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

March 31, 2020

Contents

List of Figures

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 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 is particularly intuitive that this stage In this study a simple classification problem using embedded neural network will be analyzed in order to highlight which advantages can be obtained by moving AI algorithms on embedded devices

1.1 Problem statement

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 the need of c

1.2 Summary of the work

In this project 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 special-purpose embedded platforms. The purpose of the classification task is to recognize a set of movements and positions of the human body by simply using an STM32 Nucleo board. The motivation of this early choice is based on the fact that when dealing with AI in the embedded field, one of the most tricky task for programmers is the implementation of standard aspects related to popular AI algorithms (in case of neural networks: layers, activation functions implementation and so on), sometime is possible to speed-up this long and tedious part of the development process thanks to libraries and tools provided by silicon manufacturers. All the stages from data gathering, through modeling and training the neural network, to the development of the firmware code will be discussed, remarking the advantages of recurring to embedded software tools and APIs directly provided by MCU manufacturing companies. The peculiarity of the study is the simplicity of the path that led from the network model to the embedded code. At this stage most of the work was performed by a powerful plugin integrated into the STM32CubeMX program, STM32Cube.AI realized by ST Microelectronics to facilitate bare-metal development. Another important result of this work is that the generated network code has been deployed directly on top of a real-time operating system, Miosix, showing the possibility to exploit typical real-time features of these OSs 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 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 Science and Network modeling**: 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.
- **Final Testing**: testing of the embedded neural network by performing real-time collection and classification of single batches of data samples.

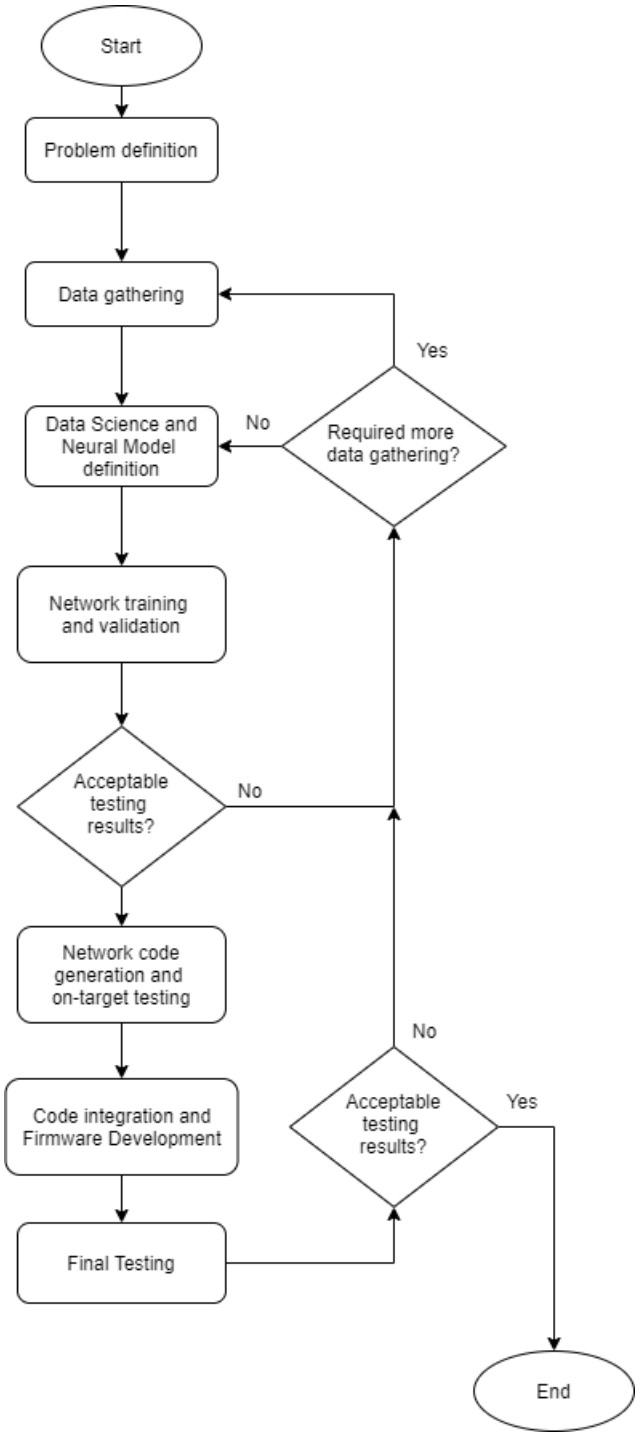


Figure 1: Design Process Chart

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.

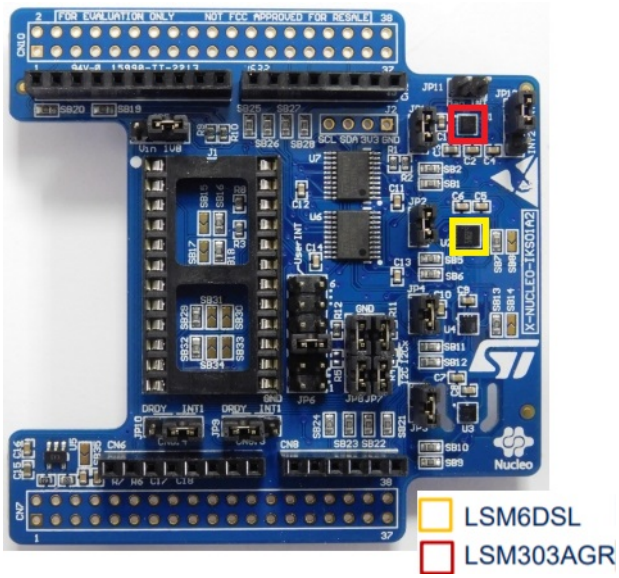
2.2 Data Gathering

This part is strongly affected by some a priori design 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:

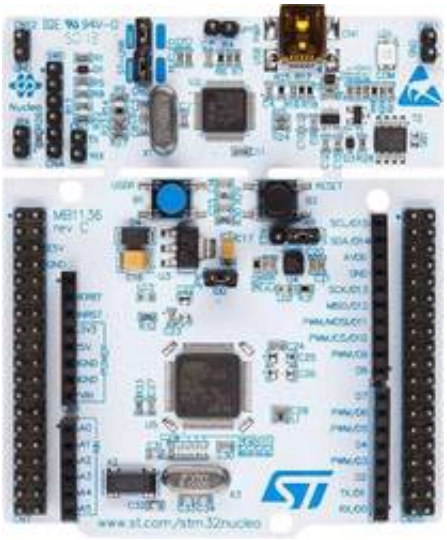
- Allows the sensors to detect directional and angular accelerations of the right lower limb which characterizes most of the movements and positions that must be classified
- Avoid the person wearing the device to be hampered during activity sessions

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)

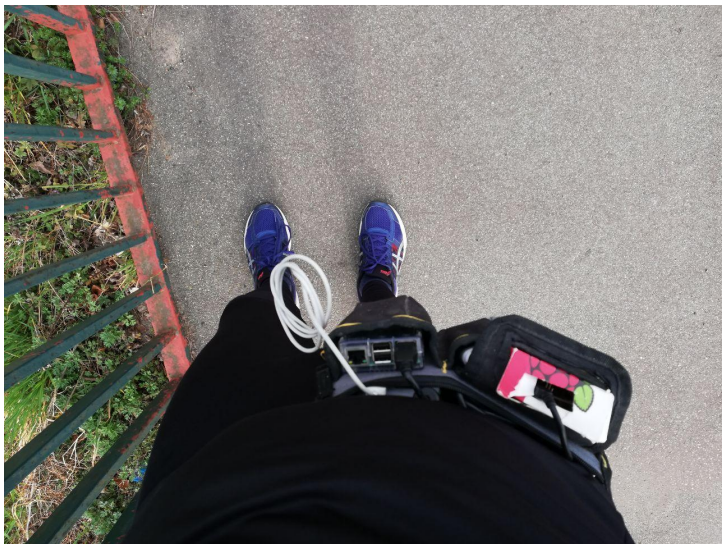


(a) IKS01A2 Sensor Board



(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	

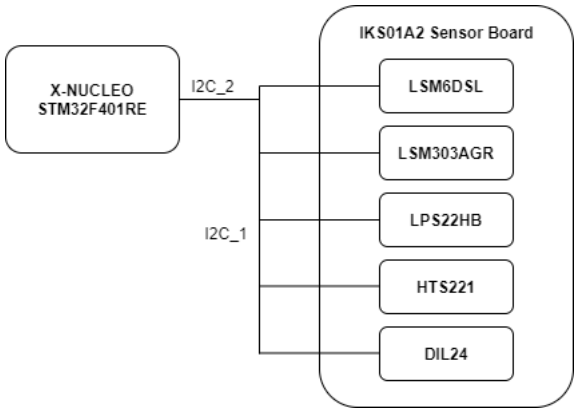
Table 16. Transfer when master is receiving (reading) one byte of data from slave

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();

for (int j = 0; j < numbByte; j++) {
        _i2c_dev::sendStart();
        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;
}
```

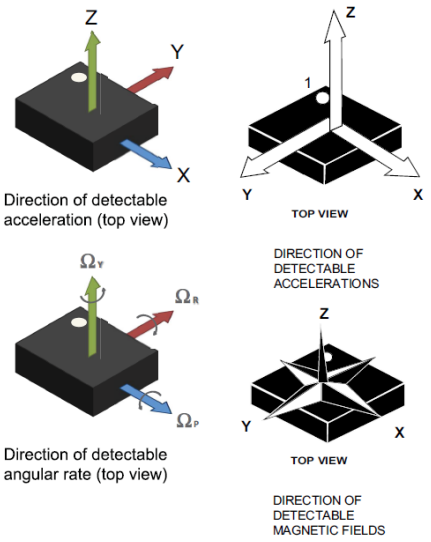
(d) i2c_helper.cpp: write

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. It runs

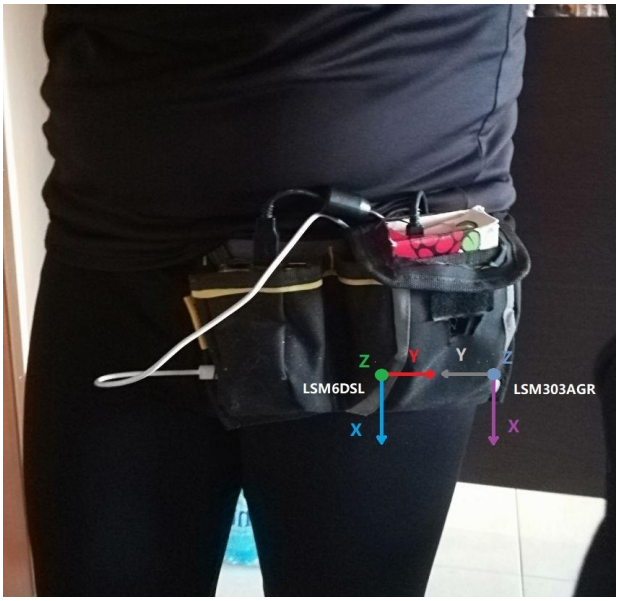
similarly for both LSM6DSL and LSM303AGR, setting the values to those registers that are more likely to be used for gathering purpose. In both classes, with slight differences, `io_read()` and `io_write()` act as a I2C interface to allow read and write operation respectively, Some of them, such as

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:

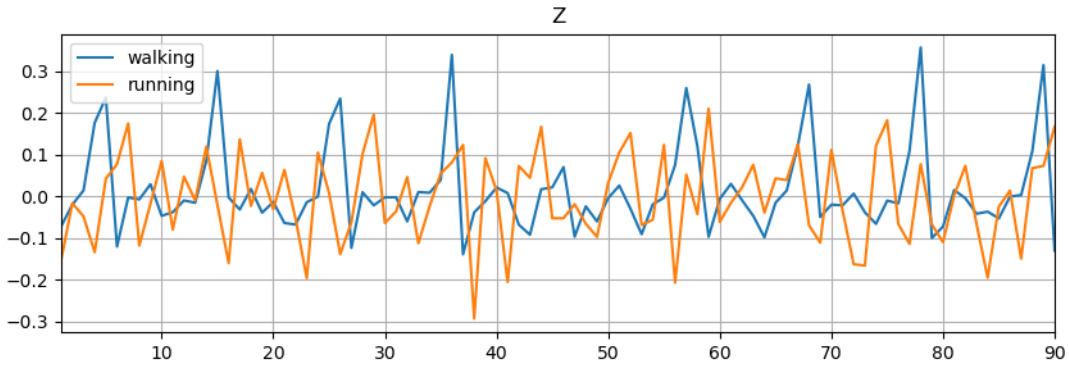


Figure 6: LSM6DSL Accelerometer: running and walking

This can be noticed from the "weird" behaviour in running trend with respect to walking, that on the other hand presents accentuated, but more distanced peaks (representing various steps).
An easier comparison regards jumping and standing activities which instead, present more distinguishable regularities on X:

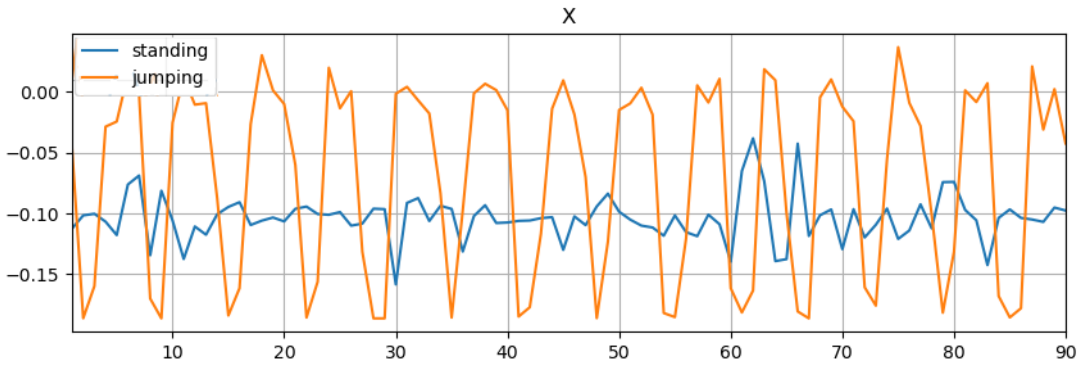


Figure 7: LSM6DSL Accelerometer: jumping and standing

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

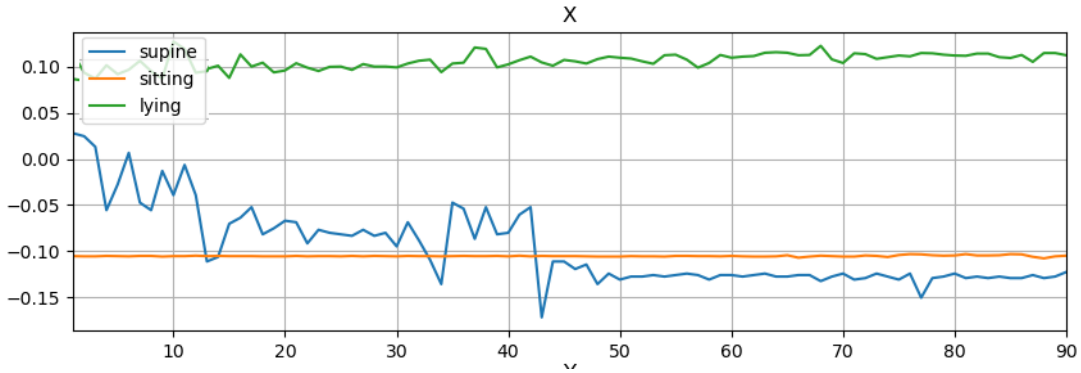


Figure 8: LSM6DSL Accelerometer: sitting, supine, lying on side

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 needs not so many samples to learn them. Although other sensor trends do not show interesting regularities, from another perspective the importance of magnetometer in the pattern recognition is remarked by seeing directly to the distribution of a bunch of data points in a 3D scattering plot:

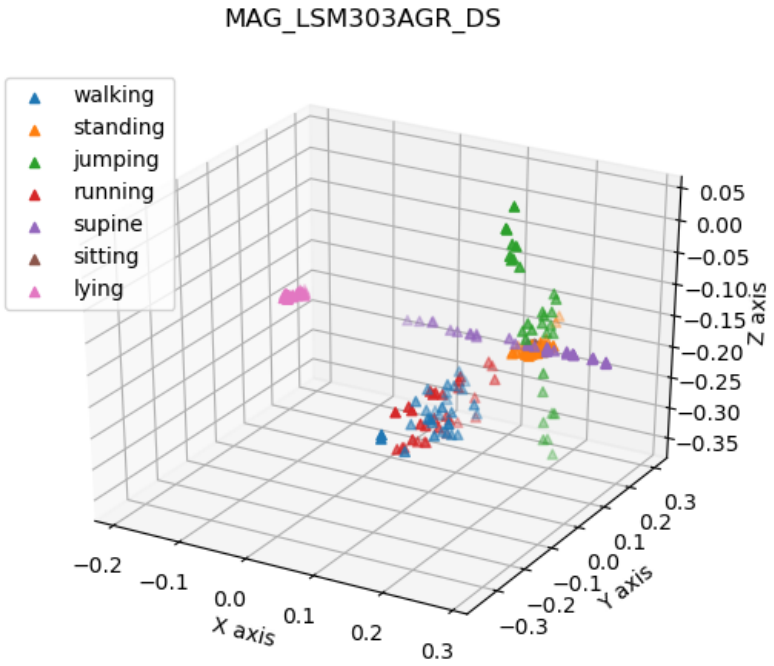
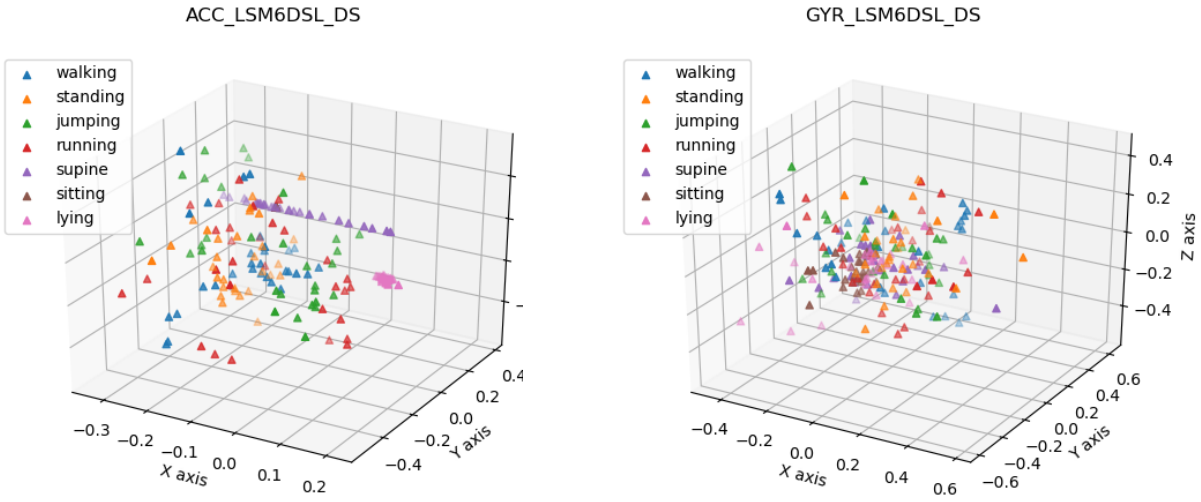


Figure 9: LSM303AGR Magnetometer, Scattering Plot

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 plot of the LSM6DSL's accelerometer and gyroscope is 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. But beside a common LSTM-based model an alternative solution is adopted, a simple Convolutional Neural Network (CNN) to demonstrate that even under constrained hardware, it is possible to deploy computationally expensive models.
The first one consists in a 4-layer LSTM architecture, 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.

- **4 LSTM** cells: to capture patterns on different time scales. The specific number of layers does not have a real explanation, is the combination that has given the best performance so far. Each of these has 32 hidden nodes, the activation and recurrent activation functions chosen are *tanh* and *sigmoid* respectively.

- **Fully-connected** layer: composed of 7 neurons implementing a *softmax* function, these last step of the neural network is the real classifier, it evaluates a score per each activity class starting from the state of the last LSTM step.

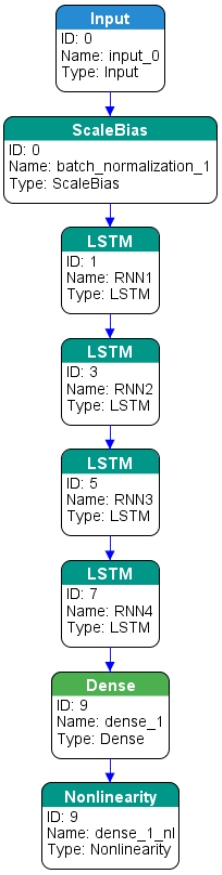


Figure 11: RNN model graph

Here some implementation details:

```
def RNN_model():

    hid_nodes_lstm = 32
    rnn_model = Sequential()
    rnn_model.add(BatchNormalization(input_shape=(WINDOW_SAMPLES, SENS_VALUES)))
    rnn_model.add(LSTM(units=hid_nodes_lstm, return_sequences=True, name='RNN1'))
    rnn_model.add(Dropout(0.2))
    rnn_model.add(LSTM(units=hid_nodes_lstm, return_sequences=True, name='RNN2'))
    rnn_model.add(Dropout(0.2))
    rnn_model.add(LSTM(units=hid_nodes_lstm, return_sequences=True, name='RNN3'))
    rnn_model.add(Dropout(0.2))
    rnn_model.add(LSTM(units=hid_nodes_lstm, return_sequences=False, name='RNN4'))
    rnn_model.add(Dropout(0.2))
    rnn_model.add(Dense(num_classes, activation='softmax'))
```

The other solution consists in a small Convolutional Neural Network, with the following architecture:

- **Batch Normalization**
- **2D Convolutional** layer: performs the convolution to learn a set of 24 filters. In this case batches are represented as groups of matrices 30×12, so can be thought as a sort of image and each filter scans all the "image" pixels

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.

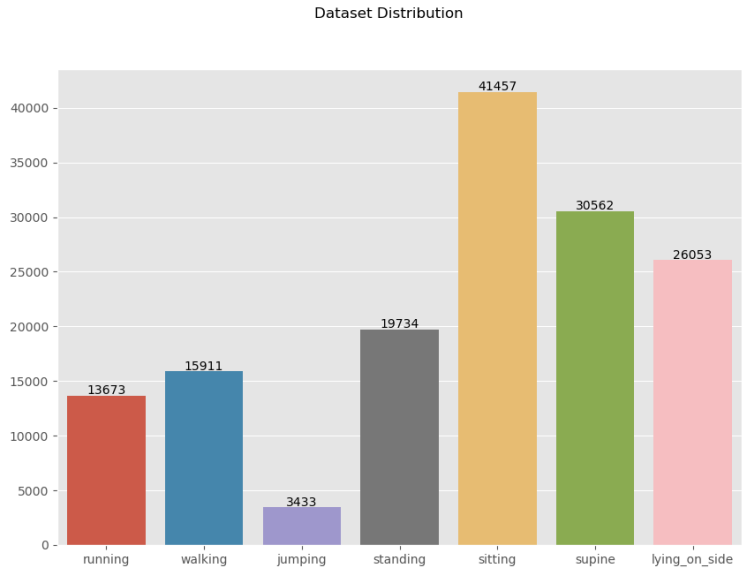


Figure 12: Dataset Distribution

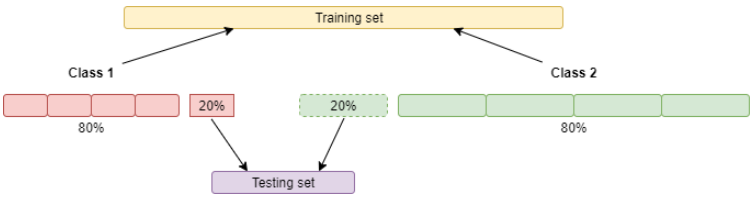


Figure 13: Dataset partitioning

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.

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 features.
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 into STM32CubeMX that requires some preliminary configuration including the creation of a new project and some specification of the development board used. 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).

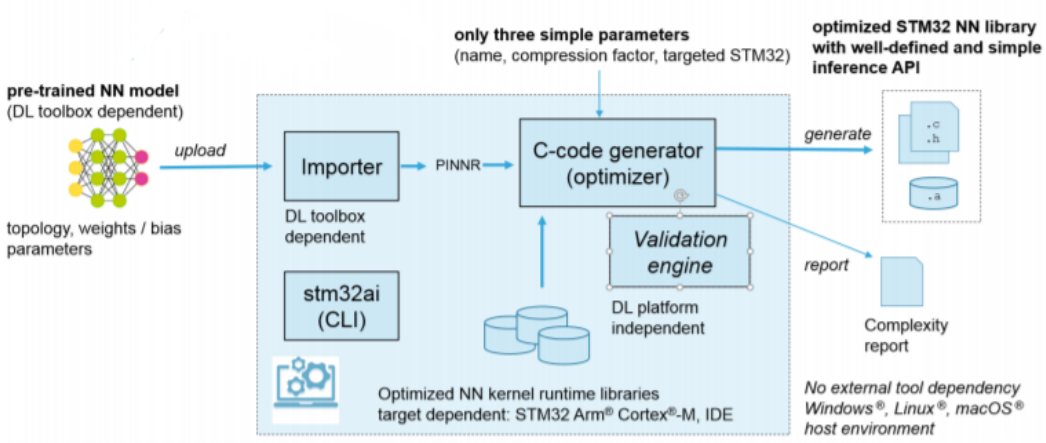


Figure 14: X-CUBE AI Engine

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.
So the first step, the model is imported into the project Furthermore, it is important to highlight another interesting aspect that raises from a first analysis of the model summary created during training phase:

Layer (type)	Output Shape	Params
BatchNormalization	(None, 30, 12)	48
LSTM1 (LSTM)	(None, 30, 32)	5760
LSTM2 (LSTM)	(None, 30, 32)	8320
LSTM3 (LSTM)	(None, 30, 32)	8320
LSTM4 (LSTM)	(None, 32)	8320
FCN1 (Dense)	(None, 7)	231
Total params:		30,999
Trainable params:		30,975
Non-trainable params:		24

Table 1: Recurrent Neural Network: Model Summary

2.6 Code Integration and Final Firmware Development

A challenging part in this work is represented by the integration It includes the following folders:

- **Drivers**: containing architecture-specific source files and in particular the implementation of network layer operations and non-linear activation functions
- **Inc**:

The main goal of this work is to show the resulting performance in terms of accuracy and response time required for the network to provide a single prediction

3 Experimental evaluation

3.1 Experimental setup

3.2 Results

Please make sure you explicitely cite the tables. Table 2 shows a complex table. Please also make sure you provide each table with an exhaustive caption. Captions for tables must be placed before not after the tables.

Table 2: Summary of the test scenarios. $M_1 \xrightarrow{\alpha} M_2$ 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		$0-3$	$2-3$
big 1	vips	3	1.00	$4-7$	$4-5 \xrightarrow{\dagger} 4-7$
	ferret†	1		$4-7$	$6-7$
big 2	freqmine†	2	1.25	$4-7$	$4-5$
	blackscholes	3		$4-7$	$6-7 \xrightarrow{\dagger} 4-7$
big 3	facesim	2	1.25	$4-7$	$4-5$
	bodytrack	3		$4-7$	$4-7$
big 4	facesim	3	1.50	$4-7$	$4-5 \xrightarrow{\dagger} 4-7$
	blackscholes†	3		$4-7$	$4-7$

4 Conclusions and Future Works

References

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