Deliver the Intelligence at the Edge via On-Device Training of Deep Learning Models on Microcontrollers

Rawan Nawaiseh, Fabrizio De Vita, Dario Bruneo

Università degli Studi di Messina
Messina, Italy

SmartMe.io s.r.l.
Messina, Italy

7th Italian Workshop on Embedded Systems, Bari 22-23 September

Introduction

The pervasive diffusion of Cyber Physical Systems (CPSs) profoundly changed the way we interact with the surrounding environment.

The advent of Artificial Intelligence (AI) acted as a catalyzer for the widespreading of machine learning applications into a new type of devices usually defined Intelligent Cyber Physical Systems (ICPSs).

Challenges:

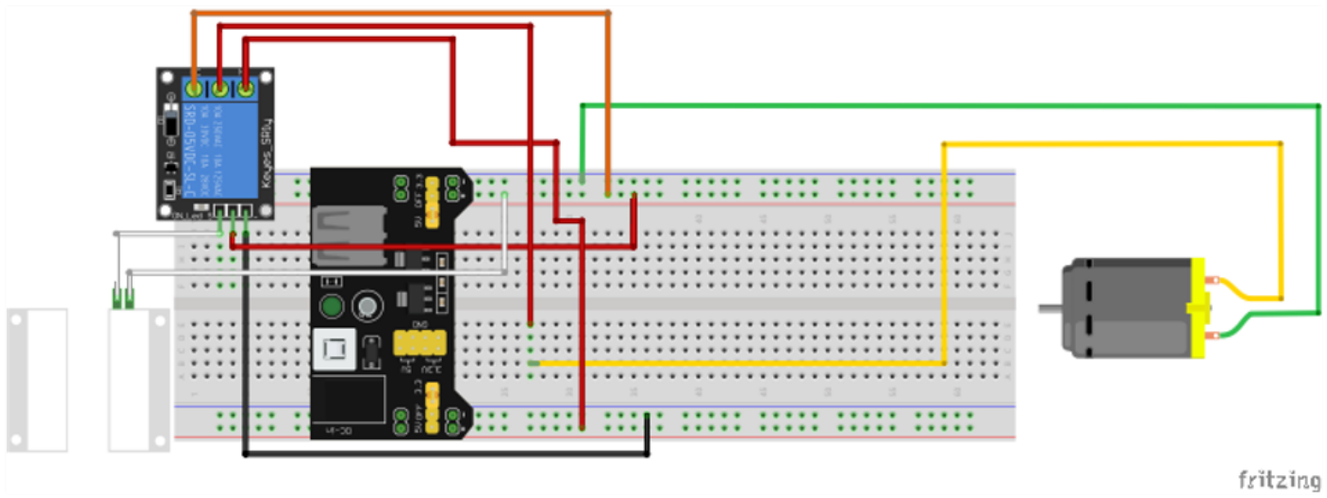
- Limited computing power;
- Battery driven devices.

Until recently the use of hybrid Cloud/Edge infrastrcutures has been the most used solution, but becomes ineffective when there is the necessity of learning new patterns in real-time.

Solution:

Realize an On-Device training/inference system that is able to learn and generate predictions without the need of external components.

Testbed description



In this work, we focused on the implementation of an anomaly detection algorithm to assess the working conditions of a DC motor.

The testbed consists of four components:

- DC motor;
- Relay;
- Magnetic switch;
- ICPS (running the anomaly detection algorithm).

The testbed has been designed to inject anomalies into the power supply system.

We analyze the vibrations generated by the motor to evaluate its "health" state.

ICPS Platform

Deploy ML models on smart boards with Micro Controlling Unit (MCU)

- low power consumption;
- real-time inference (MCU run algorithms);
- low cost hardware.

Cloud-JAM specifics:

•ARM M4 MCU (STM32F401RET6U);
•512 Kbytes of Flash memory;
•96 Kbytes of RAM;
•3D accelerometer.

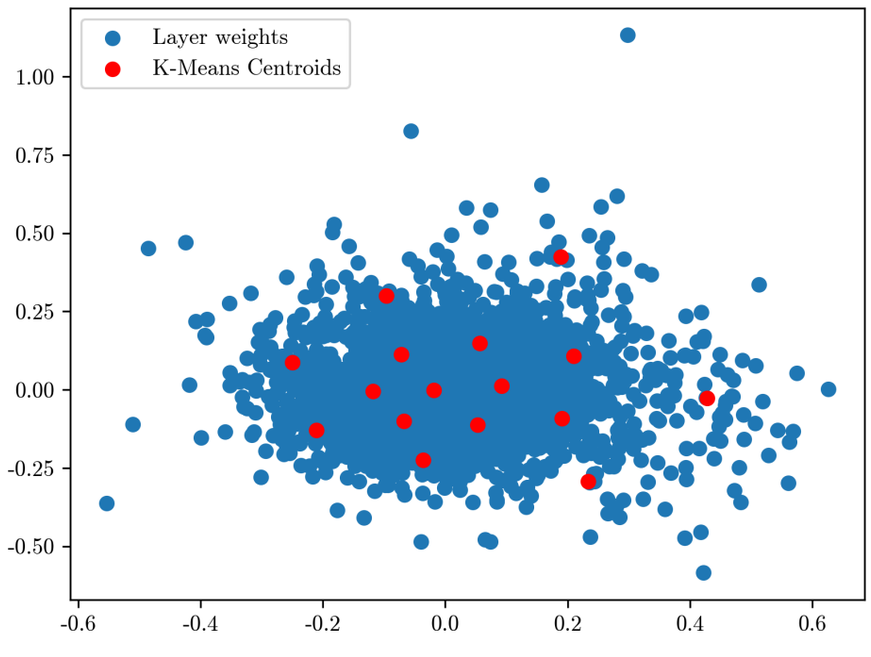
Challenge: Run AI algorithms into constrained devices, while maintaining a high level of performance.

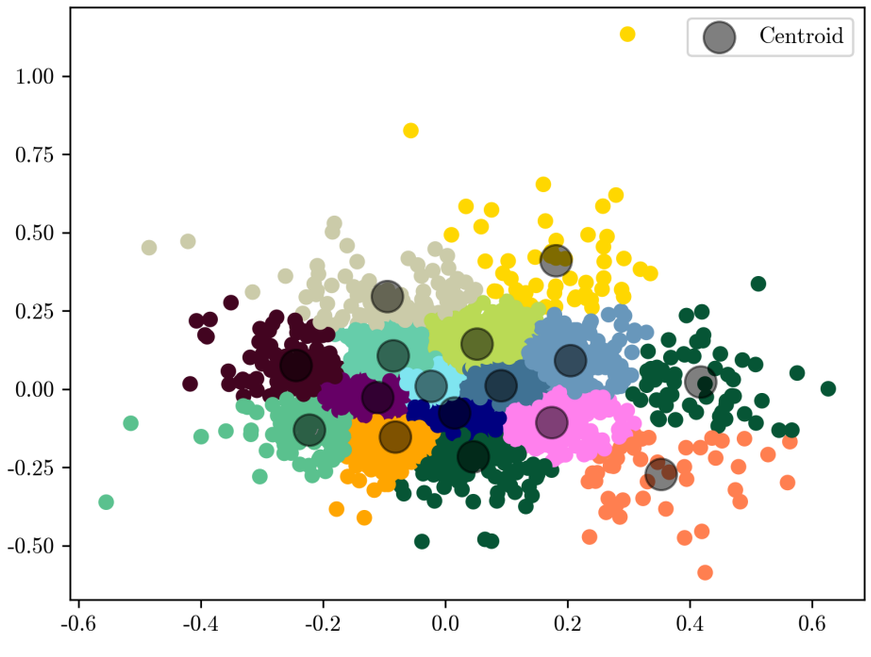
Solution:

Two conversion techniques:

- Weights compression;
- Quantization.

Weights compression





Weights compression is a viable solution to reduce the model memory footprint to fit the hardware constraints.

Applicable only to dense layers where the most part of weights is concentrated.

K-Means algorithm is used to cluster the layer weights into a reduced number of centroids.

The number of centroids depends on the target_factor fixed for the compression as follows:

$$n_{centroids} = 2^{32/target\_factor}$$

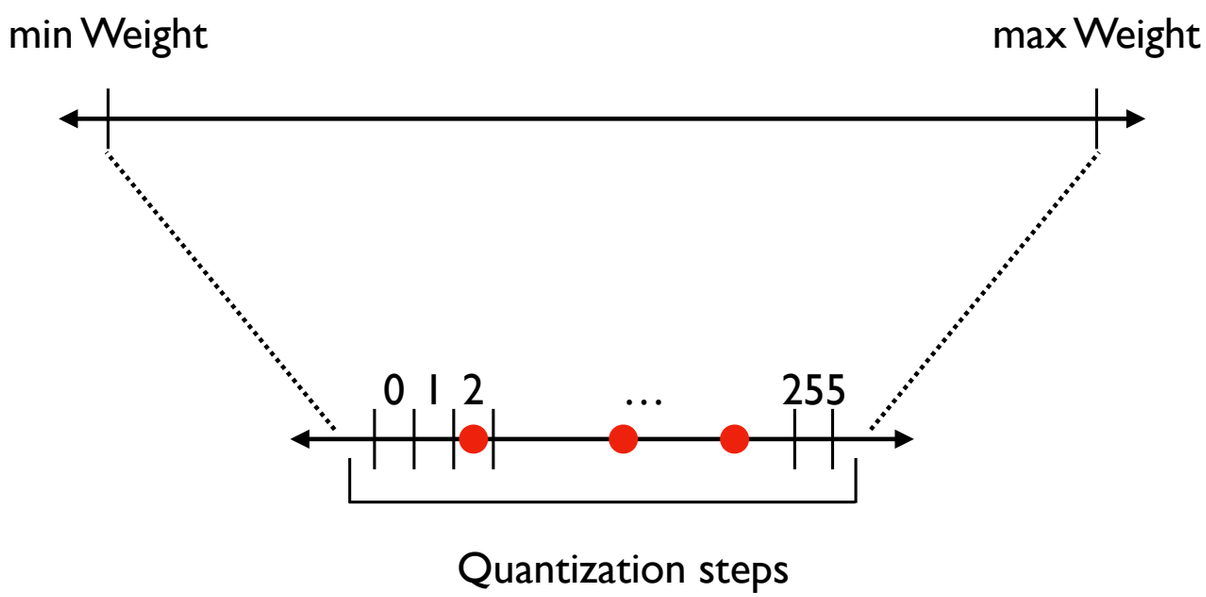
Quantization

Quantization reduces the memory footprint of the model while improving CPU performance

- Weights, Biases, and activation functions are converted from Float to 8 bit precision;
- Generation of an optimized C code.

Two types of quantization supported

- Integer quantization;
- Fixed-point quantization (Qm,n format).

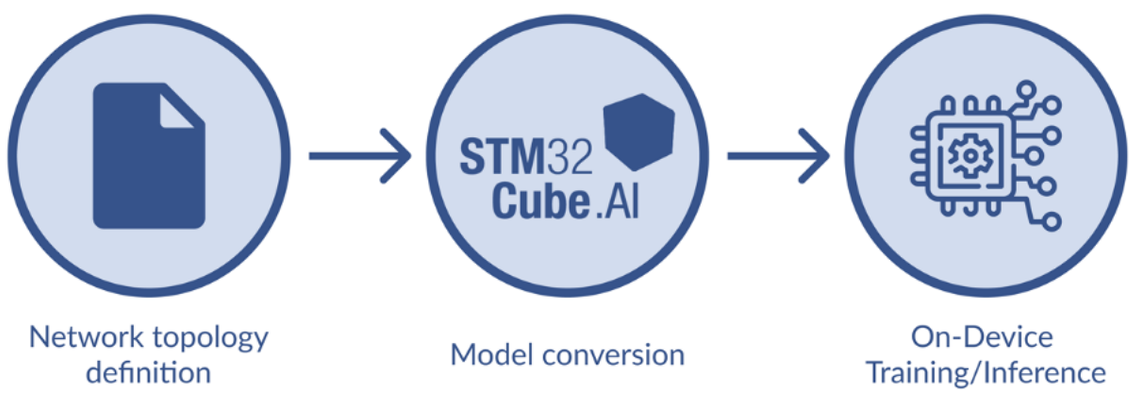


On-Device Training

Unfortunately, weights compression and quantization techniques solve only the problem related to the memory footprint, but they are subject to some limitations:

- Hyperparameters tuning (number of centroids, type of quantization, number of bits, etc…);
- Model re-training could be necessary (quantization);
- Can be applied only to pre-trained models.

Thanks to the STM32 Cube AI tool it is possible to enable an on-device training/inference process starting from a network topology.



Echo State Networks

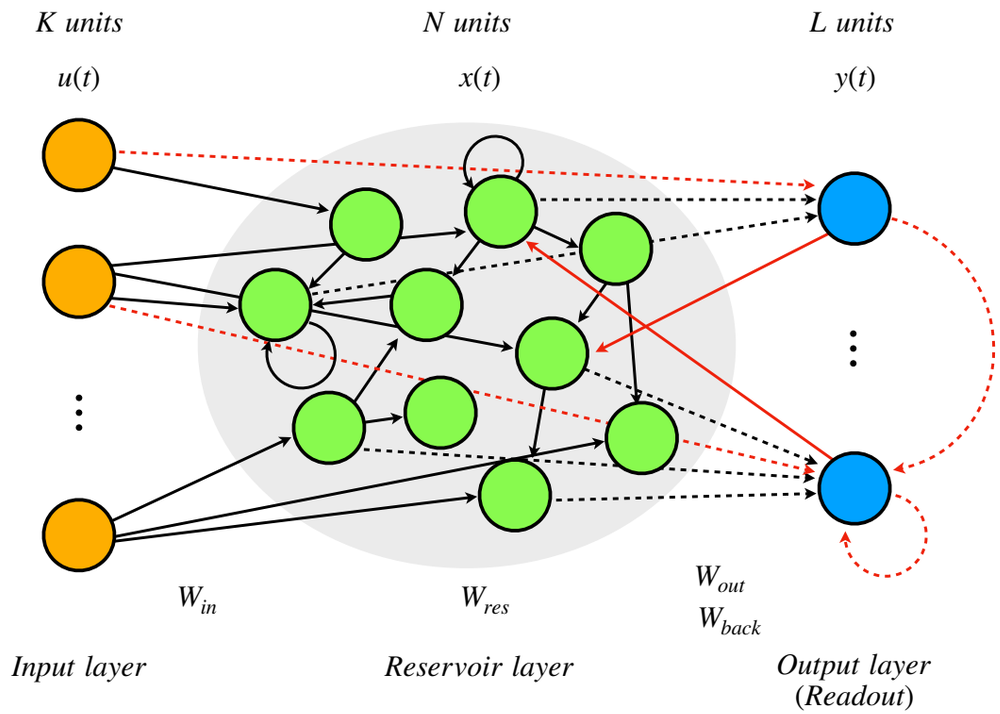
An ESN is a Recurrent Neural Network (RNN) with a sparse randomly connected recurrent structure (the *reservoir*) and an output part called *readout*.

State equation

$$x(t+1) = f(W_{in} \cdot u(t+1) + W_{res} \cdot x(t) + W_{back} \cdot y(t))$$

Output equation

$$y(t+1) = g(W_{out} \cdot [u(t+1), x(t+1), y(t)])$$

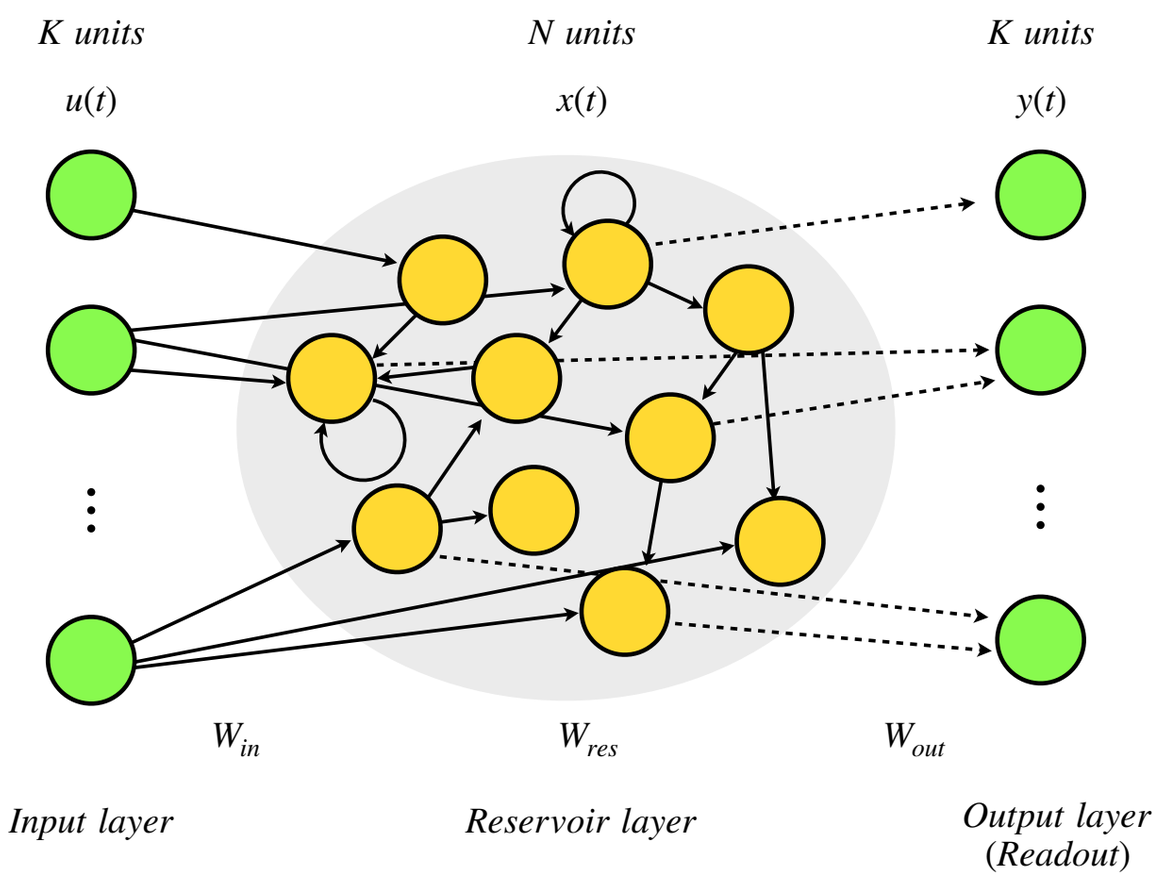


The reservoir weights (i.e., W_{res}) are randomly set and remain fixed during the entire training procedure.

The only trainable weights are those connecting the reservoir and the readout (i.e., W_{out})

- W_{out} weights can be computed solving a linear regression problem;
- Reduced model complexity;
- Faster training.

ESN Autoencoder



The use of an Autoencoder topology allowed us to address the anomaly detection problem with an unsupervised approach (no need of labeled datasets).

The ESN Autoencoder analyzes the vibration data generated from the testbed to learn the temporal patterns describing a normal behavior.

Custom ESN layer

Algorithm 1: Custom ESN layer interface.

```
class EsnCustomLayer(keras.layers.Layer):
    def __init__(self, K, N, L, ρ, sparsity):
        super(EsnCustomLayer, self).__init__(**kwargs)
        self.ESN_NEURON_IN ← K
        self.ESN_NEURON_RESERVOIR ← N
        self.ESN_NEURON_OUT ← L
        self.ESN_SPECTRAL_RADIUS ← ρ
        self.ESN_SPARSITY ← sparsity
        self.ESN_PREVIOUS_STATE ← [0, 0, ..., 0]
        W_in, W_res ← generateWeights( )

    def fit(self, train_data):
        W_out ← solveMultivariateLinearRegression()

    def call(self, input):
        state ← f(W_in · input + W_res · self.ESN_PREVIOUS_STATE)
        output ← g(W_out · [input, state])
        self.ESN_PREVIOUS_STATE ← state
```

Unlike traditional layers, the **fit** has been overriden to compute the output weights solving a multivariate linear regression problem in a closed form.

Considering a scenario with a low number of features the use of a closed form solution is preferred over backpropagation for several reasons:

- Lower number of hyperparameters (e.g., optimizer, learning rate, training epochs, etc.);
- Computationally less intensive;
- It always guarantees the optimal solution.

On-Device Anomaly Detection

Algorithm 2: Loop section of the anomaly detection algorithm.

```
1  while True do
2      if USR button was pressed then
3          training_samples ← Collect_training_dataset( );
4          training_complete, THR ←
             On_device_training(training_samples);
5          Blink_LED( ) ;    /* Give feedback to
             the user */
6      end
7      if training_complete then
8          sample ← Collect_data( );
9          reconstruction ← Predict(sample);
10         error ← MSE(sample, reconstruction);
11         if error ≥ THR then
12             Turn_ON_LED( ) ; /* Give feedback
                 to the user */
13         else
14             Turn_OFF_LED( ) ;          /* Give
                 feedback to the user */
15         end
16     end
17 end
```

On-Device training:

The ESN Autoencoder is trained to learn the patterns which describe a normal condition.

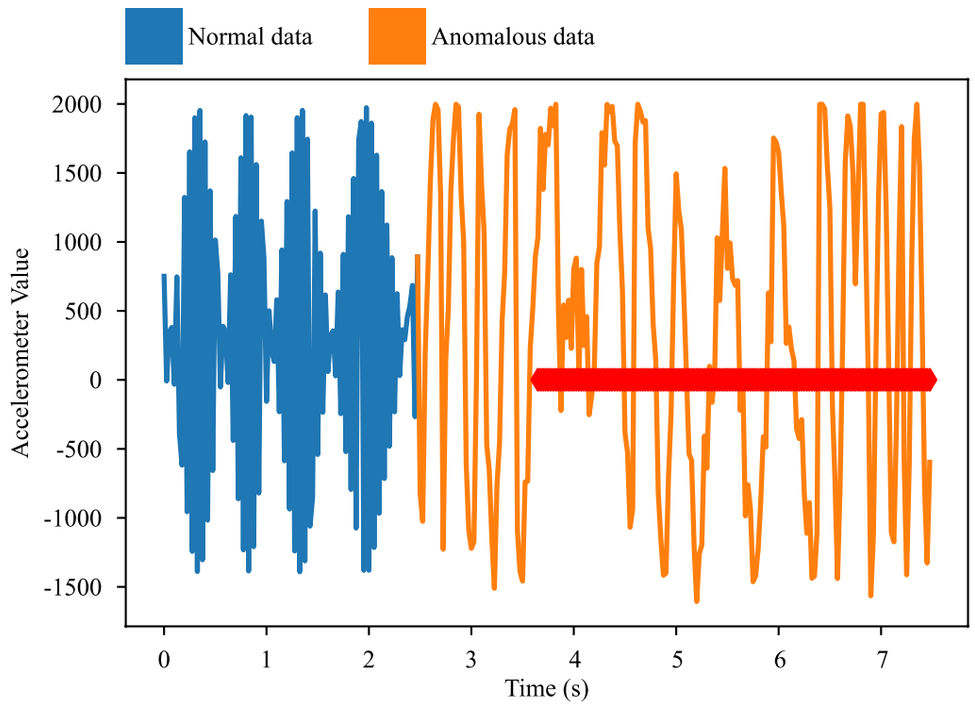
On-Device inference:

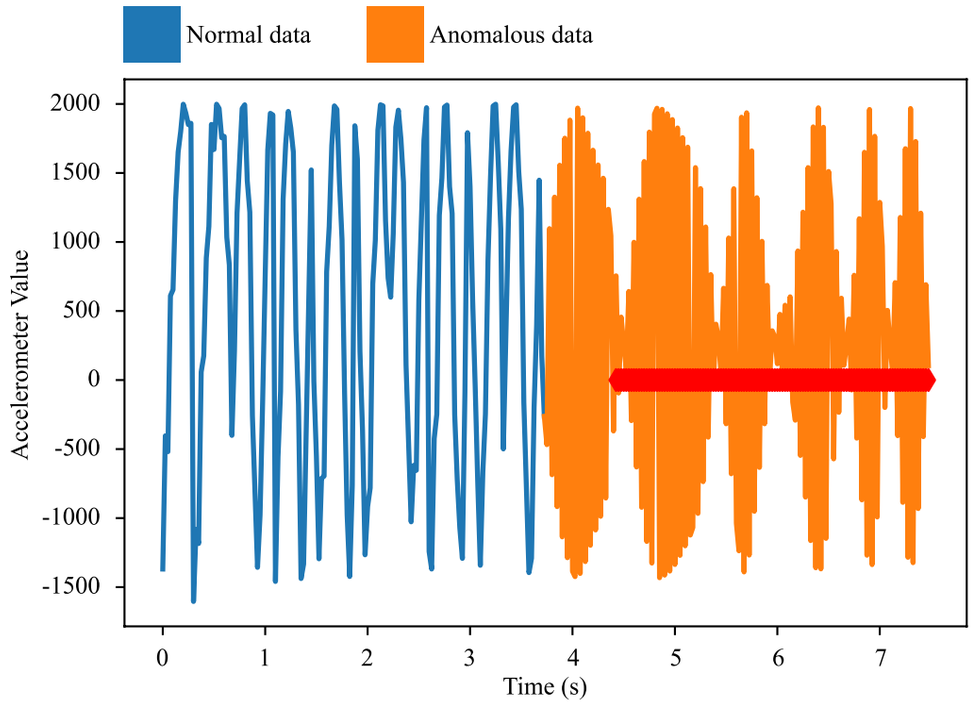
After the training phase, we use the Mean Squared Error (MSE) to discriminate between normal and anomalous vibration sequences.

$$threshold = \widetilde{MSE_{train}} + \sigma(MSE_{train})$$

$$anomaly(t) = \begin{cases} 1 \ if \ MSE > threshold \\ 0 \ otherwise \end{cases}$$

Experimental Results

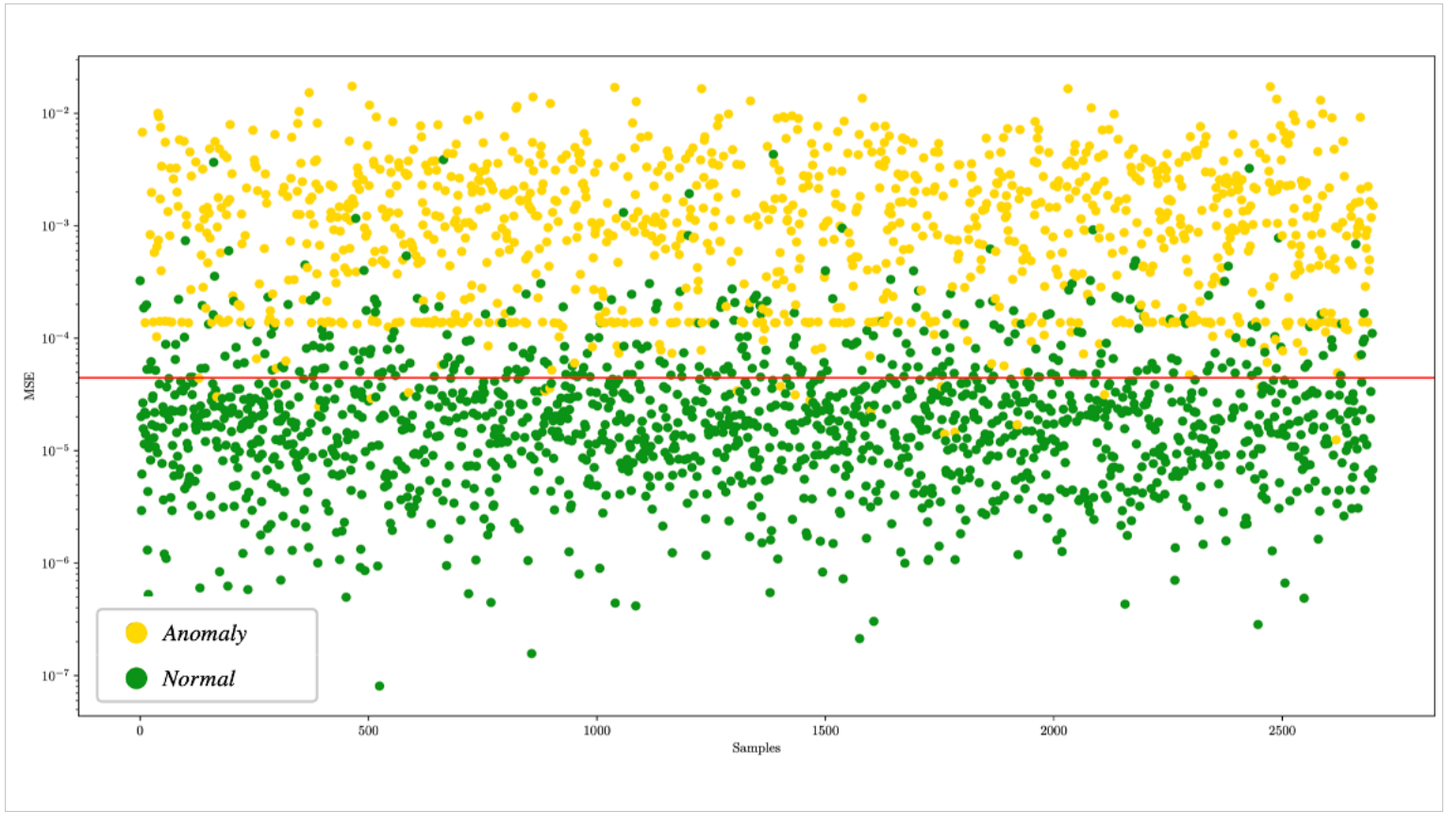




We conducted two sets of experiments acting on the magnetic switch of the testbed to change in real-time the DC motor power supply.

In both the experiments the algorithm has been able to detect the occurrence of anomalies with an acceptable delay (about 1 second).

Experimental Results (cont'd)



We conducted another experiment to verify the goodness of the computed threshold.

The MSE threshold computed during the training phase (see the red line) allowed to reach very good performance with a precision of **0.808**, a recall of **0.983**, and a F1-score of **0.887**.

These results, albeit preliminary, encourage us to test our system in a real industrial scenario.

Thank you !

Rawan Nawaiseh
SmartME.io s.r.l. (Italy)
rawan@smartme.io

Fabrizio De Vita
Department of Engineering
University of Messina (Italy)
fdevita@unime.it

Dario Bruneo
Department of Engineering
University of Messina (Italy)
dbruneo@unime.it
dario@smartme.io