UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

**DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE**

**CORSO DI LAUREA MAGISTRALE IN**

**COMPUTER ENGINEERING**

# "CONDITION MONITORING BASED ON ANOMALOUS SOUND DETECTION VIA AUTOENCODERS"

**Relatore: Carlo Fantozzi**

**Laureando: Mirko Vettori**

**Correlatore: Ph.D.  Roberto Bortoletto**

**ANNO ACCADEMICO 2021 – 2022**

**Data di laurea 18/10/2022**

# Abstract

Condition Monitoring is the process of monitoring a parameter of a particular machine, for the purpose of identifying developing anomalies.
In this thesis, in cooperation with *221e S.r.l.* during an internship, an autoencoder-based Condition Monitoring system is proposed, with the aim of detecting anomalies in machines using sound signals.
Sound indicators in Condition Monitoring offer multiple advantages over more traditional metrics like temperature, vibration, or voltage. Anomalies can be detected before major malfunctions occur, the machine can be monitored without physical contact and a large set of different anomalies can be detected.
The system was developed and evaluated on three real user scenarios provided by the company. Different conditions and settings, with customized data acquisitions and training of the model, were considered.
In the end, an embedding of the monitoring solution into the microcontroller multi-sensor board *STWIN* is considered and validated on the device.

# Sommario

Per monitoraggio condizionale si intende il processo di monitoraggio di un parametro in un macchinario, allo scopo di identificare sviluppi di anomalie.
In questa tesi, in collaborazione con *221e S.r.l.* in occasione di un tirocinio, si vuole proporre un sistema di monitoraggio condizionale basato su un autoencoder, allo scopo di identificare anomalie nei macchinari grazie all'utilizzo di segnali sonori.
Gli indicatori sonori nel monitoraggio condizionale offrono molti vantaggi rispetto a metriche più tradizionali come la temperatura, le vibrazioni ed il voltaggio. Le anomalie possono essere rilevate con largo anticipo rispetto alla comparsa di gravi malfunzionamenti, la macchina può essere monitorata senza necessità di contatto fisico con il sensore e possono essere rilevati molti tipi di anomalie diverse.
Il sistema è stato sviluppato ed applicato su tre casi studio pratici forniti dall'azienda, ognuno con diverse condizioni e caratteristiche, con acquisizioni dati e training specifici del modello.
Alla fine è discussa e considerata un'applicazione end to end del sistema all'interno del microcontrollore *STWIN* con una validazione del modello in ambiente embedded.

# 1. Introduction

Condition Monitoring is a field of study that is gaining more and more interest in recent years. It consists of the monitoring of a specific metric or parameter of a machine, in order to identify anomalous behaviors that might eventually lead to system failures.

Condition Monitoring systems are nowadays essential to detect anomalies and prevent catastrophic failures in delicate machines in industrial and domestic environments. These systems however are often just based on pure threshold strategies on a directly measured phenomenon: like vibration, temperature, or voltage. When the particular sensor detects a value outside of a prefixed normal range an anomaly is detected and adequate risk-containing actions can be performed.

More recently, however, with the rise of Artificial Intelligence (AI) and machine learning, it is possible to build Condition Monitoring systems based not only on simple direct measurements, but on more complex patterns and parameters.

Sound is an example of a complex set of parameters that often cannot be automatically interpreted without the help of AI. Sound-based Condition Monitoring opens the door to new possibilities: very different anomalies can be detected by the same system, which is no longer solving a linear problem with a simple threshold-based strategy, but it is considering a whole complex spectrum of parameters that can be handled with the help of machine learning algorithms.

In this work, in cooperation with *221e S.r.l.* on the occasion of a working internship, the basics of machine learning and digital sound are covered, in order to promote and argue the advantages of a semi-supervised sound-based Condition Monitoring strategy.

Sound-based Condition Monitoring systems listen to the sounds emitted by the monitored machine in order to classify them as normal or anomalous sounds. They have the advantage of being less invasive compared to traditional Condition Monitoring systems since they do not require direct contact with the monitored machine. They also can be more general, sound provides more information than temperature or mechanical vibrations, and can characterize a larger range of different anomalies, even if they are not known a priori.

An original sound-based Condition Monitoring system is then presented, which is based on an autoencoder followed by a threshold-based strategy, which is dynamically handled at training time by an original *threshold selection algorithm*, in a semi-supervised way.

The system is then applied to three different case studies provided by *221e S.r.l.*, with some original data acquisitions, also on the ultrasound spectrum, in order to test the system on very different environments and prove its flexibility. The results of all the experiments are reported and discussed.

In the end, for the last case study, an embedded end-to-end application of the system into the multi-sensor board *STWIN*, manufactured by *STMicroelectronics*, is also discussed, with proper conversion of the model and validation of the autoencoder directly inside the board.

## 1.1. Company Profile

*221e* is an innovative startup established in 2012 and based in Italy with business units located within Padova, Treviso, and Bergamo.

The firm leverages the convergent advances in the fields of IT, microelectronics, sensors, and control algorithms for the development of miniaturized wireless embedded systems.

*221e* operates along a two-layer business model offering:

1. OEM (Original Equipment Manufacturer) services to third-party clients that need a technology partner for their product development; this translates into R&D contracts, usually followed by commercial agreements for the supply of the engineered systems or the licensing of the technology.
2. Finished products, currently in the form of general-purpose multi-sensors hardware platforms, to direct customers or distributors.

The market targeted is Wearable Devices and those industries linked to the so-called Internet of Things.

Applications in these markets are infinite, and this is what the name *221e* stands for.

*221e* is still a small business yet growing fast, building on its identity of a company that is innovation-driven and entrepreneurship-fueled, surfing the innovation wave of IoT and wearable devices.

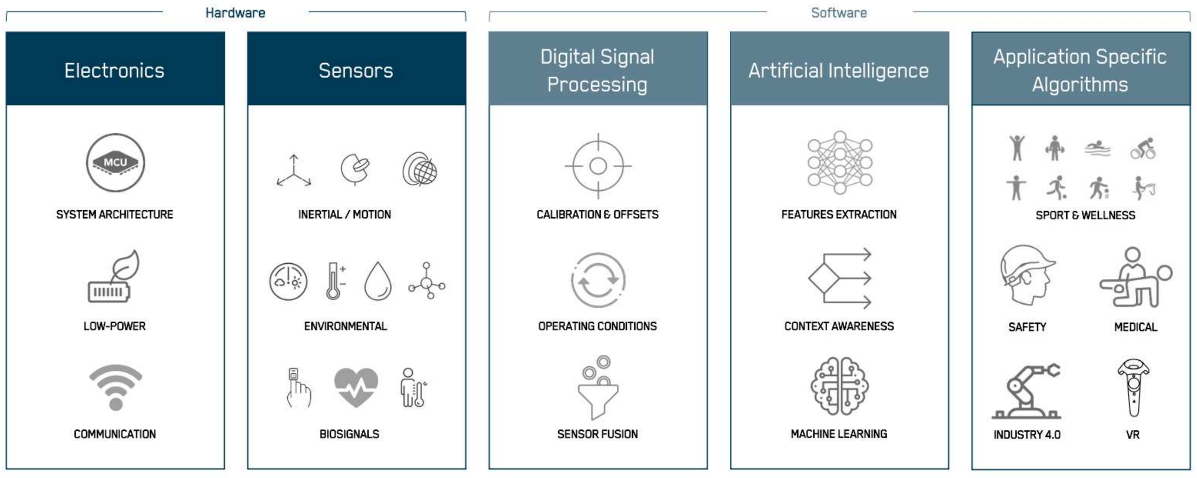


**Figure 1**: Technology ecosystem of the company (221e S.r.l)

## 1.2. Goals

The main goal of this thesis is to investigate and experiment with the potentiality of audio signals in the field of Condition Monitoring, in order to open the future possibility for *221e* to add audio-based solutions to its portfolio.

The aim of the company is to explore machine learning solutions for anomalous sound detection problems and to prove such solutions can be implemented with low resources in an IoT environment.

With the addition of Condition Monitoring dedicated microphones to *221e* own multi-sensor boards, the offer of the company technology ecosystem could be expanded. The coordination with the already existing sensors can also provide future sensor-fusion solutions for Condition Monitoring applications.

# 2. Background and Techniques

This chapter covers all the theoretical aspects necessary to understand the proposed study, as well as discussions on solutions for Sound-based Condition Monitoring problems.

## 2.1. From Machine Learning to Deep Learning

Machine learning is the use of algorithms and statistical models to effectively perform a task, without having to explicitly program the instructions for how to perform that task. Instead, the algorithms learn to perform the desired function from provided data, extracting statistical information from them in a process known as training.

Supervised learning uses a training dataset where each sample is labeled with the correct output. These labels are normally provided by humans inspecting the data. Typically, this is a time-intensive and costly process. In unsupervised learning, on the other hand, models are trained without access to labeled data. This is often used for cluster analysis (i.e., automatic discovery of sample groups) or anomaly detection, by learning the statistical distribution of data points [2].

Deep learning is a subfield of machine learning leveraging on models inspired by the structure and function of the brain called artificial Neural Networks (NNs).

Despite the big success of deep learning, it also has some disadvantages: like the need for a large amount of data and the difficulties in interpreting the reasons behind some outputs, also known in AI as the explainability problem [6].

In the following the concept of Neural Networks is expanded by presenting the two most relevant architectures used to solve many different problems, including Condition Monitoring scenarios.

### 2.1.1. Fully Connected Neural Networks

A basic and illustrative type of neural network is the Fully Connected Neural Network or Multi-Layer Perceptron (MLP). It consists of an input layer, one or more hidden layers, and an output layer.

Each layer consists of a number of neurons. Each neuron of one layer is connected to all the neurons in the preceding layer. This type of layer is therefore known as a fully-connected, or densely-connected layer. The input to the network is a 1-dimensional vector. If the data is multi-dimensional (like an image), it must be flattened to a 1-D vector.

Each neuron computes its output as a weighted sum of the inputs, offset by a bias and followed by an activation function *f*. The most common activation functions for hidden and input layers are ReLU, Sigmoid, and Tanh. Instead, the activation function of the output layer may change according to the specific output that we want to obtain [3].

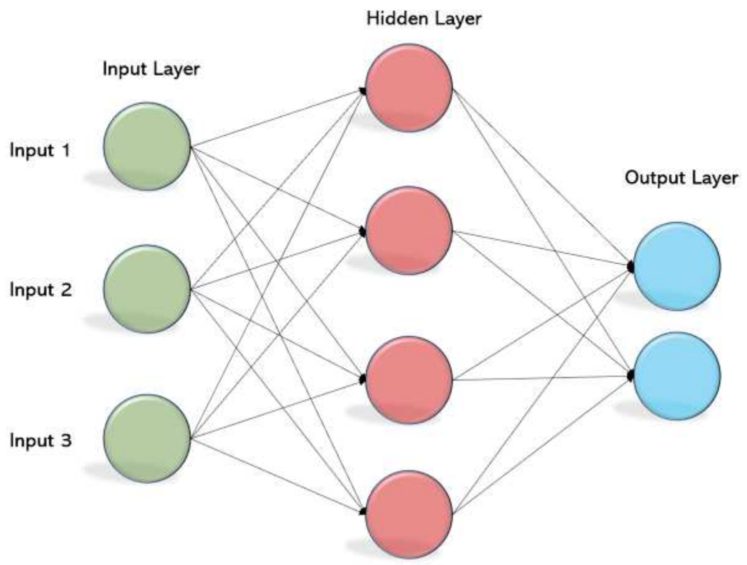


**Figure 2**: Example of a MLP with single hidden layer (becominghuman.ai)

Making predictions with a neural network is done by applying the data as inputs to the first layer, then computing all the following layers until the final outputs. This is often called forward propagation.

Considering a simple MLP with three layers, the computations taking place at every neuron in the output and hidden layer are as follows [11],

$$o(x) = G(W_2 h(x) + b_2)$$

$$h(x) = s(W_1 x + b_1)$$

with bias vectors $b_1$, $b_2$; weight matrices $W_1$, $W_2$ and non-linear activation functions $G$ and $s$.

The set of parameters to learn is:

$$\theta = \{W_1, b_1, W_2, b_2\}.$$

By the Universal Approximation theorem, we know that a neural network with at least one hidden layer like the previously described one, with appropriate non-linear activation function (e.g. ReLU, Sigmoid, Tanh), can approximate any linear function with arbitrarily low error [26].

Adding multiple hidden layers to the network will allow learning multiple representations of the inputs, and it may reduce the number of hidden units required to approximate a particular function. However, there is no standard way of choosing the best architecture of a neural network and it usually requires some parameter tuning, in order to find a model that works “well enough” for a particular problem.

### 2.1.2. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a particular type of neural network that are well suited for 2D input data. That makes them a useful tool in Computer Vision but also for audio processing.

CNNs main features are:

- **Weights sharing**: many weights are shared in the network, this makes the neural network more sparse and lighter
- **Convolution**: puts the input images through a set of convolutional filters (or kernels), each of which activates certain features from the images. The resulting matrix of the convolution operation $*$ between an input image $f$ and a kernel $h$ is defined as:

$$G[m,n] = (f * h)[m,n] = \sum_j \sum_k h[j,k] f[m-j, n-k]$$

- **Pooling**: simplifies the output by performing nonlinear down sampling, further reducing the number of parameters that the network needs to learn

Convolutional and pooling layers are usually alternated many times to reduce dimensionality and learn different features at every layer.

At the final layers of the CNN the representation can be flattened to apply dense layers (depending on the application).

CNNs are translation invariant: meaning that a local pattern of features in an input image has the same meaning to the network regardless of its position in space according to the two axes. Scale and rotation invariance can also be achieved using apposite data augmentation techniques [3].

### 2.1.3. Training Neural Networks

Neural Networks are trained through numerical optimization of an objective function (i.e., also called loss function). The general optimization method that is used is Gradient Descent (GD) [3], or some similar variants, i.e. Adam [15].

A popular and memory-efficient GD variant is mini-batch Gradient Descent. The training data is processed in multiple fixed-size batches of data, and the loss function and model parameters updates are computed per batch. That means that not all the training data has to be kept in memory at the same time, which allows training on very large datasets. The batch size is a hyperparameter and must be set high enough for the batch loss to be a reasonable estimate of the loss on the full training set, but small enough for the batch to fit into memory. A smaller batch size allows the training process to converge quicker at the cost of more noise on the loss plot, instead, a larger batch size provides slower training but with less noise on the loss plot.

Backpropagation is the way to compute the gradient used to update the weights after every batch, it works by propagating the error in the last layers “back” towards the input, using the partial derivative with respect to the inputs from the layer before it. This makes it possible to compute the gradients for each of the weights (and biases) in the network. Once the gradients are known, the weights are updated by taking a step in the negative direction of the gradient. The step is kept small by multiplying with the learning rate, a hyperparameter usually in the range of $10^{-7}$ to $10^{-2}$. Too small learning rates can lead to the model getting stuck in bad minima, while too large learning rates can cause training to not converge.

The mini-batch Gradient Descent optimization with backpropagation can be summarized in the following procedure.

1. Sample a mini-batch of data
2. Forward propagate the data to compute output and calculate the loss
3. Backpropagate the errors to compute error gradients in the entire network
4. Update each weight by moving a small amount against the gradient
5. Go to 1) until all batches of all epochs are completed

Gradient Descent is not guaranteed to find a globally optimal minimum, but with suitable choices of hyperparameters can normally find local minima that are good-enough [3].

### 2.1.4. Condition Monitoring: Supervised vs Unsupervised Techniques

A specific and relevant application of machine learning and deep learning is Condition Monitoring, which represents the focus of interest in this work.

Condition Monitoring, as discussed in Section 1., is the process of monitoring a set of parameters of condition in machinery (i.e. vibration, temperature, sound), in order to discriminate anomalies from the normal behavior of the machine.

Anomalies, often referred to as outliers, abnormalities, rare events, or deviants, are data points or patterns in data that do not conform to a notion of normal behavior. Anomaly detection, then, is the task of finding those patterns in data that do not adhere to expected norms, given previous observations. The capability to recognize or detect anomalous behavior can provide highly useful insights across industries. Flagging unusual cases or enacting a planned response when they occur can save businesses time, costs, and customers.

Hence, Condition Monitoring has found diverse applications in a variety of domains, including IT analytics, network intrusion analytics, medical diagnostics, financial fraud protection, manufacturing quality control, marketing, social media analytics, and more.

Condition Monitoring approaches based on machine learning can be categorized in terms of the type of data needed to train the model. In most use cases, it is expected that anomalous samples represent a very small percentage of the entire dataset. Thus, even when labeled data is available, normal data samples are more readily available than abnormal samples. This assumption is critical for most applications today.

When learning with supervision, machines learn a function that maps input features to outputs based on example input-output pairs. In the framework of Condition Monitoring the goal of the algorithms is to include application-specific knowledge in the anomaly detection process.

With sufficient normal and anomalous examples, the anomaly detection task can be reframed as a classification task where the machines can learn to accurately predict whether a given example is an anomaly or not. That said, for many anomaly detection use cases, as it was previously mentioned, the proportion of normal versus anomalous examples is highly unbalanced; while there may be multiple anomalous classes, each of them could be quite underrepresented.

With unsupervised learning, machines do not possess example input-output pairs that allow the system to learn a function that maps the input features to outputs. Instead, they learn by finding structure within the input features. Because, as previously mentioned, labeled anomalous data is relatively rare, unsupervised approaches are more popular than supervised ones in the anomaly detection field [2]. That said, the nature of the anomalies one hopes to detect is often highly specific. Thus, many of the anomalies found in a completely unsupervised manner could correspond to noise, and may not be of interest to the task at hand.

A sort of middle ground is represented by semi-supervised approaches, employing a set of methods that take advantage of large amounts of unlabeled data as well as small amounts of labeled data. Many real-world anomaly detection use cases are well suited to semi-supervised learning: there are usually a large number of normal examples available from which to learn, but relatively few examples of abnormal classes of interest.

Following the assumption that most data points within an unlabeled dataset are normal, one can train a robust model on an unlabeled dataset and adjust its performance (tuning only a few parameters of the model) by using a small amount of labeled data to set a discriminative threshold.

### 2.1.5. Evaluation of Anomaly Detection Systems

As previously stated, in anomaly detection applications it is expected that the distribution between the normal and abnormal classes may be highly skewed. This is commonly referred to as the class imbalance problem [5].

A model that learns from such skewed data may not be robust; it may be accurate when classifying examples within the normal class, but perform poorly when classifying anomalous examples.

For example, suppose that in a certain scenario anomalies happen in 5% of cases: a dumb classifier that always classifies samples as normal will achieve an accuracy of 0.95, but it would not be able to correctly classify any anomaly in practice.

By the example above it becomes trivial that the traditional accuracy metric (total number of correct classifications divided by total classifications) is insufficient in evaluating the skill of an anomaly detection model.

Sensitivity and specificity were introduced as metrics to address this problem:

- Sensitivity (True Positive Rate) refers to the probability of a positive test, conditioned on truly being positive

  $$TPR = \frac{TP}{TP+FN}$$

- Specificity (True Negative Rate) refers to the probability of a negative test, conditioned on truly being negative

  $$TNR = \frac{TN}{TN+FP}$$

These two metrics depend on the choice of the classification threshold and the increasing of one of them often leads to a decrease in the other. So, a tradeoff between the two is essential and the optimal choice depends on the nature of the problem.

Going back to the example above of the dumb classifier: by always classifying samples as normal it would achieve TNR = 1 but TPR = 0, making it clear that the classifier did not really learn any knowledge from the data despite the high accuracy achieved.

Other important metrics are the ROC curve and the AUC value [12]. The Receiver Operator Characteristic (ROC) curve is an evaluation metric for binary classification problems. It is a probability curve that plots the True Positive Rate against the False Positive Rate (FPR = 1 - TNR) and essentially separates the "signal" from the "noise". The Area Under the ROC Curve (AUC) is the measure of the ability of a classifier to distinguish between classes and it is used as a summary of the ROC curve. AUC is largely the most used evaluation metric in anomaly detection.

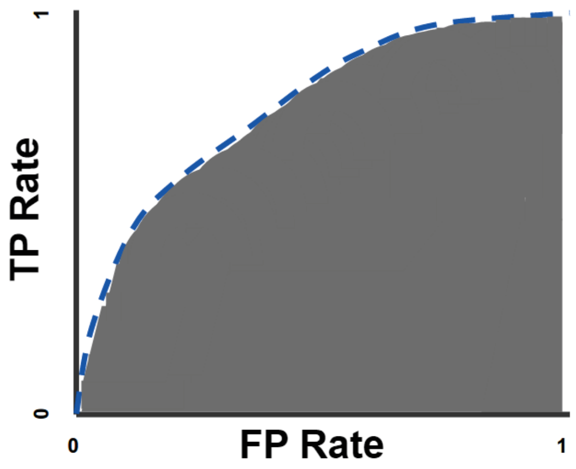


**Figure 3**: ROC curve representation [12]: AUC is represented by the gray area

## 2.2. Internet of Things and Edge AI

IoT (Internet of Things) is a growing field that refers to the interconnection using the internet of objects, usually low cost, with sensors and/or actuators embedded in their architecture.
An IoT network can be seen as three subsequent abstraction layers. The edge layer is the lowest level, where all the edge nodes ("things") are. The aggregation layer is the middle-level layer, it contains IoT gateways and micro datacenters, in this level the data are locally aggregated from many different edge sources. The top level is the cloud, where the data is further aggregated and where most of the analytics, storage, and computing take place.
IoT structure can be well suited for Condition Monitoring tasks: the edge nodes can be battery-powered sensors acquiring data from one or multiple machines and they can send raw data to higher levels where all the heavier computation takes place to classify the data between normal or abnormal. This has the advantage of taking the computation away from low-performance devices and allows aggregation between multiple sources, using complex and resourceful AI solutions at the cloud level. However, this solution usually provides high latency making the implementation of real-time applications difficult, furthermore, it can raise privacy concerns since all the sensitive data still needs to be sent to the cloud for processing.
An alternative approach that is gaining more and more popularity is to bring some AI to the edge level. With the continuous evolution of microcontrollers, the embedded CPUs and MLUs (i.e. Machine Learning Units) are now capable of running simple analytics directly in the edge nodes. This approach has the advantage of lowering the latency allowing real-time applications at the edge level and providing security of data since the raw data is not shared in the cloud.
For sound-based Condition Monitoring in particular the approach of edge AI is the winning one, since sound data is usually particularly large (especially when dealing with ultrasounds) and often sensitive from a privacy point of view. The cloud can still be used to perform data aggregation between multiple sensors to perform non-real-time analysis, and it can use already pre-processed data instead of raw ones.
The previously exposed argument motivates the need to implement analytics at the edge level, however, it requires the capability to adapt AI solutions to low-performance and low-power environments, which can be really challenging. However, in this thesis solutions like *X-CUBE-AI* [30] will be presented, to easily adapt AI solutions and constantly check all memory and computing resources available during development using the compatible boards.

## 2.3. Digital Sound Representation and Processing

Physically, sound is a variation in pressure over time. To process the sound with machine learning, it must be converted to a digital format. The acoustic data is first converted to analog electric signals by a microphone and then digitized using an Analog-to-Digital Converter (ADC) [22].

In the digitization process, the signal is quantized in time at a certain sampling rate (ODR), and the amplitude is quantized at a certain bit-depth. These two parameters affect the quality of the digital audio representation. The frequencies that can be perceived by the human ear go from 20 Hz to 20 kHz, by sampling at higher rates we can also capture ultrasounds that are not audible. The Nyquist theorem says that in order to obtain a lossless digitalization of the signal the ODR should be at least twice the highest frequency that appears in the original signal [22].

The quantization bit-depth instead is the number of bits used to represent each sample, so it represents the resolution of the digital sound.

In the previously described representation, sound is a one-dimensional sequence of numbers, sometimes referred to as a waveform. A visual example of a waveform is provided in figure 4.

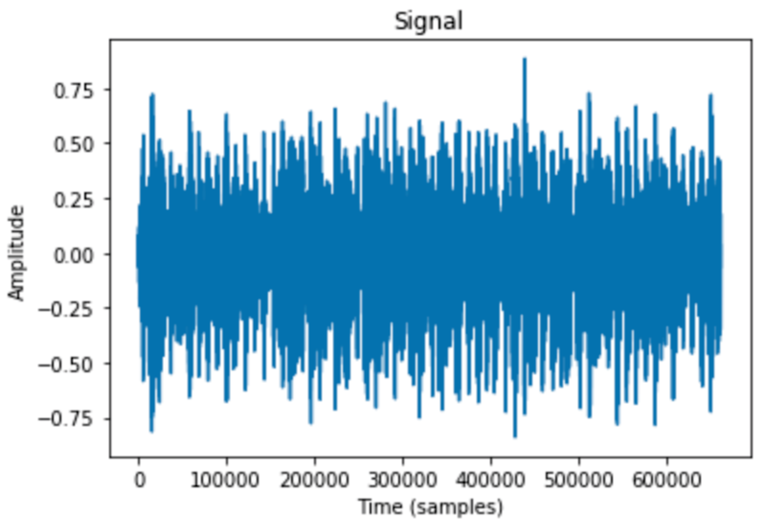


**Figure 4**: Example of a waveform

Recordings can have multiple channels of audio, but for machine learning single channel data (mono-aural) is still common.

### 2.3.1. Anomaly Detection: Sound-based Condition Monitoring

Anomaly detection can be performed using different kinds of Condition Monitoring sensors:

- Vibration sensors can be used to detect an encrease in the vibration level, which can lead to machine failures
- Thermal sensors can be used to constantly monitor the temperature change of the machine, allowing the detection of anomalies connected to overheating
- Voltage sensors can monitor the machine electric emission, in order to detect peaks in the voltage, which can be an index of machine degradation

All these sensors are widely used in industrial applications. However, they present some limitations that may restrict the capability of the Condition Monitoring system to provide a more global and comprehensive understanding of the health of the machine. Some small problems such as misalignment, looseness, loss of lubrication, and imbalance often go unnoticed until they develop into a more serious problem like overheating. An elevated temperature often indicates potential and serious problems. But it might take a long time for the temperature to grow over a certain warning threshold.

The advantage of using the previously defined sensors in a Condition Monitoring setting is their simplicity: it is sufficient to define a threshold outside of which the value is to be considered anomalous, so the problem reduces to a linear classification over one parameter. Sound, however, represents a more sophisticated and complex metric for anomaly detection tasks and it cannot be handled with a linear model, but with the rapid growth and advancement of AI and machine learning, sound-based Condition Monitoring has made a breakthrough [7]. This technology uses the sound emitted by the machine to detect anomalies. The anomalies can be often detected at an early stage and before other classic monitoring strategies (i.e. vibration, temperature) would detect them.

Each machine has its own "sound fingerprint" and an AI system can learn and adapt to every individual situation. Even if the frequency is outside the range detected by the human ear (e.g., ultrasound frequencies).

The sound is a leading indicator for Condition Monitoring, as machines often emit different sounds when there are experiencing problems, even before they escalate into something serious. The differences in the emitted sound can be easily detected by sound sensors and the system is able to report them as anomalies. Therefore, sound-based Condition Monitoring is a good choice to accurately asses the machine condition through audio data, and at a very early stage.

Another advantage of sound-based Condition Monitoring is that it is a non-intrusive monitoring solution: no physical contact is required between the machine and the sensor. The sound sensor will only need to be placed in a spot close to the monitored machine, and not phisically attached to it. This can provide high versatility and flexibility to the monitoring solution [7].

However, there are also some disadvantages of sound-based Condition Monitoring systems. One of them is the sensitivity to environmental noise; this can be mitigated by a quality training set that will provide the model with the knowledge it needs to distinguish useful information from noise. Other disadvantages are the classical ones related to all machine learning systems, like the high demand for training data or explainability.

The high demand fot training data, in this case, is a big problem only for supervised systems, since data regarding anomalies, as previously discussed, is normally absent or very limited.

On the other hand, the explainability problem remains not easily solvable. In AI explainability is defined as the problem of reconstructing the exact logical reasons behind a specific output of an AI algorithm [6]. While for linear fully-threshold based monitoring systems the reasons for an anomalous classification of an input are usually clear (e.g. temperature exceeds the maximum threshold), for sound-based systems the reasons behind an anomalous classification might usually be not clear or not directly understandable, even if usually accurate, like it happens in most machine learning based applications.

In Table 1 the previously discussed pros and cons of sound-based solutions are summarized.

| **Pros** | **Cons** |
|---|---|
| Non-intrusive | Sensitivity to noise |
| Early detection of anomalies | High demand of data |
| Detection of wide range of anomalies | Explainability of results |

**Table 1**: Pros and cons of sound-based solutions for Condition Monitoring

Regarding the possible machine learning models that can be used for Sound-based Condition Monitoring, we want to immediately discard all the fully supervised techniques for the reasons discussed in Section 2.1.4.: anomalous data cannot be assumed to be easily accessible as normal data.

The simpler unsupervised model that comes to mind is k-Nearest Neighbors used to detect outliers [9], but it is very inefficient for high dimensional data. Another unsupervised solution is Isolation Forest [18] which is a model specifically created for outlier detection; this solution is very efficient but, being fully unsupervised, it cannot use knowledge from anomalous data in case it is available, making the parameters difficult to tune, with a risk of high noise sensitivity. Autoencoder-based models on the other hand represent a more flexible solution, they can be used in an unsupervised or semi-supervised way, using also some knowledge from anomalous data when available, this makes the models more specific and less vulnerable to noise [14], making them the predominant solution regarding sound-based Condition Monitoring in recent literature [21].

## 2.4. Autoencoders

Autoencoders represent a core topic for this work since they are the chosen model to deal with the Condition Monitoring tasks ahead.

Autoencoders are a particular type of unsupervised neural network that aims to reconstruct its input in the output performing dimensionality reduction of linear and non-linear data on hidden layers, hence they are more powerful than Principal Component Analysis (PCA) which can reduce dimensionality with restriction to a linear map [14]. Their architecture can be divided into encoder and decoder parts.

The encoder architecture takes the input and performs dimensionality reduction layer by layer. The last layer is also called the encoding layer, or latent representation which represents the input in a compressed dimensionality. The decoder architecture is specular to the encoder and from the encoding layer it augments the dimensionality of the representation in an analog way, and the same dimension of the input is obtained in the output layer.

The latent representation layer performs the role of bottleneck and it can be seen as a sort of regularization within the architecture of the network, the loss of information that this layer causes is a way to avoid overfitting and prevents the autoencoder from learning the identity function, which is something that would not be of any use for machine learning tasks [3].

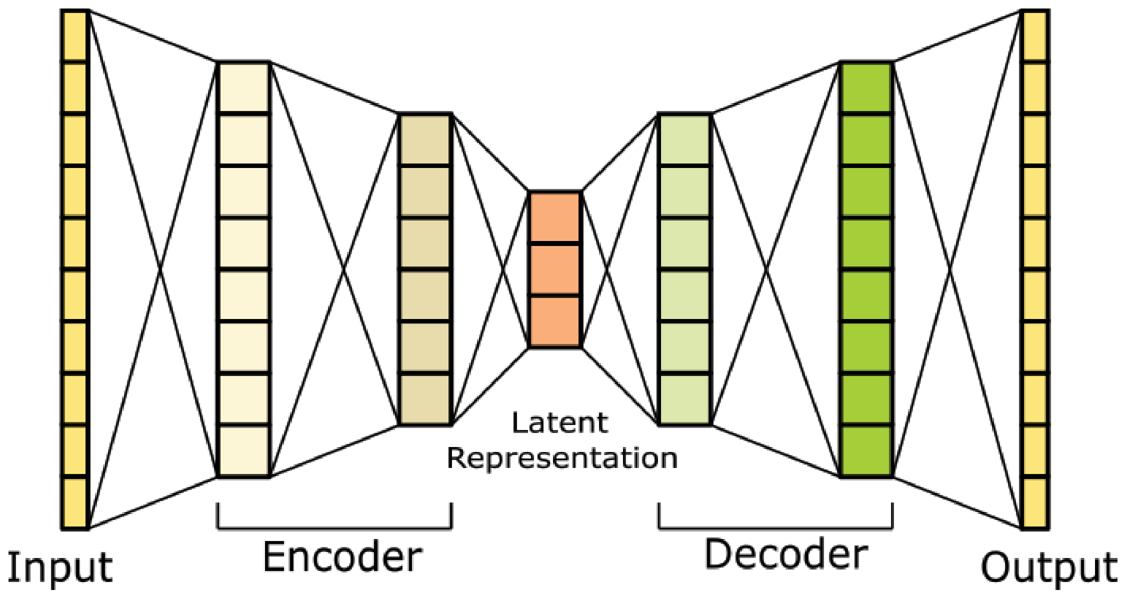


**Figure 5**: Representation of the architecture of a fully connected autoencoder (html.it)

The number of hidden layers is arbitrary, even if they need to be in an odd number to maintain its symmetrical structure. An example can be seen in figure 5. The squares represent the units of every layer that are fully connected with the units of the following layer. The sizes of the layers in the encoder and decoder can be set freely according to the needs of the problem, the latent representation layer, however, should always be the smaller layer in size to serve as a bottleneck for the previously stated reasons. The working principle is the same as the standard fully connected neural networks, and we take advantage of the ability of the network to create different representations of the input for every hidden layer in order to change its dimensionality bringing it to a minimum (in the latent representation layer) and then back to the original size in the output layer.

An autoencoder can also be convolutional. In this case, the encoder is formed from convolutional layers in order to reduce the size of the representation. The latent representation is a small size fully connected layer obtained after a flattening of the dimensions. The decoder is made by reshaping the data to its original dimensions and adding de-convolutional layers (inverse operation of convolution) mirroring the structure of the encoder to go back to the original size in the output layer. An example can be seen in figure 6.

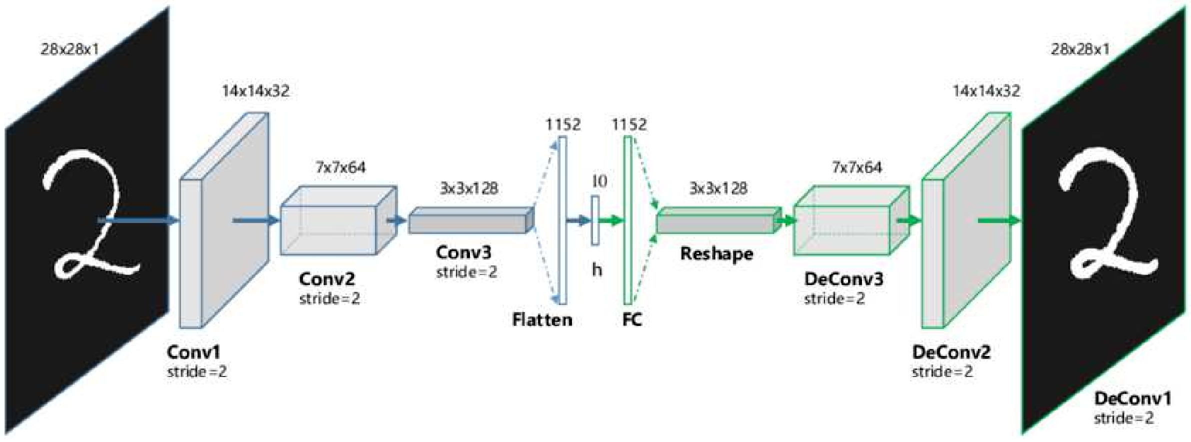


**Figure 6**: Representation of the architecture of a convolutional autoencoder (researchgate.net)

The reconstruction error is the error that represents the loss of information of the output according to the original input (Mean Square Error or Binary Cross-Entropy can be used). Autoencoders are trained using GD, where the loss to be minimized is the reconstruction error between the output and the input.

Autoencoders are often used to perform data compression, by taking the representation of the encoding layer, but also to build denoising systems, or for transfer learning applications.

## 2.5. Spectrogram

Spectrograms are another important concept to understand the experiments in this thesis since they are the chosen way to represent the audio data that is given in input to an autoencoder.

Sounds of interest often have characteristic patterns not just in time (temporal signature) but also in frequency content (spectral signature). Therefore it is common to analyze audio in a time-frequency representation. A spectrogram is the graphic representation of sound intensity as function of its time and frequency.

A common way to compute a spectrogram from an audio waveform is by using the Short-Time Fourier Transform (STFT). The STFT operates by splitting the audio up into short consecutive chunks and computing the Fast Fourier Transform (FFT) to estimate the frequency content for each chunk. To reduce artifacts at the boundary of chunks, they are partially overlapped (typically by 75%) and a window function, such as the Hann window [19], is applied before computing the FFT. With the appropriate choice of window function and overlap, the STFT is invertible.

In figure 7 an example of a spectrogram is shown, usually spectrograms are displayed using a logarithmic scale since most of the audible information in the spectrum is in lower frequencies. On the Y-axis we have the frequencies from zero to the ODR of the acquisition, the X-axis represents time and it goes from zero to the duration of the audio segment. The colors inside the spectrogram represent the intensity in dB of each frequency for every moment in time.

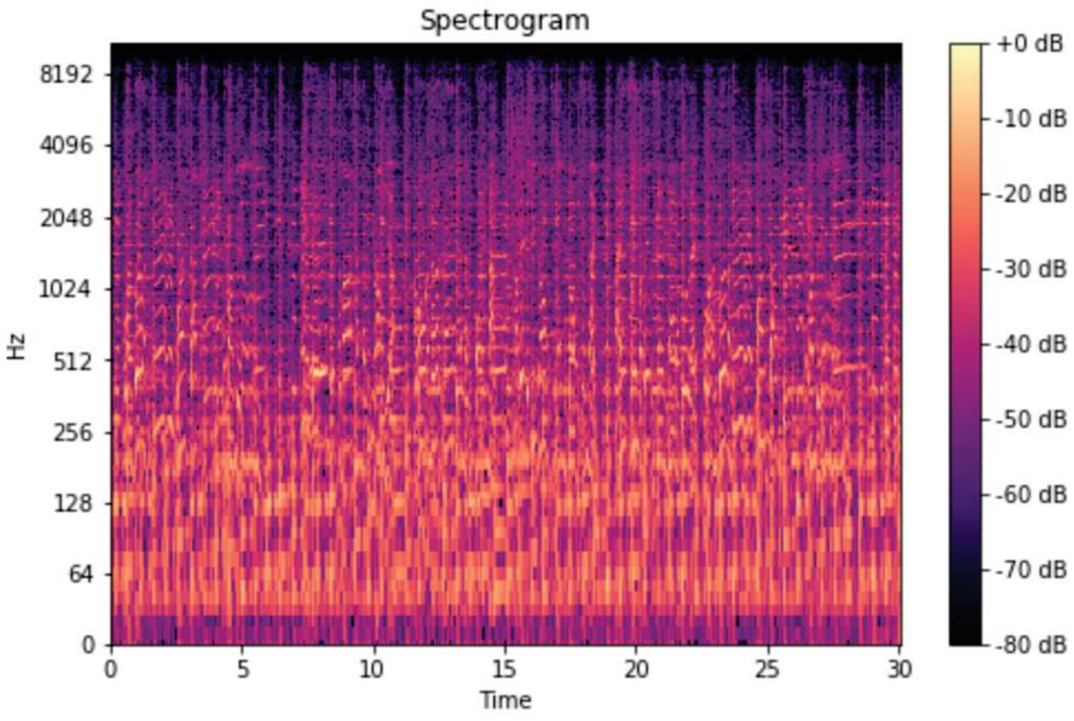


**Figure 7**: Example of a spectrogram

The length of every FFT window is determined by the *n_fft* parameter while the *hop_length* parameter decides the overlapping percentage between consecutive chunks [4].
The final width of the spectrogram is then computed as follows:

$$width = (N \times sample\_rate) \div hop\_length$$

where N is the duration of the audio in seconds.
The height is directly connected to the *FFT-Length* parameter and it equals half of its value.
Spectrograms can be in general treated as images and in machine learning settings are in general considered as such. However, it is important to keep in mind that spectrograms encode precise and different information along the two axes, differently from standard images.

### 2.5.1. Mel spectrogram

The more complex the input to a machine learning system is, the more processing power is needed both for training and inference. Therefore one would like to reduce the dimensions of inputs as much as possible. A regular spectrogram often has considerable correlation (redundant information) between adjacent frequency bins and is often reduced to 30-128 frequency bands using a filter-bank. Several different filter-bank alternatives have been investigated for audio classification tasks, such as 1/3 octave bands, the Bark scale, Gammatone, and the Mel scale. All these have filter spacing that increases with frequency, mimicking the human auditory system.

The spectrogram that results from applying a Mel-scale filter-bank is often called a Mel spectrogram. In figure 8 the Mel-scale is represented in the Y-axis.

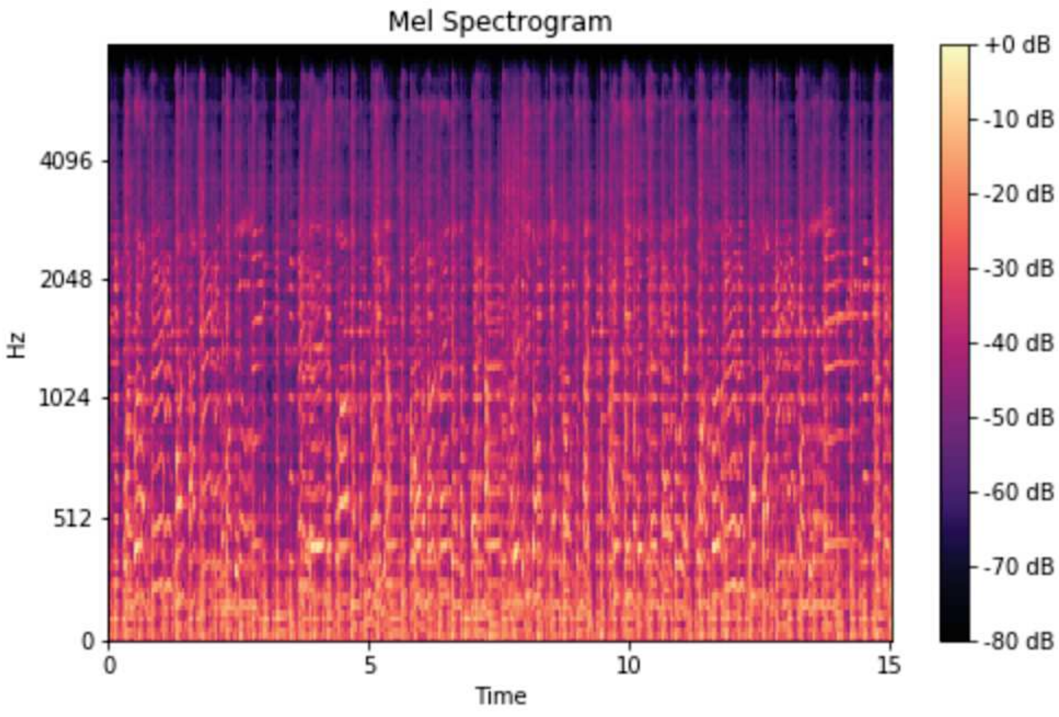


**Figure 8**: Example of a Mel spectrogram

The width of a Mel spectrogram is computed in the same way as for regular spectrograms, while the height is set by the *n_mels* parameter which also represents the frequency resolution of the output [4].

Mel spectrogram is the most used sound representation for machine learning applications, due to its compact structure and similarity of the used scale to the perception of the human ear.

## 2.6. Fully Connected vs Convolutional Autoencoders for Condition Monitoring

Fully connected neural networks are the most straightforward way to design an autoencoder, however, they accept only one-dimensional data as input. The input in sound-based Condition Monitoring is usually bidimensional since spectrogram representation is widely used for sound machine learning applications. In order to use spectrograms as inputs a flattening of the input is needed, to obtain one-dimensional data.

Autoencoders can also have a convolutional architecture to deal directly with 2D inputs. Some concerns, however, have arisen in the scientific community about the usage of convolutional neural networks with spectrograms [25]. The translation invariance property of convolutional neural networks gives them great value in the computer vision field, i.e. in object detection, the features of objects are independent of their location in the image. However, in a spectrogram, this is not ideal. A particular local pattern in a spectrogram represents very different sounds if translated on the axes: a translation on the X-axis would mean changing the time of that sound, and a translation on the Y-axis would mean a change in frequencies, drastically commuting the meaning of that local pattern. For those reasons, the spectral properties of sound are strictly non-local.

Fully connected autoencoders on the other hand have the capability of learning the spatial global features of spectrograms in a more effective way, even if they usually necessitate far more parameters with respect to the weight-sharing approach in convolutional neural networks.

Another advantage of fully connected autoencoders with respect to the convolutional counterpart is their adaptability to different case studies, meaning the number of architectural changes that are needed to adapt the system to new data sizes. In fully connected autoencoders it is possible to change in the code only the sizes of the flattened input/output without any further modification. Instead for convolutional approaches, a change in the size of the input would require a change in possibly every layer, since the output size of the 2D image depends directly on all the previous layers.

# 3. Tools

In this chapter the hardware and software tools used to carry on the experiments are presented, with particular emphasis on the acquisition board features and the development environment.

## 3.1. Multi-sensor Acquisition Board

The choice of the acquisition board was crucial for the sake of the experiments. The flagship multi-sensor acquisition board of *221e* is the *MUSE* (see Appendix A) multi-sensor system, specifically designed for inertial and environmental data acquisitions. The company is considering expanding its functionalities in the future in audio-based Condition Monitoring settings.

The choice of the board was made in favor of the *STWIN* by STMicroelectronics, mostly because of its embedded ultrasound analog microphone, which is particularly fit for Condition Monitoring settings.

### 3.1.1. STWIN - SensorTileWireless Industrial Node

The *STWIN* (*STEVAL-STWINKT1*), commercialized by STMicroelectronics, is a development kit and reference design to simplify prototyping and testing of advanced industrial IoT applications such as Condition Monitoring and predictive maintenance. The kit supports BLE (Bluetooth Low Energy) wireless connectivity through an on-board module.

Hardware main key features:

- Multi-sensing wireless platform implementing vibration monitoring and ultrasound detection
- Built around *STWIN* core system board with processing, sensing, connectivity and expansion capabilities
- Micro SD Card slot for standalone data logging applications
- Wireless BLE 4.2 (on-board), and wired RS485 and USB OTG connectivity
- Wide range of industrial IoT sensors: vibration sensor, 3D accelerometer and 3D Gyro, inertial measurement unit with machine learning core, 3-axis magnetometer, digital absolute pressure sensor, humidity and temperature sensor
- Industrial grade digital *MEMS* microphone (*IMP34DT05*) and wideband analog *MEMS* microphone (*MP23ABS1*)

- Other kit components: Li-Po battery 480 mAh, *STLINK-V3MINI* debugger with programming cable and Plastic box

U2: HTS221 relative humidity and temperature sensor
U3: LPS22HH digital absolute pressure sensor
U6: STTS751 low-voltage digital local temperature sensor
U8: TS922 rail-to-rail, high output current, dual operational amplifier
U9: ISM330DHCX 3D acc. + 3D gyro iNEMO IMU with machine learning core
U11: IIS3DWB ultra-wide bandwidth (up to 6 kHz), low-noise, 3-axis digital vibration sensor
U12: IIS2DH ultra-low-power high performance MEMS motion sensor
U13: IIS2MDC ultra-low-power 3-axis magnetometer
M1: MP23ABS1 analog MEMS microphone
M2: IMP34DT05 industrial grade digital MEMS microphone

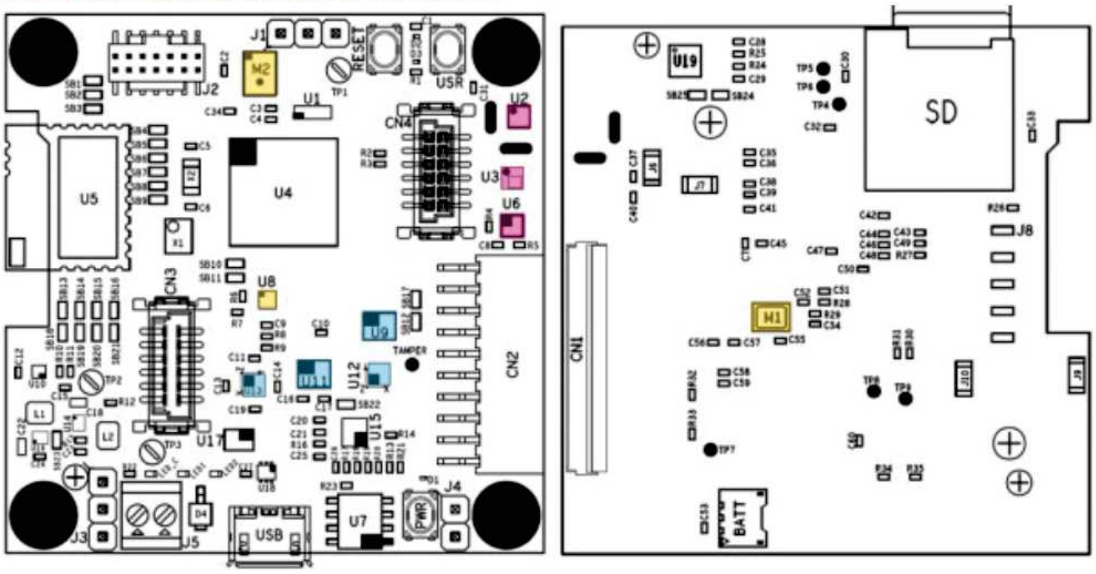


**Figure 9**: Architecture of STWIN (STMicroelectronics)

*STWIN* belongs to the *STM32* microcontrollers family, a wide set of multi-sensor boards manufactured by STMicroelectronics.

The microphone that was used for the data acquisitions in this work is the analog *MEMS MP23ABS1*, marked as “M1” in figure 9.

STMicroelectronics *MP23ABS1 MEMS* Audio Sensors are high-performance and low-power analog microphones built with a capacitive sensing element and an integrated circuit interface. The sensors have a sensitivity of -38dBV ±1dB, an acoustic overload point of 130dBSPL and 135dBSPL with a minimum 64dB signal-to-noise ratio. These *MP23ABS1* sensors are available in a package compliant with reflow soldering and are guaranteed to operate over an extended temperature range from -40°C to 85°C. Typical applications include smart speakers, mobile phones, and wearables. This particular microphone embedded in *STWIN* represents a good choice to acquire data for Condition Monitoring purposes.

In the development kit, another small hardware element is present, the *STLINK-V3MINI*, which is a standalone debugging and programming mini probe for *STM32* microcontrollers. This piece of hardware is essential to load premade or custom firmwares into the *STWIN* board; it was used to load the standard data acquisition firmware and will be used in the following sections to upload the on-device custom validation firmware.

### 3.1.2. Standard Data Acquisition Firmware

The *FP-SNS-DATALOG1* [27] function pack (used for all data acquisitions on *STWIN* in this work) implements High Speed Datalog application for *STEVAL-STWINKT1* and other *STM32* microcontrollers. It offers a global solution to save data from one or multiple sensors set with their individual configurations using a JSON standard format.
*FP-SNS-DATALOG1* is part of the *STM32Cube* softwares for *STM32* devices.
Software main features:

- High-rate (up to 6 Mbit/s) data acquisition software solution:
    - BLE app for real-time control and setup of the system
    - API to enable integration into any data analysis tool
    - Timestamping for synchronization of different sensor data
    - Provided Python frameworks for analysis of sensor data
    - Real-time control tools for Python and C++

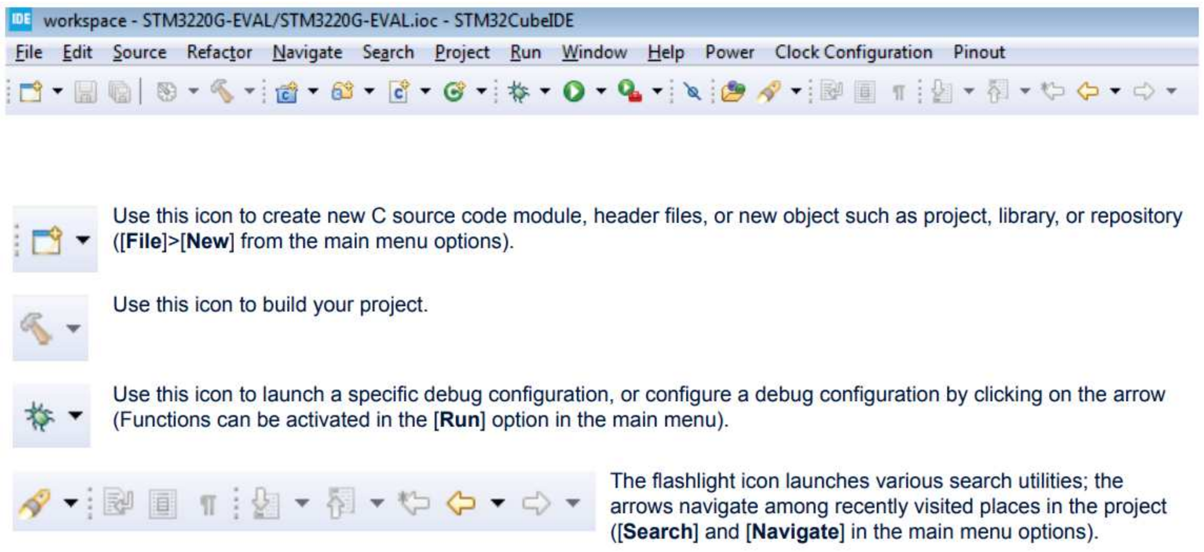
This software allows the acquisition of acoustic data using the embedded *MEMS* analog microphone at the sampling rate (ODR) of 192 kHz, an extremely high frequency that allows capturing a wide range of ultrasounds, using a bit-depth of 16.
Once the acquisition is completed a folder is generated with one (or more, depending on how many sensors were used) *.mat* file containing the acquisition data and a JSON text file containing all the information about the acquisition for each active sensor (i.e. ODR for an audio acquisition). Using the Python libraries associated with this firmware it is possible to convert the raw data to *.wav* using the original acquisition folder.

## 3.2. Development Environment

The main IDE used to develop and test the whole software pipeline is *JupyterLab* [16], which allows a full Python implementation.
Regarding the embedded porting and evaluation on *STWIN*, some *STM32Cube* tools provided by STMicroelectronics were used. *STM32Cube* is an ecosystem that provides software tools to allow and facilitate the development on *STM32* microcontrollers.

### 3.2.1. STM32CubeIDE

*STM32CubeIDE* [28] is an all-in-one multi-OS development tool, which is part of the *STM32Cube* software ecosystem. *STM32CubeIDE* is an advanced C/C++ development platform with peripheral configuration, code generation, code compilation, and debug features for *STM32* microcontrollers and microprocessors. It is based on the Eclipse/CDT framework [10]. It allows the integration of hundreds of existing plugins that complete the features of the Eclipse IDE.
*STM32CubeIDE* integrates *STM32* configuration and project creation functionalities from *STM32CubeMX* to offer an all-in-one tool experience and save installation and development time. After the selection of an empty *STM32*, or pre-configured microcontroller or microprocessor from the selection of a board or the selection of an example, the project is created and initialization code generated. The resulting initialization code is already a fully working firmware ready to be uploaded to the device or to be further customized. At any time during the development, the user can return to the initialization and configuration of the peripherals or middleware and regenerate the initialization code with no impact on the user custom code.
*STM32CubeIDE* includes build and stack analyzers that provide the user with useful information about project status and memory requirements, which are usually the biggest obstacle when working on microcontrollers, due to their limited hardware.
*STM32CubeIDE* also includes standard and advanced debugging features including views of CPU core registers, memories, and peripheral registers, as well as live variable watch, Serial Wire Viewer interface, or fault analyzer.
In this work, *STM32CubeIDE* will be used as the standard platform to generate the validation code for the *STWIN* board, a particular initialization code to test the proposed system on an embedded environment.


workspace - STM3220G-EVAL/STM3220G-EVAL.ioc - STM32CubeIDE
File Edit Source Refactor Navigate Search Project Run Window Help Power Clock Configuration Pinout

Use this icon to create new C source code module, header files, or new object such as project, library, or repository ([**File**]>[**New**] from the main menu options).

Use this icon to build your project.

Use this icon to launch a specific debug configuration, or configure a debug configuration by clicking on the arrow (Functions can be activated in the [**Run**] option in the main menu).

The flashlight icon launches various search utilities; the arrows navigate among recently visited places in the project ([**Search**] and [**Navigate**] in the main menu options).

**Figure 10**: STM32CubeIDE main functionalities (STMicroelectronics)

### 3.2.2. STM32CubeMX

*STM32CubeMX* [29] is a graphical tool that allows a very easy configuration of *STM32* microcontrollers and microprocessors, as well as the generation of the corresponding initialization C code for the Arm Cortex-M core or a partial Linux Device Tree for Arm Cortex-A core, through a step-by-step process (see Appendix B).

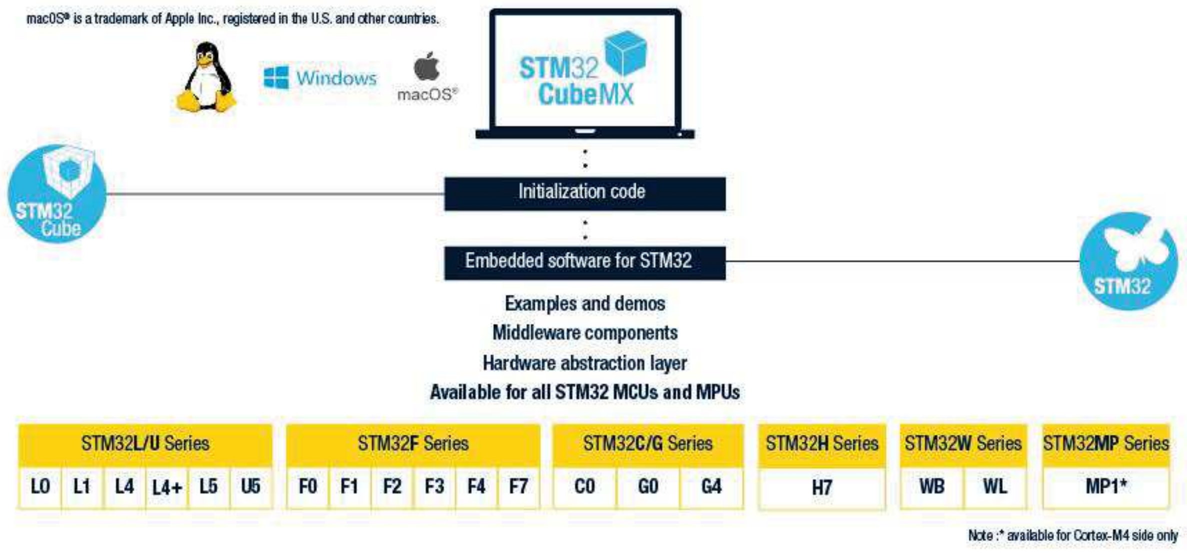


**Figure 11**: Usage diagram of STM32CubeMX (STMicroelectronics)

### 3.2.3. X-CUBE-AI

*X-CUBE-AI* [30] is a *STM32Cube* Expansion Package and it extends *STM32CubeMX* capabilities with automatic conversion of pre-trained AI models, including neural networks and classical machine learning models like isolation forest, support vector machine (SVM), and K-means, and it provides integration of generated optimized library into custom projects.
The *X-CUBE-AI* Expansion Package also offers several means to validate Artificial Intelligence algorithms both on desktop PC and *STM32*, as well as measure performance on *STM32* devices without user handmade ad hoc C code.
In figure 12 the general architecture of *X-CUBE-AI* Expansion Package is displayed. The displayed procedure will be used in later sections to optimize the autoencoder model for a *STWIN* embedded validation.

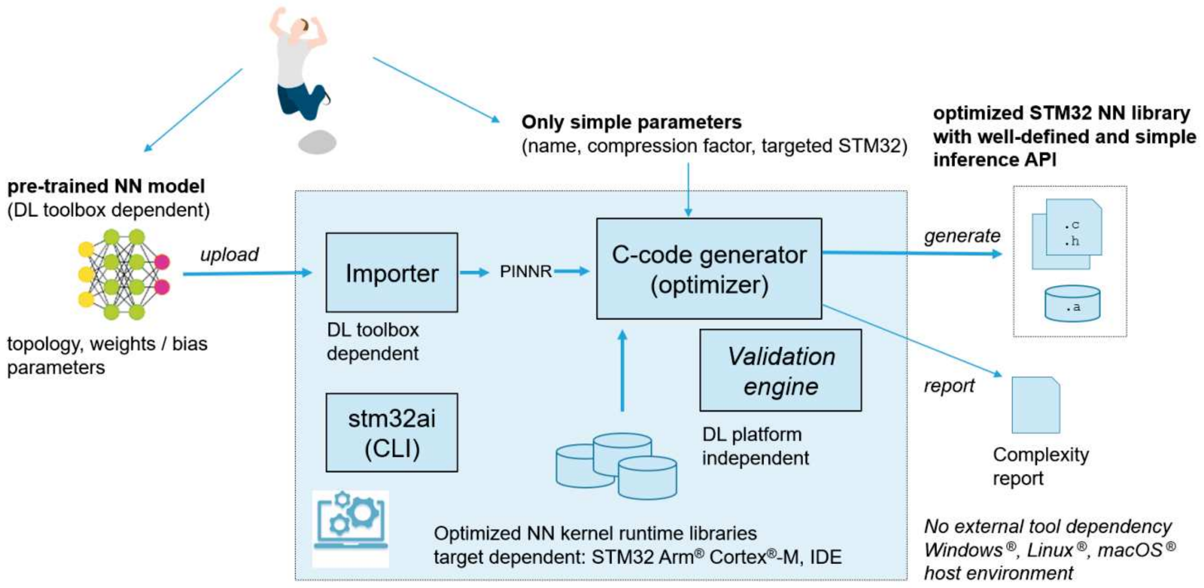

Only simple parameters
(name, compression factor, targeted STM32)
optimized STM32 NN library with well-defined and simple inference API
pre-trained NN model
(DL toolbox dependent)
upload
topology, weights / bias parameters
Importer
PINNR
C-code generator (optimizer)
DL toolbox dependent
Validation engine
DL platform independent
stm32ai (CLI)
generate
.c
.h
.a
report
Complexity report
Optimized NN kernel runtime libraries
target dependent: STM32 Arm® Cortex®-M, IDE
No external tool dependency
Windows®, Linux®, macOS® host environment

**Figure 12**: X-CUBE-AI general architecture [30]

This software platform provides a standard way to implement AI solutions directly in embedded devices without the need to send the raw data to be processed to other devices, providing latency and privacy advantages.
*X-CUBE-AI* can be divided into 3 main pillars:

- **Graph optimizer**: automatically improve performance through graph simplifications and optimizations that benefit the implementation to the target device.
- **Quantized model support**: the graphs can be quantized to compress the information from FP32 to Int8 with the minimum loss of accuracy, providing code validation on

target, displaying latency, accuracy and memory usage. Extreme compression can also be achieved with a mixed binary Int1 + Int8 quantization.

- **Memory optimizer**: optimize the memory allocation to get the best performance while respecting the constraints of the specific embedded design. The network is generated in a different binary file to enable a standalone model upgrade.

This solution provides an efficient and hardware-specific approach. In figure 13 a comparison of performances with TensorFlow Lite for Microcontrollers [13] is shown. It can be seen that the specificity of *X-CUBE-AI* to *STM32* microcontrollers can provide better performances in every aspect. *X-CUBE-AI* is compatible with all *STM32* devices.

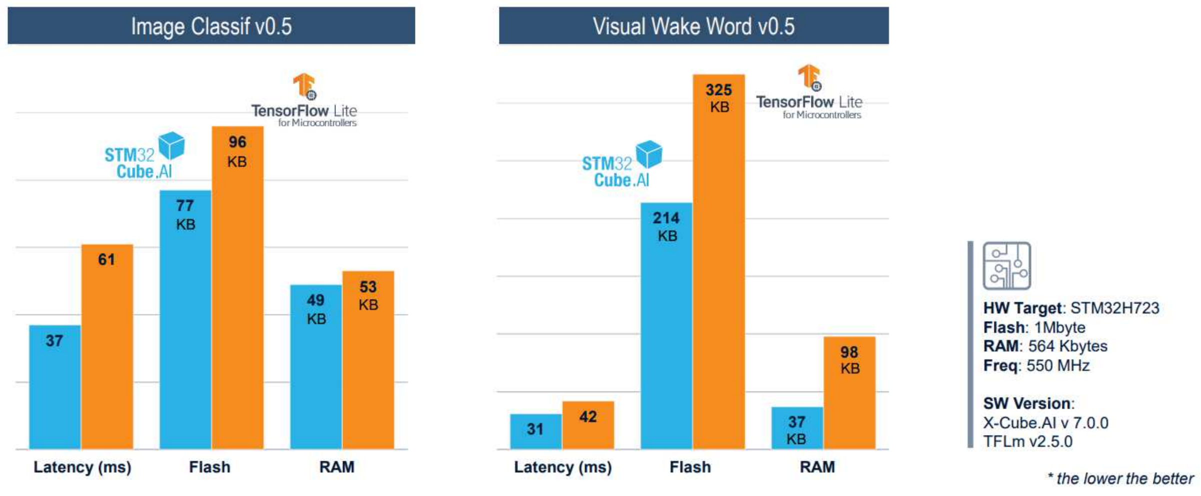


**Figure 13**: Performance of X-CUBE-AI vs TFLite for Microcontrollers for two different problems [30]

# 4. Methods

The Condition Monitoring system presented in this work aims to be:

- ***Versatile***
  It can be used in totally different contexts and applications regarding anomaly detection, any machine that emits mechanical sounds can be monitored.
- ***Easy to adapt***
  Very few modifications are needed to adapt the system to a different application.
- ***General***
  No assumptions are made about the nature of the anomalies that might occur.
- ***Not invasive***
  The system can be deployed on a microcontroller and inference can happen without the device being in direct contact with the monitored machine.
- ***Flexible***
  The threshold to identify anomalies can be algorithmically estimated, if we have some samples of anomalies, or can be set manually to match the required system sensitivity to anomalies.

The system is a semi-supervised classifier based on a fully connected autoencoder. This decision was made in order to make it suitable for most real case applications: as previously stated it is reasonable to assume that abnormal samples are not always available, or available in very low numbers. The proposed system can work with few (or none) abnormal samples.

The autoencoder is trained on the normal *sound fingerprint* of the monitored machine so that at the end it will learn to reconstruct with a low error all the standard noises in normal conditions.

The working principle is that, at inference time, when an anomaly happens the autoencoder will have an increase in the reconstruction error between input and output. If the reconstruction error is larger than a defined threshold an anomaly is detected [14].

This solution allows us not to make any assumptions about the nature of the anomalies. That is because the system, rather than learning all the possible types of anomalies that can occur, which are not usually known a priori, rather simply identifies when the input sounds deviate from the normal learned *sound fingerprint*.

The system was tested and evaluated on three case studies provided by 221e S.r.l:

1. The first case study involves the MIMII dataset [23], a well-known dataset for anomaly detection, this case study had the purpose to be a starting point useful to tune

the parameters and create a general architecture. Furthermore, with this case study, it is possible to compare the results with a baseline model found in the literature.

2. The second case study concerns a client of 221e that manufactures lift electric motors. The purpose of this experiment was to experience the use of ultrasound frequencies in the designed system, dealing with a more complex and custom dataset acquired with the *STWIN* board using its ultrasound embedded microphone.
3. The third case study was about a cooling fan of an electric power supply. In this case, an original data acquisition was performed, and the anomalies were simulated. This third experiment deals with an easier setting but it aims to reduce the dimensions and number of computations of the system by doing some minor architectural changes and considering an embedded porting of the model on the *STWIN* board, by finally performing an on-device validation procedure of the autoencoder.

## 4.1. Datasets

In every case study the dataset is organized in a way that best suited the specific problem, using the following structure:

1. Data is split into audio segments of the same length
2. Normal data is divided into training, validation, and test set
3. Abnormal data is divided into threshold train set and test set
4. Some form of data augmentation is applied

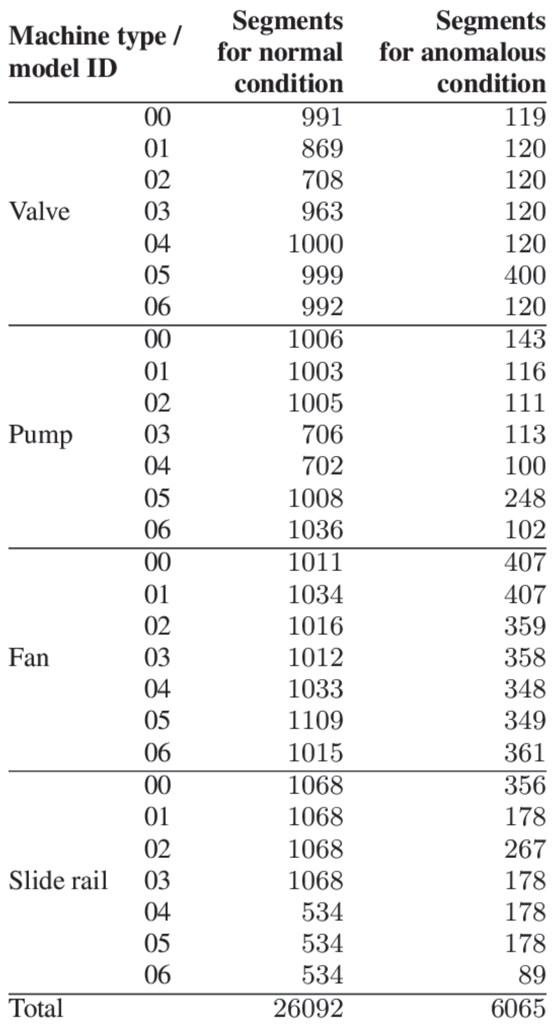
### 4.1.1. Case Study 1: MIMII Water Pump

The MIMII dataset is a sound dataset for malfunctioning industrial machine investigation and inspection [23]. It contains the sounds generated from four machine types: valves, pumps, fans, and slide rails. Each machine type is divided into 7 machine ids. Specifically, the machine id denotes an identifier for each individual machine of the same machine type.

The data for each model contains normal sounds (from 5000 seconds to 10000 seconds) and anomalous sounds (about 1000 seconds) already divided into audio segments. In Table 2 a complete representation of the dataset is provided.

To resemble a real-life scenario, various anomalous sounds were recorded (e.g., contamination, leakage, rotating unbalance, and rail damage). Also, the background noise recorded in multiple real factories was mixed with the machine sounds. The sounds were recorded by an eight-channel microphone array with 16 kHz sampling rate and 16 bit-depth.

The MIMII dataset assists benchmarks for sound-based machine fault diagnosis. Users can test the performance for specific functions (e.g., unsupervised anomaly detection, transfer learning, noise robustness).



| Machine type / model ID | | Segments for normal condition | Segments for anomalous condition |
|---|---|---|---|
| | 00 | 991 | 119 |
| | 01 | 869 | 120 |
| | 02 | 708 | 120 |
| Valve | 03 | 963 | 120 |
| | 04 | 1000 | 120 |
| | 05 | 999 | 400 |
| | 06 | 992 | 120 |
| | 00 | 1006 | 143 |
| | 01 | 1003 | 116 |
| | 02 | 1005 | 111 |
| Pump | 03 | 706 | 113 |
| | 04 | 702 | 100 |
| | 05 | 1008 | 248 |
| | 06 | 1036 | 102 |
| | 00 | 1011 | 407 |
| | 01 | 1034 | 407 |
| | 02 | 1016 | 359 |
| Fan | 03 | 1012 | 358 |
| | 04 | 1033 | 348 |
| | 05 | 1109 | 349 |
| | 06 | 1015 | 361 |
| | 00 | 1068 | 356 |
| | 01 | 1068 | 178 |
| | 02 | 1068 | 267 |
| Slide rail | 03 | 1068 | 178 |
| | 04 | 534 | 178 |
| | 05 | 534 | 178 |
| | 06 | 534 | 89 |
| Total | | 26092 | 6065 |

**Table 2**: MIMII dataset structure [23]

For the first case study, the data from the pump id_00 are used: 1006 normal samples and 143 abnormal samples of 10 seconds each. The monitored machine is a water pump that in normal conditions performs suction, or discharge, from a water pool. Possible anomalies can be represented by leakage, contamination, or clogging. For these reasons the dataset appears to be quite challenging: the normal sounds have high variance and in many cases are indistinguishable from the abnormal sounds by the human ear.

Normal data was divided into training (70%), validation (15%), and test set (15%).

Abnormal data was divided into threshold train set (50%), and test set (50%): a high percentage of abnormal data needed to be kept for testing to provide valid test results given the low volume of available anomalies.

Data augmentation has been applied to 20% of the normal samples adding Gaussian noise. The data were already chunked into 10 seconds samples with ODR = 16 KHz.

### 4.1.2. Case Study 2: Lift

For this case study, the data was acquired using the *MEMS* microphone (*MP23ABS1*) on *STWIN* board. The monitored machine is an electric motor to operate a lift manufactured by a customer of the 221e company.

The data was acquired at the maximum sampling frequency, so ODR = 192 KHz, to be able to capture ultrasounds, and chunked into 1-second samples.

The original work by *221e* [32], on which the acquisitions are taken, had the purpose to identify anomalies in the form of partial discharges. In order to do that the damage was simulated by removing the sheath of one of the motor phase cable. With this setup the following tests were performed:

- Test #1: "Reference" acquisition, where the damage was not present. The motor was powered at $400 \cdot \sqrt{2}V$
- Test #2: Short between two phases, with a supply of $400 \cdot \sqrt{2}V$
- Test #3: Short between two phases, with a supply of $480 \cdot \sqrt{2}V$
- Test #4: Short between phase and ground, with a supply of $400 \cdot \sqrt{2}V$

In test #2, #3 and #4 intensity peaks on the ultrasound frequencies can be detected on the waveform of the data after applying a high pass filter. The assumption is that they represent partial discharge events [32], visible in figure 14.

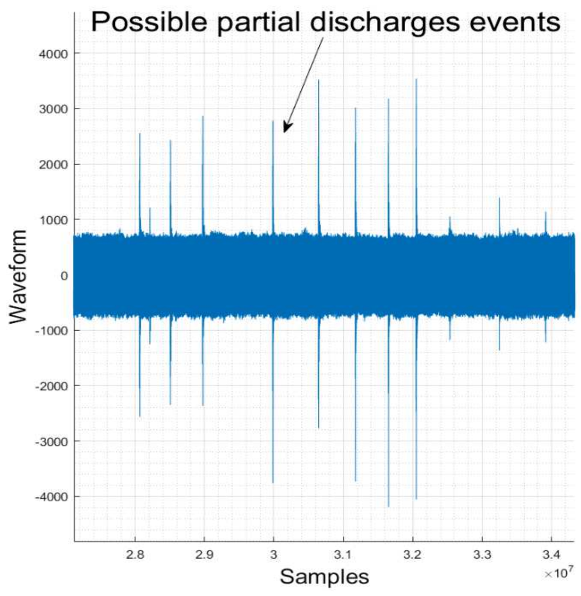


**Figure 14**: Waveform of test #3 acquisition

Partial discharge events are a very unpredictable phenomenon, both in terms of pattern in time, and frequency on the spectrum; the most probable frequency range is between 10KHz and 400kHz [1].

For the scope of this experiment, since the intensity peaks on the ultrasound are sparse and not regular, all four test data were joined and high-pass filtered to remove most of the environmental noise. After that, the data was chunked every 192,000 quantization segments, and the chunks containing a peak with intensity higher than 3,000 were considered anomalies, and all the remains were considered normal sounds to train the autoencoder, according to the assumption made by 221e S.r.l. So the obtained data consist of 125 normal samples and 35 abnormal samples.
Data augmentation has been applied in the form of time-shifting to compensate for the very low data availability.
Normal data has been divided into training (60%), validation (20%), and test set (20%). Validation and test set percentages are higher than the ones on the previous case study in order to obtain more reliable test metrics, given the lower data availability.
Abnormal data was divided, as in the previous case study, into threshold train set (50%), and test set (50%).

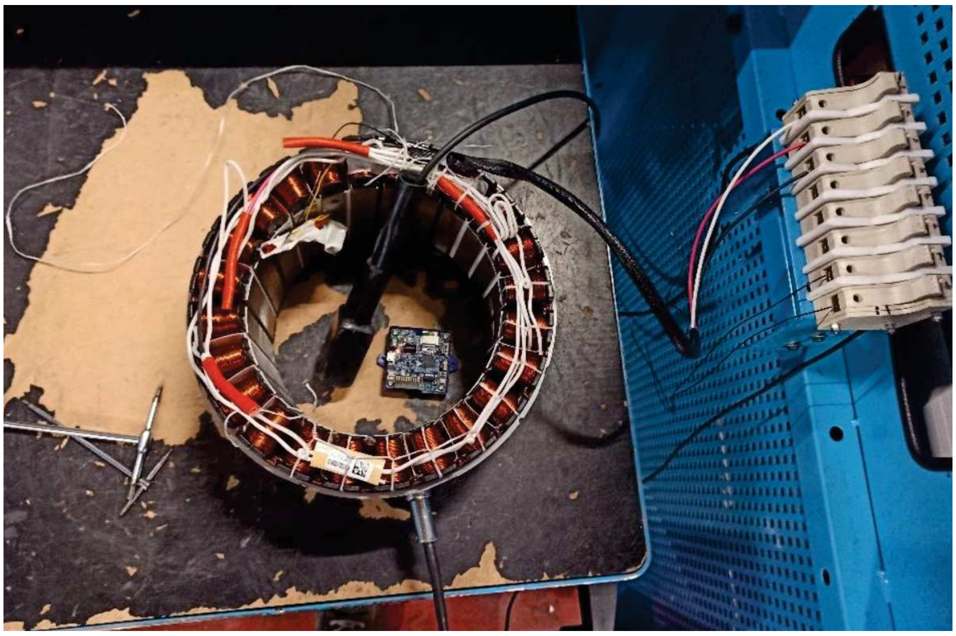


**Figure 15**: Data acquisition setup (221e S.r.l, 2021)

### 4.1.3. Case Study 3: Fan

The data were acquired specifically for the sake of this experiment using the *MEMS* microphone mounted on the *STWIN* board.

The object of the acquisition is a commercial power supply used to convert voltage from 200-220 V to 12 V. The monitored sound is produced by a fan that automatically regulates its speed based on the working temperature. For the anomalous data acquisition, an anomaly was simulated by partially obstructing the fan with a small piece of paper.

The firmware used for the acquisitions is the *FP-SNS-DATALOG1* firmware [27]. The acquisitions were stored on an SD-card on a battery-powered *STWIN*.

Data augmentation was performed by adding Gaussian noise to 20% of normal training data.

Normal data was divided into training (70%), validation (15%), and test set (15%).

Abnormal data was divided into threshold train set (50%), and test set (50%).

The sampling frequency was set to ODR = 8 KHz, the data were chunked into 1 second samples.

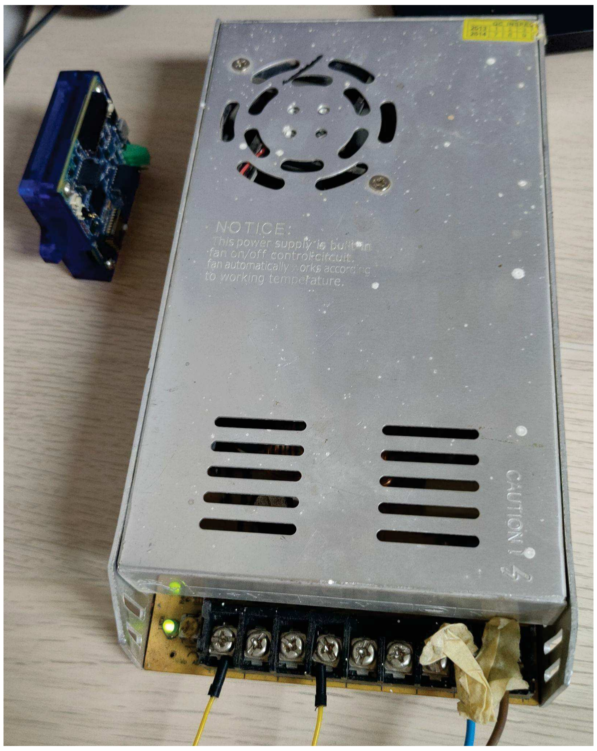

NOTICE:
This power supply is built-in fan on/off control circuit. fan automatically works according to working temperature.
CAUTION

**Figure 16**: Photo of the data acquisition procedure

## 4.2. Signal Processing

The proposed software architecture is designed to be a general pipeline, easy to adapt to different applications.

At inference time the input data will be pre-processed to obtain a normalized and flattened Mel spectrogram. The data will then run through the autoencoder, which was trained on the normal sounds of the monitored machine. Then the reconstruction error is computed, given the input and output of the autoencoder, and compared to a threshold that was set during training by a *threshold selection algorithm*. If the reconstruction error exceeds the threshold an anomaly is detected, otherwise, the input is classified as normal.

This solution is crucially different from a simple threshold approach, because the mapping of the data into the linear space is handled by the autoencoder which is trained on the normal data of any specific case. This has the advantage of being able to deal even with more complex scenarios in a standard way, at the cost of higher computational demand and a loss in simplicity.

In this section, all the steps of the pipeline are described, providing information also about the training process. In figure 17 the whole software pipeline is shown, it is composed of three subsequent blocks: data pre-processing, the autoencoder, and the threshold check.

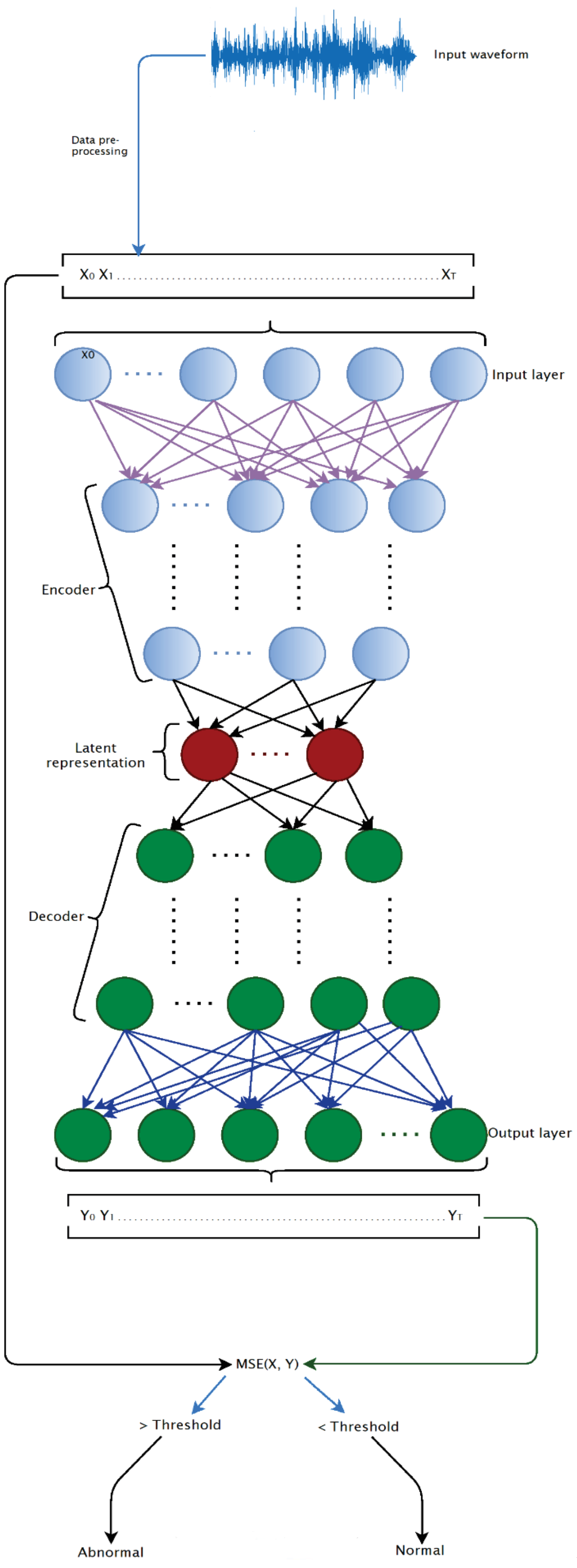


**Figure 17**: Software pipeline

## 4.2.1. Data Pre-processing

As previously stated, the data used for training the autoencoder are only normal sounds of the machine.

The data pre-processing phase is composed of the following steps:

1. Chunk the data into samples of the same length as waveforms (Section 2.3.).
2. Generate Mel spectrograms (Section 2.5.1.) from each chunk using Librosa software library [4].
3. Split the data into training, validation, and test set.
4. Flatten the data in all three datasets, obtaining unidimensional data.
5. Normalize the data between zero and one.

Data augmentation can be optionally applied in the form of Gaussian noise or by time-shifting the chunks before generating the Mel spectrograms.

Assume having audio data with sampling rate of 16 kHz that are chunked into 10 seconds segments. Every sample would then have $16,000 \times 10 = 160,000$ quantization segments. Dividing it by the *hop_length* which is set to 512 for all case studies we would obtain $160,000 \div 512 = 312.5$ so the final spectrogram will have a length of 313. With the Librosa parameter *n_mels* we can also decide the frequency resolution that will represent the height of the final spectrogram, which is set to 128 for all case studies. So the final size of the flattened vector after all the pre-processing would be $128 \times 313 = 40,064$.

## 4.2.2. Architecture of the Autoencoder

The model is designed using Keras [8], and is composed of the following layers:

- InputLayer: shape input_shape
- Dense: shape 500, activation ReLU
- Dense: shape 500, activation ReLU
- Dense: shape 10, activation ReLU
- Dense: shape 500, activation ReLU
- Dense: shape 500, activation ReLU
- Dense: shape input_shape, activation Sigmoid

Where *input_shape* is equal to the size of the flattened input spectrogram and depends on the ODR (sampling rate) and the chosen data size.

The loss function to be used for training is the Mean Square Error (MSE), which also represents our reconstruction error which will be the parameter to be monitored for the Condition Monitoring task.
The activation function for the hidden layers is ReLU (Rectified Linear Unit), while for the output layer the Sigmoid function is used to keep the output between 0 and 1, coherently to the applied input normalization.
The autoencoder needs to be trained on the specific application before being applied to the end-to-end pipeline of the model. As previously stated, the training involves only the normal inputs and the optimization algorithm Adam [15] is used to minimize the MSE between the input and the output.
The validation loss is used as the monitored parameter for early stopping, so the training stops when the validation loss stops improving for a fixed number of epochs.
Now the details of the architecture and training for each case study are presented, in order to show how this architecture can be adapted to very different settings.

**Case Study 1: MIMII Water Pump**

- Input shape: 40064
- Architecture:
    - Dense layer #1 (units: 500, ReLU)
    - Dense layer #2 (units: 500, ReLU)
    - Dense layer #3 (units: 10, ReLU)
    - Dense layer #4 (units: 500, ReLU)
    - Dense layer #5 (units: 500, ReLU)
    - Dense layer #6 (units: 40064, Sigmoid)
- Learning
    - Epochs: 50
    - Batch size: 32
    - Early stopping (on val_loss) patience: 10
    - Optimizer: Adam (learning rate: 0.001)
- Total parameters: 40,616,074

This case study was used to decide the general architecture of the autoencoder, all the parameters were tuned using a 5-fold cross-validation [20] and the purpose was to find a standard setting that would work for different datasets with only minor changes. The tuned hyperparameters were the size of the latent representation layer of the autoencoder (between 10 and 50), the batch size (between 32 and 64), and the learning rate (between 0.001 and 0.0001), considering a total number of 40 different models, given by all the combinations between the hyperparameters multiplied by the number of folds, in order to find the previously shown configuration. The size of the layers of the encoder and decoder was chosen large enough to ensure a good representation of the input size without being too heavy from a computational point of view for the available hardware. The early stopping patience and number of epochs were chosen by trial and error ahead of the 5-fold cross-validation, in order to let the train stop by early stopping when further improvements on the loss are unlikely.

**Case Study 2: Lift**

- Input shape: 48128
- Architecture:
  - Dense layer #1 (units: 500, ReLU)
  - Dense layer #2 (units: 500, ReLU)
  - Dense layer #3 (units: 10, ReLU)
  - Dense layer #4 (units: 500, ReLU)
  - Dense layer #5 (units: 500, ReLU)
  - Dense layer #6 (units: 48128, Sigmoid)
- Learning
  - Epochs 100
  - Batch size: 16
  - Early stopping (on val_loss) patience: 10
  - Optimizer: Adam (learning rate: 0.001)
- Total parameters: 48,688,138

The input size increased compared to the previous case study, even though the chunk size is significantly lower (1 second vs 10 seconds in the previous case study). This is due to a large increase in the sampling rate (192 KHz vs 16 KHz in the previous case study) that allows the spectrogram to be sensitive to ultrasound frequencies. This feature inevitably makes the network large in the number of parameters, also complicating the training process from a computational point of view; that is also why the learning parameters were changed with respect to the previous case study. The higher number of epochs makes the learning longer, providing more iterations to make the loss improve and the lower batch size brings a faster convergence of the loss plot.

**Case Study 3: Fan**

- Input shape: 2048
- Architecture:
    - Dense layer #1 (units: 100, ReLU)
    - Dense layer #2 (units: 100, ReLU)
    - Dense layer #3 (units: 10, ReLU)
    - Dense layer #4 (units: 100, ReLU)
    - Dense layer #5 (units: 100, ReLU)
    - Dense layer #6 (units: 2048, Sigmoid)
- Learning
    - Epochs: 50
    - Batch size: 32
    - Early stopping (on val_loss) patience: 10
    - Optimizer: Adam (learning rate: 0.001)
- Total parameters: 434,058

Since the conditions of this experiment were easier than the previous ones (the anomalies, in this case, could easily be detected by the human ear), the challenge for this experiment was to simplify the architecture of the network in order to create a model that could fit directly into the *STWIN* memory for embedded applications.
The weights lowering and input dimensionality reduction, due to the lower sampling rate, allowed the autoencoder to be represented by 434,058 parameters; instead of the 48,688,138 parameters of the *lift* case study.

### 4.2.3. Threshold Selection

The procedure of threshold selection is to be performed after the unsupervised training of the previously defined autoencoder on normal data. The parameter for which we want to define a classification threshold is the reconstruction error of the autoencoder.

A lower threshold would mean higher sensitivity and lower specificity, while a higher threshold would mean a lower sensitivity and higher specificity of the final classifier. However, if some anomalous data are available a good threshold can be obtained by choosing one, among some hypotheses, that maximizes the AUC of the classifier.

One of the outputs of the thesis is the original *threshold selection algorithm.*

The idea of the algorithm started by studying the AUC metric and how it represents the quality of a classifier (Section 2.1.5.). The initial idea was to create an algorithm that chooses the best threshold, among some hypotheses, according to the AUC metric. This had the problem of giving a very low control on the tradeoff between sensitivity and specificity. So an additional constraint was added to the output, only thresholds with specificity higher than 0.95 are considered in the search. This choice is made on the assumption that it is convenient to prioritize specificity over sensitivity, to avoid false positives and detection of false anomalies. This is usually preferable in Condition Monitoring settings, where the system should be transparent and not invasive, except when an anomaly is detected with a high probability. An exception can be situations where we are dealing with a very high-risk machine, like an airplane motor or a nuclear reactor, where we would highly prefer having false alarms rather than overlooked anomalies.

With the exception of the discussed tradeoff between sensitivity and specificity, there is no other hyperparameter to be set according to the specific application, this makes the algorithm a general tool that was applied without any need for custom changes to all three case studies.

The proposed algorithm requires at least a small set of anomalous data. The case in which no anomalous data is available at all will also be later discussed.

```
Threshold selection algorithm (pseudocode)
INPUT:
lossN (list containing the reconstruction errors on normal data),
lossA (list containing the reconstruction errors on abnormal data),
k (number of threshold hypotheses to be investigated)
OUTPUT: bestThresh

if min(lossA) >= max(lossN):   // the data points are linearly separable
    return ( max(lossN) + min(lossA) ) / 2
thresholds = k possible thresholds equally distributed between avg(lossN) and max(lossA)
yClass = a binary array of the true predictions on both lossN and lossA
bestThresh = 0
bestAUC = 0
for threshold in thresholds:
    predClass = a binary array of the classifier predictions using the current threshold
                on both lossN and lossA
    AUC = AUC between predClass and yClass
    specificity = specificity of the classifier with current threshold
    if AUC > bestAUC and specificity >= 0.95:
        bestAUC = AUC
        bestThresh = threshold
return bestThresh
```

As can be observed the algorithm returns the halfway value between the maximum of *lossM* and the minimum of *lossA* if the two classes of data are linearly separable. Otherwise, it performs a search between *k* hypotheses to find the best threshold according to the specified conditions. Larger values of *k* improve the precision of the algorithm at the cost of execution time.

For all the case studies in this work *k* = 1000 is used, while the data used to get the reconstruction errors for *lossN* and *lossA* were respectively the validation normal data and the threshold train abnormal data (50% of the abnormal dataset).

The use of the described *threshold selection algorithm* makes the classifier semi-supervised: the total number of parameters of the classifier are the ones of the autoencoder plus the threshold parameter; the parameters of the autoencoders are all unsupervised, while the threshold is the only supervised parameter.

This solution represents a good tradeoff between the generality of an unsupervised model and the accuracy of a supervised model.

In extreme cases, in which we do not have access to any kind of abnormal data, the *threshold selection algorithm* cannot be used. However, the threshold can still be estimated by dividing the mean of the normal reconstruction errors by their standard deviation [24]. In this last case, we would lose the only supervised block of the pipeline, making the system completely unsupervised, and also unable to be properly tested since no anomalous data can be used for evaluation.

## 4.3. Embedded Solution

Neural networks are usually performance-demanding models: this is why embedded portings on low-energy and low-performance devices are often challenging. Seeing the dimensionality reduction achieved by the network, for the third case study an embedding for *STWIN* is possible, differently from the network architectures in the previous, more complex, case studies.

The network of case study 2 weighs 154.95 MB over an available memory on *STWIN* of 2 MB. That is why the reduction of the network for case study 3 described in Section 4.2.2. was essential for the sake of this experiment. With the original architecture (as in the first two case studies) the network would have still weighted 9.79 MB, despite the small input size. That is the reason that brought to the lighter architecture in case study 3 and made an embedded porting of the network on *STWIN* possible.

This experiment is motivated by the will of *221e* to bring AI to the Edge level, in order to pave the way for future end-to-end real-time demo on the device. For that goal it was decided to create a porting of the machine learning part of the pipeline, which is the autoencoder, for the *STWIN* board, taking advantage of the *STMicroelectronics* development ecosystem for *STM32* microcontrollers described in Section 3.2..

### 4.3.1. Process

In order to avoid ambiguity, the term "validation" in this section refers to the testing of the Autoencoder from inputs to outputs in the embedded environment using test data, in order to stay coherent with the terminology of *X-CUBE-AI*.

Before handling the validation, the previously described autoencoder was saved and exported in TensorFlow Lite format after training.

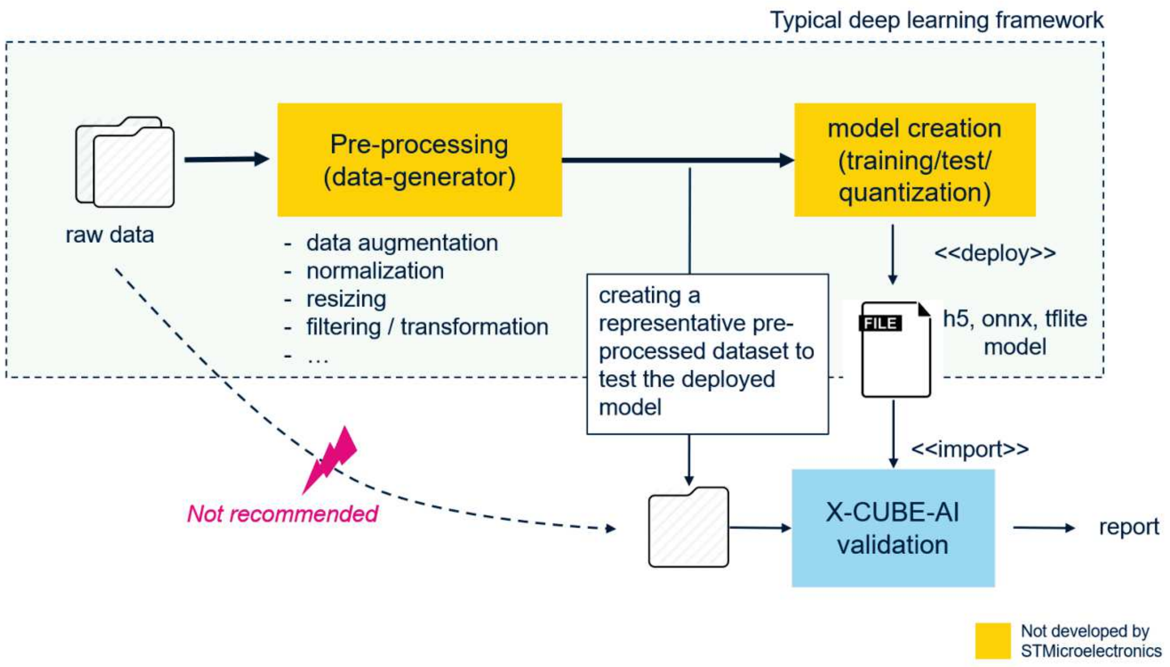
The validation C code was generated using the board selection tool on *STM32CubeMX*, selecting the *STEVAL-STWINKT1* device and importing the *X-CUBE-AI* expansion for a validation application.

To achieve reasonable inference times the minimum heap size was set to 8000 on the linker settings, while the minimum stack size was set at 800.

Without any compression, the imported network occupies 1.67 MB over the 2 MB Flash memory available on the board. In order to be safer, also considering a possible future end-to-end embedded demo, where also memory to store the rest of the pipeline would be needed, a low compression was applied, achieving a memory size of 512 KB over 2 MB, which is reasonable for a future end-to-end application. The used RAM is 11.95 KB over 640 KB available.

After the previous steps, the C firmware was generated for the *STM32CubeIDE* and loaded to the device using the *STLINK-V3MINI*.

Two validation procedures have been performed, one for normal data and one for abnormal data. In any case, the input data for the validation represents test data, after the pre-processing phase, exported in CSV format from Python. In figure 18 a schematic representation of the validation procedure is provided. The sequence of steps needed to reproduce the experiment is described in Appendices B and C.



Typical deep learning framework
raw data
Pre-processing (data-generator)
- data augmentation
- normalization
- resizing
- filtering / transformation
- ...
model creation (training/test/ quantization)
<<deploy>>
FILE
h5, onnx, tflite model
creating a representative pre-processed dataset to test the deployed model
<<import>>
Not recommended
X-CUBE-AI validation
report
Not developed by STMicroelectronics

**Figure 18**: Representation of the validation procedure (STMicroelectronics)

# 5. Evaluation and Results

In this chapter the results for each case study are presented and described, including the results of the embedding of the autoencoder on the *STWIN*.

## 5.1. Case Study 1: MIMII Water Pump

As previously stated, this case study was essential in order to set a general standard setting for the experiments, tune the parameters, and compare the results to an important baseline from the literature.

The achieved average reconstruction error on new data is 0.0041 for normal data, 0.0128 for abnormal data. The threshold returned by the *threshold selection algorithm* is 0.0063 and the two classes are not linearly separable.

In Table 3 the values of the evaluation metrics for the end-to-end classification of test data are shown.

| | |
|---|---|
| Accuracy | 0.9238 |
| Specificity | 0.9868 |
| Sensitivity | 0.7917 |
| AUC | 0.8892 |

**Table 3**: MIMII water pump case study results

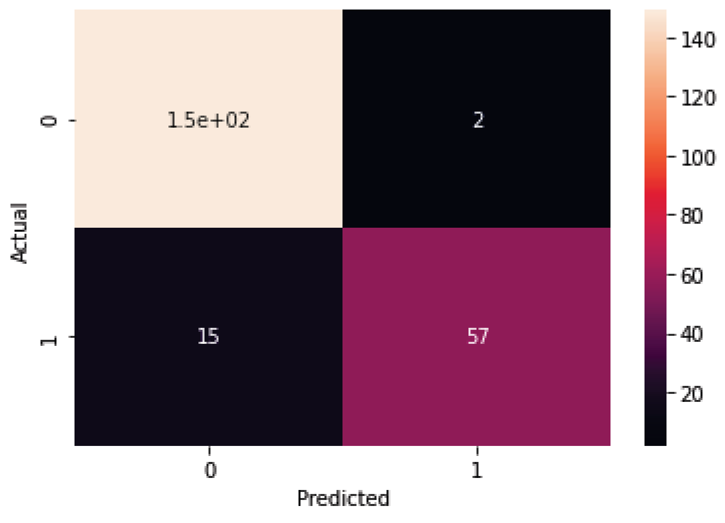


**Figure 19**: MIMII water pump case study confusion matrix over test data.

0: normal samples

1: abnormal samples

| **Work** | **AUC** |
|---|---|
| This work | 0.8892 |
| Baseline model [17] | 0.6715 |

**Table 4**: MIMII water pump case study AUC comparison with baseline

The baseline model listed in Table 4 is an autoencoder-based model presented as a baseline for a DCase challenge regarding Machine Condition Monitoring [17]. The reported AUCs are on the same machine ID of the MIMII Dataset. As can be seen from the results the proposed Condition Monitoring system performs considerably better than the baseline model.

The results of this experiment are positive. It can be observed from Table 3 that the *threshold selection algorithm* correctly gives priority to a high value of specificity, guaranteeing a low false positive rate. Given the generality of the system, meaning the capability of detecting types of anomalies that have never been observed before, it was decided that it is preferable to give priority to a low false positive rate, even at the cost of a higher false negative rate, in order not to risk false alarms. However, this preference can be easily changed by changing the conditional parameters of the *threshold selection algorithm* (Section 4.2.3.), adapting the results to different needs and applications.

In figure 20 and figure 21 we can see two examples of the reconstruction of spectrograms made by the autoencoder. From these two particular examples, we can clearly see what was discussed in Section 4.: the autoencoder reconstructs the normal sounds more easily for its unsupervised nature, while it struggles to reconstruct some particular features of the abnormal sounds.

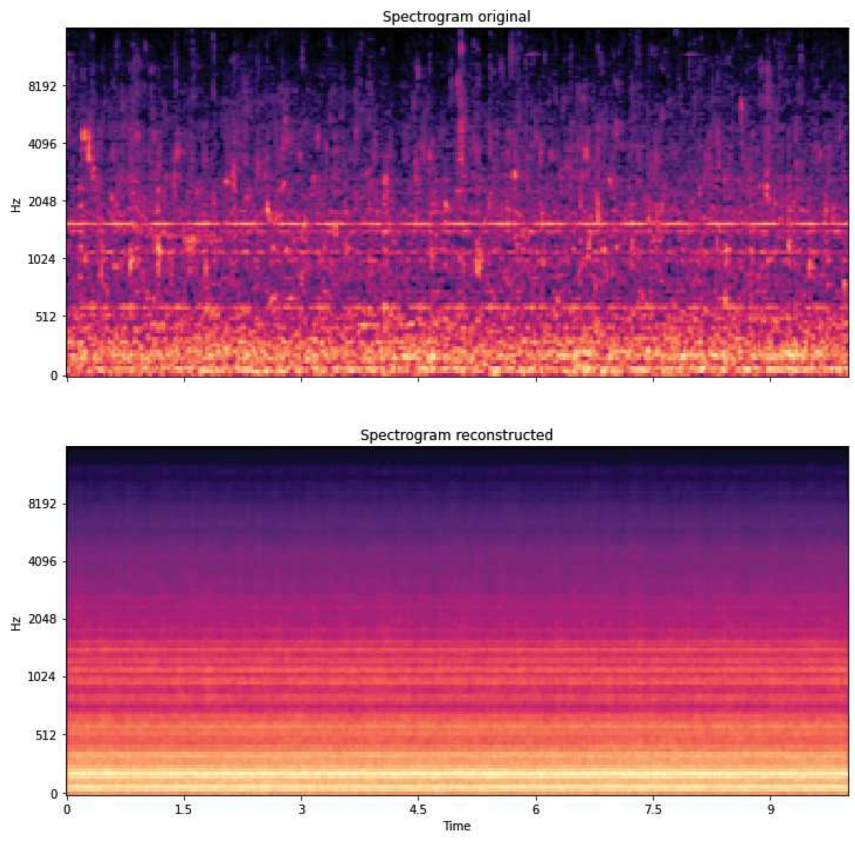


**Figure 20**: Original and reconstructed spectrogram of a normal sample

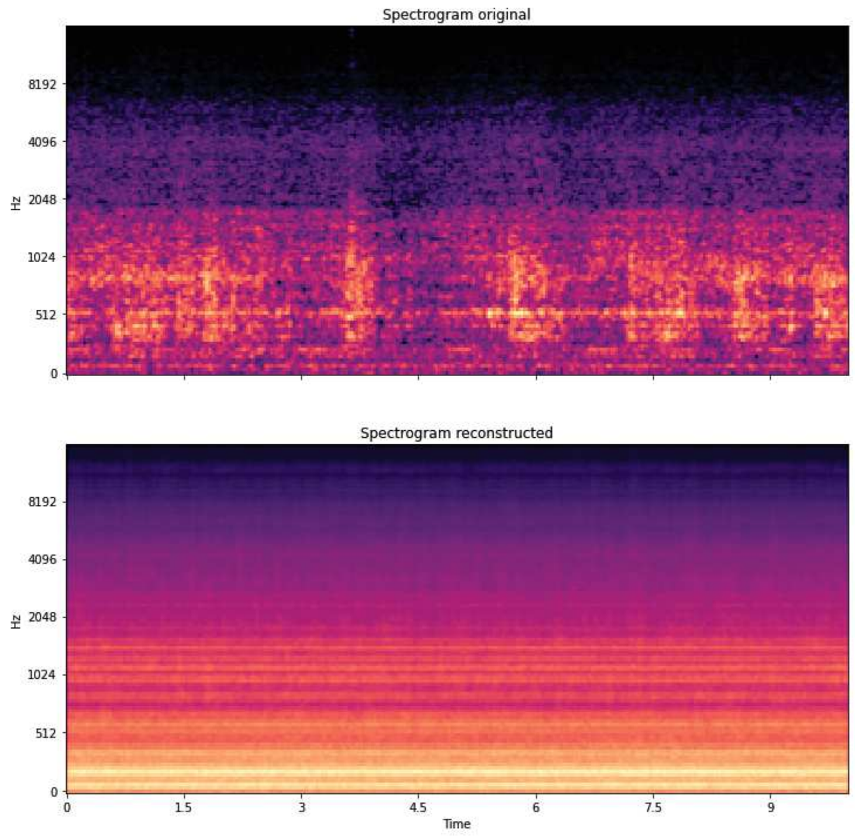


**Figure 21**: Original and reconstructed spectrogram of an abnormal sample

## 5.2. Case Study 2: Lift

This case study had the purpose to prove the effectiveness of the system in an original and more challenging environment with the use of ultrasound frequencies.

The average reconstruction error on new data is 0.0008 for normal data, 0.0056 for abnormal data. The threshold returned by the *threshold selection algorithm* is 0.0016 and the two classes are not linearly separable.

In Table 5 the values of the evaluation metrics for the end-to-end classification of test data are shown.

| | |
|---|---|
| Accuracy | 0.9401 |
| Specificity | 0.9469 |
| Sensitivity | 0.8824 |
| AUC | 0.9146 |

**Table 5**: Lift case study results

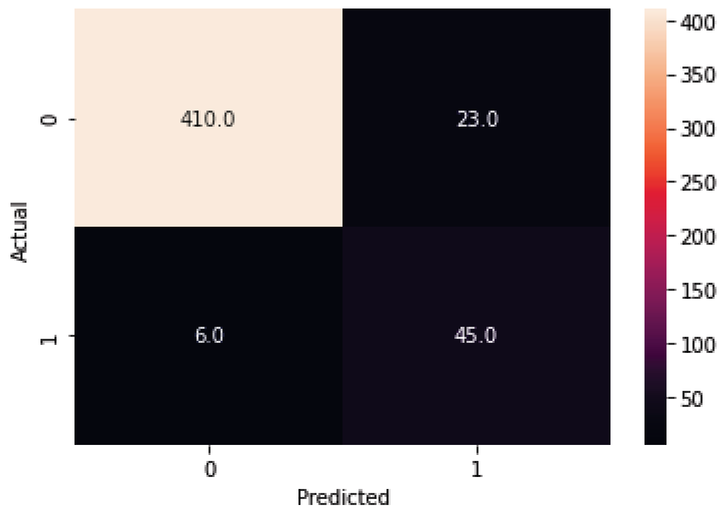


**Figure 22**: Lift case study confusion matrix over test data.
0: normal samples
1: abnormal samples

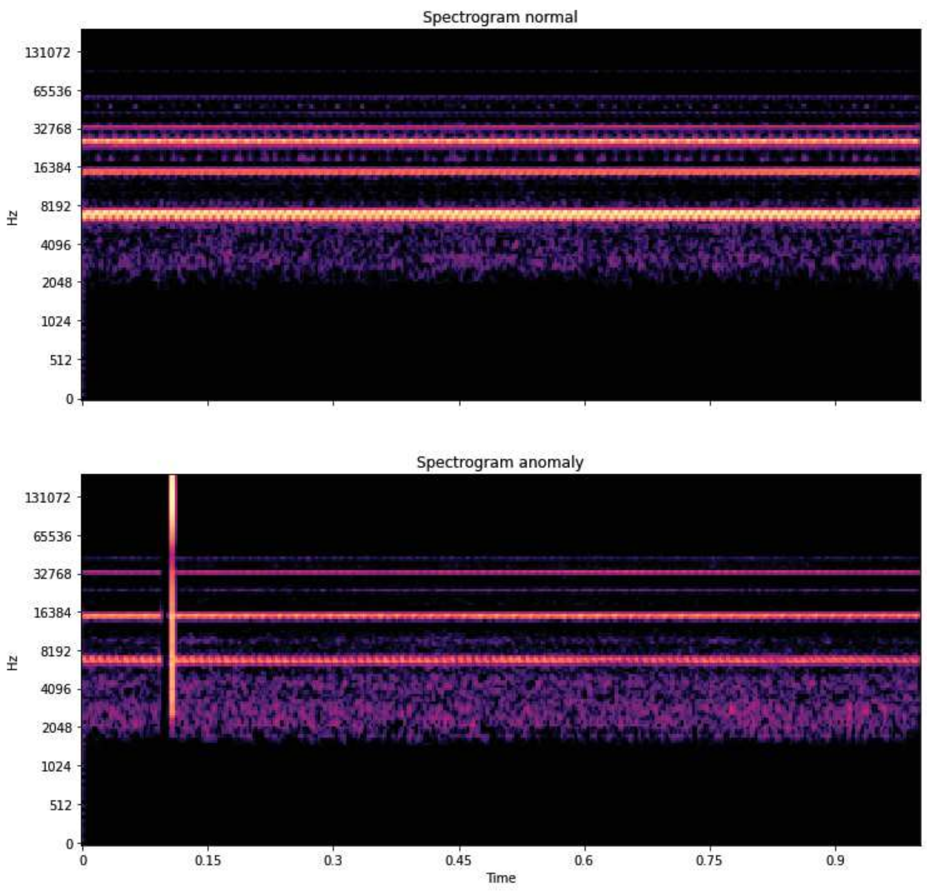


**Figure 23**: Comparison between spectrograms of a normal and an abnormal sample

As previously stated, these results are based on the hypothesis that amplitude peaks on higher frequencies represent partial discharges, so a particular class of anomalies. As we can see from a clear example in figure 23: the anomaly can be seen as the vertical colored line that in this case appears on a high variety of frequencies, and the black area on lower frequencies depends on the application of the high pass filter.

These results can be considered satisfactory based on the premises of the previous study conducted by the *221e* company, also considering the particular complexity of the problem, working with data acquired in different conditions of the motor. It is also worth considering that the training has been quite computationally intensive and it needed more epochs than the previous experiment to find the best results. In figure 24 we can see the loss slowly going down epoch after epoch till epoch 84, after that the improvements stopped. Even small changes (order of $10^{-4}$) in the reconstruction error of normal samples can influence a lot the performance of the system, which is why it is important to reach the minimum possible reconstruction error on normal samples.

The advantage of using this system, instead of a simple intensity threshold strategy, is the generality that is gained: other types of anomalies can be possibly identified other than partial discharges, because of the semi-supervised nature of the solution. With a simpler threshold-based system only partial discharges could be detected as anomalies, so the system would not make use of any additional knowledge on the normal *sound fingerprint* of the machine, possibly totally overseeing anomalies with different characterizations.
In figure 25 and figure 26 the spectrograms of a normal and an abnormal sound are represented, with their respective reconstructions of the autoencoder. In the represented case the anomaly is clear and visible in order to show the difference in the reconstruction, however, the inputs in this case study are in general complex and with a large variety both in terms of normal and abnormal samples.

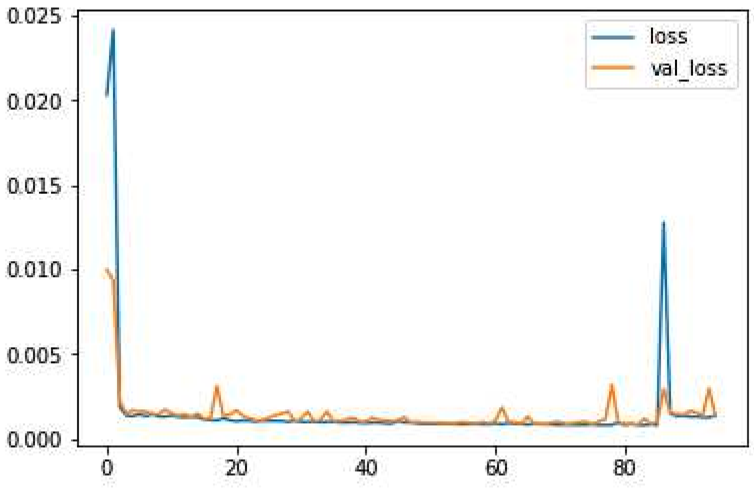


**Figure 24**: Plot of the *loss* and *val_loss* during training

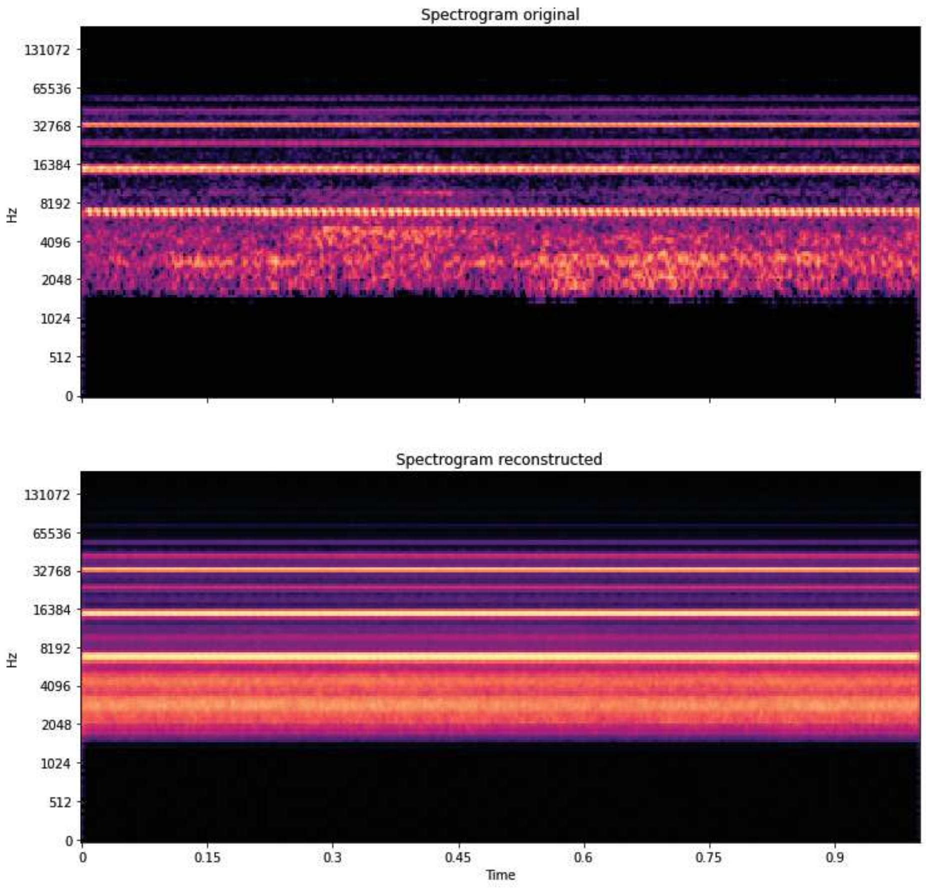


**Figure 25**: Original and reconstructed spectrogram of a normal sample

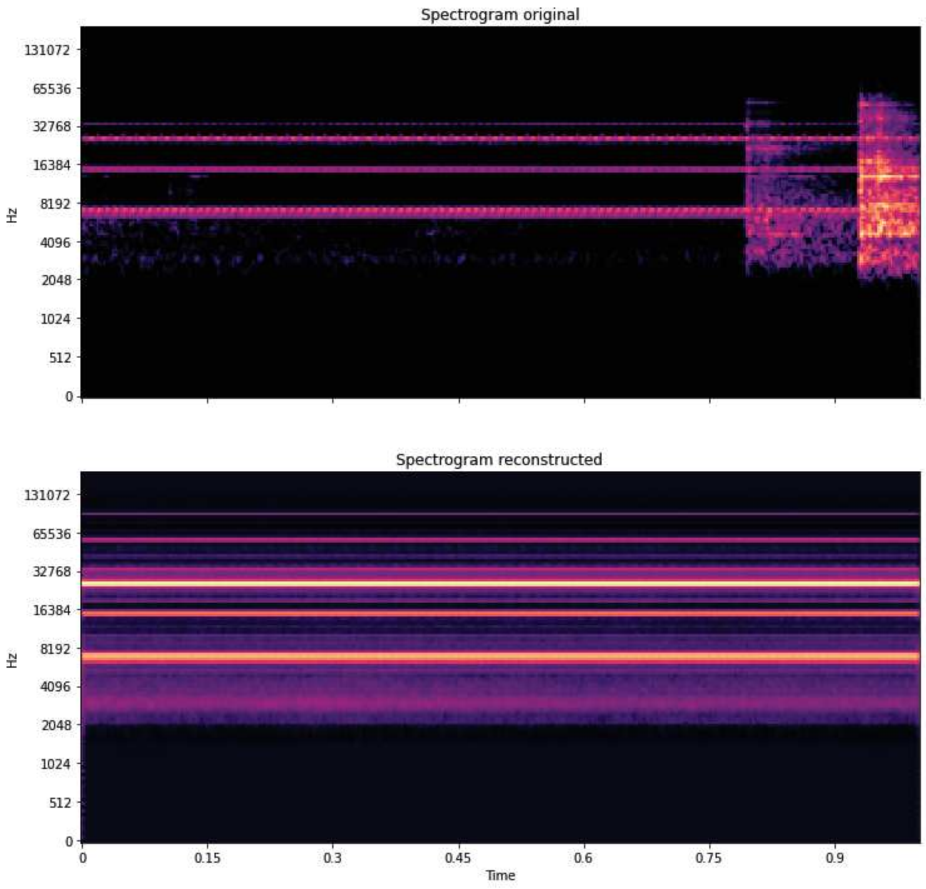


**Figure 26**: Original and reconstructed spectrogram of an abnormal sample

## 5.3. Case Study 3: Fan

The goal of this case study was to reduce the dimensionality of the model in order to make it suitable for embedding environments.

The average reconstruction error on new data is 0.0017 for normal data, 0.0542 for abnormal data. The threshold returned by the *threshold selection algorithm* is 0.0261 and it achieved data separability between the two classes with a distance between the two closest points of 0.0453.

In Table 6 the values of the evaluation metrics for the end-to-end classification of test data are shown.

| | |
|---|---|
| Accuracy | 1.0 |
| Specificity | 1.0 |
| Sensitivity | 1.0 |
| AUC | 1.0 |

**Table 6**: Fan case study results

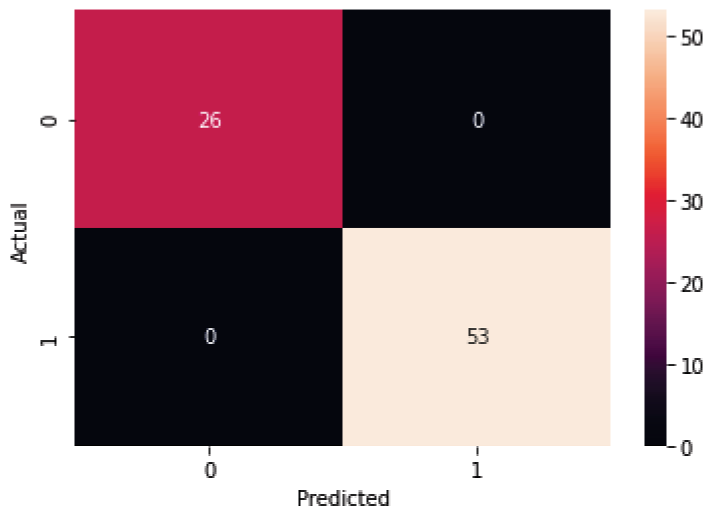


**Figure 27**: Fan case study confusion matrix over test data.
0: normal samples
1: abnormal samples

As expected, the system in this case performs a perfect classification. The normal and abnormal samples are mapped into linearly separable points on the reconstruction error space. As a consequence of that, the *threshold selection algorithm* returned the halfway value between the closest extremes of the two classes.

In figure 28 and figure 29 the difference between a normal and abnormal sample is clearly visible in the spectrograms.

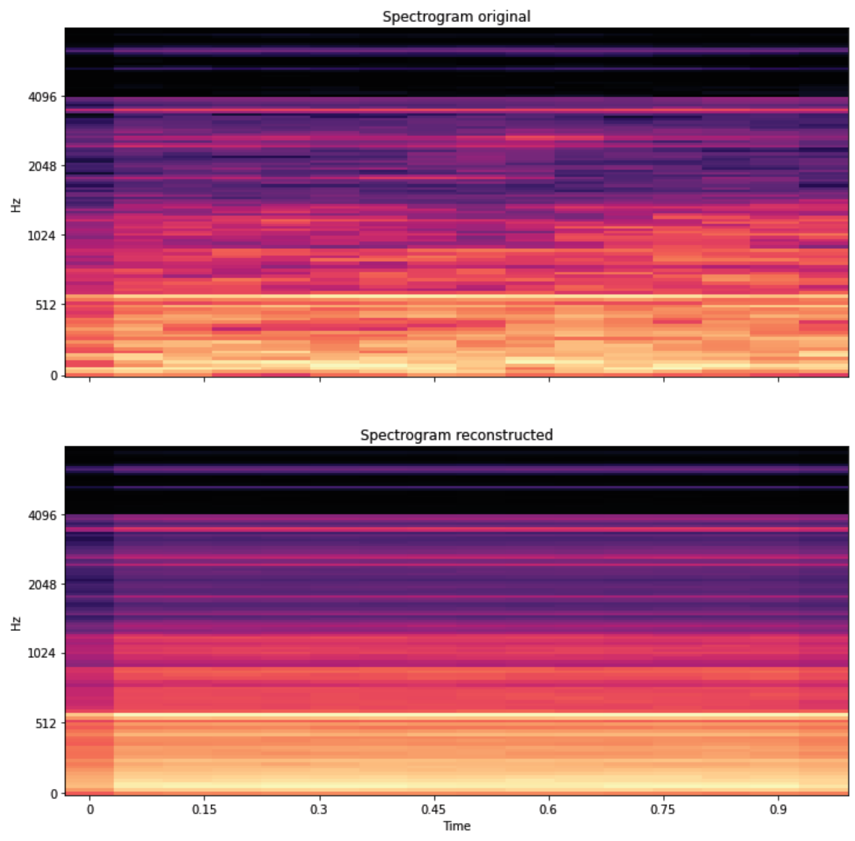


**Figure 28**: Original and reconstructed spectrogram of a normal sample

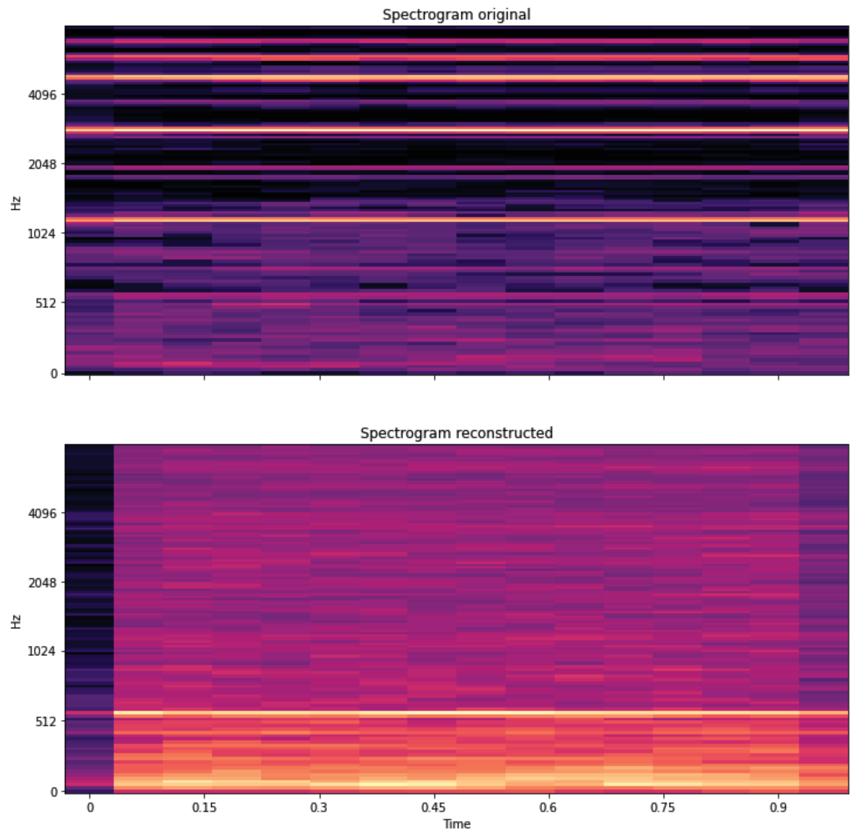


**Figure 29**: Original and reconstructed spectrogram of an abnormal sample

### 5.3.1. Validation on STWIN

The following displayed results concern the validation procedure on the autoencoder after the process described in Section 4.3.1.

| Output | mse | mean | std |
|---|---|---|---|
| stm32 c-model | 0,0016594 | 0.0003319 | 0.0407342 |
| original model | 0,0016585 | 0.0003252 | 0.0407233 |

**Table 7**: Results of validation on normal samples

| Output | mse | mean | std |
|---|---|---|---|
| stm32 c-model | 0,0542411 | -0.1671737 | 0.1621552 |
| original model | 0,0542204 | -0.1671140 | 0.1621528 |

**Table 8**: Results of validation on abnormal samples

From Table 7 and 8 it can be easily observed that the compression did not relevantly compromise the performance of the model. The MSE of both normal and abnormal samples increased slightly with a difference in the order of $10^{-6}$ for normal samples and $10^{-5}$ for abnormal samples, way smaller than the distance between the two classes, still ensuring a perfect classification.
It is also interesting to notice how the mean and standard deviation change from normal to abnormal samples. This is also due to the normalization in the pre-processing phase, which is done according to the mean and the standard deviation of the training set (normal data).
Regarding the inference time, the average time taken for one iteration (from input to output) is 53 ms with the clock set at the maximum frequency of 120 MHz, so an average of 6,360,000 CPU cycles per iteration. In Table 8 it can be observed the average inference time for a single input layer by layer.
These results demonstrate that an embedding application of the model is possible. Of course, for a complete end-to-end demo the pre-processing phase should be implemented on the *STWIN* as well and put at the head of the model. The output frequency of the system should then be set according to the overall performance of the full pipeline.
An additional observation is that the false positive rate could be further decreased in a real-time embedded application of the system, by assuming that anomalies do not last only

for one time unit, but continue for a greater amount of time. This is particularly reasonable in this case study: the time the fan takes to perform a full spin is largely smaller than the time of a chunk acquisition, and an anomaly on the fan would reasonably show at each spin under the same conditions. In this way, an anomaly will be notified if the system returns only abnormal classifications for a fixed time window of N outputs. However, in more complex situations like case study 2, where anomalies appear with unpredictable frequency, this assumption is not valid.

In an embedded battery-powered environment the anomalies can be notified via the LEDs of the *STWIN,* or by employing the BLE (Bluetooth Low Energy) module on board to send the information to an external device to incorporate the node into a wider IoT network where data aggregation with other sensors can be performed in order to build more complex Condition Monitoring systems.

```
c_id  desc           output                      ms       %
---------------------------------------------------------------------------------
0     Dense (0x104)  (1,1,1,100)/float32/400B    23.688   44.7%
1     NL (0x107)     (1,1,1,100)/float32/400B    0.016    0.0%
2     Dense (0x104)  (1,1,1,100)/float32/400B    0.824    1.6%
3     NL (0x107)     (1,1,1,100)/float32/400B    0.016    0.0%
4     Dense (0x104)  (1,1,1,10)/float32/40B      0.089    0.2%
5     NL (0x107)     (1,1,1,10)/float32/40B      0.004    0.0%
6     Dense (0x104)  (1,1,1,100)/float32/400B    0.118    0.2%
7     NL (0x107)     (1,1,1,100)/float32/400B    0.016    0.0%
8     Dense (0x104)  (1,1,1,100)/float32/400B    0.824    1.6%
9     NL (0x107)     (1,1,1,100)/float32/400B    0.016    0.0%
10    Dense (0x104)  (1,1,1,2048)/float32/8192B  24.068   45.4%
11    NL (0x107)     (1,1,1,2048)/float32/8192B  3.322    6.3%
---------------------------------------------------------------------------------
                                                 53.001 ms
```

**Table 9**: Information on average execution time layer by layer for one input

# 6. Discussion and Conclusions

In this thesis, the advantages and disadvantages of sound-based Condition Monitoring systems were discussed, stating that they are more general (can identify more types of anomalies), and less invasive (do not require physical contact) compared to more classical systems using temperature, mechanical vibrations, or voltage. On the other hand, they are more sensitive to noise and they show the typical problems of deep learning based solutions: high demand for training data and explainability.

The high demand for training data and noise are two related problems since a good quality dataset can give the system the knowledge it needs to extract the signal from the irrelevant noise during training. An unsupervised or semi-supervised approach can also significantly mitigate the problem of data demands since it would need only normal sound data for training, and only a low or inexistent amount of abnormal data. That is also coherent with the specifics of the problem of Condition Monitoring: anomalies are not usually known a priori.

An original sound-based Condition Monitoring system was then presented. The pipeline of the system consists of three different phases: a data pre-processing sequence, an autoencoder, and a threshold-based strategy. The data pre-processing phase uses the Fourier transform to obtain Mel spectrograms from audio chunks and performs normalization and flattening. The autoencoder, which is trained only on normal pre-processed data, tries to reconstruct the input in the output. In the end, the MSE between input and output is computed and compared to a threshold value, dynamically decided by the proposed original *threshold selection algorithm* after the training of the autoencoder. If the computed MSE is higher than the threshold an anomalous sample is detected.

The system has then been applied to three case studies: a water pump from the MIMII Dataset, a lift motor produced by a customer company of *221e S.r.l.*, and a cooling fan of an electric power supply provided by *221e*. The results of all three case studies are positive and they all had very different settings and aims.

The *MIMII water pump* case provided an important reference to a baseline system in the literature, in order to confirm the practical usability of the system. This first experiment was also used to properly tune the used parameters of the *threshold selection algorithm* and set the tradeoff between Sensitivity and Specificity choosing to give priority to the latter.

The *lift* case study allowed the system to be tested on a less standard environment, which involved the use of original data acquired by *221e S.r.l.* with the *STWIN* board at a high sample rate, allowing the system to actually work with real anomalies manifesting on ultrasound frequencies.

The *fan* case study required original data acquisitions using the *STWIN* board. The aim, in this case, was to reduce the dimensions of the model in order to study a porting inside the *STWIN* board, performing an embedded validation of the autoencoder on the device. This experiment set the basis for an eventual future end-to-end implementation, by *221e S.r.l.*, of the whole system on board an *STM32* device.

To conclude, sound-based Condition Monitoring systems represent a valid solution and can be easily adapted to very different practical cases even in embedded settings, paying particular attention to the noise conditions of the different environments and the quality of the data acquisitions.

# Appendices

## A) 221e: Muse

*MUSE* is a low-power, miniaturized, wireless multi-sensor logger, incorporating state-of-the-art sensing technology into a compact, robust, customizable, and easy-to-use solution.

*MUSE* combines inertial and environmental sensors together with onboard flash storage, wireless connectivity, automated power on/off functions, and regulated rechargeable power, providing a versatile system for data acquisition in a multipurpose fashion.

*MUSE* runs proprietary algorithms that range from orientation estimation to embedded time-frequency analysis for predictive maintenance applications, from driving and vehicle monitoring routines to activity tracking metrics extraction in the fields of sport, wellness, and e-health.

**Figure 30**: *MUSE* (221e S.r.l)

Applications:

- Inertial Measurement Unit
- Attitude and Heading Reference System
- Acceleration, angular rate, and motion tracking
- Magnetic fields measurement surrounding the device
- Environmental Condition Monitoring
- Sport and e-health applications
- Vibration and structural monitoring
- Drones, robotics, high-precision systems

Product features:

- Digital inertial sensors: gyroscope (up to ±2000 dps), accelerometer (up to ±32g), magnetometer (up to ±16G)
- Digital 3-axis High Dynamic Range (HDR) accelerometer (up to ±400g)
- Temperature (accuracy up to ±0.1 °C, -40…125 °C) and relative humidity (accuracy up to ±1.5 %RH, 0…100 %RH) sensors
- Digital barometer (accuracy 0.5 hPa, 260-1260 hPa)
- Light and proximity sensor (0.0022 - 73000 lx, detection range up to 160 mm)
- Audio sensor omnidirectional digital microphone (–26 dBFS ±3 dB sensitivity, –26 dBFS ±3 dB sensitivity)
- External flash memory (Nand, 1024 Mbit) • RGB LED, buzzer and push button for user interaction • 16-pin slim stack board-to-board expansion connector
- Dedicated ARM® Cortex®-M4 CPU with FPU, up to 64 MHz speed
- Dedicated ARM® Cortex®-M0 for radio and security tasks
- USB Type-C connector
- Rechargeable battery
- Selectable device configurations, output formats and sample rates for data

The main goal of this work was to experiment and test audio-based solutions for Condition Monitoring in embedded environments in order for *221e S.r.l.* to consider the integration of an audio-based Condition Monitoring into the *MUSE* in the future.

## B) STM32CubeMX: Code Generation

A new trend is emerging from several microcontroller manufacturers. Driver code can now be configured and generated using provided tools. *STM32CubeMX* is the provided tool for generating project reports and code in *STM32* environment.
The process of code generation in *STM32CubeMX* starts with the selection of the *STM32* microcontroller on which we want to develop.
The graphical tool has four main views that reflect the four main functionalities of the software:

- **Pinout & Configuration view**: this view, shown in figure 30, includes a visualization of the microcontroller and its pins and has a vertical side toolbar. The peripheral of interest for our application can be selected from the toolbar, the tool will automatically assign them to the appropriate pin. Otherwise, the user can select a pin directly and select a supported peripheral.

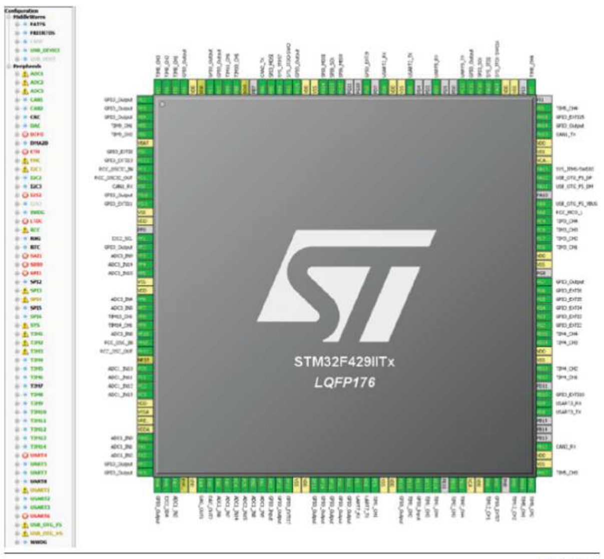


**Figure 31**: Pinout view

After selecting the peripherals we can deal with their configuration (i.e. data size, sampling rate, data conversion mode, and resolution).

The integration and configuration of extra software packs, like *X-CUBE-AI*, is also done in this view.

- **Clock Configuration view**: This view is shown in figure 32 and provides a good overview of the clock tree. It enables the developer to choose between external and internal clock sources. Clock frequencies in the clock hierarchy will be automatically adjusted after every change. Invalid clock configurations are shown in red making it easy to fix problems and find incompatibilities.

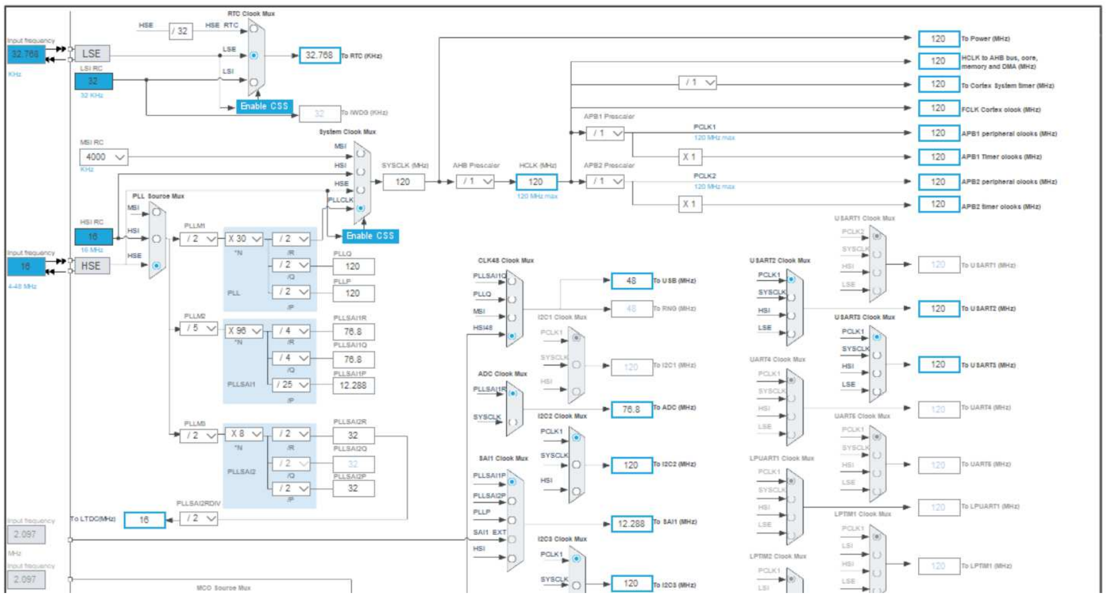
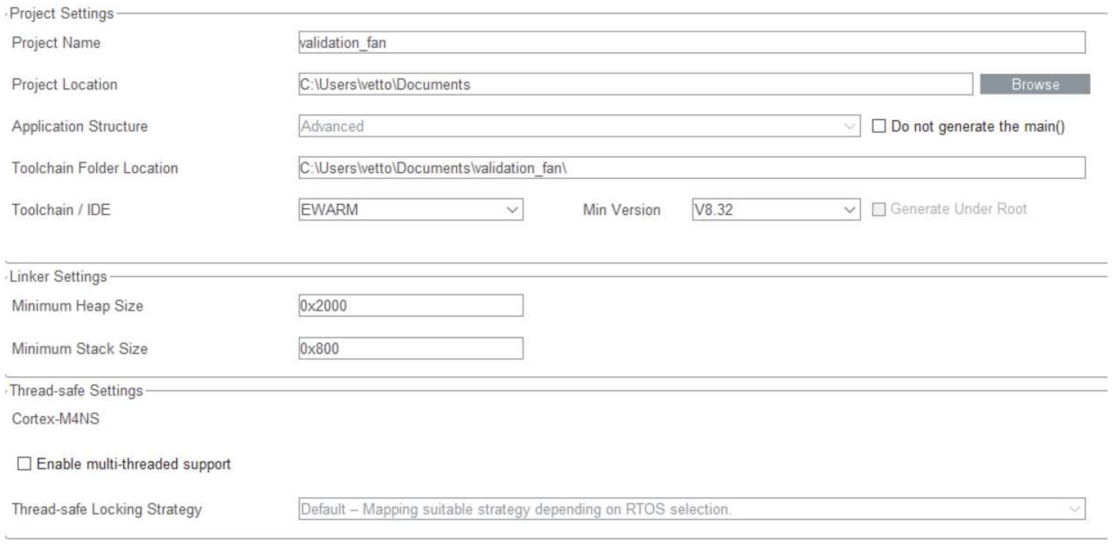


**Figure 32**: Clock tree

- **Project Manager view**: from this view, we can set all the settings regarding the generation of the project code, like the project name, location and the IDE to be used for generation (i.e. *STM32CubeIDE*). It is then possible to set the minimum heap size and minimum stack size under the linker settings. Multithreading strategies, if any, can be selected in this view.



Project Settings

| | |
|---|---|
| Project Name | validation_fan |
| Project Location | C:\Users\vetto\Documents |
| Application Structure | Advanced |
| Toolchain Folder Location | C:\Users\vetto\Documents\validation_fan\ |
| Toolchain / IDE | EWARM |
| Min Version | V8.32 |

Browse

☐ Do not generate the main()

☐ Generate Under Root

Linker Settings

| | |
|---|---|
| Minimum Heap Size | 0x2000 |
| Minimum Stack Size | 0x800 |

Thread-safe Settings

Cortex-M4NS

☐ Enable multi-threaded support

| | |
|---|---|
| Thread-safe Locking Strategy | Default – Mapping suitable strategy depending on RTOS selection. |

**Figure 33**: Project Manager view

- **Tools View**: from the tools view, the developer has a recap of all the settings made up till now, with the full sequence of operations displayed in a loop. This view is also used to calculate approximately how much current the microcontroller, with the selected peripherals and settings, would consume.

The view lets the developer set parameters such as supply voltage, clock frequency, run/sleep/standby mode, RAM voltage, enabled peripherals, etc. In lower-power applications, the microcontroller will often sleep most of the time, only waking up periodically to check for events or when an interrupt occurs. This means that the microcontroller most likely has at least two modes with very different parameter values. An example is shown in figure 34.

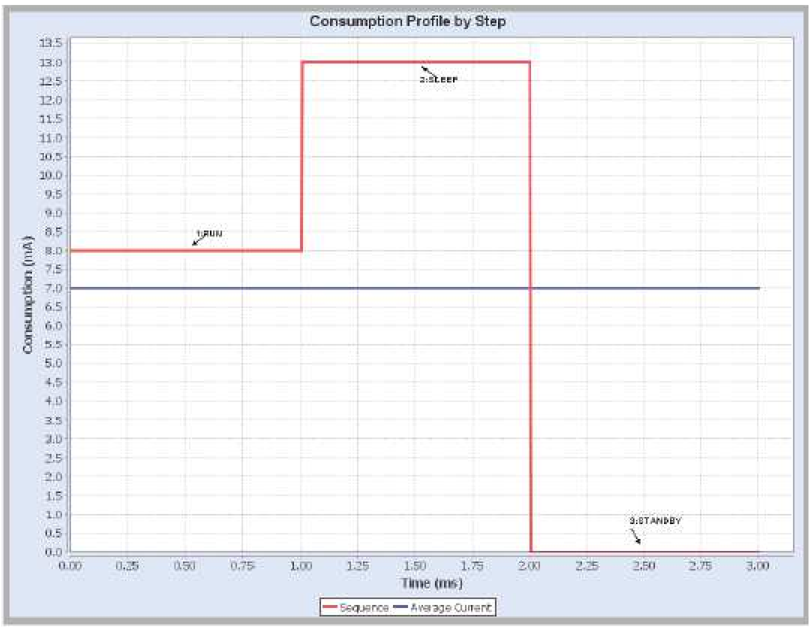


**Figure 34**: Power consumption example

In addition, the most regular batteries can be selected from a list and the approximate battery life can be estimated for a full charge.

When everything is set it is sufficient to click on "Generate code" and the tool will generate the custom firmware for the selected board. Once the process is complete the selected IDE will open with the C project ready to be compiled and loaded onto the device.

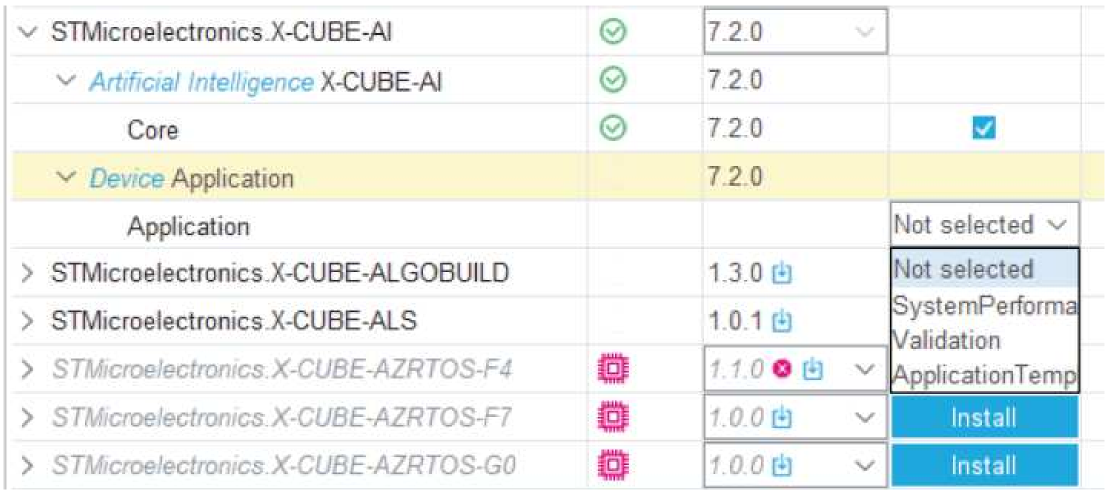
## C) X-CUBE-AI: Porting of a Neural Network

In *STM32CubeMX* it is possible to include a neural network in the generated firmware, it will be included in the project as a library. *X-CUBE-AI* is the official software expansion for AI embedded portings on *STM32* devices provided by *STMicroelectronics*.

In this appendix, the step-by-step process to add a neural network in *STM32CubeMX* is described.

1. In the Python environment where the network was designed and trained save locally the network in a compatible format, i.e. Keras, TFLite, ONNX.
2. Open *STM32CubeMX* and select the desired board to start a new project.

3. From the Pinout & Configuration view select “Software Packs” and “Select components”.
4. From this view find the “STMicroelectronics.X-CUBE-AI” voice. Install it and select the Artificial Intelligence core and the desired application as in figure 35.

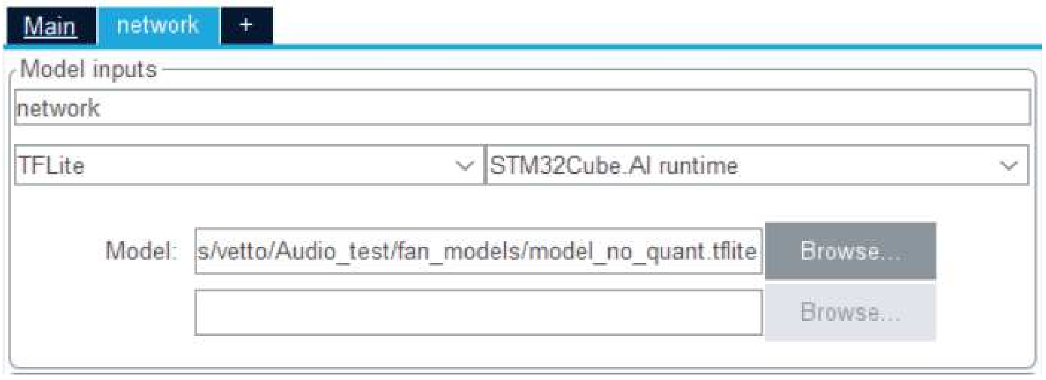


| | | | |
|---|---|---|---|
| STMicroelectronics.X-CUBE-AI | ✓ | 7.2.0 | |
| *Artificial Intelligence* X-CUBE-AI | ✓ | 7.2.0 | |
| Core | ✓ | 7.2.0 | ☑ |
| *Device* Application | | 7.2.0 | |
| Application | | | Not selected |
| STMicroelectronics.X-CUBE-ALGOBUILD | | 1.3.0 | Not selected |
| STMicroelectronics.X-CUBE-ALS | | 1.0.1 | SystemPerforma<br>Validation |
| *STMicroelectronics.X-CUBE-AZRTOS-F4* | | *1.1.0* | ApplicationTemp |
| *STMicroelectronics.X-CUBE-AZRTOS-F7* | | *1.0.0* | Install |
| *STMicroelectronics.X-CUBE-AZRTOS-G0* | | *1.0.0* | Install |

**Figure 35**: Selection of X-CUBE-AI software expansion

The possible applications are three. *SystemPerformance* will generate the code to allow the accurate measurement of the neural network inference CPU load and memory usage. *Validation* provides a setup to check the accuracy of the embedded model compared to the original. *ApplicationTemplate* provides a general template to further customize the pipeline from the C code also allowing multi-network support. If no Application is selected the tool will just create the library with the included neural network leaving the rest of the firmware untouched.

5. In this following view, like in figure 36, select the format of the network and “STM32Cube.AI runtime” for the conversion, then provide the path to the model.



Model inputs

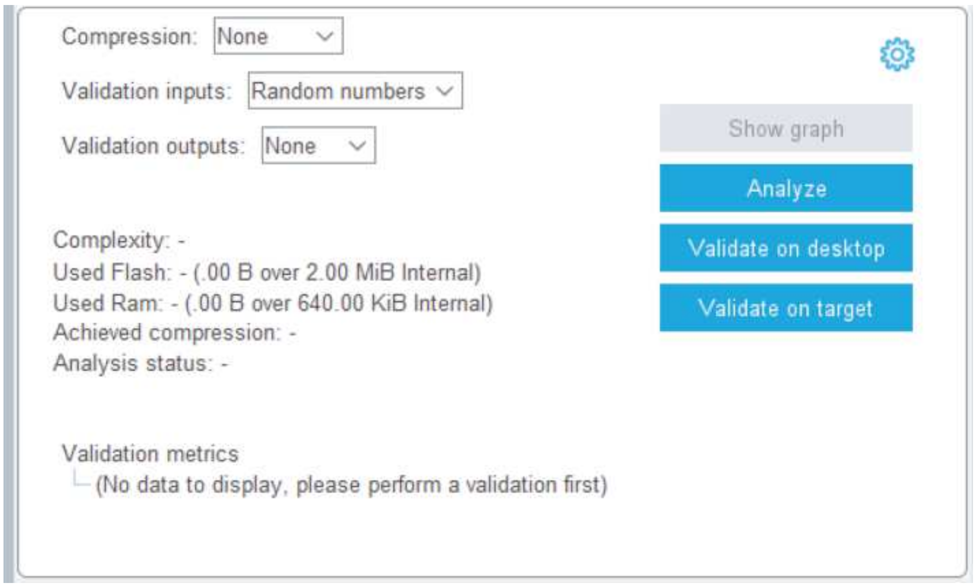
network

TFLite | STM32Cube.AI runtime

Model: s/vetto/Audio_test/fan_models/model_no_quant.tflite

**Figure 36**: Model inputs

6. In the final window, in figure 37, click on “Analyze” leaving everything untouched. After the validation process, the tool will provide information about the complexity of the model, in terms of used Flash and RAM. Depending on the results we can change the compression level to lower the requirements. Three types of compression are available: low, medium, and high. It is recommended to proceed from low to high and analyze the network every time to check how the requirements change. Invalid network configurations will be marked in red.


Compression: None
Validation inputs: Random numbers
Validation outputs: None

Complexity: -
Used Flash: - (.00 B over 2.00 MiB Internal)
Used Ram: - (.00 B over 640.00 KiB Internal)
Achieved compression: -
Analysis status: -

Validation metrics
(No data to display, please perform a validation first)

**Figure 37**: Network conversion

After obtaining a compatible network we can click on “Validate on desktop” to perform inference with random numbers on the network and check that everything works with the original input size. Validation can be performed also with custom data proving on validation inputs and outputs appropriate CSV data, they can be exported using Numpy from Python. With the validation process, it is possible to verify on the output txt file the differences in performance between the original and the converted embedded model.

With “Validation on target” the validation process can be performed directly on the board if properly connected via USB, in order to see the exact inference time needed per sample in the embedded environment.

After these passages, we can proceed with the standard configuration in *STM32CubeMX* and set everything else freely.

By clicking on “Generate code” and opening the generated project we can see that an extra folder called “X-CUBE-AI” will be created inside the project. The folder contains the library needed to access inference with the converted network, and inside the file "network_data_params.c" all the converted parameters of the network are stored.

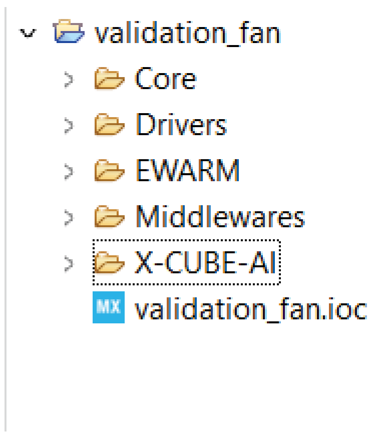


**Figure 38**: Folders of the generated project in STM32CubeIDE

# References

[1] *Acoustic emission signatures of arcs and sparks*. (1980). IEEEXplore. https://doi.org/10.1109/icei.1980.7470905

[2] Bajaj, V. (2020, August 8). *Unsupervised Learning For Anomaly Detection | by Vardaan Bajaj*. Towards Data Science. Retrieved September 15, 2022, from https://towardsdatascience.com/unsupervised-learning-for-anomaly-detection-44c55a96b8c1

[3] Bengio, Y., Courville, A., & Goodfellow, I. (2016). *Deep Learning*. MIT Press.

[4] Brian, M., Raffel, C., Liang, D., Ellis, D. P., McVicar, M., Battenberg, E., & Nieto, O. (2015). *librosa: Audio and music signal analysis in python* (0.9.1).

[5] Brownlee, J. (2019, December 23). *A Gentle Introduction to Imbalanced Classification*. Machine Learning Mastery. Retrieved September 15, 2022, from https://machinelearningmastery.com/what-is-imbalanced-classification/

[6] C3 AI. (2022). *Explainability*. C3.ai. https://c3.ai/glossary/machine-learning/explainability/

[7] Chan, T. (2022, January 13). *Predictive Maintenance: Using Sound as the Primary Indicator*. Groundup.ai.

[8] Chollet, F. (2015). *Keras*. GitHub. https://github.com/fchollet/keras

[9] Dang, T. T., Ngan, H. Y.T., & Liu, W. (2015). *Distance-based k-nearest neighbors outlier detection method in large-scale traffic data*. 2015 IEEE International Conference on Digital Signal Processing (DSP).

[10] Eclipse Foundation. (2022). *Eclipse*. Eclipse. https://www.eclipse.org/

[11] Evangeline, P., & Raj, P. (Eds.). (2020). *The Digital Twin Paradigm for Smarter Systems and Environments: The Industry Use Cases*. Elsevier Science.

https://www.sciencedirect.com/topics/computer-science/multilayer-perceptron#:~:text=The%20equation%20w1x,data%20that%20are%20linearly%20separable.

[12] Google. (2022, July 18). *Classification: ROC Curve and AUC | Machine Learning*. Google Developers. Retrieved September 20, 2022, from https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc

[13] Google. (2022, July 22). *TensorFlow Lite for Microcontrollers*. TensorFlow. Retrieved September 20, 2022, from https://www.tensorflow.org/lite/microcontrollers

[14] Khandelwal, R. (2021, January 20). *Anomaly Detection using Autoencoders | by Renu Khandelwal*. Towards Data Science. Retrieved July 6, 2022, from https://towardsdatascience.com/anomaly-detection-using-autoencoders-5b032178a1ea

[15] Kingma, D. P., & Ba, J. L. (2015). *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*. https://arxiv.org/pdf/1412.6980.pdf

[16] Kluyver, T., Regan-Kelley, B., Perez, F., Granger, B., Bussonnier, M., Frederic, J., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., & Willing, C. (2016). *Jupyter Notebooks -- a publishing format for reproducible computational workflows*.

[17] Koizumi, Y., Kawaguchi, Y., Imoto, K., Nakamura, T., Nikaido, Y., Tanabe, R., Purohit, H., Suefusa, K., Endo, T., Yasuda, M., & Harada, N. (2020, March 2). *Unsupervised Detection of Anomalous Sounds for Machine Condition Monitoring*. DCASE

[18] Liu, F. T., Ting, K. M., & Zhou, Z.-H. (2009). *Isolation Forest*.

[19] MathWorks. (2022). *Hann (Hanning) window*. MathWorks. https://it.mathworks.com/help/signal/ref/hann.html

[20] Mohan, A. (2019). *K- Fold Cross Validation For Parameter Tuning | by Arun Mohan*. DataDrivenInvestor. Retrieved September 17, 2022, from

https://medium.datadriveninvestor.com/k-fold-cross-validation-for-parameter-tuning-75b6cb3214f

[21] Muller, R., Illium, S., Ritz, F., & Schmid, K. (2021). *Analysis of Feature Representations for Anomalous Sound Detection*. Proceedings of the 13th International Conference on Agents and Artificial Intelligence.

[22] Nahrstedt, K. (2012). *Digital Audio Representation*.

[23] Purohi, H., Tanabe, R., Ichige, K., Endo, T., Nikaido, Y., Suefusa, K., & Kawaguchi, Y. (2019). *MIMII Dataset: Sound Dataset for Malfunctioning Industrial Machine Investigation and Inspection*. https://arxiv.org/abs/1909.09347

[24] Rajan, S. (2021, May 20). *Anomaly Detection using AutoEncoders | A Walk-Through in Python*. Analytics Vidhya. Retrieved July 23, 2022, from https://www.analyticsvidhya.com/blog/2021/05/anomaly-detection-using-autoencoders-a-walk-through-in-python/

[25] Rothmann, D. (2018). *What's wrong with CNNs and spectrograms for audio processing?* towardsdatascience. https://towardsdatascience.com/whats-wrong-with-spectrograms-and-cnns-for-audio-processing-311377d7ccd

[26] Sahay, M. (2020, June 6). *Neural Networks and the Universal Approximation Theorem | by Milind Sahay*. Towards Data Science. Retrieved September 6, 2022, from https://towardsdatascience.com/neural-networks-and-the-universal-approximation-theorem-8a389a33d30a

[27] STMicroelectronics. (2021, January 8). *FP-SNS-DATALOG1 - STM32Cube High Speed Datalog function pack for STWIN evaluation kits*. STMicroelectronics

[28] STMicroelectronics. (2022). *STM32CubeIDE* (1.9.0).

[29] STMicroelectronics. (2022). *STM32CubeMX* (6.5.0).

[30] STMicroelectronics. (2022). *X-CUBE-AI* (7.2.0).

[31] Tagawa, Y., Maskeliunas, R., & Damaševiˇcius, R. (2021). *Acoustic Anomaly Detection of Mechanical Failures in Noisy Real-Life Factory Environments*. https://dcase.community/documents/workshop2019/proceedings/DCASE2019Workshop_Purohit_21.pdf

[32] 221e S.r.l. (2021). *Predictive maintenance – Data analysis for ***** lift motors* [221e S.r.l. intellectual property].