Politecnico di Milano - Department of Electronics, Information and Bioengineering

POLITECNICO
MILANO 1863

Movements and Activity Recognition using STM32CubeAI-generated Neural Network

[Coding Project]

Student Giorgio Cozza
ID 10649461

Course Advanced Operating System (Computer Science and Engineering)
Academic Year 2018-2019

Advisor Federico Terraneo
Professor William Fornaciari

April 19, 2020

Contents

List of Figures 3

List of Tables 3

1 Introduction 4
1.1 A Possible Scenario . . . 4
1.2 Summary of the work . . . 4

2 Design and implementation 5
2.1 Problem Definition . . . 6
2.2 Data Gathering . . . 6
2.3 Data Analysis and Neural Model Design . . . 11
2.4 Network Training and Validation . . . 15
2.5 Network Code Generation and on-target Testing . . . 17
2.6 Code Integration and Final Firmware Development . . . 18

3 Experimental evaluation 20
3.1 Experimental setup . . . 20
3.2 Results . . . 20

4 Conclusions and Future Works 21

List of Figures

1 Design Process Chart 5
8 LSM6DSL Accelerometer: running and walking on X axis 11
9 LSM6DSL Accelerometer: running and walking on Z axis 12
10 LSM6DSL Accelerometer: jumping and standing 12
11 LSM6DSL Accelerometer: sitting, supine, lying on side 12
12 LSM303AGR Magnetometer, Scattering Plot 13
14 RNN model graph 14
15 Dataset Distribution 15
16 Dataset partitioning 15
18 Confusion Matrix: Testing 17
19 X-CUBE AI Engine 17
21 Class Diagram 19

1 Introduction

As the AI goes further in its evolution, together with explosion of interconnected IoT devices, the need to move the computation to the "edge" gets more urgent. Wereable devices relying on AI algorithms (such as neural networks) most of the times must stream data to general-purpose devices (such as smartphones and laptops) or to cloud services. The fact that, in real-time applications this represents a serious issue, is known, as well as the numerous legal concerns regarding data manipulation and privacy. All these problems get more mitigated as data and computation are kept local. But close to these debats there is a more subtle problem related to scalability. As heterogeneus data sources increase in number, in fact it becomes computationally expensive to scale a cloud-hosted AI application, in such cases is required to process a huge amount of different kinds of information for instance from a wide network of distributed sensor nodes. It may help, in such cases to let small sensing devices to solve simple classification tasks without involving external services, or even think to partition a complex AI task in smaller ones, moving part of the problem to the edge of a large infrastructure. This kind of scenario now are more feasable thanks to new tools such as STM32CubeAI, that speeds up prototyping on embedded platforms using neural networks.

1.1 A Possible Scenario

Let suppose to be in an hospital with several patients with serious pathological conditions or mental illness. In some case a patient is constrained in its movements, he cannot wake up from bed doing particular stressful movements or take specific positions, it may also be that such movements are symptoms of agitation that requires immediate medical intervention. These kind of patients must be monitored by mean of wereable sensors to detect anomaly conditions. Such system most of the times consists in a centralized architectures based on a AI algorithm processing data acquired from heterogeneous sensors distributed all over the building complex, in charge of classifying the condition of each specific patient. As the number of patients to be observed grows up it would be more efficient to allow each sensing device to provide directly the information related to the posture/movement of the patient instead of processing huge amount of data acquired from inertial sensors. This work aims to show that even under constrained hardware it is possible to achieve such a similar task.

1.2 Summary of the work

So a possible solution to the above mentioned problem is proposed in sort of toy example that can perfectly represent the potential effects of moving classification tasks on embedded platforms. The goal of this work is to solve a simple classification task involving a set of movements and postures of the human body by simply using an STM32 Nucleo board. All the stages from data gathering, through modeling and training the neural network, up to the implementation of the final firmware code will be discussed, remarking the advantages of recurring on specific tools to improve efficiency in all development stages. The peculiarity of this project is the simplicity of the process that led from the network model to the embedded code. At this step most of the work is performed by a powerful plugin integrated into the STM32CubeMX program, STM32CubeAI realized by ST Microelectronics to facilitate bare-metal development. Another important result is that the generated network code has been deployed directly on top of a real-time operating system, Miosix, showing the potential benefits of typical characteristics of this OS in the field of edge AI.

2 Design and implementation

It may help to have an high-level view of all the steps of the development process before starting to describe them in detail:

- **Problem Definition**: discussion about the topic of the project, identification of candidate sensors for data collection, first strategy of the design process and framework to be used for the AI part.
- **Data Gathering**: after being provided with all the required hardware set, development of a basic set of drivers and a firmware for data collection
- **Data Analysis and Network Model Design**: writing and testing of a set of python scripts to manage dataset files, study collected data by computing and plotting some statistics and define a first architecture of the neural network.
- **Network Training and Validation**: training and testing the network iteratively by changing settings, possibly defining new network models, or collecting new data sequence if required, until acceptable performance result is achieved.
- **Network Code Generation and on-target Testing**: set-up of the ST tool STM32CubeAI within STM32CubeMX environment, providing the model obtained in the previous step, performing validation on desktop and on target (on the board) and generate the C code of the input model.
- **Code Integration and Firmware Development**: integration of the output code within the Miosix environment, development of preprocessing routines (if required) and all the other aspects of the final firmware.
- **Experimental Evaluation**: testing of the embedded neural network by performing collection and classification of single batches of data samples in different operating modes.

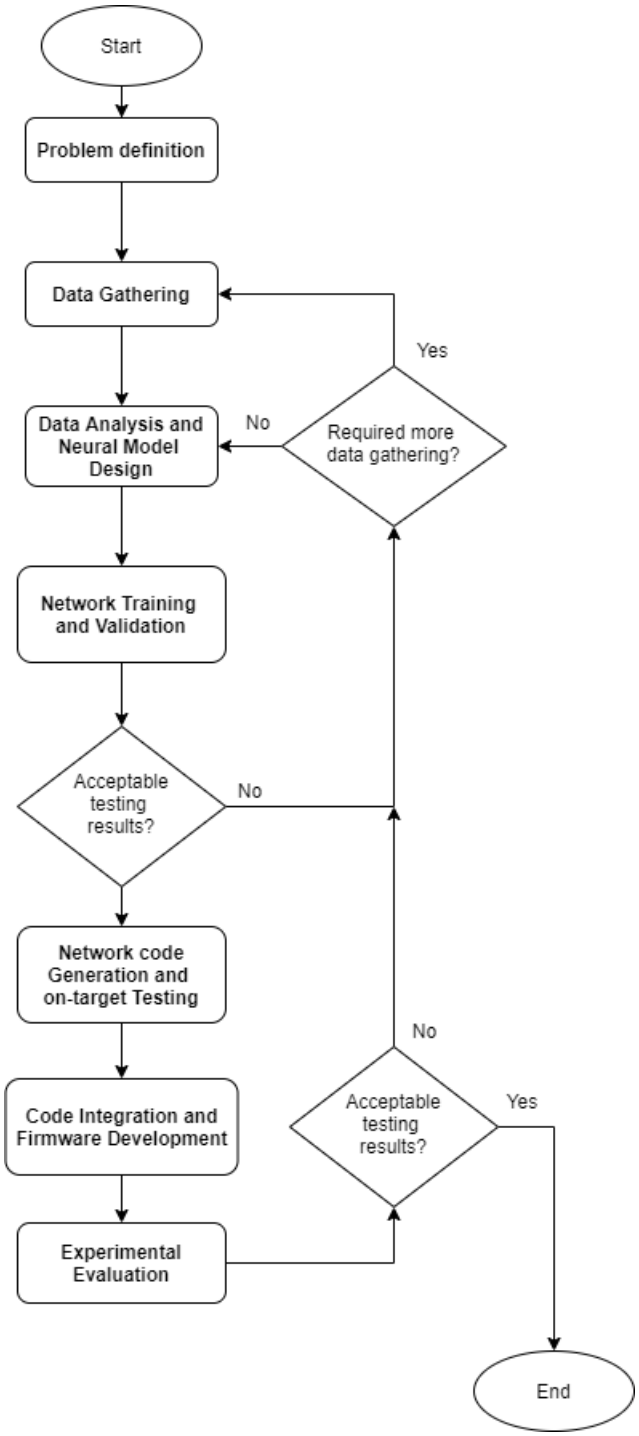


Figure 1: Design Process Chart

2.1 Problem Definition

The obviousness of this first step should not mislead from its purpose, as well-known the design cost related to possible errors at first steps is unavoidably amplified through all the subsequent ones. Beyond this observation, some key decisions at this step were taken by looking ahead to the data gathering stage and estimating the possible data demand of the AI algorithm. Classifying movements and positions of the human body is a task that lives naturally in the time domain, so is crucial to understand in detail the phenomenon to be observed and which type of information to be gathered.
There are many ways to measure movements and positions, *inertial* sensors such as the *accelerometer* and *gyroscope* are well-established solutions. *Magnetic* sensors (e.g:*magnetometers*) can also help in capturing motion patterns if used in combination with accelerometers and gyroscopes. Some research work in fact, adopt this approach to detect position and orientation of body parts by using sensor fusion techniques [1] and the patterns to be extracted from data are strongly shaped by this information. The problem in this case, is more abstract and requires a considerable amount of data to be collected, from many sources, to obtain accurate classification results. This observation is then accentuated considering the cost in terms of time and efforts required to perform data gathering, neural networks as well-known are "data hungry" and regardless their effectivness, no acceptable results can be achieved without a rich dataset.

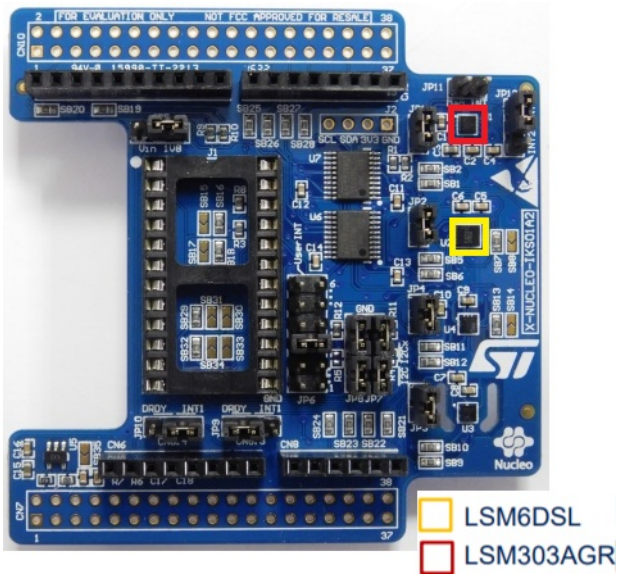
2.2 Data Gathering

This part is strongly affected by some a priori decision. As stated before this work relies on the STM32CubeAI plugin to speed-up the process that allows to obtain the initializing firmware code of the neural network from some framework-generated model, so only STM32-family hardware is used. After a deep analysis of topic-related studies it has been decided to use a single sensing device located at inguinal level (right-side). This location revealed to be extremely useful for two reasons:

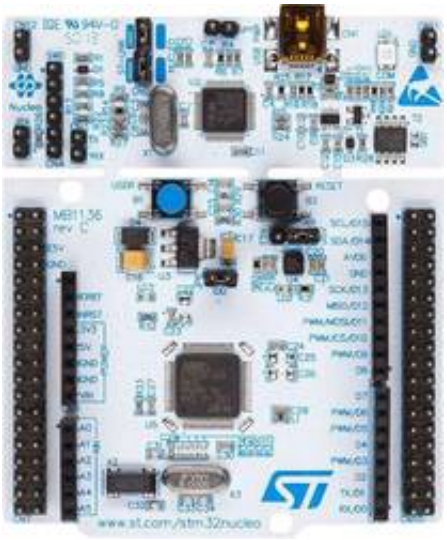
- Allows the sensors to detect directional and angular accelerations of the right lower limb which characterizes most of the movements and positions that must be classified
- Avoid the person wearing the device to be hampered during activity sessions

From hardware perspective, a first evaluation of the model complexity and resource avalability suggested two different solutions:

- NUCLEO-F401RE: a prototyping board based on a STM32-family MCU, the F401RE (ARM Cortex M4).
- IKS01A2 Sensor board, equipped with 3 inertial sensors (LSM6DSL 3D Accelerometer and 3D Gyroscope and LSM303AGR 3D Accelerometer), 1 magnetic sensor (LSM303AGR Magnetometer), 1 temperature and humidity sensor (HTS221) and 1 pressure sensor (LPS22HB)

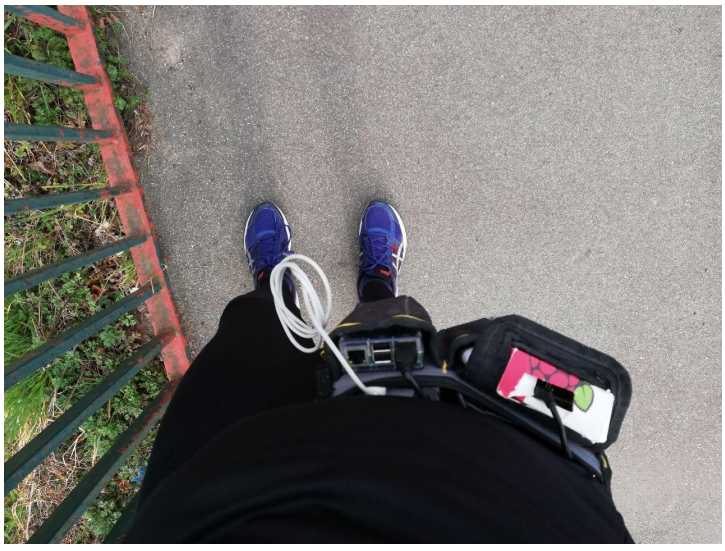


(a) IKS01A2 Sensor Board



(b) STM32F401RE Board

A first problem to be solved at this stage consists in finding a feasable way to store sensor-collected information.
Unfortunately it is not possible to use memories on board. On the other hand, the intuitive solution of streaming data samples being directly connected to the laptop through the USB port, is unfeasable because of ambient and cable size constraints. Although Miosix provides support to manage a filesystem on an external SD card, this possibility has been discarded due to the unavailability of a proper board expansion. This forced at the beginning to consider an Arduino MKR1000 board as a WiFi shield to stream data to a laptop using a smartphone as hotspot, but attempts to transfer data from the Nucleo board to the MKR1000 using I2C met some problems and discouraged this approach. In any case the increasing size of the packed boards make them uncomfortable to be worn.
A final and simple solution to the mentioned problems relies on a simple Raspberry Pi Model 3 besides the sensor board:



(a) Data gathering activity

(b) used devices

The sensing device streams data in form of comma-separated values that are stored by the Raspberry into a file, while a simple python program on backend is in charge of creating and managing all the activity session files during data collection.
The implementation of such script is straightforward, less trivial is the design and implementation of the MEMS drivers considering the fact that 4 different sensors must be used. Because of time constraints at the beginning, an already implemented version of the drivers were used, this to speed up data gathering stage. The following discussion will be focused instead on a version implemented afterwards, not included in the final test.
An initial remark regards the design pattern adopted for this part. Four sensors are splitted into 2 physical SoCs: the LSM6DSL (accelerometer and gyroscope) and the LSM303AGR (accelerometer and magnetometer), so 2 hardware-proxy classes have been realized `LSM6DSLAccGyr` and `LSM303AGRAccMag`, both implementing methods to setup control registers and read values from on-chip sensors. The expansion board instead communicates with the MCU using I2C. Since Miosix provides a low-level implementation of I2C, this was exploited to realize a class to perform read and write operations from the sensor board registers. Among the possible configurations by which is possible to connected the sensors, the expansion has been set up with a single shared line connected to all the MEMS' of the IKS01A2, the communication steps to read/write one or more bytes, moreover are generally the same (as specified by the protocol), from such specifications the implementation is straightforward:

Table 14. Transfer when master is writing one byte to slave

Master	ST	SAD + W		SUB		DATA		SP
Slave			SAK		SAK		SAK	

Table 15. Transfer when master is writing multiple bytes to slave

Master	ST	SAD + W		SUB		DATA		DATA		SP
Slave			SAK		SAK		SAK		SAK	

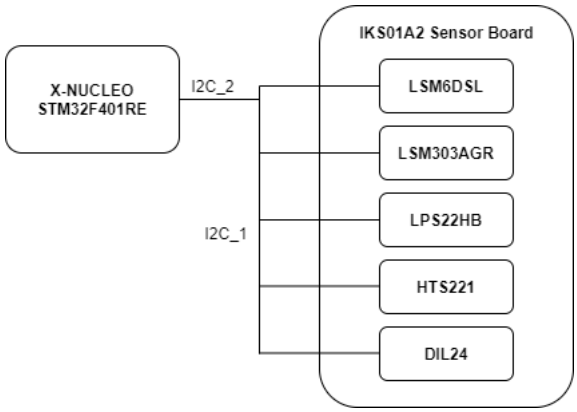
Table 16. Transfer when master is receiving (reading) one byte of data from slave

Master	ST	SAD + W		SUB		SR	SAD + R			NMAK	SP
Slave			SAK		SAK			SAK	DATA		

Table 17. Transfer when master is receiving (reading) multiple bytes of data from slave

Master	ST	SAD+W		SUB		SR	SAD+R			MAK		MAK		NMAK	SP
Slave			SAK		SAK			SAK	DATA		DAT A		DATA		

(a) I2C Communcation



(b) I2C Connection

```
_i2c_dev::init();
_i2c_dev::sendStart();

for (int j = 0; j < numbByte; j++) {
        if (_i2c_dev::send((unsigned char)devAddr)) {
                if (_i2c_dev::send((unsigned char)(regAddr + j))) {
                        _i2c_dev::sendRepeatedStart();
                        unsigned char sl_addrr = (unsigned char)(devAddr + 1);
                        if (_i2c_dev::send((unsigned char)sl_addrr)) {
                                *(buf + j) = _i2c_dev::recvWithNack();
                                _i2c_dev::sendStop();
                                delayUs(10);
                        }
                }
                else
                        return false;
        }
        else
                return false;
}
return true;
}
```

(c) i2c_helper.cpp: read

```
_i2c_dev::init();

for (int j = 0; j < numByte; j++) {
        _i2c_dev::sendStart();
        if (_i2c_dev::send((unsigned char)devAddr)) {
                if (_i2c_dev::send((unsigned char)(regAddr + j))) {
                        if(!_i2c_dev::send((unsigned char) * (buf + j)))
                                return false;
                }
                else
                        return false;
        }
        else
                return false;

}
_i2c_dev::sendStop();
delayUs(10);
return true;
}
```

(d) i2c_helper.cpp: write

Before proceeding to read data from MEMS, it is required to configure the control registers. Activation and initialization are performed in a single step implemented by the `init()` method.

```
/* LSM303AGRAccMag::io_read() */

bool io_read(uint8_t* pBuffer, uint8_t RegisterAddr, uint16_t NumByteToRead,
                               uint8_t dev_address , uint8_t reg_mask = 0xFF)
{
        uint8_t i = 0;
        for (i = 0; i < NumByteToRead; i++) {
                if (!((I2CHelper::getInstance())->read(pBuffer + i,


                        return false;
                *(pBuffer + i) &= reg_mask;
        }
        return true;
}
```

(a) LSM303AGR.h: io_read method

```
/* LSM303AGRAccMag::io_write() */

bool io_write(uint8_t* pBuffer, uint8_t RegisterAddr, uint16_t NumByteToWrite,
                                uint8_t dev_address , uint8_t reg_mask = 0xFF)
{
        uint8_t i = 0;
        uint8_t * tmp_buf = new uint8_t[NumByteToWrite];
        for (i = 0; i < NumByteToWrite; i++) {
                if (!((I2CHelper::getInstance())->read((tmp_buf + i),
                                                       dev_address,
                                                       RegisterAddr + i, 1)))
                        return false;
                else {
                        *(tmp_buf + i) &= ~reg_mask;
                        *(tmp_buf + i) |= *(pBuffer + i);

                        if (!((I2CHelper::getInstance())->write((tmp_buf + i),
                                                        dev_address,
                                                        RegisterAddr + i,
                                                        1, false)))
                                return false;
                }
        }
        delete[] tmp_buf;
        return true;
}
```

(b) LSM303AGR.h: io_write method

It runs similarly for both LSM6DSL and LSM303AGR, sets default values in those registers that are more likely to be used for data gathering. In both classes, with slight differences, `io_read()` and `io_write()` act as a I2C interface to allow read and write operation respectively involving device, register addresses and bit masks
Crucial aspects of sensor configuration are:

- **Output data rate**: representing the frequency at which the device provides new values
- **Full scale value**: the range of values that the sensor can represent, changing the FS, the sensitivity is affected as well.
- **Sensitivity**: the minimum variation of the quantity to be measured that can be percepted by the MEMS. Sensitivity can also change according to the power mode in which the sensor is used.

All these values are set and shown in the initial steps of the program execution, specifically in this work:

LSM6DSL Accelerometer	
ODR	208 Hz
Full scale	2.0 g
Sensitivity	0.061 [mg/LSB]

LSM6DSL Gyroscpe	
ODR	208 Hz
Full scale	2000 dps
Sensitivity	70.0 [mdps/LSB]

LSM303AGR Accelerometer	
ODR	100 Hz
Full scale	2.0 g
Sensitivity	3.9 [mg/LSB]

LSM303AGR Magnetometer	
ODR	100 Hz
Full scale	49.12 g
Sensitivity	1.5

Table 1: Data Gathering settings

As it can be noticed, both the accelerometers have been set to the same full scale value, to mantain data coherent with previous datasets already collected, whereas an high value (2000dps) has been chosen for the gyroscope to increase the sensitivity of the MEMS to even small movements of the leg (typical of some positions). In all the cases the configuration can be changed by modifying the `IKS01A2_config.h` .
Axes values are read almost the same way for all the sensors, since by design data registers are contiguous a single read scan of 6 bytes (2 byte-value/axis) is performed, raw bytes are then composed and multiplied by the sensitivity. The `LSM303AGR::get_acc_axes()` is slightly complex, since it must consider the power mode in which the sensor is used that affects the sensitivity value:

```
/*******  LSM6DSL::get_acc_axes()  *****************************************/

bool LSM6DSLAccGyr::get_acc_axes(int32_t * aData){

    uint8_t tmp_val[6] = {0,0,0,0,0,0};
    int16_t raw_val[3] = {0,0,0};
    float sens;

    if (!LSM6DSLAccGyr::io_read((uint8_t *)tmp_val, LSM6DSL_OUTX_L_XL,6))
        return false;


    raw_val[0] = ((((int16_t)tmp_val[1]) << 8 ) + (int16_t)tmp_val[0]);
    raw_val[1] = ((((int16_t)tmp_val[3]) << 8 ) + (int16_t)tmp_val[2]);
    raw_val[2] = ((((int16_t)tmp_val[5]) << 8 ) + (int16_t)tmp_val[4]);

    if (!LSM6DSLAccGyr::get_acc_sensitivity(&sens))
        return false;

    aData[0] = (int32_t)(raw_val[0] * sens);
    aData[1] = (int32_t)(raw_val[1] * sens);
    aData[2] = (int32_t)(raw_val[2] * sens);

    return true;
}

/******************************************************************************/
```

(a) LSM6DSL.cpp: get_acc_axes()

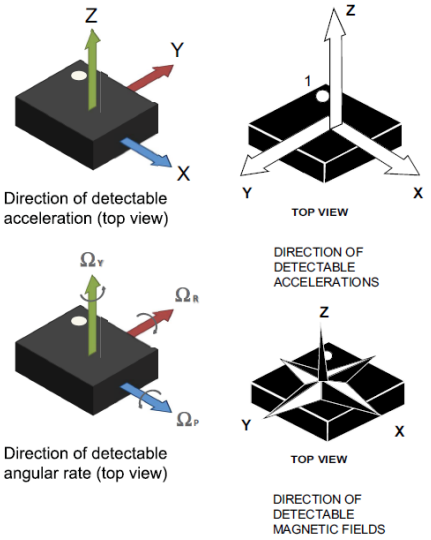
```
/*******  LSM303AGRAccMag::get_acc_axes  *************************************/

    if ( a_lp == LSM303AGR_ACC_LP_ENABLED && \
            a_hr == LSM303AGR_ACC_HR_DISABLED ) {
        shift = 8;
    }else if ( a_lp == LSM303AGR_ACC_LP_DISABLED && \
                a_hr == LSM303AGR_ACC_HR_DISABLED ) {
        shift = 6;
    }else if ( a_lp == LSM303AGR_ACC_LP_DISABLED && \
                    a_hr == LSM303AGR_ACC_HR_ENABLED ) {
        shift = 4;
    }else
        return false;

    if (!get_acc_sensitivity(&sens))
        return false;

    sens *= 1000.0f;

    if (!LSM303AGRAccMag::io_read((uint8_t *)tmp_val, LSM303AGR_ACC_OUT_X_L,6,
                                    LSM303AGR_ACC_I2C_ADDRESS, 0xFF))
        return false;


    raw_val[0] = (((((int16_t)tmp_val[1]) << 8 ) + (int16_t)tmp_val[0]));
    raw_val[1] = (((((int16_t)tmp_val[3]) << 8 ) + (int16_t)tmp_val[2]));
    raw_val[2] = (((((int16_t)tmp_val[5]) << 8 ) + (int16_t)tmp_val[4]));

    raw_val[0] = (((raw_val[0] >> shift) * sens + 500)) / 1000;
    raw_val[1] = (((raw_val[1] >> shift) * sens + 500)) / 1000;
    raw_val[2] = (((raw_val[2] >> shift) * sens + 500)) / 1000;


    aData[0] = (int32_t)(raw_val[0]);
    aData[1] = (int32_t)(raw_val[1]);
    aData[2] = (int32_t)(raw_val[2]);

    return true;
}

/******************************************************************************/
```

(b) LSM303AGR.cpp: get_acc_axes()

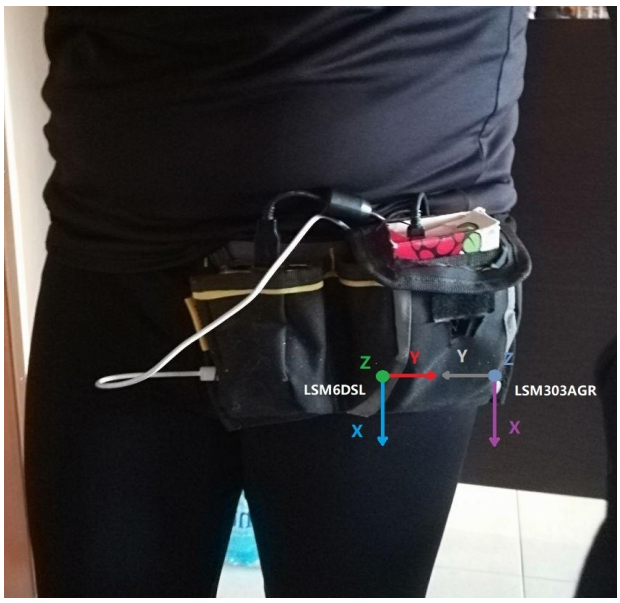
In the implementation of the method `LSM303AGR:: get_acc_axes()` the variables `a_hr` and `a_lp` contain values that encode the resolution and power modes at which the device is used and according to such information a proper shifting is applied to the read values.
Another interesting aspects of the LSM6DSL (iNEMO) is the maximum ODR the device is able to reach: 1660 Hz. Under strict power requirements, this high rate may represent a problem, fortunately both the sensors offer the possibility to store data in a FIFO buffer in order to burst all the read values at once and allows the MCU to sleep in the meanwhile. Since the low output data rate, this possibility has not been exploited.

2.3 Data Analysis and Neural Model Design

Almost immediately after data gathering it is required to plot some information about the collected datasets to outline a first strategy for a possible neural network model. A preliminary intuitive analysis is carried out by plotting the temporal distribution of a set of sequences (time windows) of data points from some activity files. This helps to infer some patterns that could reasonably be extracted by the algorithm.
In order to understand the trend of each activity type, is important to show the orientation of the MEMS with respect to the body:



(a) MEMS orientation



(b) Axes w.r.t the body

From MEMS positions it is possible to remark some considerations. Intuitively, it is expected to have some regular behaviour of the directional acceleration towards Z for walking and running activities:

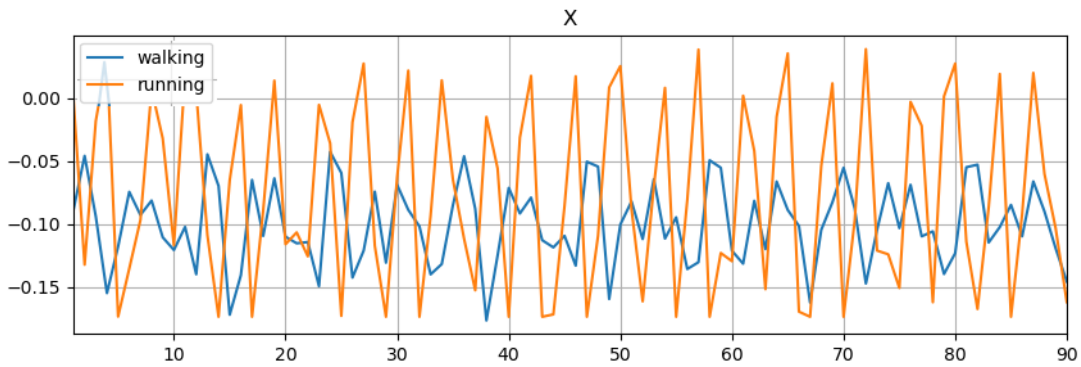


Figure 8: LSM6DSL Accelerometer: running and walking on X axis

This can be noticed from the "weird" behaviour in running trend with respect to walking, that on the other hand presents less accentuated and more distanced peaks (representing various steps).
While on Z there are not appreciable information that can be extracted:

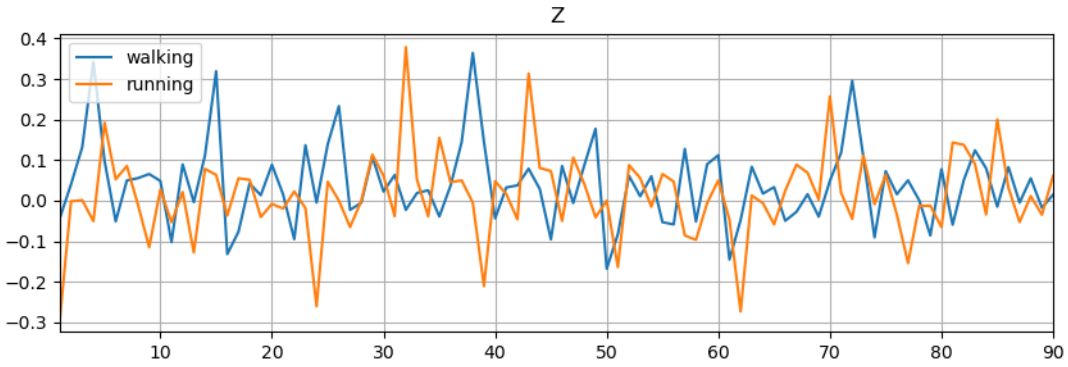


Figure 9: LSM6DSL Accelerometer: running and walking on Z axis

An easier comparison regards jumping and standing activities which instead, present more distinguishable regularities on X:

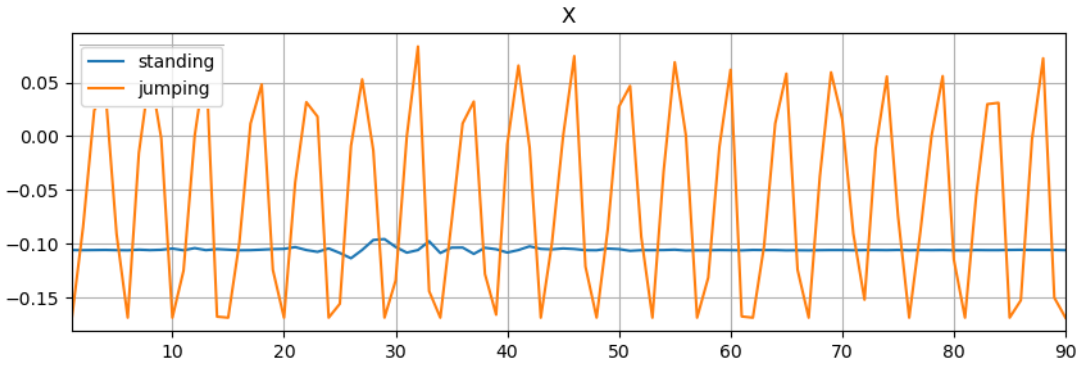


Figure 10: LSM6DSL Accelerometer: jumping and standing

Same observation for supine, lying on side and sitting positions:

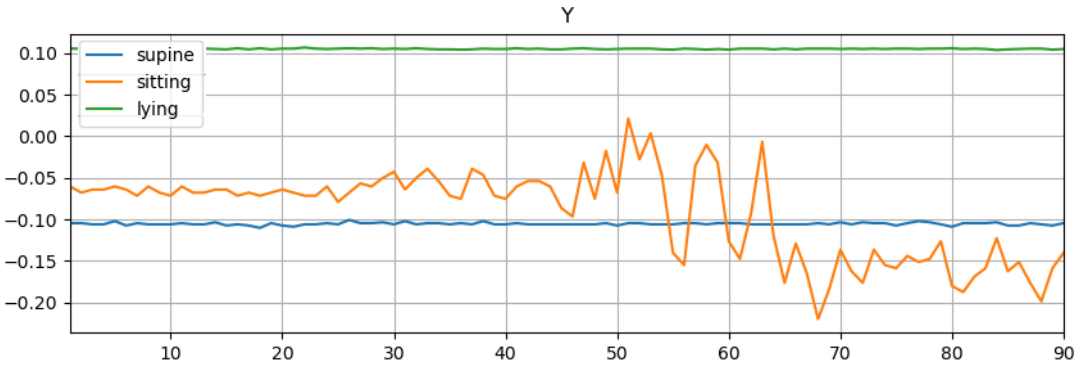


Figure 11: LSM6DSL Accelerometer: sitting, supine, lying on side

This kind of analysis also helps to understand qualitatively the amount of efforts to dedicate in further session of data collection: if trends show evident patterns, it is reasonable to think that the AI algorithm does not need so many samples to learn them. Although other sensor trends do not show interesting regularities, the role of the magnetometer in pattern recognition is inferred from the distribution of a bunch of data points in a 3D scattering plot:

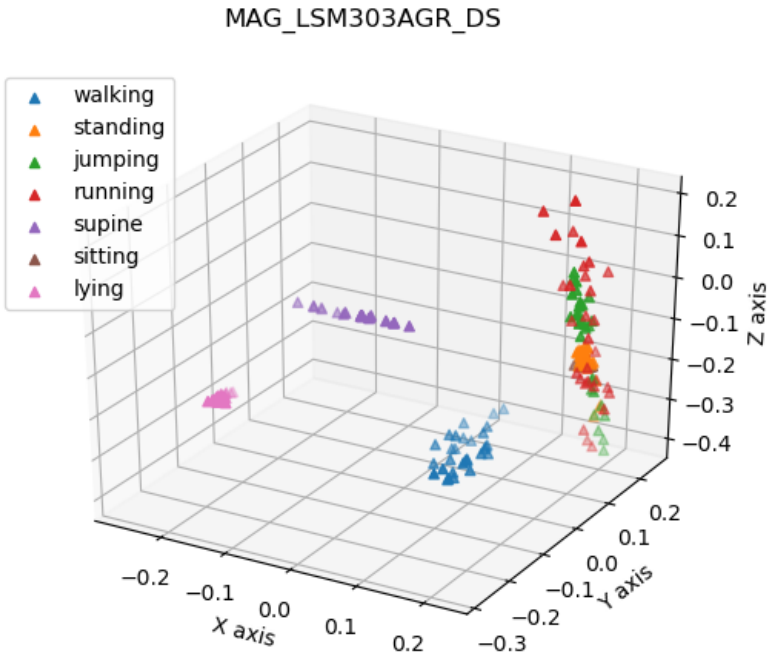
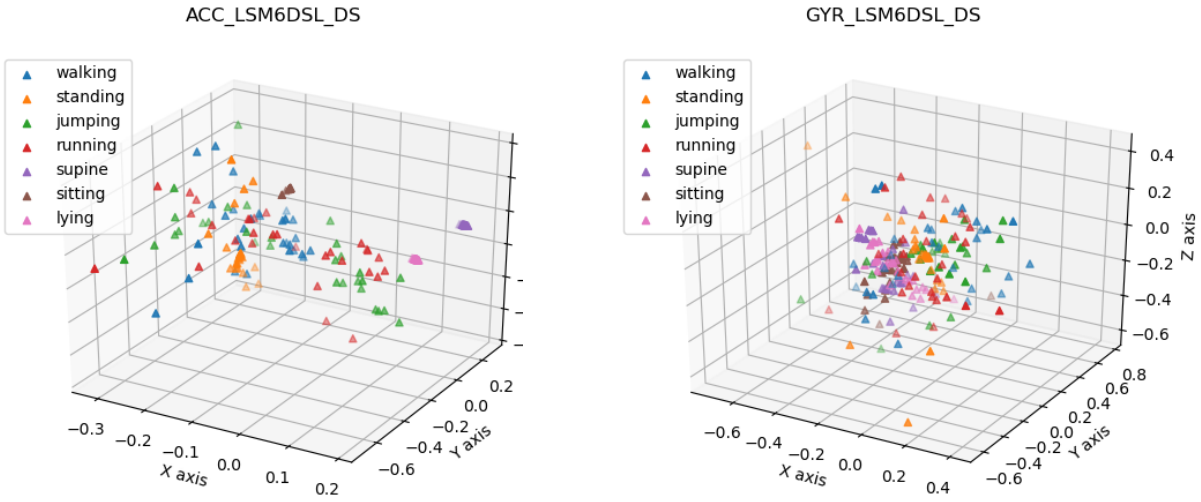


Figure 12: LSM303AGR Magnetometer, Scattering Plot

Most of the activity distributions appear to be distinguishable, but as it can be noticed, it is very difficult to discriminate between running and walking, or between standing and jumping in a certain range of values. The scattering plots of the LSM6DSL's accelerometer and gyroscope are even more confused and difficult to be analyzed:



(a) LSM6DSL Accelerometer, Scattering Plot

(b) LSM6DSL Gyroscope, Scattering Plot

This is the reason why it is not so easy to write an algorithm to classify among 7 activities, by doing some statistical evaluation on samples in a very tight time window. This consideration does not preclude the possibility to use other machine learning techniques to solve this problem, but since such

techniques have been frequently used in many similar research works, it has been decided to rely on neural networks this time.
Focusing on the architecture, Recurrent Neural Networks (RNN) are an established standard in classification problems that involve time series. Besides this LSTM-based model an alternative solution was developed originally, a simple Convolutional Neural Network (CNN) that unfortunately did not provide satisfying results for this specific task. It will not be discussed in this report.
The RNN solution consists in a single-cell LSTM architecture on top of a classifier, characterized by the following layers:

- **Batch Normalization**: to perform batch normalization of input data. Precisely, this layer performs *standardization* of the input batch according to the following formulas:

$$y_i = \gamma \hat{x_i} + \beta \qquad \hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

  in which x_i is a data point of the batch, μ_B and $\sigma_B$is the batch mean and variance respectively, whereas γ and β are paramters used to properly scale the normalization result, ϵ at the end avoids zero-division.

- **single LSTM** cell: characterized by 64 hidden nodes, with *tanh* as activation function and *sigmoid* as recurrent activation function. This part of the architecture is in charge of capturing time dependences in data sequence

- **3 Fully-connected** layers: to classify features extracted by the LSTM cell, first layers (FCN1 and FCN2) composed of 32 neurons using a *rectified-linear unit* as non-linear activation function and a last step of *softmax* normalization .

Here some implementation details:

```
# **************   Recurrent Neural Network model   ***************************************
def RNN_model():
    hid_nodes_lstm = 64
    fcn_nodes = 32
    rnn_model = Sequential()
    rnn_model.add(BatchNormalization(input_shape=(WINDOW_SAMPLES, SENS_VALUES)))
    rnn_model.add(LSTM(units=hid_nodes_lstm, return_sequences=False, name='LSTM1'))
    rnn_model.add(Dropout(0.2))
    rnn_model.add(Dense(fcn_nodes, activation='relu', name='FCN1'))
    rnn_model.add(Dropout(0.2))
    rnn_model.add(Dense(fcn_nodes, activation='relu', name='FCN2'))
    rnn_model.add(Dropout(0.2))
    rnn_model.add(Dense(num_classes, activation='softmax', name='FCN3'))
    model_summary(rnn_model, mod_type='RNN')
    return rnn_model
# ****************************************************************************************
```

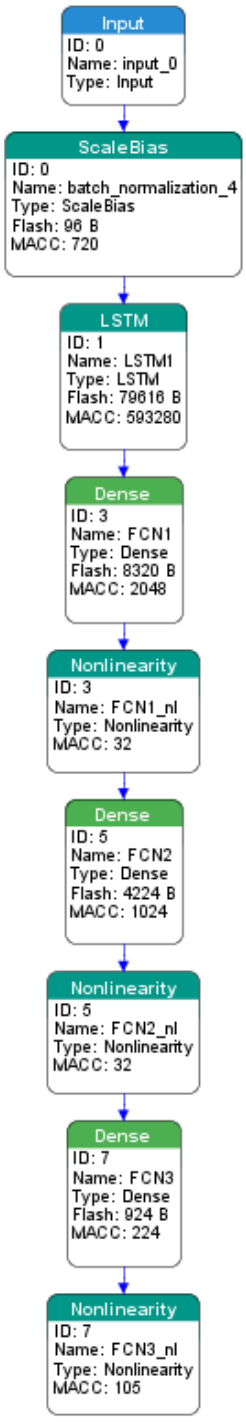


Figure 14: RNN model graph

2.4 Network Training and Validation

This step of the development process, starts from some preprocessing of the collected dataset files. Since each data gathering session is fully contained in a single csv file, it is required at first to group all the files associated to the same activity into a single one by using `merge_session_files` function. After that, samples within each dataset are extracted into a dictionary to simplify possible use of preprocessing routines and prepared to be fed as input to the network for the training phase. Before proceding, it is important to remark that, in order to obtain reliable results the *k-fold cross-validation* technique is used in the testing part, so different models of the same architecture are trained and the relative performances are then averaged, whereas at the end only the best accuracy and loss performance models are saved.

A first look to the sample distribution per class, helps to understand that there is a risk of biasing the model performance on the activity with the largest number of samples. The dataset in fact, is strongly unbalanced due to the particular efforts required to collect data for some class with respect to the others (e.g: see jumping) and this factor represents a limitation in the final result.
The number of models to train reflects the number of folds. The technique as well-known expects to leave one fold out and use the remaining ones for training, evaluating the generalization error on the excluded fold at the end. This procedure repeated for all the k folds produces k different models.

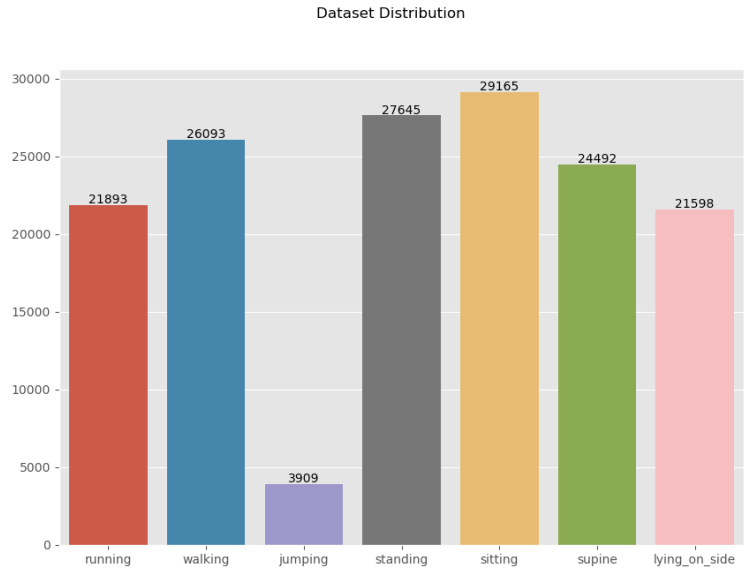


Figure 15: Dataset Distribution

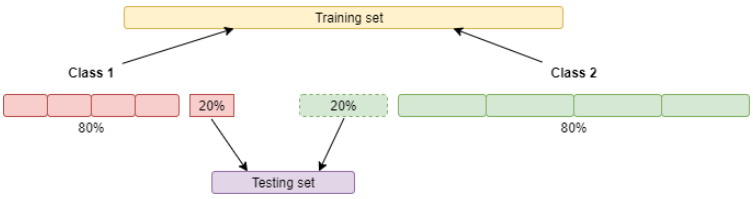


Figure 16: Dataset partitioning

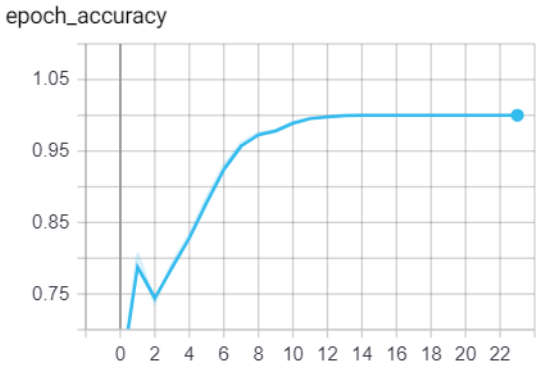
In order to compensate this unbalanced distribution, fold partitioning is performed on each activity subset and the partitions then are all grouped together.
One of the most important features of STM32Cube.AI, is that it allows to perform model testing directly on target either with randomly generated values or with a test set properly stored in a csv file. This possibility suggested to develop a function, `test_on_csv` in order to automatically create such file from the test partition of the dataset.
As it can be noticed from the code, at a certain point preprocessed data are stacked in a categorical manner, this condition forces the algorithm to learn parameters per class, biasing the final result. As usual in such cases, an effective solution is to shuffle the entire dataset adding stochasticity in the way in which different data samples are fed to the network during training, this solution is implemented by the `shuffle_dataset` method.

After data preprocessing, each network instance is trained sequentially for 60-80 epochs with 32 samples per batch, using RMSProp as optimization algorithm. Whatever solution is chosen the training settings are more or less the same except for the number of epochs which is tuned according to the adopted solution.

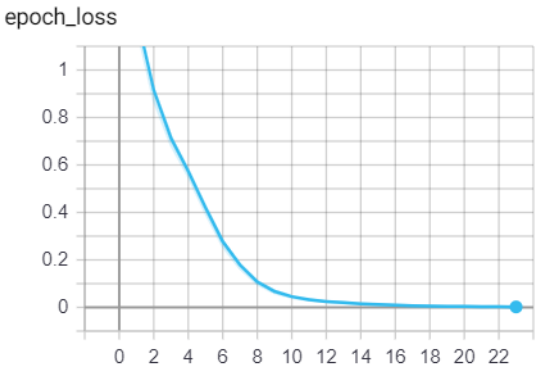
Layer (type)	Output Shape	Params
BatchNormalization	(None, 30, 12)	48
LSTM1 (LSTM)	(None, 64)	19712
FCN1 (Dense)	(None, 32)	2080
FCN2 (Dense)	(None, 32)	1056
FCN3 (Dense)	(None, 7)	231
Total params:		23,127
Trainable params:		23,103
Non-trainable params:		24

Table 2: Recurrent Neural Network: Model Summary

As the number of trainable parameters suggests, the network architecture is not so complex. Although this seems an advantage from computational perspective, it implies that the model is affected by an high variance and it tends therefore to overfit and as experimented in final testing, regardless good validation results, the application presents some slight performance degradation under changes in working conditions.
Hence different Dropout layers are added at training time and Early Stopping regularization technique has been adopted to prevent overfitting as well.



(a) Validation accuracy trend through epochs



(b) Validation loss trend through epochs

The training algorithm converges sharply after few epochs. From final validation considering all the trained models the average results appears better than expected, with:

- **Average Validation Accuracy**: 0.9927
- **Average Validation Loss**: 0.0204

More optimistic results, instead come from the best accuracy model (fold 1) as shown in the confusion matrix (Figure 18), that gives even a perfect prediction of all testing cases, the estimation however is even too optimistic:

- **Validation Accuracy**: 1.0
- **Validation Loss**: 1.682e-3

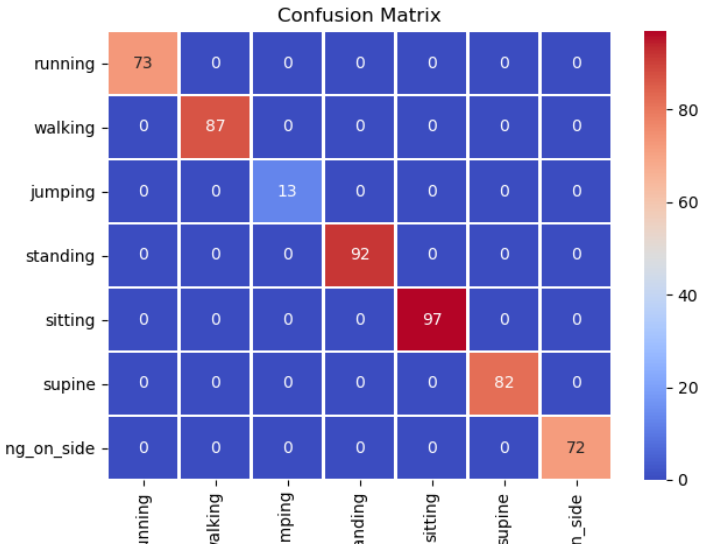


Figure 18: Confusion Matrix: Testing

As confirmed with other trained instances of the same architecture, some of them completely mispredict jumping cases, so even if the chosen model is the best one, it should not be expected to have a perfect classifier at the end. The use *k-fold cross validation* to train multiple classifiers opens up the possibility to use model ensemble techniques to obtain more reliable results, averaging predictions from all the trained networks. Unfortunately the complexity of the current architecture allows a single instance of the network to be executed on the board because of the constrained hardware.

2.5 Network Code Generation and on-target Testing

STM32CubeMX, as mentioned in previous discussions, is in charge of generating the C-code from the Keras model, precisely an initialization of the network on which is possible to build a specific application, so what is really important in this step is that the developer is not worried about the embedded implementation of each network layer details.
Before proceeding it is important to outline how the tool works in order to understand the way the network model is treated within the program environment, the AI plugin in fact, lives within STM32CubeMX that requires some preliminary configuration including the creation of a new project and some specification of the development board used.

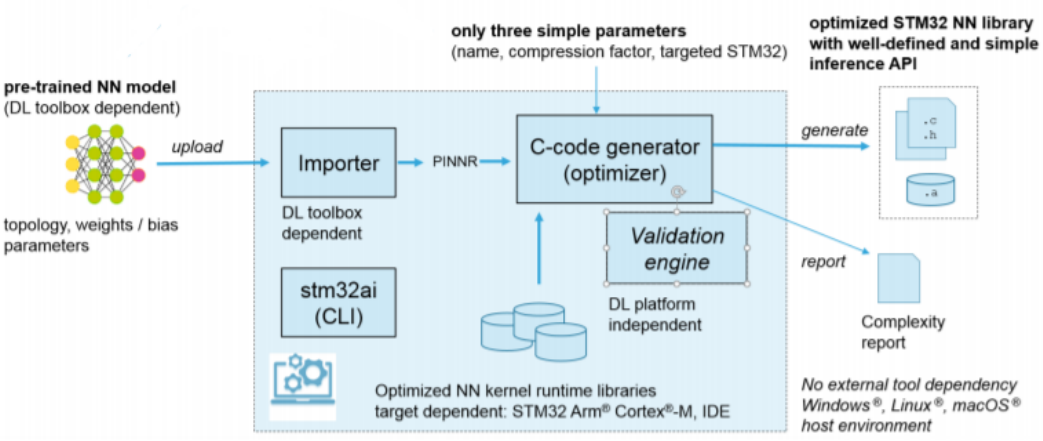
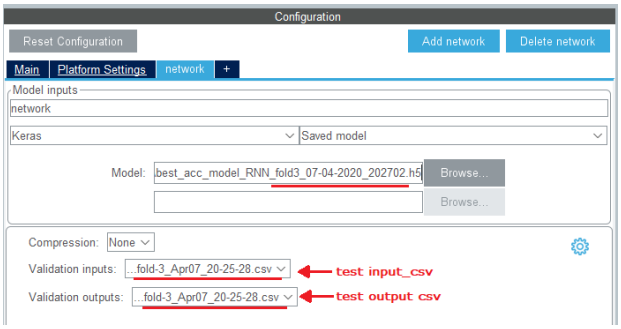


Figure 19: X-CUBE AI Engine

After that, one of the h5 model files produced during training is imported and from the specific framework model, the tool builds a platform-independent representation (PINNR).
This new version of the model is used by the *Validation Engine* to perform validation either locally or directly on the target board and the *C-code generator* that generates the network code. All this procedure can be done mainly on the GUI. However, STM32CubeAI offers a command-line interface that is exploited in this work to automatize validation and code generation and speed up this development stage.
In any case, in order to do cross-validation, the related code must be generated, compiled and flashed. This particular kind of application is provided by the tool and it is used to test the neural network directly on target and comparing validation performance between the original model, the x86 and stm32-based code.



(a) Selection of model and validation files

Complexity: 597465 MACC
Flash occupation: 93.18 KBytes
RAM: 1.70 KBytes
Achieved compression: -
Analysis status: done

Evaluation status	Acc	RMSE	MAE
x86 C-model	100.0%	0.004269	0.000425
stm32 C-model	-	-	-
original model	100.0%	0.004269	0.000425
X-cross	100.0%	0.000000	0.000000

L2R: 7.92619481e-09

Complexity: 597465 MACC
Flash occupation: 93.18 KBytes
RAM: 1.70 KBytes
Achieved compression: -
Analysis status: done

Evaluation status	Acc	RMSE	MAE
x86 C-model	-	-	-
stm32 C-model	100.0%	0.004269	0.000425
original model	100.0%	0.004269	0.000425
X-cross	100.0%	0.000000	0.000000

L2R: 7.87645416e-09

(b) Validation results on x86 and stm32

According to both the results 100% accuracy is perfectly coherent with testing performance obtained in Python scripts at training step. Since small differences in flash occupation, moreover no compression has been applied to the model.
The same operation as mentioned previously can be performed by mean of a simple batch script `validate_NN.bat` that unfortunately requires a preliminar generation of the Middleware code that can be done only once from the GUI version of X_CUBE_AI, after that the script is able to generate model dependent files and performs the validation automatically.

2.6 Code Integration and Final Firmware Development

This part of the development process begins from creating an application that easily exploits the client API generated by STM32CubeAI to compute and show one or multiple predictions from a set of data batches processed by the network. The `NN` class for instance is perfectly suitable for such purpose, it encapsulates essential information and implements methods to create, initialize and run a single instance of the neural network, so it simply uses the generated interface. As in data gathering phase, `circularQueue` is used for buffering sensor data as they are read. Furthermore, to support some preprocessing on collected samples, another class has been created, `data_proc` it implements a normalization routine and some other useful operations such as `get_argmax` and `get_mode` that compute the index of the maximum value and the mode given an input vector of integers respectively, they are ad-hoc optimized functions that help in prediction results visualization. Unfortunately, no preprocessing operation is performed on the input batch in this work since a batch normalization layer already

does standardization on it, in general is possible to do any kind of batch preprocessing just implementing the desidered function in `data_proc` and call it in `NN::prepareData` since an instance reference of such class is passed to this method.

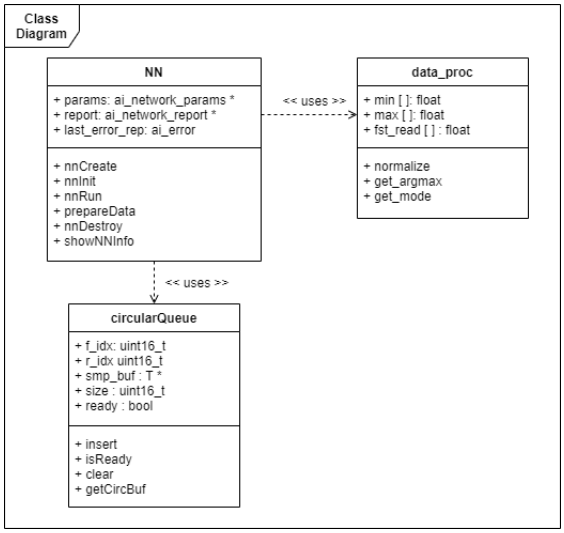
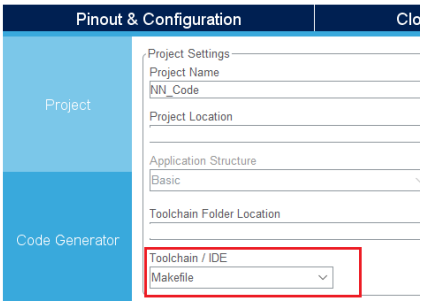
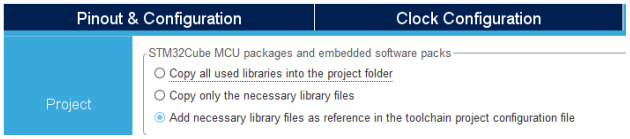


Figure 21: Class Diagram

Focusing on code generation, in order to correctly integrate the required source and object files within Miosix environment, STM32CubeAI is configured as follow:



(a) Selection of model and validation files



(b) Validation results on x86 and stm32

Regardless the selected application (*aiSystemPerformance, aiValidation, ApplicationTemplate*), the overall structure of the generated code is the same and includes the following directories:

- **Inc**: in which all header files related to the specific AI application are located, in this folder only *network.h* and *network_data.h* are specifically required in final application
- **Middlewares**: that as the name suggests includes the crucial *network_runtime* library, containing the implementation of each network specific layer and features and other header files
- **Src**: containing source application specific source-files. In the final firmware only *network.c* and *network*

To automatize code generation and installation a small batch script `build_NN.bat` performs a preliminar analysis of the keras model, exploiting the STM32CubeAI CLI generates and install all the needed sources into Miosix folder, then builds the binary file using Miosix toolchain and writes it into the F401RE's flash. Of course this simple installer is still under development and must be improved and tested properly, it does not update neither the Makefile nor the client application in case, some additional source files are added into the project or the name of the neural network is changed. Some external files moreover, must be manually included since they are related to some memory routines in glibc *eabi_memcpy.c* and *eabi_memclr.c*. Edited versions of such files are located in `additional_files` .The remaining part of the firmware resides in the `main.cpp` , in which:

- the input and output buffers (`ai_buffer`) are defined using client API's macros
- neural network parameters are allocated and associated to the neural network with `NN::nnInit` method
- `LSM6DSLAccGyr` and `LSM303AGRAccMag` are enabled and initialized with the configuration given in the `IKS01A2_config.h` header file

The `main` function runs iteratively reading from all the 4 sensors until an entire batch of 30 samples is collected, then feeds all the data to the neural network that performs a single prediction through `NN::nnRun` . An interesting aspect that will be analyzed in further section is related to a mechanism developed to suppress mispredictions due to "noisy" transitions between an activity/posture and another. This simple algorithm consists in buffering several predictions provided by the neural network and computing the mode on such set. The number of predictions mentioned is defined by `PRED_SIZE` in `main_config.h` and can be changed arbitrarily, however this affects the time required to provide the prediction of an activity or posture.

3 Experimental evaluation

3.1 Experimental setup

In order to not perturb the conditions in which initial training datasets were collected the experimental setup is exactly the one adopted in data gathering phase. Since it is required to visualize network results while doing testing activities it has been decided to rely on VNC Viewer app to establish a connection between the Raspberry Pi and a tablet. In testing part `data_gathering.py` is adapted and reused to show the output of the Nucleo board.

3.2 Results

Unfortunately due to the difficulty on keeping track of prediction results in "real-time" mode only the "noise-suppressing" solution has been tested with the results summarized in Table 3

Table 3: Testing . $M_1 \xrightarrow{\alpha} M_2$ means that an application mapping is changed from M_1 to M_2 after application α has terminated.

	Description of the workload			Cores allocation	
Name of scenario	Application	Threads	$\frac{Threads}{Cores}$	HMP	HMP w/policy
LITTLE 1	ferret†	1	1.00	0 − 3	0
	vips	3		0 − 3	$1-3 \xrightarrow{\dagger} 0-3$
LITTLE 2	freqmine†	2	1.25	0 − 3	0 − 1
	blackscholes	3		0 − 3	$0-3 \xrightarrow{\dagger} 0-3$
LITTLE 3	bodytrack†	3	1.25	0 − 3	0 − 1
	facesim	2		0 − 3	0 − 3
LITTLE 4	facesim	3	1.50	0 − 3	$0-1 \xrightarrow{\dagger} 0-3$
	blackscholes†	3		0 − 3	2 − 3

4 Conclusions and Future Works

There are many perspectives from which is possible to improve this work, first of which the adopted device. As it can be noticed in Figure 3a it is pretty uncomfortable to wear such equipment, there are many disturbing factors that affect negatively data gathering. ST Microelectronics provides in such case a very fancy solution, more packed and easy to be located on the body, the *SensorTile STEVAL-STLCS01V1*, that includes all the adopted MEMS'. Such device together with a more rigorous methodology in collecting data would improve considerably the neural network performance. This last point in fact, is another aspect to be improved: the training dataset is entirely profiled on a single person, movements and positions follow rigid sequences with few variations, so a tight range of cases are really provided during training phase, this is unfortunately caused by the considerable efforts requested to gather data for a single person and the unavailability of a range of subjects with different athletical skills and physical characteristics to improve the quality of the dataset.
Another interesting future goal is to enable the application to host a more complex neural model in order to open the way to a more specific and useful AI task also related to physical activity.

References

[1] Angelica Munoz-Melendez. Irvin Hussein Lopez-Nava. Wereable intertial sensor for human motion analysis: A review. 2016.