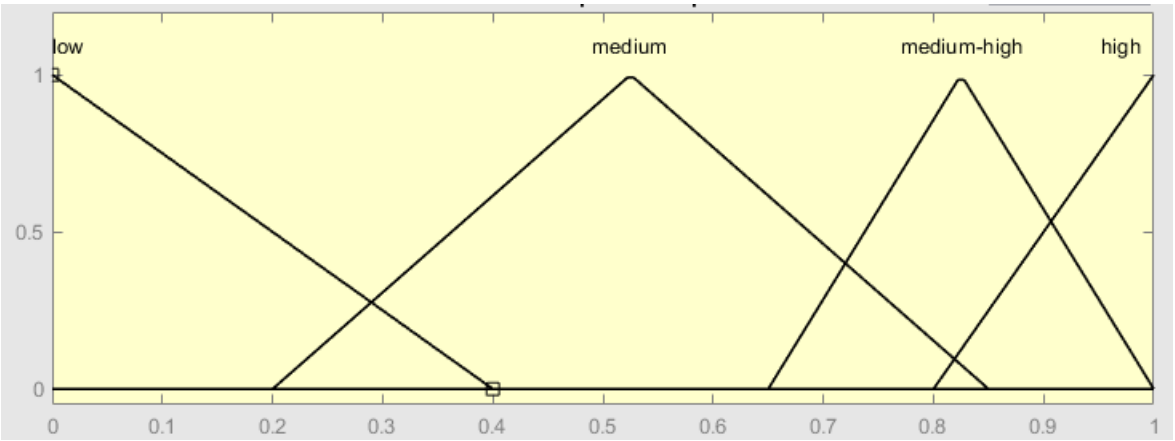

Figure 11 - Membership functions of feature 1

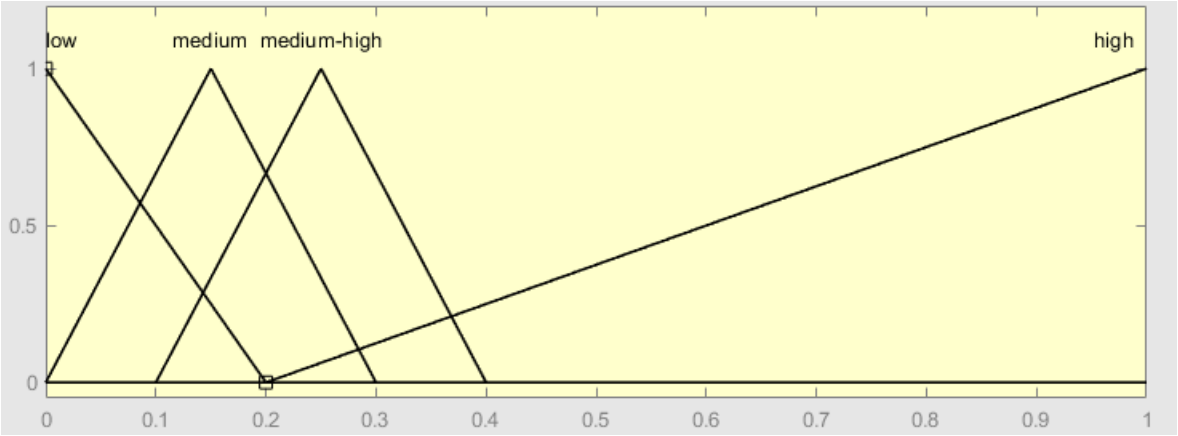

Figure 12 - Membership functions of feature 2

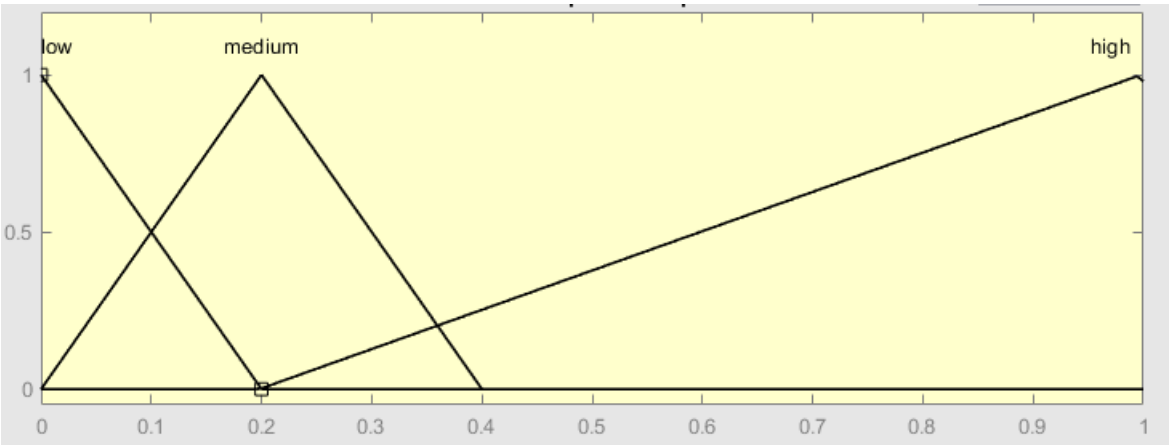

Figure 13 - Membership functions of feature 3

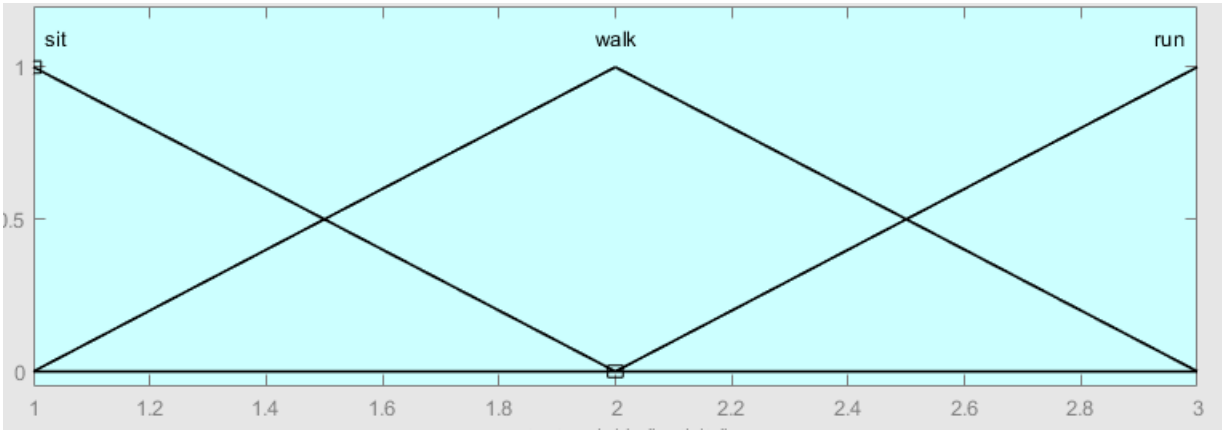

Figure 14 - Membership function of the output variable

Finally, in order to write the fuzzy rules, we examined the distribution of the three features for the activities of each individual, as illustrated in Figure 15. We thus created 39 rules, with different weights, to attain an acceptable level of precision (Figure 16 - Fuzzy rules).

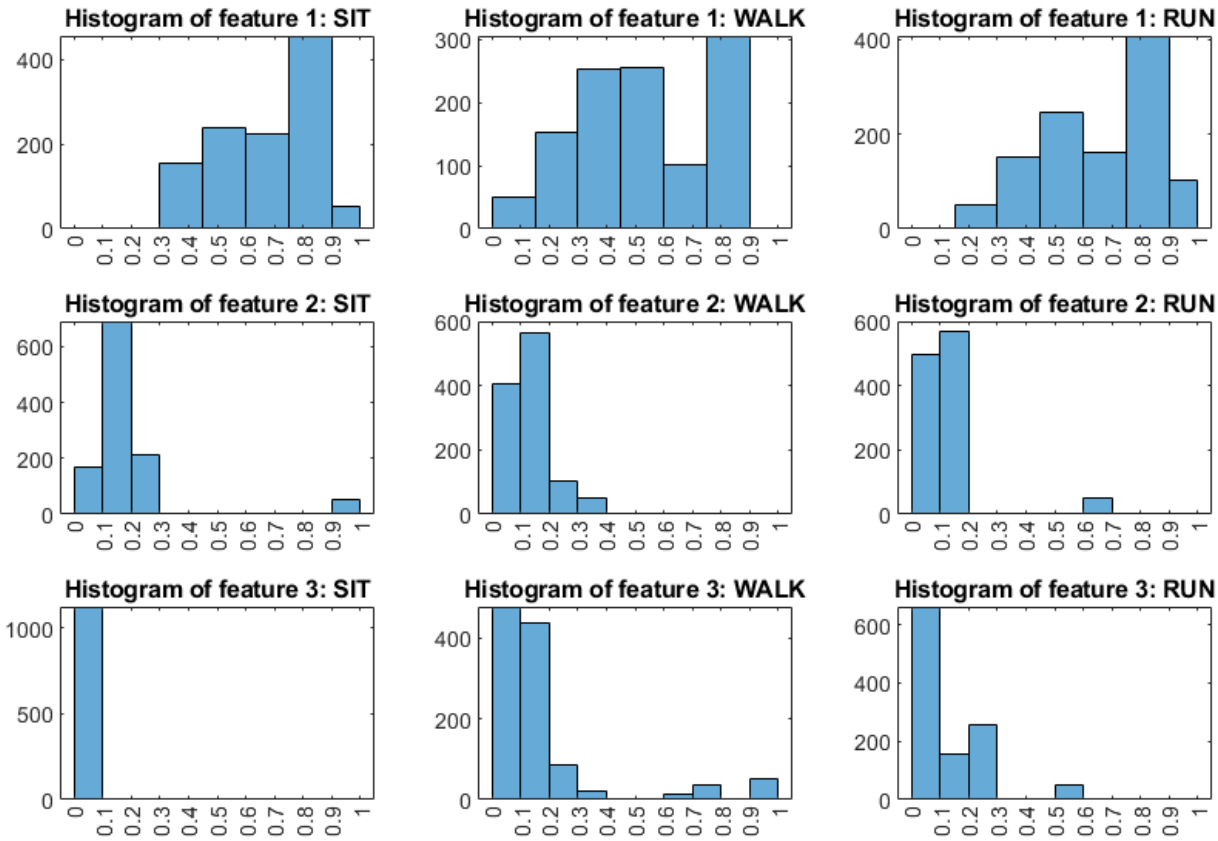

Figure 15 - Distribution of features for each activity class

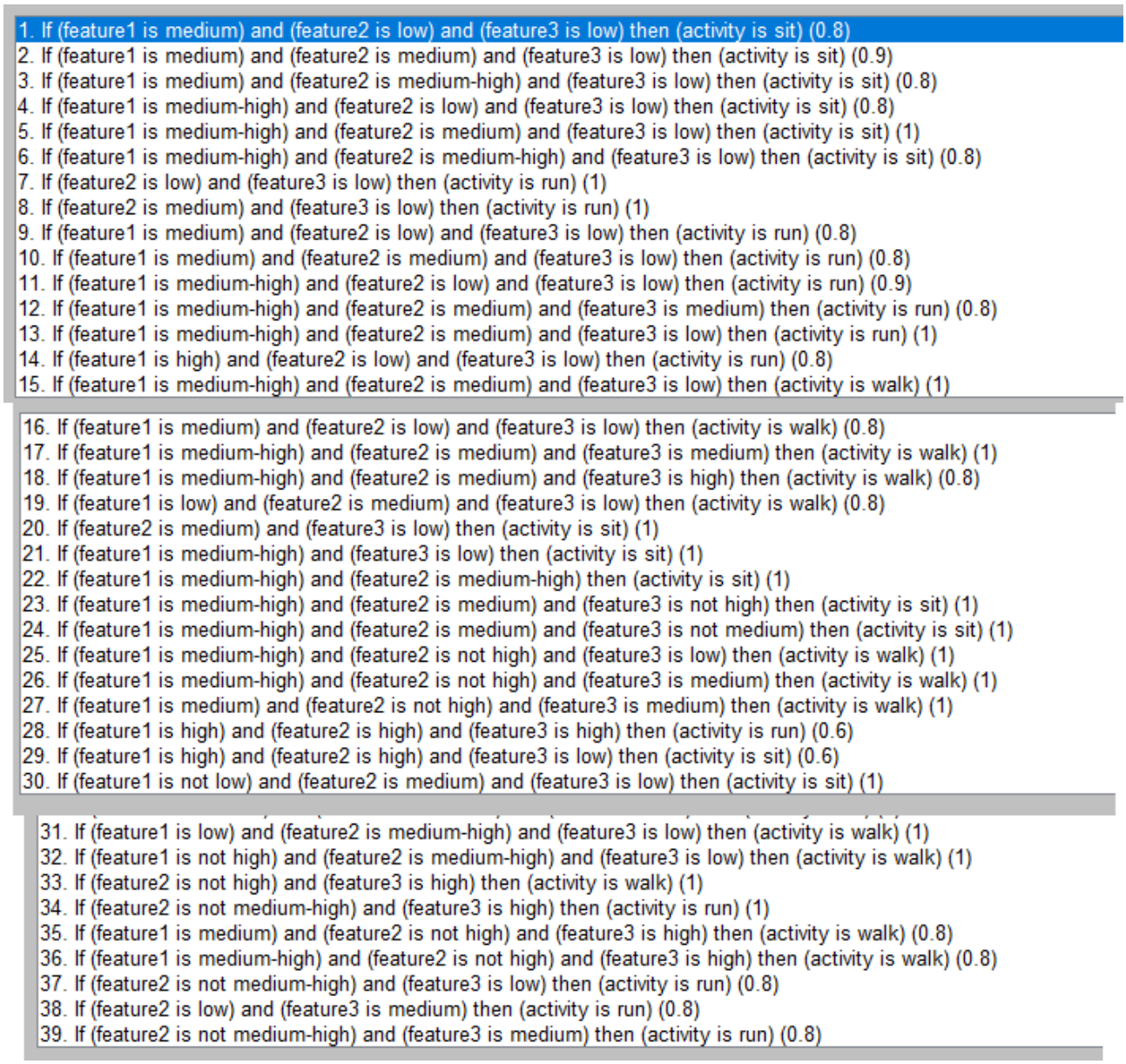

Figure 16 - Fuzzy rules

Regarding the configuration of the fuzzy inference system (and, or, implication, aggregation, defuzzification methods), as illustrated in Figure 17, the standard methods for a Mamdani system were implemented, except for the **defuzzification** method, which was set to **Smallest of Maximum** (***som***) due to its superior performance.

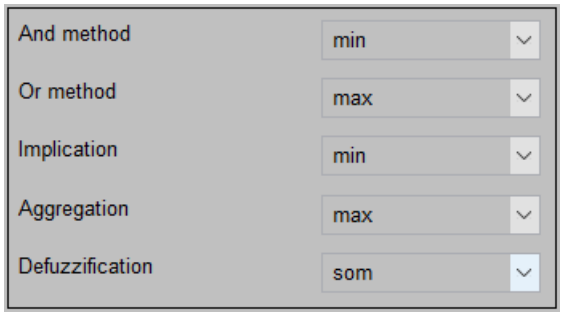

Figure 17 - FIS configuration

The results of the Mamdani FIS are shown in Table 12. Unfortunately, we were unable to achieve satisfactory precision.

AVERAGE	SIT	WALK	RUN
51.485443%	77.450980%	52.495544%	24.509804%

Table 12 - Results of Mamdani FIS

3.3.2. TSK Fuzzy System

Besides manually designing a Mamdani fuzzy system via the Fuzzy Logic Designer toolbox, we attempted an alternative approach: **automatically generating a FIS using data** (***genfis*** function) and **tuning it with an adaptive neuro-fuzzy algorithm** (the ***tunefis*** function).

Firstly, we created a completed fuzzy system comprising complete input/output membership functions and rule set.

We compared various **clustering methods**, to determine the optimal one. The findings are presented in Table 13.

METHOD	AVERAGE	SIT	WALK	RUN
Grid Partitioning (3 MF)	**87,492573%**	**82,620321%**	**97,682709%**	**82,174688%**
Subtractive Clustering	69,994058%	64,438503%	91,443850%	54,099812%
FCM Clustering	76,024955%	76,559715%	80,481283%	71,033868%

Table 13 - Results of clustering method tuning

As can be observed, the **Grid Partitioning** method produce the best outcomes.

Then, we tuned the **number of membership functions**. The results are shown in Table 14.

Grid Partitioning				
MEMBERSHIP FUNCTION #	**AVERAGE**	**SIT**	**WALK**	**RUN**
3	87,492573%	82,620321%	97,682709%	82,174688%
4	**95,008913%**	**92,602496%**	**96,880570%**	**95,543672%**
5	96,078431%	93,137255%	98,841355%	96,256684%
6	97,623292%	97,147950%	97,950089%	97,771836%

Table 14 - Results of number of membership funtions tuning

As it can be observed, the **Grid Partitioning method with 4 membership functions** presented the optimal trade-off.

4. Deep Neural Network

4.1. Improve ECG estimation using a Convolutional Neural Network (CNN)

In this section, the focus is on estimating ECG values by utilizing a **Convolutional Neural Network** (CNN). The CNN was designed to estimate the standard deviation of the ECG signal, since the MLP developed in section 3.1.1 exhibited inferior performance compared to mean value estimation.

To produce the data from time series, we partitioned each biophysical signal into **windows consisting 5000 time steps**. These windows were organized into sets and utilized as input for network training.

In order to determine the best CNN architecture to achieve optimal results, we went through a number of major steps that made significant improvements to the network, until we arrived at the final neural network.

These steps involved modifying the network itself, its layers, and the values of its hyper parameters:

- Modifying the **number of convolutional layers**
- Modifying the **number and size of the filters**
- Changing the **training algorithm**

4.1.1. Starting Architecture

The **base block** of our Convolutional Neural Network architecture consists of:

- A **1-D convolutional 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
- A **ReLU layer**, to avoid the vanishing gradient problem
- A **1-D average pooling layer**, which performs down-sampling by segmenting the input into 1-D pooling zones and calculating then the average of each region

The blocks were connected in a cascade fashion. The **final block is connected to**:

- A **1-D global average pooling layer**, which performs down-sampling by outputting the average of the time dimension of the input
- A **fully connected layer** with a **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 Figure 18.

The initial configuration of the architecture was determined through trial-and-error, specifically regarding the selection of the local and global pooling layers (max or average).

The training options for this neural network are outlined in Figure 19.

```
numFilters = 8;

layers = [
    sequenceInputLayer(11)

    convolution1dLayer(5, numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    reluLayer

    averagePooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    convolution1dLayer(3, 2 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    reluLayer

    averagePooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    globalAveragePooling1dLayer

    fullyConnectedLayer(100)
    reluLayer

    fullyConnectedLayer(1)
    regressionLayer
];
```

Figure 18 - Starting architecture of CNN

```
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 = 'gpu', ...
    Plots = 'training-progress', ...
    Verbose = 1, ...
    VerboseFrequency = 1 ...
);
```

Figure 19 - Training options CNN

Using this neural network and these training options, an **R-value of 0.52572** was obtained for **validation** and **0.54629** for **training**.

These values were used as starting point, then modifying the network and tuning its values to achieve better performance.

4.1.2. Modification of the number of convolutional layers and the number and size of filters

As an initial step to enhance network performance, we increased the number of base **blocks** in our CNN **from 2 to 4**. Nevertheless, this alternation alone did not result in substantial improvements. As a result, we opted to **augment** both the **filter size** and the quantity of **filters**. We maintaining the training **options unaltered**.

The configuration of the new network can be seen in Figure 20.

```
numFilters = 64;

layers = [
    sequenceInputLayer(11)

    convolution1dLayer(7, numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    reluLayer

    averagePooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    convolution1dLayer(5, 2 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    reluLayer

    averagePooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    convolution1dLayer(5, 3 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    reluLayer

    averagePooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

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

    averagePooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    globalAveragePooling1dLayer

    fullyConnectedLayer(100)
    reluLayer

    fullyConnectedLayer(1)
    regressionLayer
];
```

Figure 20 - CNN improvement

Using this neural network, an **R-value of 0.68342** was obtained for **validation** and **0.79269** for **training**.

To further improve performance, we tried to determine the **optimal combination** of **convolutional layers** and **filter number and size.**

All of these tests are visible and comparable in Table 15.

As can be seen, the initial combination turned out to be the best among all the others.

NUMBER OF CONVOLUTIONAL LAYER	NUMBER OF FILTERS	FILTER SIZE FOR CONVOLUTIONAL LAVER	VALIDATION R-VALUE	TRAINING R-VALUE
4	**64**	**[9 7 7 5]**	**0.73814**	**0.83123**
5	64	[9 7 7 5 5]	0.64274	0.75334
4	64	[11 9 9 5]	0.71263	0.78297
4	64	[15 11 11 9]	0.6967	0.78401

Table 15 - Tuning of CNN

4.1.3. Addition of dropout layer

Another performance improvement was made by adding a **Dropout Layer**. This Dropout Layer is designed to reset input elements to zero with a specified probability, aimed at **preventing overfitting**.

We tested several probability values for the dropout layer, as illustrated in Table 16.

NUMBER OF CONVOLUTIONAL LAYER	NUMBER OF FILTERS	FILTER SIZE FOR CONVOLUTIONAL LAVER	PROBABILITY DROPOUT LAYER	VALIDATION R-VALUE	TRAINING R-VALUE
4	**64**	**[9 7 7 5]**	**0**	**0.73814**	**0.83123**
4	64	[9 7 7 5]	0.5	0.64924	0.72351
4	64	[9 7 7 5]	0.3	0.66804	0.76452
4	64	[9 7 7 5]	0.2	0.70467	0.78403

Table 16 - Tuning of dropout layer

As can be observed, the inclusion of a dropout layer led to a decrease in performance. Therefore, we opted **not to add any dropout layers.**

4.1.4. Change of the training algorithm

We attempted to change the **training algorithm type**, also tuning the hyperparameters and making slight alterations to the CNN architecture, in order to see if there would be an improvement in performances.

The best results obtained with each algorithm are shown in Table 17.

TRAINING ALGORITHM	VALIDATION	TRAINING
Adam	0.75456	0.84139
Sgdm	**0.77998**	**0.8756**
Rmsprop	0.68894	0.78476

Table 17 - Tested CNN training algorithm

In particular, to test the SGDM algorithm, we had to alter the network architecture from the one used to test the ADAM algorithm. This was necessary due to the issue of the ***exploding gradient***, that caused overflow. So, we introduced and fine-tuned several parameters, including the ***Momentum*** and the ***GradientThreshold***. The latter is utilized to limit the gradient value if it exceeds the set threshold. In addition, we had to establish the ***InitialLearnRate*** option to ***0.1*** to expedite convergence, as with lower values this convergence rate was very slow and an acceptable network accuracy was unattainable within the limited number of Epochs, as with the other examined algorithms. Thus, also in the case of the SGDM algorithm, we conducted tests with various network architectures. Specifically, we experimented with different numbers of convolutional layers, starting with 2 layers, as well as varying the number and size of filters. Our findings revealed that the optimum network architecture for the SGDM algorithm comprised again 4 convolutional layer, albeit with a different filter size.

4.1.5. Final architecture

The **best Convolutional Neural Network** and the best training options found are shown in Figure 23 and Figure 24 respectively, and are obtained with the **SGDM** training algorithm. The best results obtained are then shown in Figure 21 and Figure 22.

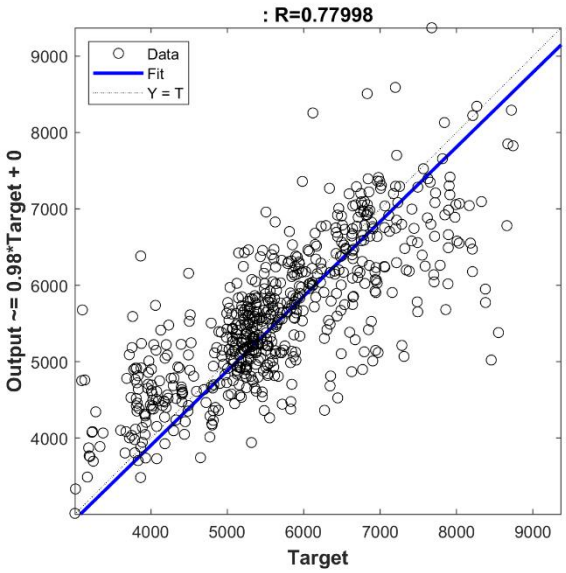

Figure 21 - Validation Regression Plot

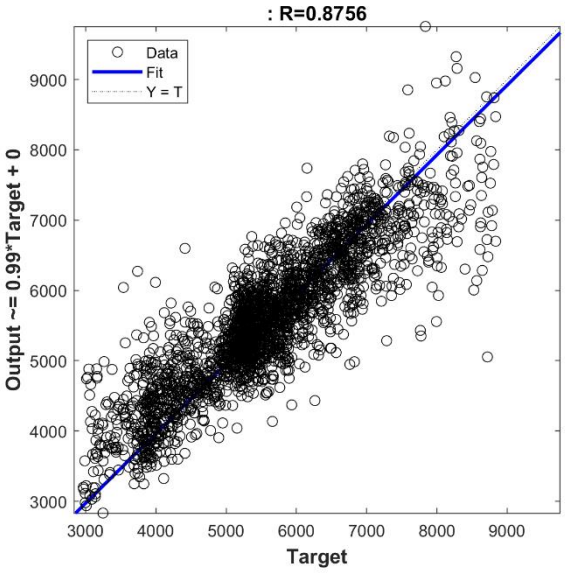

Figure 22 - Training Regression Plot

```
numFilters = 64;

layers = [
    sequenceInputLayer(11)

    convolution1dLayer(9, numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    reluLayer

    averagePooling1dLayer(4, 'Stride', 4, 'Padding', 'same')


    convolution1dLayer(7, 2 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    reluLayer

    averagePooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    convolution1dLayer(5, 3 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    reluLayer

    averagePooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    convolution1dLayer(3, 3 * numFilters, 'Stride', 2, 'Padding', 'same')
    batchNormalizationLayer
    reluLayer

    averagePooling1dLayer(4, 'Stride', 4, 'Padding', 'same')

    globalAveragePooling1dLayer

    fullyConnectedLayer(100)
    reluLayer

    fullyConnectedLayer(1)
    regressionLayer
```

Figure 23 - Final CNN architecture

```
options = trainingOptions('sgdm', ...
    ...
    MaxEpochs = 30, ...
    MiniBatchSize = 64, ...
    Shuffle = 'every-epoch' , ...
    ...
    Momentum = 0.9, ...
    InitialLearnRate = 0.1, ...
    LearnRateSchedule = 'piecewise', ...
    LearnRateDropPeriod = 10, ...
    LearnRateDropFactor = 0.1, ...
    L2Regularization = 0.01, ...
    GradientThresholdMethod='global-l2norm', ...
    GradientThreshold=0.9, ...
    ...
    ValidationData =  {XTest TTest}, ...
    ValidationFrequency = 30, ...
    ...
    ExecutionEnvironment = 'gpu', ...
    Plots = 'training-progress', ...
    Verbose = 1, ...
    VerboseFrequency = 1 ...
```

Figure 24 - Final CNN training options

4.2. Predict ECG values using Recurrent Neural Networks (RNN)

As detailed in the project specifications, the neural network receives as input one or more signals through the values they take in **different time intervals [t – k, t]**, with **t** representing the **start time** of the interval (window) and **k** being the **width of the window**.

To achieve optimal network performance, we fine-tuned by adjusting the **windows size**, number of **hidden neurons** and the **training algorithm**.

Firstly, we attempted to optimize **tuning** by varyingthe **window size.** We tested values in the range **[20, 60]**, while maintaining a fixed number of hidden neurons to a random value (30 hidden neurons). Our findings indicate that the **optimal window size** is **40**, as shown in Table 18.

WINDOW SIZE	RMSE ON VALIDATION SET
20	0.030621
30	0.026917
40	**0.023840**
50	0.024357
60	0.027323

Table 18 - Tuning of Window size

Then, we attempted to **adjust** the **number of hidden neurons** by testing values within the range **[20, 50]**. Based on our analysis, the **optimal number of hidden neurons** was **45**, as illustrated in Table 19.

HIDDEN NEURONS	RMSE ON VALIDATION SET
20	0.030083
25	0.029191
30	0.023840
35	0.025928
40	0.026614
45	**0.023490**
50	0.024429

Table 19 - Tuning of number of hidden neurons

Finally, we attempted to improve the model's performance by altering the training algorithm. It was determined that **rmsprop** is the **optimal training algorithm**, as shown in Table 20.

TRAINING ALGORITHM	RMSE ON VALIDATION SET
adam	0.023490
sgdm	0.069008
rmsprop	**0.016599**

Table 20 - Tuning of training algorithm

The final network and its training options are shown in Figure 25.

```
num_channels = 1;
num_hidden_neurons = 45;

layers = [
    sequenceInputLayer(num_channels)
    lstmLayer(num_hidden_neurons, 'OutputMode', 'last')
    fullyConnectedLayer(num_channels)
    regressionLayer
];
```

```
%% Specify Training Options

options = trainingOptions('rmsprop', ...
    MaxEpochs = 30, ...
    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' ...
```

Figure 25 - Final RNN architecture and training options

In Figure 26, a partial comparison between the predicted and expected values can be observed. Where the dotted line represents the predicted values while the solid line depicts the expected values.

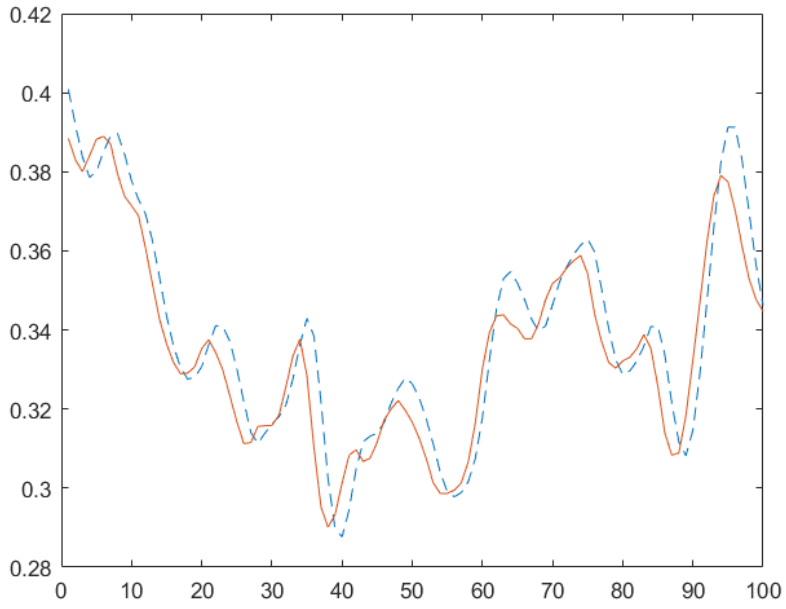

Figure 26 - Results RNN