



Master's Degree in **Mechatronic Engineering**  
Commission of **Computer Engineering, Cinema and  
Mechatronics**

---

# Machine Learning Applications for PLC-Based Industrial Automation

Smart Industrial Solutions with the *Finder  
Opta<sup>TM</sup>* Programmable Logic Controller

---

A.Y. 2023/2024

October 2024

**Supervisors**

Prof. Andrea MURA  
Prof. Luigi MAZZA

**Candidate**

Riccardo MENNILLI

# Abstract

In recent years, efforts have been made towards the integration of smart digital technologies into traditional industrial processes, allowing for the collection of vast amounts of data across all aspects of the manufacturing cycle, in what has been referred to as the “Fourth Industrial Revolution”, or “Industry 4.0”. Machine learning tools, particularly neural networks, have been playing an increasingly relevant role within this context, thanks to their ability to handle complex multidimensional data, uncover hidden relationships extrapolating patterns, predict the future evolution of the process and thus aid decision-making, ultimately improving efficiency.

In this work, we aim to showcase how a next-generation programmable logic controller like the Finder Opta™ - which integrates traditional PLC features with a powerful processor, several connectivity options and access to the Arduino language and ecosystem - enables breakthrough applications within the “Industry 4.0” framework, even supporting small neural networks to handle on-device classification or regression tasks. While deploying a neural network to a resource- and memory-constrained device poses significant challenges, it can be very advantageous as it eliminates the need for a central cloud infrastructure, at least for simpler tasks, with many benefits in terms of latency, security and privacy. This trend, known as edge computing, is indeed a very active area of research.

The thesis is structured around three case studies, where we demonstrate how to address the hardware limitations while still deploying meaningful machine learning algorithms, and how the Arduino platform allows for straightforward software development, facilitating the transition towards intelligent manufacturing even for smaller entities. To lay the groundwork and explore the tools at our disposal, the first application involves training an MLP network to classify an input waveform and deploying such a model to the Opta™ PLC with TensorFlow Lite for Microcontrollers. Then, we transition to a more complex convolutional neural network to infer the rotational speed of two rolling bearings, starting from the spectrogram of an audio recording of the test bench in operation; this represents a preliminary study in the field of anomalous sound detection for predictive maintenance. Shifting focus away from machine learning for the last application, the Opta™ is tasked with controlling a wave energy converter, exploiting its internet connection capabilities to analyze forecast data about wave height, thereby protecting the system from damage due to excessive stress on the floating device.

# Contents

List of abbreviations	1
<b>1 Introduction and Objectives</b>	<b>3</b>
1.1 Hints about Deep Neural Networks	5
1.1.1 Training a Neural Network - Supervised Learning	8
1.1.2 Convolutional Neural Networks	11
1.2 Machine Learning on Microcontrollers	15
1.2.1 TensorFlow	16
1.2.2 TensorFlow Lite for Microcontrollers	17
<b>2 Finder Opta™: a next-generation "Programmable Logic Relay"</b>	<b>19</b>
2.1 Programmable Logic Controllers	19
2.1.1 PLC Programming Languages	20
2.2 Arduino Ecosystem	22
2.3 Finder Opta™ - Technical Characteristics	23
<b>3 Case Study I: Waveform Identification Model</b>	<b>27</b>
3.1 Test Bench Setup and Objectives	27
3.1.1 Requirements	27
3.1.2 Test Bench	28
3.2 Workflow Overview	29
3.3 Building a Neural Network	30
3.3.1 Training Data Collection	30
3.3.2 Machine Learning Model Configuration	36
3.3.3 Model Training, Performance Evaluation and Tuning	40
3.3.4 Final Model Testing	46
3.4 Model Deployment to the Finder Opta™	47
3.4.1 TensorFlow Lite Conversion	47
3.4.2 Arduino Application	50
3.4.3 Preliminary Application Unit Testing	57
3.5 Overall System Testing	58
3.6 Conclusions	61

<b>4</b>	<b>Case Study II: Towards Predictive Maintenance</b>	<b>63</b>
4.1	Machine Learning applied to Predictive Maintenance . . . . .	64
4.1.1	Anomalous Sound Detection . . . . .	66
4.2	Edge Computing and Machine Learning . . . . .	66
4.3	Test Bench Setup and Objectives . . . . .	67
4.3.1	Test Bench . . . . .	67
4.3.2	Requirements . . . . .	70
4.4	Workflow . . . . .	71
4.5	Building a Neural Network . . . . .	72
4.5.1	Training Data Collection . . . . .	72
4.5.2	Machine Learning Model Configuration . . . . .	79
4.5.3	Model Training and Performance Evaluation . . . . .	82
4.6	Model Deployment to the Finder Opta™ . . . . .	84
4.6.1	TensorFlow Lite Conversion . . . . .	85
4.6.2	Arduino Application . . . . .	86
4.6.3	Preliminary Application Unit Testing . . . . .	92
4.7	Overall System Testing and Conclusions . . . . .	94
<b>5</b>	<b>Case Study III: Application to a Wave Energy Generator</b>	<b>97</b>
5.1	Experimental Setup . . . . .	97
5.2	Internet Connectivity and Infrastructure . . . . .	99
5.2.1	Workflow and Requirements . . . . .	99
5.2.2	Arduino Cloud Connection . . . . .	100
5.2.3	Mbed OS Parallel Thread - LEDs Management . . . . .	102
5.2.4	Unit Testing . . . . .	104
5.3	Emergency Operations . . . . .	106
5.3.1	Requirements . . . . .	106
5.3.2	Marine Weather Forecast . . . . .	107
5.3.3	State Machine . . . . .	113
5.3.4	Unit Testing . . . . .	117
5.4	Human-Machine Interface . . . . .	120
5.4.1	Requirements . . . . .	121
5.4.2	Arduino Cloud Dashboard . . . . .	121
5.4.3	Data Logging to Google Sheets . . . . .	123
5.4.4	Unit Testing . . . . .	127
5.5	Conclusions and Future Developments . . . . .	128
<b>6</b>	<b>Conclusions</b>	<b>129</b>

<b>Bibliography</b>	<b>131</b>
<b>Appendices</b>	<b>141</b>
Appendix A Finder Opta™	141
Appendix B Case Study I	143
Appendix C Case Study II	153
Appendix D Case Study III	167

---

## List of abbreviations

Abbreviation	Meaning
ADC	Analog-to-Digital Converter
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
CNN	Convolutional Neural Network
FFT	Fast Fourier Transform
FPU	Floating-Point Unit
GPU	Graphics Processing Unit
HMI	Human Machine Interface
IDE	Integrated Development Environment
I/O	Input/Output
IoT	Internet of Things
ISR	Interrupt Service Routine
IT	Information Technology
MAE	Mean Absolute Error
ML	Machine Learning
MLP	MultiLayer Perceptron
MSE	Mean Squared Error
NN	Neural Network
Op-Amp	Operational Amplifier
OTA	Over-The-Air
PLC	Programmable Logic Controller
PC	Personal Computer
RTOS	Real-Time Operating System
RAM	Random Access Memory
RUL	Remaining Useful Life
SGD	Stochastic Gradient Descent
TFLite	TensorFlow Lite



# Chapter 1

## Introduction and Objectives

In an era where Artificial Intelligence (AI) and Machine Learning (ML) algorithms are becoming ever more pervasive in essentially all areas of technology and research [1], industrial automation and maintenance can be no exception. Recent advances in deep learning have enabled breakthrough capabilities for the industrial sector, such as the power to gain insight about all aspects of the manufacturing process, recognizing intricate patterns and complex trends from seemingly unrelated sources. Such insight can be an invaluable asset for decision-making, optimizing uptime, efficiency and productivity [2]. Industrial maintenance can also benefit greatly from predictive practices based on machine learning and "big data", as these can detect a machine's earliest signs of failure from the analysis of historic tendencies, alongside real-time sensor data from the plant, minimizing downtime and allowing for repairs to be carried out only when necessary [3].

Though AI has accelerated progress considerably, it can be regarded as part of a larger "Fourth Industrial Revolution", or "Industry 4.0" [4], a trend towards the integration of *smart*, digital devices and advanced data analysis practices into traditional industrial processes. The Industry 4.0 transition was facilitated by the dramatic reduction in costs of silicon chips, and the breakthrough of the internet, with the associated spread of connected devices and telecommunication technologies [5]: only a deeply interconnected network of sensing devices and actuators, managed by a powerful central infrastructure, can handle the complexity and scale of modern manufacturing, and allow for effective data analysis models to operate in real-time. Contemporary Industry 4.0 practices increasingly emphasize wireless connectivity, promoting direct access to the internet or to a shared network for each device involved in the process. This allows for control and data collection to occur remotely, minimizing human effort, overcoming physical limitations, and enhancing scalability. Known as the "Internet of Things" (IoT) [6], this trend fits into the more general framework of "Cloud Computing", where a powerful centralized infrastructure made of one or multiple servers is in charge of resource-intensive operations [7].



While the benefits of cloud computing are undeniable, and often the only possible choice of architecture due to the ever-growing complexity in process data, it does present some drawbacks. Specifically, in terms of bandwidth and latency, since data needs to be transferred from the terminal devices to the cloud and back, and in terms of security and privacy [8].

The work presented in this thesis, which touches on all aspects mentioned thus far, is meant to address the disadvantages of cloud computing with a change of paradigm. Instead of relying on a central infrastructure, all computing operations are devolved to the edge devices, emphasizing data processing closer to the source, limiting slow transfers and reducing privacy and security concerns, in what is referred to as "Edge Computing" [9].

Indeed, the aim of the thesis is to showcase how Deep Learning algorithms can be deployed effectively to industrial edge devices, such as a Programmable Logic Controllers (PLCs), enabling many of the advantages of the Industry 4.0 revolution, including predictive maintenance and intelligent manufacturing, with on-device solutions, despite the limited computing resources that microcontrollers can provide. The latter is the most critical aspect to investigate, as the constraints on memory and processing power place a strict limit on the size of ML algorithms and on their operations [10]. Special-purpose techniques must be utilized to optimize the model for execution on embedded platforms, which often lack most dependencies available on desktop or mobile. The workflow and the tools to apply are introduced later in the chapter, in section 1.2.

Employed throughout this work is a next-generation PLC device, the Finder Opta™. Jointly developed by Finder, a leading Italian manufacturer of electromechanical components, and Arduino, the company behind the renowned electronic prototyping platform, it offers a powerful Arm Cortex-M processor, alongside several connectivity options, precisely targeting novel IoT and machine learning applications. In addition to traditional PLC features, such as a rugged construction suitable for industrial environments, compatibility with industrial voltages and currents, and support for the IEC 61131-3 PLC programming languages, it is built leveraging the Arduino ecosystem, with access to its countless pre-built libraries and services, as well as the Arduino programming language. Not only does this guarantee a high level of flexibility at limited costs, but also makes innovation more accessible, even for smaller entities, perhaps with no highly specialized IT personnel. For a more in-depth analysis of the device, refer to chapter 2.

The main content of the thesis is represented by three case studies:

- **Case Study I** (chapter 3)  
An introductory application to explore the machine learning workflow on the Finder Opta™ and the available tools, laying the groundwork for more

advanced implementations. The Opta™ is tasked with identifying an input waveform among three possible options (*sinusoidal, triangular, square*), using a basic neural network model trained on a dataset collected by the PLC itself.

- **Case Study II** (chapter 4)

A preliminary exercise in view of further research in the field of predictive maintenance on mechanical machinery. On the basis of an audio signal of a test bench in operation, recorded via inexpensive electret microphones, the Opta's role is to infer the rotational speed of the main shaft and support bearings. This is achieved on-device thanks to advanced signal processing, involving the FFT algorithm, and a more complex convolutional neural network architecture, which pushes the PLC hardware to its limits. Future developments of the case study include an expansion to anomalous sound detection, to monitor the health of the mechanical parts, precisely with predictive maintenance in mind.

- **Case Study III** (chapter 5)

Shifting emphasis away from deep learning, the last application is meant to showcase the potential of Industry 4.0 and IoT practices when the focus is on remote control and monitoring. The target system is an offshore wave energy generator, where the Opta's connectivity capabilities and integration with online services such as Arduino Cloud prove extremely useful for monitoring an otherwise inaccessible device. In addition to ensuring efficient operations, the Opta™ is in charge of the system's safety, using predictive methods based on the marine weather forecasts for the working site.

Before delving deep into the analysis of each case study, it is necessary to provide some general hints about the machine learning algorithms to apply, specifically deep neural networks. The following section introduces the key terminology and concepts, and aims to provide a basic understanding of the deep learning workflow. However, the rigorous mathematical treatment that the topic would warrant is beyond the scope of this work.

## 1.1 Hints about Deep Neural Networks

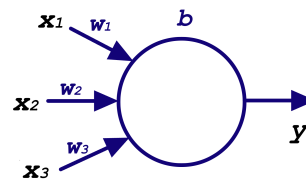
Among all machine learning algorithms, deep Artificial Neural Networks (ANNs) have experienced the most rapid surge in popularity, starting from the early 2010s, owing to their exceptional performance in pattern and image recognition, as well as the recent breakthroughs in natural language processing [11]. Being the archetypal AI algorithm, ANNs are the natural choice to demonstrate the potential of machine learning practices on industrial edge devices. Be advised that, throughout this work, the term Machine Learning (ML) is used synonymously with Artificial Neural Networks, as these are the only ML algorithms treated in the case studies.

Feedforward neural networks are computational structures inspired by the human brain, composed of multiple interconnected nodes (Fig. 1.2), that typically process input data for classification or regression problems. Each node, or *neuron*, is characterized by a value, its *activation*, that loosely mimics how biological cells fire in the brain: information propagates through the network by selectively activating certain neurons, depending on the features that each node is meant to identify.

Mathematically, they provide a means to model complex phenomena, and to represent any arbitrary nonlinear relationship between the inputs and the outputs. The nodes, typically organized in layers, are computational units that apply a weighted sum to the inputs, modulated by a nonlinear activation function  $\sigma$ . This is how the forward propagation of information occurs within a network.

$$y = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

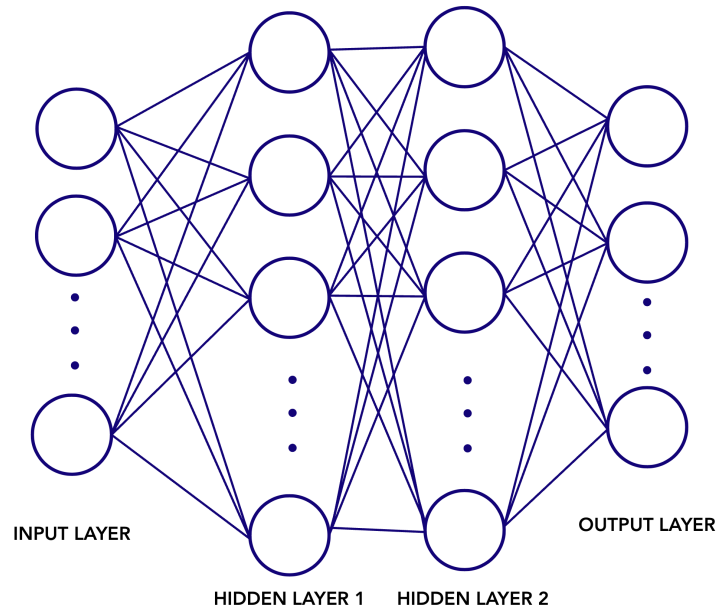
where  $\mathbf{x} \in \mathbb{R}^n$  is the input vector,  
 $y \in \mathbb{R}$  the output,  $\mathbf{w} \in \mathbb{R}^n$  a vector  
of weights  $b \in \mathbb{R}$  a scalar bias, and  
 $\sigma : \mathbb{R} \rightarrow \mathbb{R}$  the activation function



**Figure 1.1:** Neuron Diagram

Weights and biases are the fundamental parameters that determine how signals travel through the network, and they are the key behind the power and generality of ANNs. Traditionally, pattern recognition or classification algorithms required the features to identify to be hand-crafted - with each weight and bias term selected manually -, a time-consuming and inefficient process, that lacks scalability. In contrast, within the framework of **supervised learning**, these parameters are *learned* autonomously by the network in a process called *training*, thus overcoming the bottleneck. Training allows for an *optimal* set of parameters to be found for the task at hand, and it involves feeding the network numerous examples of the items to classify (millions, in our "big data" era), with the aim of minimizing a relevant error metric that measures how the model performs. In fact, neural networks and learnable weights have existed since the 1950s [12], but hardware and software limitations prevented the dramatic advances of the field that are seen today.

The foundational deep architecture consists of a number of fully-connected layers, each comprising several neurons. This is referred to as a Multilayer Perceptron (MLP), since a single neuron unit is also known as a *perceptron*. More hidden intermediate layers result in "deeper" networks, characterized by a greater number of parameters but also an increased capacity to model even complex relationships, and detect high-level patterns within the input data.



**Figure 1.2:** Example of a Multilayer Perceptron Structure (2 hidden layers)

Denoting  $n_l$  the number of layers, and labelling them  $L_1, \dots, L_{n_l}$ , the parameters of the network are  $(W, b)$ , where  $W_{ij}^{(l)}$  is the weight of the connection between unit  $j$  in layer  $(l)$  and unit  $i$  in layer  $(l+1)$ , and  $b_i^{(l)}$  the bias associated with unit  $i$  in layer  $(l+1)$ . If layer  $(l)$  features  $s_l$  nodes, then  $W^{(l)} \in \mathbb{R}^{s_{l+1}, s_l}$  and  $b^{(l)} \in \mathbb{R}^{s_{l+1}}$ . Using this notation, the activations  $a^{(l+1)}$  can be computed with the following recursion, a generalization of the single perceptron:

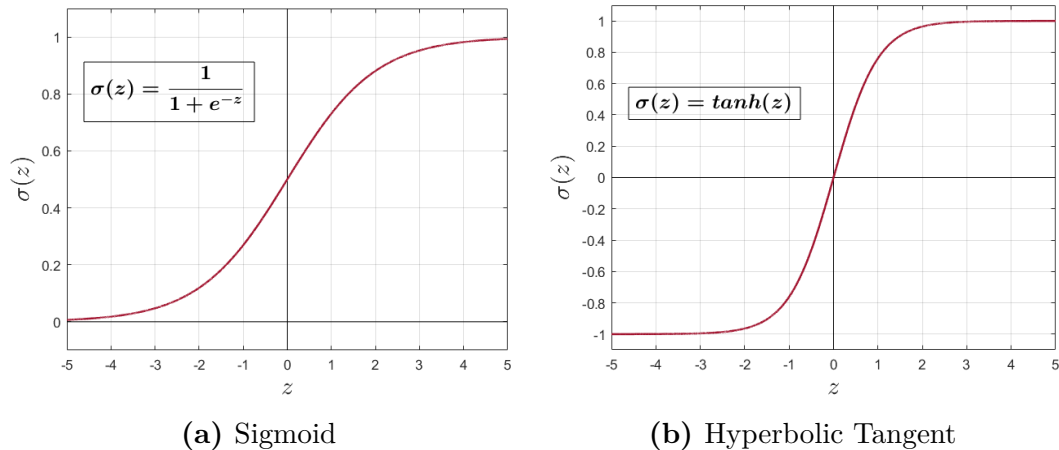
$$a^{(l+1)} = \sigma \left( W^{(l)} a^{(l)} + b^{(l)} \right)$$

### Universal Approximation and Activation Functions

Before discussing the training process, it is worth highlighting how rigorous mathematical results have proven the ability of feedforward neural networks to approximate arbitrarily well wide classes of functions [13]. However, without a nonlinear function  $\sigma$  modulating the activation of each perceptron, the entire network would reduce to a linear regression [14]. Linearity would severely limit its capacity to identify complex features, as a structure of arbitrary depth would be equivalent to a single neuron.

A traditional activation function is the sigmoid (Fig. 1.3a), which maps the output of the weighted sum to the interval  $[0, 1]$ . Essentially, with the sigmoid, each neuron

applies the logistic regression [15], which is ideal for binary classification problems, and helps prevent quantities from exploding, causing numerical instability.



**Figure 1.3:** Common Activation Functions

Other common choices include the hyperbolic tangent (Fig. 1.3b), which centers the data around 0, the softmax, and the ReLU (section 3.3.2).

### 1.1.1 Training a Neural Network - Supervised Learning

The aim of the training process within the framework of supervised learning is to find a set of parameters  $(W, b)$  that minimizes the error between the model's predictions and the known training examples, according to a given metric known as *cost function*. Ultimately, the true goal is for the model to generalize effectively to *unseen* inputs, preventing underfitting or overfitting of the training data.

#### Cost Function

Given a single training example  $(x^i, y^i)$ , different criterions can be employed to measure the mismatch (loss) between the expected output  $y^i$  and the predicted output  $\hat{y}^i = (a^{(n)})^i$ , corresponding to the activations in the output layer. Examples include the mean squared error (MSE), the mean absolute error (MAE), and cross-entropy [14]. More details about the various options will be provided when designing the network architecture for each case study.

Denoting this loss as  $\mathcal{L}(\hat{y}^i, y^i)$ , the overall cost function to account for all of the  $m$  examples in the training dataset can be computed as the average:

$$J(W, b) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^i, y^i)$$

Therefore, training a neural network translates into solving the following optimization problem, where the optimal set of weights  $(W^*, b^*)$  is the one minimizing the cost function  $J$ :

$$(W^*, b^*) = \arg \min_{(W, b)} J(W, b)$$

Although more complex versions are routinely found in the literature, for a more robust training process (i.e. regularization terms can be included to reduce overfitting) [16], this foundation is sufficient for the purposes of this work.

## Optimization and Backpropagation

The optimization algorithms to minimize cost function  $J(W, b)$  are largely based on gradient descent, a first-order iterative descent method [17]. At each iteration, the algorithm updates a candidate point of minimum following a descent direction, and terminates when the predefined exit criterion is satisfied. Typically, the path is determined by the negative gradient of the cost function, which coincides with the direction of steepest descent: given a cost function  $f_0 : \mathbb{R}^n \rightarrow \mathbb{R}$ , a generic point  $x_k \in \text{dom} f_0$ , and a generic direction  $v_k \in \mathbb{R}^n$ , the first-order Taylor expansion states that

$$\begin{aligned} f_0(x_k + \alpha v_k) &\approx f_0(x_k) + \alpha \nabla f_0(x_k)^\top v_k, \text{ for } \alpha \rightarrow 0 \\ \Rightarrow \lim_{\alpha \rightarrow 0} \frac{f_0(x_k + \alpha v_k) - f_0(x_k)}{\alpha} &= \nabla f_0(x_k)^\top v_k \end{aligned}$$

Therefore,  $\nabla f_0(x_k)^\top v_k$  is the local rate of variation of  $f_0$ . From the Cauchy-Schwartz inequality:

$$- \|\nabla f_0(x_k)\|_2 \|v_k\|_2 \leq \nabla f_0(x_k)^\top v_k \leq \|\nabla f_0(x_k)\|_2 \|v_k\|_2$$

It follows that the local rate of variation is minimized, i.e.  $v_k$  is the steepest descent direction, if  $v_k = -\nabla f_0(x_k) / \|\nabla f_0(x_k)\|_2$ .

The basic steps of gradient descent are reported in algorithm 1.

---

**Algorithm 1** Basic Gradient Descent

---

**Require:**  $f_0 : \mathbb{R}^n \rightarrow \mathbb{R}$  differentiable,  $x_0 \in \text{dom} f_0$

$k \leftarrow 0$

**while** stopping criterion is NOT met **do**

    determine a descent direction:  $v_k = -\nabla f_0(x_k) / \|\nabla f_0(x_k)\|_2$

    determine the step length  $\alpha > 0$

    update  $x_{k+1} = x_k + \alpha v_k$

$k \leftarrow k + 1$

**end while**

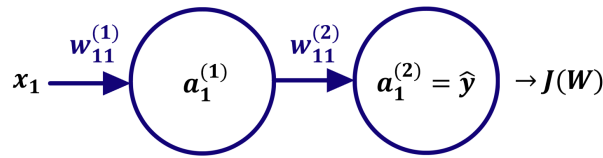
---

In the context of neural network training, one iteration of gradient descent updates parameters  $W$  and  $b$  as follows:

$$W_{ij}^{(l)} \leftarrow W_{ij}^{(l)} - \alpha \frac{\partial J(W, b)}{\partial W_{ij}^{(l)}} \quad b_i^{(l)} \leftarrow b_i^{(l)} - \alpha \frac{\partial J(W, b)}{\partial b_i^{(l)}}$$

The step length  $\alpha$  is also referred to as *learning rate*, and it represents a crucial hyperparameter to tune for an effective training process.

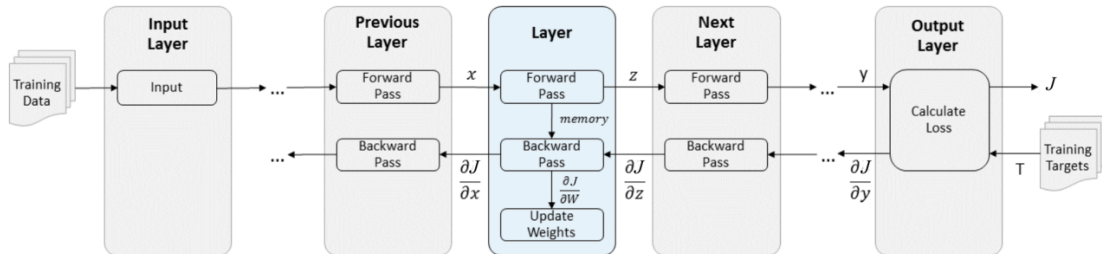
What makes gradient descent very powerful is an efficient recursive algorithm, known as **backpropagation**, to compute the gradient of the cost function  $J(W, b)$ , exploiting the chain rule of derivatives [18].



**Figure 1.4:** Neural Network with 1 input, 1 hidden neuron and 1 output

Consider a network with a single hidden layer made of one neuron (Fig. 1.4): backpropagation starts from the output  $\hat{y}$ , and computes each component of the gradient with the chain rule of derivatives, backtracking to the input. This recursive approach is numerically efficient and scalable to a huge number of parameters [18]. Intuitively, the derivatives quantify the effect of each single weight on the final loss.

$$W = \begin{bmatrix} w_{11}^{(1)} \\ w_{11}^{(2)} \end{bmatrix} \rightarrow \begin{aligned} \frac{\partial J(W)}{\partial w_{11}^{(2)}} &= \frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_{11}^{(2)}} \\ \frac{\partial J(W)}{\partial w_{11}^{(1)}} &= \frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_{11}^{(1)}} = \frac{\partial J(W)}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial w_{11}^{(1)}} \end{aligned}$$



**Figure 1.5:** Backpropagation Algorithm

In neural networks, the cost  $J(W, b)$  is highly complex, since it is a nonlinear and non-convex function of several thousands if not millions of parameters. Therefore, finding the global minimum is typically infeasible, as non-convex gradient descent is susceptible to local optima. Nonetheless, this approach still yields satisfactory results in the majority of cases.

Since the cost function is defined as the average of the loss, calculated over the entire training dataset, computing its gradient exactly at each iteration can be extremely computationally expensive, if a large number of training examples is to be used. Therefore, to improve performance, the gradient is often estimated over a subset of randomly selected training examples, in what is essentially a stochastic approximation, and thus referred to as *stochastic gradient descent* (SGD) [19].

Strictly speaking, stochastic gradient descent computes the gradient with a single training example. Instead, when a subset of the data is employed (i.e. a *mini-batch*), the algorithm takes the name of *mini-batch stochastic gradient descent*, which also takes better advantage of the parallel computational resources offered by modern GPUs. This is the norm for training neural networks nowadays, with the batch size tuned as a hyperparameter to strike a balance between training speed and accuracy [20].

Countless optimization techniques that build on these basic ideas have been proposed in the literature, with different approaches suited to different problems. Some of the more advanced algorithms, which have proven successful across a wide range of applications, are *Adam* [21], *Adadelta* [22], *Adagrad* [23], and *RMSProp* [24]. All of these rely on the idea of adaptive learning rates, to help convergence in complex optimization landscapes. In particular, *Adam* and *RMSProp* will be explored in chapter 3 and 4.

Neural network training is a complex field of research, that cannot possibly be exhausted in just a few pages. For the sake of brevity, many important topics have been omitted from this discussion, such as learning rate decay, regularization, normalization of the parameters, the problem of exploding and vanishing gradients, general hyperparameter tuning, etc.

## 1.1.2 Convolutional Neural Networks

Starting from the basic multilayer perceptron, more and more complex neural network architectures have been developed over the years, to expand the range of applicability and explore new frontiers for AI.

Nowadays, one of the most prominent fields of AI and deep learning research is *computer vision*, where the aim is for a machine to recognize objects and patterns within an image or a video [25]. However, it was discovered that fully-connected



networks are not suited for pattern recognition, since all the input features are treated independently, thus losing any spatial information. Moreover, as the input images become larger and more complex, the number of parameters would grow exponentially, resulting in huge - and inefficient - networks.

A different type of feedforward neural network, *convolutional neural networks* (CNN), proposed in the 1980s by notable computer scientists like K. Fukushima (inventor of the "neocognitron", inspired by the anatomy of the visual cortex [26]) and Y. LeCun (the first to apply backpropagation to convolutional layers [27]), turned out to address all of the issues above, and be very effective at picking up visual cues in an input image, extracting the most useful features.

To a computer, images are just matrices of numbers, or "pixels": the goal is for the model to learn features with a hierarchical approach, moving from low-level details like the edges of an object or dark spots (a few pixels), to higher-level abstractions, making up larger portions of the image. If, for example, the problem was facial recognition, the idea is for the model to start from the edges of the face, recognizing darker or lighter areas, then identify features like the eyes, ears and nose, and finally move up to the complete facial structure.

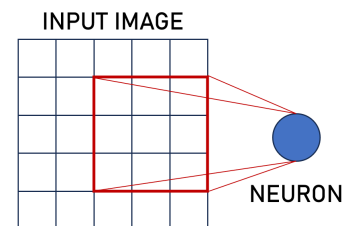
This is precisely the key idea behind CNNs, where each neuron in the first hidden layer attends to a small patch of the input array, using a sliding window to cover the entire image and define the connections between layers, preserving the spatial information.

For the neural network to extract the desired features from these patches, weights need to be applied, just like in a regular fully-connected layer: this can be carried out with the mathematical operation known as *convolution*, computed between the layers of the neural network and suitably tuned weight matrices (known as **filters**, or **kernels**). The result is a set of feature maps, indicating the presence or absence of the patterns the filters were designed to recognize.

Whereas in traditional image recognition algorithms the filters are manually adjusted to detect certain patterns, this is not a scalable approach nor does it provide acceptable results for more complex features. Instead, deep learning allows for the filters to be *learned* by the model during training, thanks to the backpropagation algorithm, yielding better and far more consistent results.

There are three main components to a CNN pipeline:

- **Convolutional Layers:** to extract the required features from the input image in hierarchical order, thanks to suitably tuned filters.
- **Pooling Layers:** to reduce the dimensionality of the layers, the deeper they are, in a down-sampling operation meant to reduce computational overhead



**Figure 1.6:** CNN Spatial Structure

and working against overfitting.

- **Fully-Connected Layers:** to process the recognized features for the regression or classification problem at hand.

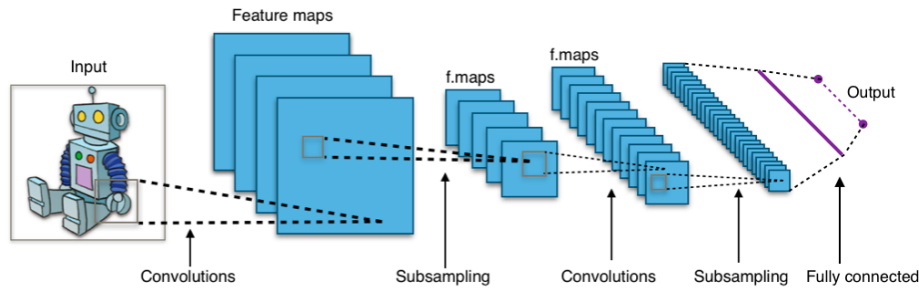


Figure 1.7: CNN Pipeline<sup>1</sup>

## Convolution

The convolution operation is the core of a CNN as it allows constructing meaningful feature maps from the input in an efficient manner, while maintaining spatial information.

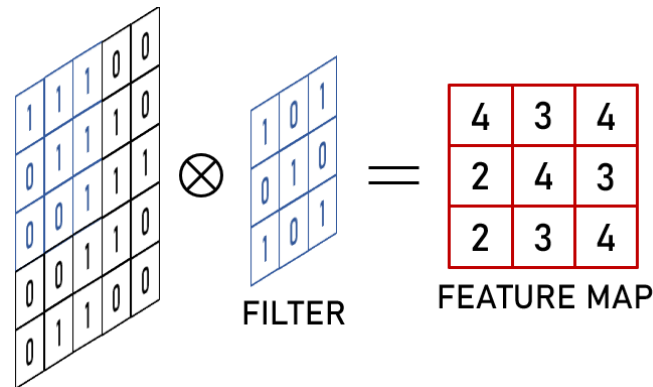
In analogy to how the activations of the neurons in regular fully-connected layers are calculated, the discrete convolution operation consists in a weighted sum of the inputs inside a patch, with the addition of a bias. For neuron  $(p, q)$  in the hidden layer, with a  $n \times m$  filter  $W$  (see Fig. 1.8):

$$\sum_{i=1}^n \sum_{j=1}^m w_{ij} x_{i+p, j+q} + b$$

The final activation is then obtained by applying a nonlinear activation function. However, it is crucial to remember that, unlike regular fully-connected layers, each neuron is only connected to its receptive field, i.e. a specific region of the previous layer, defining the spatial arrangement of the features.

Each convolutional layer is typically composed of several filters, giving it "depth", allowing to recognize several sets of features. Each "slice" of this convolutional layer correspond to a specific feature map, as illustrated by Fig. 1.7.

<sup>1</sup>Aphex34, CC BY-SA 4.0 <https://creativecommons.org/licenses/by-sa/4.0>, via Wikimedia Commons



**Figure 1.8:** Convolution between a  $5 \times 5$  image and a  $3 \times 3$  filter, designed to recognize the "X" shape

## Pooling

Pooling is a down-sampling operation, that reduces the dimensionality of hidden layers, the deeper they are. This essentially increases the dimensionality of the filters because they slide over a smaller array, thus enlarging the receptive fields and enabling the detection of higher-level, more abstract, features, while reducing the computational overhead.

Common techniques for pooling include *maximum pooling*, which takes the maximum activation over a certain window, or *average pooling*, which employs the average instead of the maximum. Both exhibit a degree of spatial invariance, making the network more robust and improving its generalization ability.

## Fully-Connected Layers

The output of convolutional and pooling layers are high-level features of the input image, which still need to be processed to solve the classification or regression problem at hand. Therefore, they are flattened and fed to regular fully-connected layers designed to infer the class of the input, or a continuous numerical value.

By carefully designing and stacking together many of these building blocks, it is possible to develop CNN architectures with remarkable performance in object recognition and computer vision. Notable examples include AlexNet (2012) [28], that achieved state-of-the-art results in large-scale visual recognition at the time of its publication, VGGNet (2015) [29] with a focus on simplicity, or MobileNet (2017) [30], aiming for computational efficiency for deployment to mobile devices. While computer vision and its exciting developments are out of the scope of this work, convolutional neural networks were introduced because of their relevance in audio recognition for case study II: in chapter 4, a CNN architecture will be

employed to infer the rotational speed of a mechanical system, by extracting the most useful features from an audio spectrogram, which is nothing more than a visual representation (an image!) of the frequency information of a signal.

## 1.2 Machine Learning on Microcontrollers

The last section of this chapter is devoted to the problem of deploying machine learning models, specifically neural networks, on resource-constrained devices such as microcontrollers, and the tools that enable such applications.

Embedded devices are small, low-power and low-cost computers with a dedicated function within larger electromechanical systems. They typically consist of a microcontroller, i.e. an integrated circuit with a processing unit and memory, along with a power supply and various I/O interfaces [31]. The exponential reduction in costs of silicon chips has made this kind of devices more and more prevalent in a wide range of consumer and industrial appliances, wherever there is a need to control a process or perform some software actions in real-time [32].

While deploying machine learning algorithms to embedded systems could enable innovative and *intelligent* real-time applications without relying on an internet connection, thus reducing latency and addressing privacy and security concerns, it is complicated by the strong constraints in terms of resources and memory of common microcontrollers<sup>2</sup>. Deep learning models are computationally intensive and require a large storage capacity, rendering special acceleration and optimization techniques, such as pruning, quantization, model distillation, etc., a necessity [32]. Energy consumption is another critical aspect: embedded sensors are often required to operate with as little as 1 mW of power<sup>3</sup>, which is not compatible with standard machine learning models, nor does it allow to transmit the information over the air. However, efforts to optimize and deploy neural networks on this kind of platforms could enable innovative data processing pipelines, and the ability to perform real-time actions based on such data [34].

Therefore, this is a crucial area of research that has garnered a lot of interest from both academia and industry. This work aims to explore some high-level tools and techniques, investigate the state of the art, and showcase the practical advantages that machine learning brings to real-world embedded applications.

---

<sup>2</sup>It is not unusual for microcontrollers to feature a few hundred kilobytes of RAM, similar amounts of flash memory, and clock speeds of just tens of megahertz.

<sup>3</sup>In order for sensors to be applied to the environment, and run for a useful amount of time without human intervention, they cannot rely on a standard power supply. Instead, they are often battery-powered, which limits considerably their computational ability [33].

The Finder Opta™ PLC is a suitable platform for AI applications since it is equipped with a dual-core Cortex-M microprocessor, on the higher end of microcontroller performance, and 2 MB of flash memory (Table 2.1). Thanks to these specifications, it provides more flexibility when it comes to computationally intensive tasks. Moreover, its integration with the Arduino ecosystem allows access to a pre-built library for machine learning, *TensorFlow Lite for Arduino*. The library leverages Google's *TensorFlow Lite* [35] runtime, specifically developed for on-device machine learning, providing a software framework to execute models trained with *TensorFlow* on 32-bit Arduino boards.

The excellent textbook *TinyML. Machine Learning with TensorFlow Lite on Arduino and Ultra-Low Power Microcontrollers* by P. Warden and D. Situnayake [36] is used as reference on this topic throughout the thesis.

### 1.2.1 TensorFlow

TensorFlow is the main software library used to develop the machine learning models in this work. Originally developed by Google, TensorFlow was released as an open-source library in 2015, and it is currently maintained by machine learning practitioners all over the world [37].

It focuses on training and deployment of deep learning models, and it is built with desktop environments in mind. For this reason, it trades size and complexity for more functionalities and performance, as computational resources are not the main concern. The traditional interface is Python, a high-level scripting language, with cross-platform support added more recently.

TensorFlow performs computations on multidimensional data arrays (i.e. *tensors*, a generalization of vectors and matrices), expressed as «stateful dataflow graphs» [38], exploiting the parallelization capabilities of modern GPUs.

### Keras

To interface with TensorFlow, the most common high-level Python API<sup>4</sup> is **Keras** [39], which was initially developed independently, and later integrated as the default TensorFlow API. **Keras** is an «approachable, highly-productive interface for solving machine learning (ML) problems, with a focus on modern deep learning» [40], which allows fast experimentation, while simplifying each step of the workflow.

It provides high-level implementations of all commonly used objects in machine learning, with its core data structures being **layers**, which encapsulate weights

---

<sup>4</sup>API ("Application Programming Interface"): software interface for different components and services to communicate, exchanging information.

and computational units, and **models**, which group layers together in a trainable network [40]. Two types of models are available: **Sequential** models, which are linear stacks of layers, each with a single input and a single output, and **Functional** models, which provide more flexibility. This work employs exclusively **Sequential** models, as the neural networks to implement are straightforward enough.

**Keras** provides built-in methods for training and evaluating the model's performance (for example: `tf.keras.Model.fit` or `tf.keras.Model.compile` [41]), letting the user focus on the high-level implementation.

More details about the **Keras** workflow and some specific methods will be provided in chapters 3 and 4.

## 1.2.2 TensorFlow Lite for Microcontrollers

TensorFlow is primarily thought for desktop and cloud environments, and does not address properly the confinements of embedded applications of machine learning. For this reason, the *TensorFlow Lite* (TFLite) project was initiated in 2017, enabling much smaller binary sizes by eliminating certain features, such as the ability to train models directly on target devices, or dropping support for more complex architectures [42]. This opened the door to experimentation with machine learning on small, mobile, memory-constrained devices, bringing advanced inference capabilities closer to the end application.

To further widen the range of target hardware capable of executing TensorFlow Lite models, a specialized version, called *TensorFlow Lite for Microcontrollers*, was introduced in 2018 [43]. The goal was to meet even more stringent requirements, such as eliminating operating system dependencies, functioning with no dedicated floating-point unit, or avoiding dynamic memory allocation, all common limitations in standard embedded development. The only requirements, to maintain compatibility with the baseline TensorFlow Lite framework, are C++ 17 and a 32-bit platform [44].

Since analyzing the inner workings of this library is beyond the scope of this work, only a few hints about its main components and workflow are reported here.

The TFLite platform can be thought of as the union of two main objects: the **TensorFlow Lite Converter** and the **TensorFlow Lite Interpreter**.

Starting from a small trained TensorFlow model, that only contains supported operations<sup>5</sup>, the **TensorFlow Lite Converter** applies the *FlatBuffers* serialization library [45] to obtain a space-efficient binary. It also supports additional layers of optimization, as it will be investigated in chapter 3. The converted TFLite model is then transformed to a C byte array, to store it in the read-only program memory

---

<sup>5</sup><https://ai.google.dev/edge/litert/microcontrollers>

of a microcontroller. This solution avoids relying on a filesystem, which is often not available, as it would create an excessive code overhead.

All of these operations will be carried out with the Python language in Google *Colab*, an online environment to execute Jupyter notebooks [46] on the cloud, exploiting Google’s powerful hardware for the computationally intensive tasks above [47].

Finally, the **TensorFlow Lite Interpreter** is in charge of efficiently executing the model on the target device, making use of the C++ library [44].

### **TensorFlow Lite for Arduino**

The TensorFlow Lite for Microcontrollers framework was made available as an Arduino library [48]. However, the official version accessible via the GitHub repository [49] only supports the Arduino Nano 33 BLE Sense board, since its integrated inertial sensors make it the ideal starting point for machine learning. Nevertheless, throughout this work, it was possible to employ an older version [50] that lacks explicit references to the Nano board peripherals, as the framework code is compatible with most Arm Cortex M-based boards, including the Finder Opta™. A different, more labor-intensive option would have involved a custom porting of the base library.

An in-depth analysis of the Arduino code architecture required by the library is available in Chapter 3, alongside a detailed description of the workflow to develop and deploy neural networks on the Finder Opta™, exploiting the software tools introduced thus far.

## Chapter 2

# Finder Opta™: a next-generation "Programmable Logic Relay"

The *Finder Opta*™ Programmable Logic Controller (PLC), or "Programmable Logic Relay", is the device of choice employed throughout this work, both in the two machine learning case studies (chapters 3 and 4), and in chapter 5 for a *smart* industrial control application.

The Finder Opta™ represents a new generation of Programmable Logic Controllers, bringing together the experience in electromechanical devices of Italian manufacturer Finder, and the open-source Arduino electronic platform. The device is characterized by a reliable industrial-grade form factor, with a powerful dual-core M4 + M7 ARM chip that can be programmed via the open-source Arduino programming language, or via traditional PLC languages. Offering a wide range of secure connectivity options, along with high-level hardware encryption, it is developed for intelligent industrial or home automation applications, offering mid to high range performance at relatively low costs<sup>1</sup>.

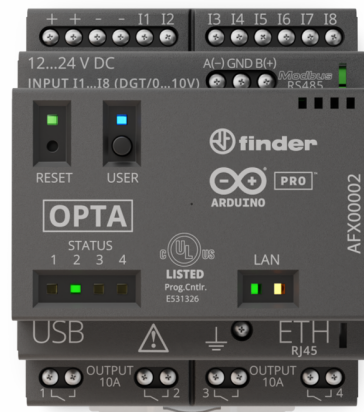


Figure 2.1: Finder Opta™

### 2.1 Programmable Logic Controllers

Programmable Logic Controllers are the most widely used technology for process control and industrial automation, having replaced almost completely traditional

---

<sup>1</sup>[https://cdn.findernet.com/app/uploads/FOCUS-ON\\_Opta\\_EN.pdf](https://cdn.findernet.com/app/uploads/FOCUS-ON_Opta_EN.pdf)



hard-wired control circuits, based on relay logic. They are industrial-grade computers that offer a high degree of reliability, modularity, flexibility, performance and cost-effectiveness, eliminating the need to physically rewire the control circuit to implement changes to the control logic [51]. Programmable logic controllers are *real-time* systems, as their output depends on the input conditions in real time. The brain of a PLC is represented by a microcontroller, that is able to implement arithmetic, logic, timing and counting operations, according to the control program stored in memory. They interface with external devices, like sensors and actuators, with suitable digital or analog input/output ports, that are graded for industrial voltages and currents.

More recent programmable logic controllers also offer advanced connectivity options, enabling remote communication via the internet or other protocols, in the framework of the "Industry 4.0" revolution, as later discussed in chapter 4.

The Finder Opta™ encompasses all the features of traditional programmable logic controller, but also offers capabilities focused on the Internet of Things (IoT, see chapter 4), machine learning, edge computing and connectivity, thanks to the integration with the open-source Arduino ecosystem.

### 2.1.1 PLC Programming Languages

A PLC program is a user-developed series of instructions that dictates the actions to execute, and what response shall correspond to any given input. A program is built using a programming language, i.e. a series of rules to specify such instructions.

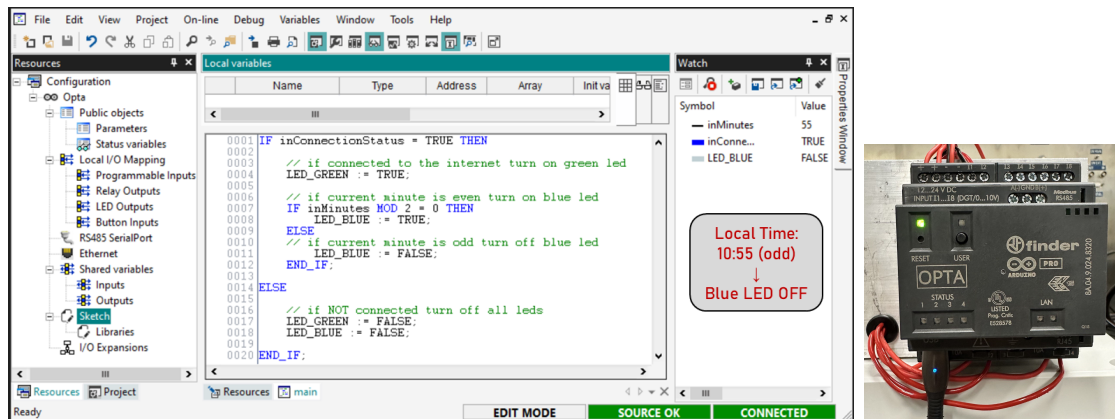
Part 3 of international standard IEC 61131, which deals with programmable logic controllers, defines five traditional programming languages: ladder diagram (LD), structured text (ST), function block diagram (FBD), instruction list (IL) and sequential function chart (SFC), with the first two being the most commonly used. Ladder diagram is a *graphical* language that originated from the schemes of relay racks, employed in traditional hard-wired logic [51]. It is particularly useful in simple but critical process control tasks, and it requires little additional training for technicians not well versed in procedural programming languages, employed in other domains. Instead, structured text is a lightweight high-level *textual* programming language, that resembles Pascal [52].

The Arduino PLC IDE<sup>2</sup> allows programming the Opta™ with any of the five IEC 61131-3 syntaxes, offering a familiar experience to PLC technicians. The software also provides configuration capabilities for all the peripherals, along with the possibility of mixing the traditional languages with Arduino sketches, bringing even more flexibility to the application.

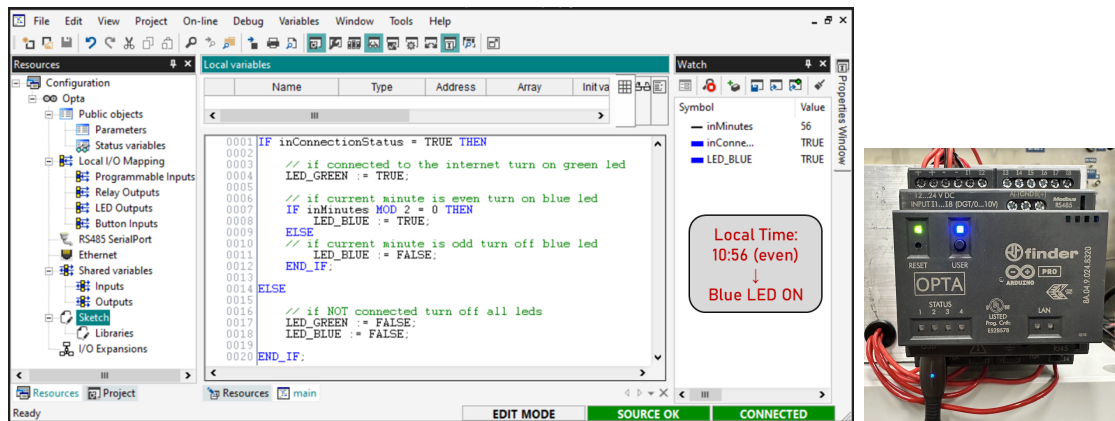
---

<sup>2</sup>IDE: Integrated Development Environment, a program suite for software development (<https://www.arduino.cc/pro/software-plc-ide>)

A simple example program illustrates the potentialities of the PLC IDE: the Arduino sketch (Appendix A.1.1) employs the `WiFi.h` library to connect to the internet, and sends an API GET request to a web service that offers the local time for a given timezone<sup>3</sup>; the JSON response is then parsed to extract the digits of the minutes from the time (`Arduino_JSON.h` library), and store them in a shared variable to be accessible by both the Arduino and the PLC environments (`PLCIn.inMinutes`). A short piece of PLC code, in Structured Text, turns on the built-in blue LED in case integer `PLCIn.inMinutes` is even, as shown in Fig. 2.2, and turns it off otherwise. The green LED instead gives a visual indication of the connection status.



(a) Structured Text code - Odd Time (10:55)



(b) Structured Text code - Even Time (10:56)

Figure 2.2: PLC IDE Programming Example

<sup>3</sup><http://worldtimeapi.org/>

More details on how the connection process works, or on how to send an API GET request to a web server and parse the JSON response are available in chapter 5. Despite the utter simplicity, this is an example of effective integration between the Arduino sketch, geared more towards the connectivity aspect, and traditional PLC programming, to manage the hardware peripherals and the core logic of the application.

In addition to the Arduino PLC IDE, the Opta™ can also be programmed via the regular Arduino IDE<sup>4</sup>, using Arduino sketches.

## 2.2 Arduino Ecosystem

One of the key selling points of the Opta™ - and a crucial motivation behind its selection for this work - is the ability to take advantage of the Arduino ecosystem. The Arduino project was born in 2005 in Ivrea, Italy, as a low-cost and rapid electronic prototyping solution for novices. Over the years, it has rapidly grown into a revolution for electronics, empowering hobbyists, students and professionals all over the world by providing straightforward and inexpensive access to microcontroller kits, a world that traditionally required strong technical skills to access [53]. Arduino's hardware and software products are open-source, under a CC-BY SA<sup>5</sup> license.

The latest developments in the Arduino world involve products for 3D printing, wearable, embedded environments, and IoT applications, aimed at widening the user base by addressing emerging research fields, and targeting more professional applications [54]. The Opta™ represents an attempt in this direction, bringing the Arduino open-source philosophy to the professional market of industrial PLCs.

Arduino boards, including the Opta™, can be programmed with C or C++, via the Arduino programming language API<sup>6</sup>. This brings special structures and methods to the baseline languages, to simplify and streamline certain operations.

The already-mentioned Arduino IDE also brings special code structuring, again with the aim of rendering the software development more straightforward. An Arduino program, called a *Sketch*, only requires two basic functions: `setup()`, which runs only once at the start of the program, and `loop()`, repeatedly executed as long as the microcontroller is powered. These two functions, and any others in the project, are linked with a stub `main()`, and converted into an executable with hexadecimal

---

<sup>4</sup><https://www.arduino.cc/en/software>

<sup>5</sup><https://creativecommons.org/licenses/by-sa/2.0/deed.en>

<sup>6</sup><https://docs.arduino.cc/programming/>

encoding, to be loaded into the firmware. A considerable flexibility is permitted in terms of syntax: for example, a function can be defined anywhere in the project, and does not require a prototype. This streamlined software development process is what makes the Arduino experience very accessible, including to those without prior programming experience.

The Arduino platform offers countless libraries, both official and third-party, to address a wide variety of applications and issues, letting the user focus on the end functionality. Several are explored in this work, including libraries to manage the Wi-Fi connection and the API calls to external services (as in the example in the previous section), to compute the FFT of audio signals, and the already-mentioned *TensorFlow Lite for Arduino*, to deploy neural networks to the PLC. Carrying out any of these tasks without pre-built libraries would require a much longer and more complex development process.

The Opta™ is based on Arm Mbed OS, an open-source operating system and development platform aimed at IoT and connectivity for 32-bit Cortex-M microprocessors, developed up until July 2024 [55]. This is the core providing drivers for all peripherals and modules. Moreover, it allows access to more advanced RTOS<sup>7</sup> functionalities like task-scheduling, memory management, interrupt handling, etc.: for instance, the Mbed OS multithreading feature is exploited in chapter 5 to execute different processes ("threads") simultaneously.

In conclusion, the Opta™ stands out as a powerful and versatile platform, that leverages the extensive resources of the Arduino community to provide an accessible yet robust solution for sophisticated industrial applications.

## 2.3 Finder Opta™ - Technical Characteristics

The main technical characteristics of the Finder Opta™ are summarized in Table 2.1. The device is available in three different versions<sup>8</sup>, with this work employing the **Opta™ Wi-Fi**, that offers the most connectivity options, including Wi-Fi and Bluetooth Low-Energy.

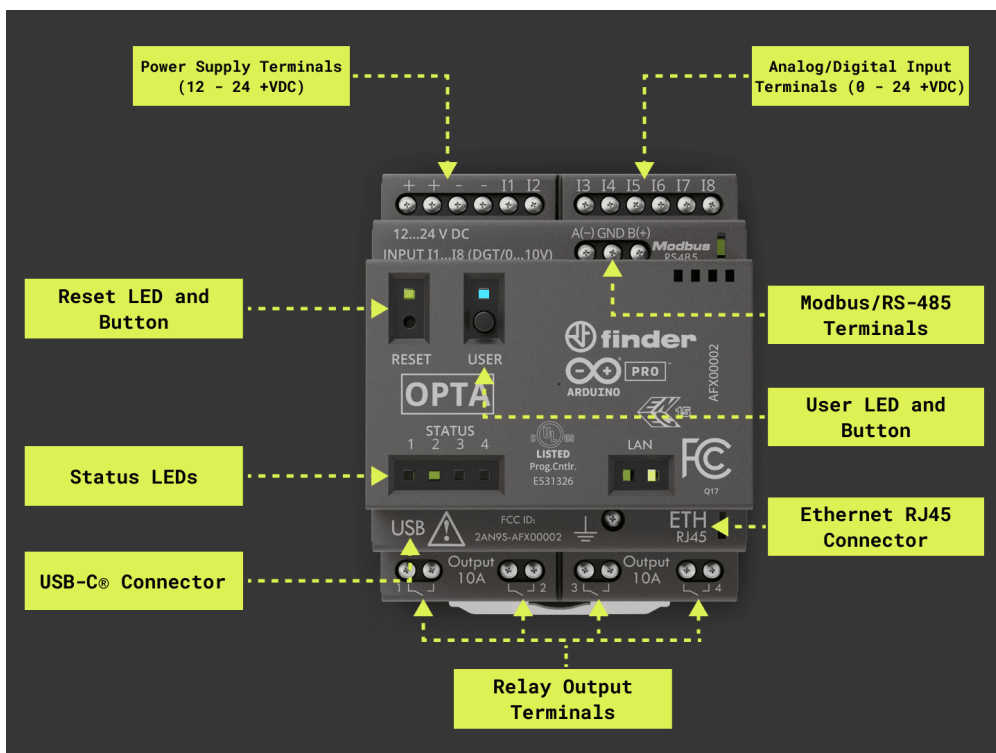
The Opta™ features a dual-core STM32H747XI microcontroller, built around two ARM cores, a Cortex-M7 (up to 480 MHz) and M4 (up to 240 MHz). It supports 2.4 GHz Wi-Fi, Bluetooth Low-Energy and Ethernet connectivity, and it features a USB-C port for programming and data logging, and an RS-485 interface for

---

<sup>7</sup>Real-Time Operating System, an OS characterized by predictability and determinism, for time-critical tasks in microcontrollers: unlike general-purpose operating systems, microcontrollers often require critical tasks to be executed within tight time boundaries.

<sup>8</sup>Catalog: <https://cdn.findernet.com/app/uploads/S8AEN.pdf>

communication with external industrial devices [56]. The Ethernet port supports the TCP/IP (full-duplex) and Modbus TCP communication protocols, with Modbus RTU available via the physical RS-485 port, without internal termination resistors. The Opta™ is also capable of performing Over-The-Air (OTA) updates, without the need to stop the process. Encryption capabilities are provided by the ATECC608B chipset, that can be used to store sensitive information, protecting them from unauthorized access, an ever-increasing concern for IoT applications [57]. Without expansion modules, eight interrupt-enabled input pins are available: they can be used either as digital (12/24 V) or as analog inputs (0 – 10 V, with 12, 14 or 16-bit resolution), depending on the software configuration. The Opta™ also offers four normally-open relay outputs, rated at 10 A and 250 VAC.



**Figure 2.3:** FINDER Opta™ Components<sup>9</sup>

A DIN rail mount ensures that the device can be easily fitted into standard industrial equipment racks.

<sup>9</sup>Source: <https://docs.arduino.cc/tutorials/opta/user-manual/>

**Finder Opta™ - Type 8A.04-8320**

Characteristics	Details
Supply Voltage	12 – 24 V
Inputs	8x Analog/Digital Inputs (8.9 kΩ impedance)
Outputs	4x Relays Normally Open
Processor	Dual-core ST STM32H747XI (Arm Cortex-M7 core up to 480 MHz, Arm 32-bit Cortex-M4 core up to 240 MHz)
Memory	1 MB RAM, 2 MB Flash
Communication	Ethernet, RS-485, Wi-Fi 2.4 GHz, Bluetooth LE 4.2, USB-C
Security	ATECC608B Crypto Microchip
Ingress Protection	IP20
Temperature Range	-20 – 50 °C

<b>Analog Inputs</b>	
Input Voltage	0 – 10 V
Input Resolution	12 – 16 bits
Input LSB Value	166 μV
Accuracy	±5% (repeatability ±2%)
<b>Digital Inputs</b>	
Input Voltage	0 – 24 V
Voltage Logic Level	VIL <sub>max</sub> : 4.46 VDC VHL <sub>min</sub> : 6.6 VDC
Input Current	1.12 mA at 10 V
Acquisition Cycle Time	10 μs
<b>Outputs</b>	
Max Current per Relay	10 A (peak 15 A)
Relay Rated Voltage	250 VAC (max 400 VAC)
Rated Load	AC1: 2500 VA, AC15: 500 VA
Relay Response Time	0 → 1: 6 ms, 1 → 0: 4 ms

**Table 2.1:** Finder Opta™ - Main Technical Characteristics [56]



# Chapter 3

## Case Study I: Waveform Identification Model

The objective of the first case study is to demonstrate how a simple neural network can be deployed on a smart PLC such as the Finder Opta™, opening the door to numerous applications within the "Industry 4.0" framework, especially in the field of predictive maintenance.

The best starting point is the analysis of a basic implementation: recognize the typology of a waveform provided as input to the PLC. Despite the objective being of limited practical relevance, it enables a review of the available tools to acquire familiarity with the workflow. Therefore, this chapter focuses on the fundamentals, summarizing and consolidating what is necessary for further research in the area.

### 3.1 Test Bench Setup and Objectives

#### 3.1.1 Requirements

Below are reported the requirements for the system. This case study involves quite a trivial classification problem, so it is reasonable to expect a high accuracy.

1. Recognize the shape of the input waveform, with an accuracy of at least **90%**, among three possible choices:
  - (a) sinusoidal
  - (b) triangular
  - (c) square
2. The range of the input waveform is 0 – 10 V, with random amplitude and random offset within such range; the frequency is constant and known (1 Hz)
3. Reduce as much as possible the size of the machine learning model
4. Reduce as much as possible the size of the training dataset



### 3.1.2 Test Bench

The experimental setup is composed of the Finder Opta™, the brain of the system where the ML model is executed, a waveform generator to provide the input signal to recognize, and three indicator lights, for a visual indication of the output.

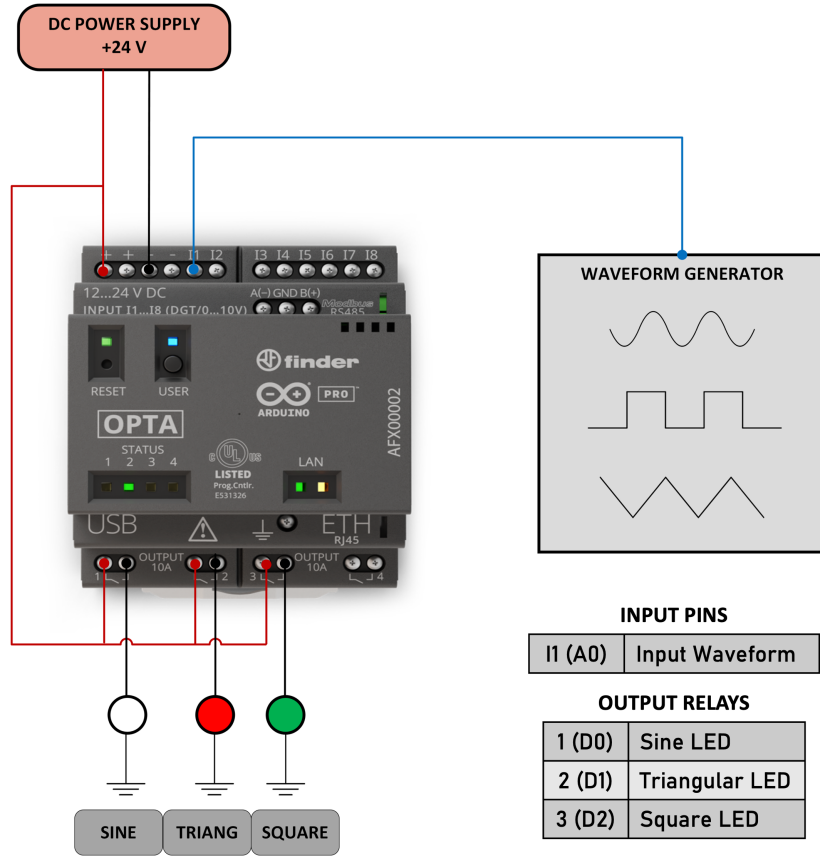


Figure 3.1: Waveform Identification - Test Bench Schematic

Instrument	Model	Qty.
PLC	Finder Opta™ Wi-Fi	1
Regulated DC Power Supply	Voltcraft FSP1243	1
Waveform Generator	Gwinstek SFG2104	1
Digital Oscilloscope	Gwinstek GDS-1054B	1
24 V Indicator Lamps	-	3

Table 3.1: Case Study I - List of Instruments

The Opta™ and the output lamps are powered by +24 V DC, as shown in the schematic of Fig. 3.1. The waveform generator is configured to provide a signal within the range 0 – 10 V, compatible with the analog input pins of the Opta™ (Table 2.1), with amplitude and offset to be manually adjusted.

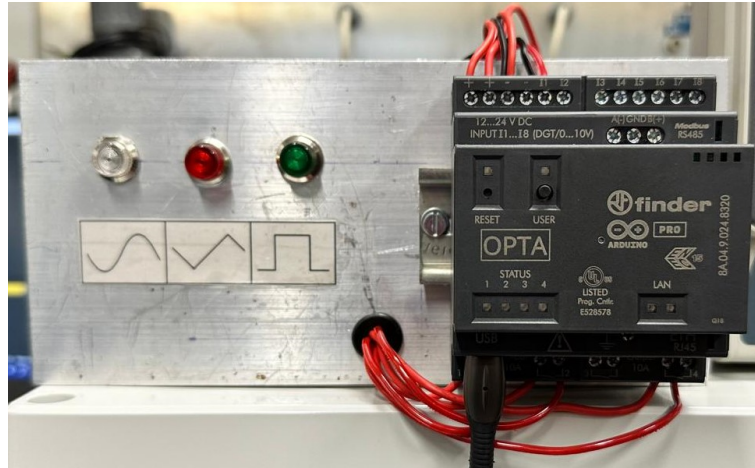


Figure 3.2: Waveform Identification - Test Bench

## 3.2 Workflow Overview

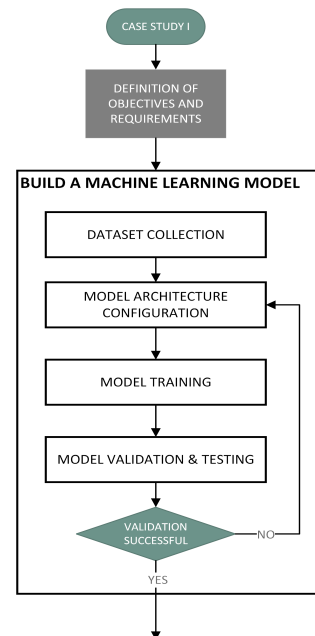
This case study, and in general all machine learning applications with microcontrollers, can be subdivided in three main tasks:

1. development of the machine learning model
2. deployment of the model to the microcontroller
3. validation of the entire system (hardware-in-the-loop testing)

These steps shall always be preceded by the specification of all the requirements and objectives of the activity (as listed in the previous section).

### Development of a machine learning model

The first task in the workflow involves the creation of the machine learning model (a neural network) to execute on the Finder Opta™.



As for all ML applications, this is preceded by the collection of a dataset to train and validate the model: a crucial step as the final accuracy largely depends on its size and quality. However, collecting a large amount of data is expensive, so it is important to find the best trade off. Once a dataset is available, an initial model architecture can be devised, and tuned iteratively as different models are trained and their performance evaluated. All of these steps will be carried out using TensorFlow and its API *Keras*, exploiting Google Colab to write Python code. Only when validation is successful, indicating that the model has achieved the desired accuracy, is it possible to move to the second step, deployment.

**Deployment to the Finder Opta™** As discussed in details in chapter 1, machine learning is a resource-intensive activity that is quite complex to carry out within the constraints of a microcontroller. Therefore, the second task of the workflow consists in converting the model to a lightweight format, with all the infrastructure to execute it on the PLC. Moreover, it is necessary to develop the application that samples the input waveform, runs inference with the model, and makes sense of the output, turning on the correct indicator light. The conversion is carried out with the TensorFlow tools, while the application is developed with the Arduino programming language, in the Arduino IDE.

**Hardware-in-the-loop Testing** By running inference on a systematic test set, employing the model deployed to the Opta™, it is possible to verify the successful integration of hardware and software, and evaluate the final performance.

## 3.3 Building a Neural Network

### 3.3.1 Training Data Collection

It is crucial to collect a high-quality dataset since it is the main ingredient to improve the performance of the resulting model. The dataset shall cover the full

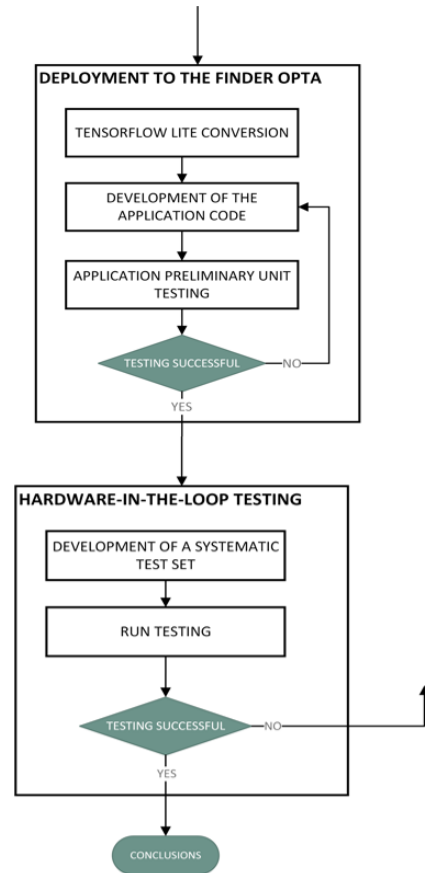


Figure 3.3: Workflow

range of conditions that can occur in the system, and, the larger it is, the more the neural network is able to learn and generalize the desired input features, without overfitting. However, collecting experimental data is time-consuming, expensive, and often quite complex, making certain trade-offs a necessity.

In this case study, collecting a training dataset translates into sampling a number of input signals, in the same working conditions that the model is going to experience. To be representative, the dataset has to include waveforms of all three kinds (sinusoidal, triangular, square), with a wide range of amplitudes and offsets.

### Arduino Application

The input signal, applied to analog pin I1 (A0) (see Fig. 3.1), is characterized by a frequency of 1 Hz: for an accurate reconstruction of its dynamics, a suitable choice of sampling time can be 5 ms, resulting in 200 samples per period. The analog-to-digital conversion chain on the Finder Opta™ is fast enough to sustain this rate, as empirically observed during the data collection.

To make the inference more robust, each training example (the input of the ML model) shall be composed of 500 samples (2.5 periods<sup>1</sup>), instead of 200 (a single period). The last configuration element to address is the resolution of the Analog-to-Digital Converter (ADC) on the input pin, set to the intermediate value of 14 bits (resolution of 0.6 mV).

According to the flowchart of the Arduino application in Figure 3.4, each training example is collected by sampling the input waveform every 5 ms, converting the readings to mV, and normalizing to obtain a value between -0.5 and +0.5 (normalization reduces the norm of the parameters involved, helping with gradient convergence and decreasing the risk of overfitting [58]).

The normalized value to log via the serial port, denoted as  $x_{norm}$ , can be calculated as follows:

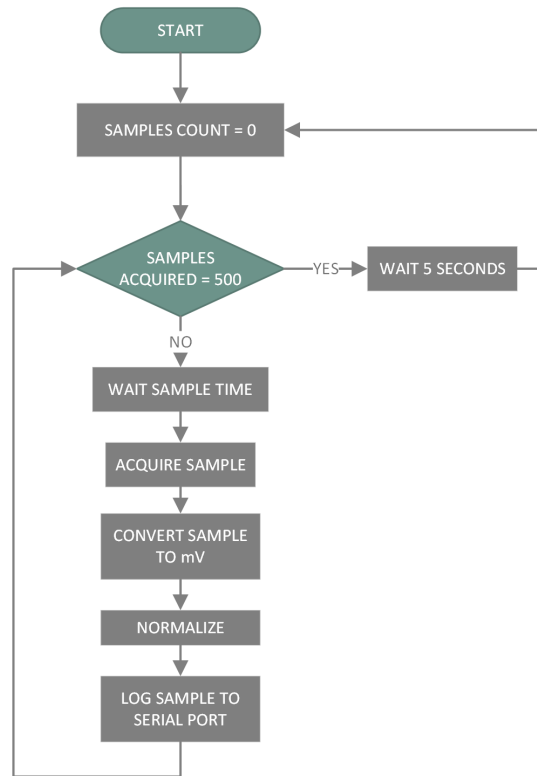
$$x = \frac{\text{sample} \cdot 10\,000 \text{ mV}}{2^{14} - 1} \Rightarrow x_{norm} = \frac{x - 5\,000 \text{ mV}}{10\,000 \text{ mV}}$$

After each training example, the program stops for 5 seconds, to allow the operator to manually modify amplitude and offset of the waveform.

Described below are the key sections of the Arduino implementation to execute on the Opta™ (the entire code is reported in the Appendix B.3.1 for reference).

---

<sup>1</sup>Number of periods in a training example =  $\frac{500 \text{ samples}}{T_{period}/T_{sample}} = \frac{500}{200} = 2.5$



**Figure 3.4:** Training Data Collection - Flowchart

The `setup()` function initializes serial communication for logging the data, and configures port A0 ( $\rightarrow$  I1) as a 14-bit analog input.

```

void setup() {
  // initialize serial communication
  Serial.begin(115200);
  while (!Serial); // wait until the serial port is open

  /* ---- Setup the Arduino Peripherals ---- */
  pinMode(A0, INPUT); // A0 -> analog input port
  // set resolution to 14 bits
  analogReadResolution(ANALOG_INPUT_RESOLUTION);
}
  
```

The `loop()` function, executed repeatedly, acquires and normalizes the samples, before transmitting them to the serial port. The dataset is encoded with floating-point numbers, the most straightforward choice when it comes to neural networks, though `floats` are quite critical when dealing with microcontrollers as they require complex operations, that are sensitive to numerical errors. However, the Finder

Opta™ features a powerful Cortex-M series processor (Table 2.1), with a dedicated floating-point unit to deal with such operations.

```
void loop() {  
  
    // check if the current training example is complete  
    // (500 samples collected)  
    if (samples_count >= NUM_SAMPLES) {  
        // if so wait 5 seconds to allow the operator to change  
        // the parameters of the input waveform  
        delay(delay_between_examples_ms);  
  
        samples_count = 0; // reset the counter for the samples  
        chrono = millis(); // save the time  
  
    } else if (millis() - chrono >= Ts) {  
        /* If a time > sample time has passed since the last sample  
        * -> record a new sample and transmit it to the serial port  
        * after conversion to millivolts and normalization */  
        Serial.println(digitalSampleToMillivolts(  
            (analogRead(A0) - 5000.0) / 10000.0));  
  
        samples_count++; // record that a new sample has been collected  
        chrono = millis(); // record the time  
    }  
}
```

## Experimental Data Collection

In general, deciding a priori how much data to collect to train an effective model is far from straightforward. However, due to the nature of this case study, building a dataset that is representative of all input conditions, with relatively low noise and few outliers, is a manageable task. It is thus reasonable to expect a "small" dataset to provide satisfactory results. Moreover, recall how the requirements in section 3.1 mandate a limited dataset size, to reduce the "cost" of the model. The initial choice is therefore of **100** examples per waveform type (total 300), with this decision validated down the line.

Figure 3.5 shows a randomly selected training example (a sinusoidal waveform, with 4.23 V peak-to-peak amplitude and +4.4 V offset), composed of 500 data points normalized between -0.5 and 0.5. In addition to validating the choice of sample time, as the dynamics of the waveform was captured correctly, the plot demonstrates that the Arduino application for data collection behaves as expected.

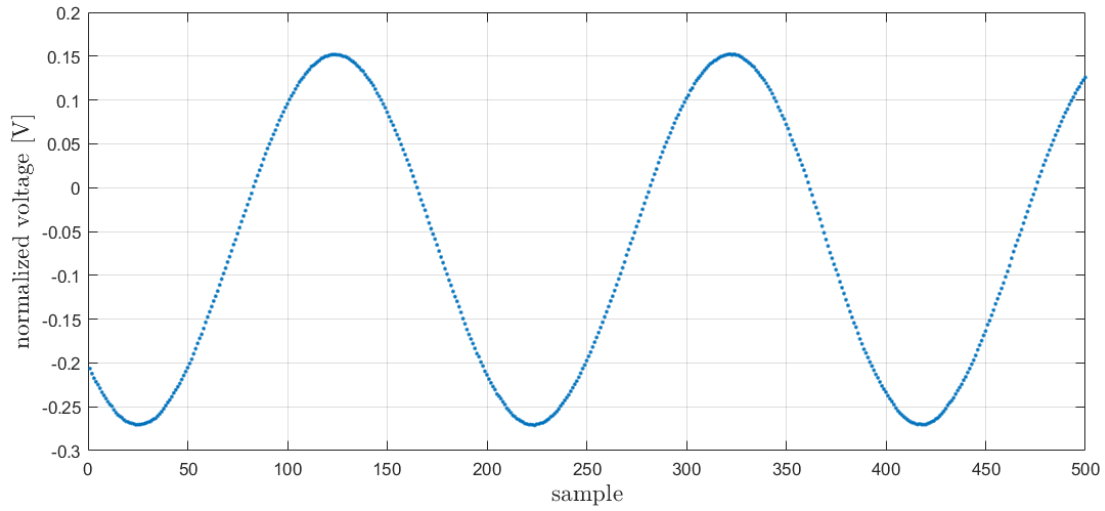


Figure 3.5: Random Training Example

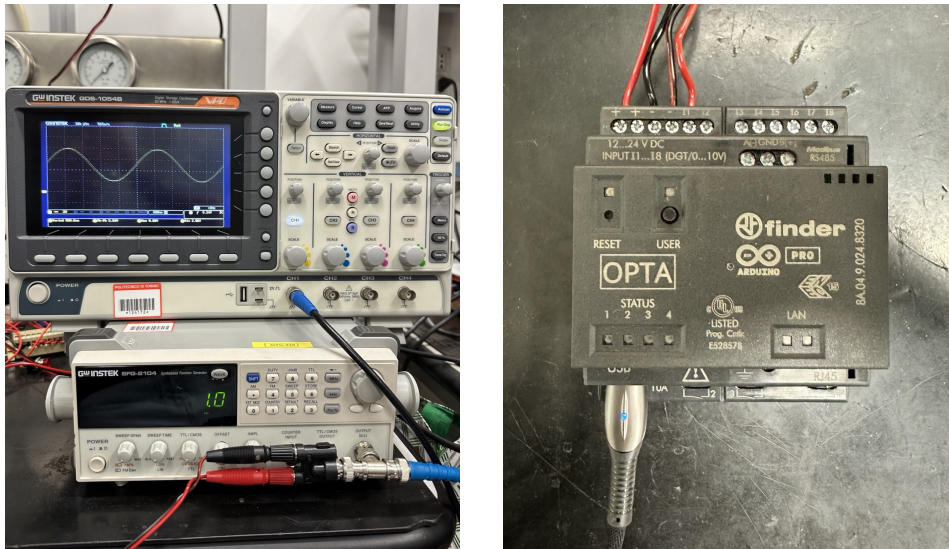


Figure 3.6: Waveform Identification - Experimental Data Collection

## Data Processing

The next step in the workflow involves processing the dataset to prepare the training of the neural network. This activity is carried out in Google Colab with the Python programming language, exploiting the `numpy`<sup>2</sup> library.

<sup>2</sup>`numpy` provides fundamental tools for scientific computing in Python (<https://numpy.org>)

The dataset, stored in three separate `.csv` files (one per waveform type), is parsed by creating a list of all the samples, which is then converted to a `numpy` array. This is the "inputs" matrix, of dimension  $300 \times 500$  (each row is a training example, made of 500 measurements).

A crucial step for supervised learning is the labelling of the training data: the neural network must be aware of what class the training examples belong to. An "outputs" matrix is thus created by stacking vertically several one-hot encoded row vectors, to serve as labels.

Employing a one-hot encoding for the output (i.e. for each example in the dataset, the output is a vector with length equal to the number of classes, of all 0 entries, except for the entry that corresponds to the index of the correct class, equal to 1; for example, a sinusoidal waveform is represented by the vector `[1 0 0]`) is a choice mandated by the loss function that is going to be selected in the next section, 3.3.2.

```
import numpy as np
import pandas as pd # to parse csv dataset files

# list of waveform types to recognize
WAVEFORMS = [
    "sine",          # -> output label [1 0 0]
    "triangular",   # -> output label [0 1 0]
    "square"        # -> output label [0 0 1]
]

SAMPLES_PER_WAVE = 500
NUM_WAVES = len(WAVEFORMS) # 3 waveforms

# create a one-hot (-> something like [1 0 0])
# encoded matrix that is used in the output
ONE_HOT_ENCODED_WAVES = np.eye(NUM_WAVES)

inputs = []
outputs = []

# read the three csv files and parse inputs and outputs
for waveform_index in range(NUM_WAVES):
    waveform = WAVEFORMS[waveform_index]

    output = ONE_HOT_ENCODED_WAVES[waveform_index]

    df = pd.read_csv("/content/" + waveform + ".csv")
    num_recordings = int(df.shape[0] / SAMPLES_PER_WAVE)

    for i in range(num_recordings):
```



```

tensor = []
for j in range(SAMPLES_PER_WAVE):
    index = i * SAMPLES_PER_WAVE + j
    tensor += [
        (df['y'][index]).astype('float32')
    ]

inputs.append(tensor)
outputs.append(output)

# convert the lists to numpy matrices
inputs = np.array(inputs).astype('float32')
outputs = np.array(outputs).astype('float32')

inputs =
[[-0.2064 -0.2122 ... 0.1227 0.1257]
 [ 0.1283 0.132 ... -0.2592 -0.2605]
 [-0.2013 -0.2034 ... 0.2863 0.2869]
 ...
 [-0.3918 -0.391 ... 0.0796 0.0796]
 [ 0.3749 0.3748 ... -0.0814 -0.0819]
 [-0.0819 -0.0819 ... -0.0818 -0.0817]]

outputs =
[[1. 0. 0.]
 [1. 0. 0.]
 [1. 0. 0.]
 ...
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]]

```

The dataset is then subdivided in three parts: a training set, to train the model, a validation set, to compare different architectures during training, and a final test set, to evaluate the performance of the selected architecture.

Since the number of training examples is quite limited, at least compared to modern "big-data" applications, a traditional ratio 60% / 20% / 20% can be reasonable. The partition is carried out randomly, to prevent the model from inferring any property from the order in which the data was collected (for the definition of function `split_dataset()` see Appendix B.3.1).

```

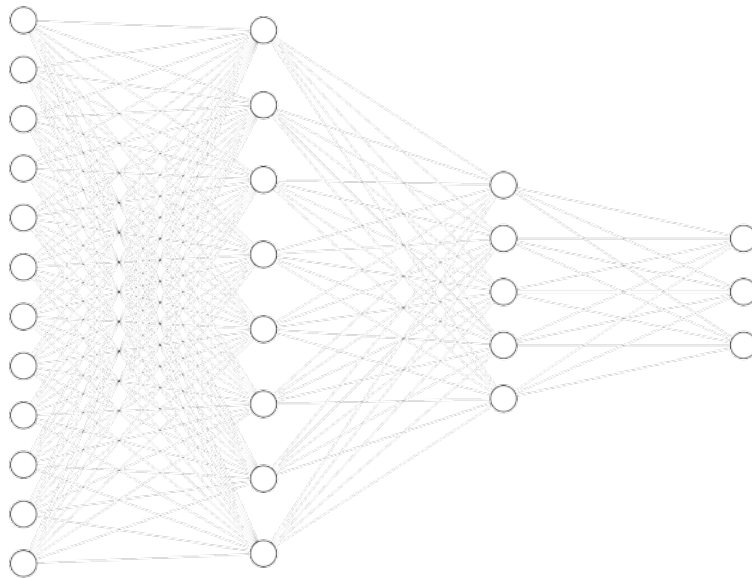
# split the dataset randomly as 60% training set,
# 20% validation set and 20% test set
outputs_train, outputs_test, outputs_validate,
inputs_train, inputs_test, inputs_validate =
    split_dataset(inputs, outputs, [0.6, 0.2, 0.2])

```

### 3.3.2 Machine Learning Model Configuration

The task at hand involves a multi-class classification scenario, where the neural network has to assign a label to an unknown input, in what is essentially a generalization of binary classification.

Various techniques are available to approach this problem [59], but the principal objective in microcontroller-based applications of neural networks is to limit the model size, and not introduce too much complexity. For this reason, a vanilla Multi-Layer Perceptron (MLP) is the most natural choice. An MLP model (Fig. 3.7) is a feedforward neural network that consists of multiple deeply-connected layers, with non-linear activation functions. As discussed in section 1.1, artificial neural networks of this kind can learn any arbitrary nonlinear relationship between the input data, assuming enough depth.



**Figure 3.7:** Generic MLP model with 2 hidden layers

In the most intuitive interpretation of an MLP, each neuron in the input layer represents a quantifiable property, or feature, of the input. In this case study, the input layer requires 500 neurons, one for each waveform sample. Such architecture does not provide the model with any temporal information, treating the samples as completely independent, and not as sequential data. Since the focus is on simplicity, this is an acceptable trade-off, although there exist more complex architectures to maintain the relationships between input features. This will be explored in the next case study, chapter 4, where a Convolutional Neural Network is employed to analyze audio spectrograms.

Given the goal of assigning one of three labels to the input, the output layer of the network must comprise three neurons, with their activation value representing the probability that the input belongs to each class.

This marks the starting point for the development of the model architecture. It is

now necessary to delve deeper into the selection of the number of hidden layers, hidden neurons, and corresponding activation functions, of the loss criterion to quantify the model performance, of the optimization algorithm, and of all the other hyperparameters in the machine learning pipeline, as outlined in section 1.1.

## Loss Function

The most common choice in terms of loss criterion for a multi-class classification problem is the *cross-entropy* function [60], also known as *log* loss. This function assigns a score to the average difference between two probability distributions, specifically the true probability that the training example belongs to a given class ( $\mathbf{0}$  or  $\mathbf{1}$ ), and the predicted probability. By minimizing this score, the aim is for the predicted probability of each class (the output of the NN) to match the true probability, thus obtaining a correct estimate [61].

For discrete distributions  $p$  and  $q$ , the cross-entropy function has the following shape:

$$H(p, q) = - \sum_x p(x) \log(q(x))$$

Since the comparison of different loss functions is out of the scope of this work, the log loss is adopted without further analysis. The choice will be validated during model testing.

Cross-entropy for multi-class classification is available in Keras as `categorical_crossentropy`. It is important to note how `categorical_crossentropy` requires that the output labels (0 or 1) are one-hot encoded [62]. This could be problematic when the number of output classes is huge, as the size of the output would explode, though it is not the case here. A suitable alternative would be `sparse_categorical_crossentropy`, which only requires the index of the correct class instead of a one-hot encoded vector [63].

Recall how the matrix `outputs`, to label the training examples, was encoded as one-hot (section 3.3.1), precisely in anticipation of such choice of loss function.

As discussed in section 1.1.1, the loss function refers to a single training example: the total cost that the optimization algorithm aims to minimize is the sum of the log loss over all the training examples. No regularization term is introduced to keep the process as simple as possible.

## Optimization Algorithm

The most straightforward choice nowadays, when it comes to optimization algorithms for neural networks, is mini-batch stochastic gradient descent (section 1.1.1). It is implemented in Keras with the `SGD` class [64], which can be a good starting

point to train the model, although better performance is expected from more sophisticated versions, as will be investigated down the line.

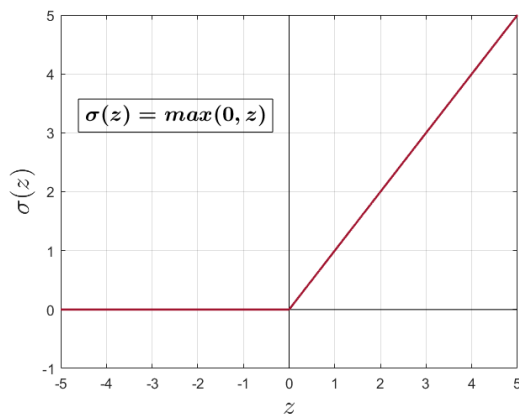
The default learning rate  $\alpha = 0.01$  is selected initially, since no information about how the optimizer will perform is available at this stage.

### Initial Model

Developing a neural network is a highly empirical process, and little a priori guidance is available. One can only guess the initial values of the hyperparameters, which are then tuned in response to how the training process behaves. Keeping in mind that simplicity is crucial, an initial MLP model made of two hidden layers, of 50 and 15 nodes respectively, seems a reasonable guess. These are followed by the output layer, with 3 nodes (one for each class: *sine*, *triangular* and *square*), and preceded by the input layer comprising 500 neurons (one for each sample).

What remains to be determined is the nonlinear activation function for each layer. An obvious choice for the output because of the use of `categorical_crossentropy`, is the *softmax*, i.e. a smooth approximation of the one-hot *argmax* function [14]. It converts a vector of real numbers into a probability distribution, applying the exponential to each element, thus amplifying the maximum, and normalizing by the sum. Such an activation function provides a smooth output (a requirement to be able to compute the gradient during backpropagation), in the form of a probability distribution, exactly what is needed to compute the cross-entropy loss.

$$\text{softmax } \sigma : \mathbb{R}^k \rightarrow (0,1)^k \quad \sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^k e^{z_j}}, \text{ for } i = 1, \dots, k \text{ and } \mathbf{z} \in \mathbb{R}^k$$



**Figure 3.8:** ReLU

Another straightforward choice, at least initially, is the use of the rectified linear unit ReLU as activation for the hidden layers. The ReLU (Fig. 3.8) was empirically found to guarantee faster training in almost all situations, given its constant slope in the region of positive abscissa, in contrast with the more traditional sigmoid or hyperbolic tangent, whose slopes tend to 0 for activations that are far from the origin (Fig. 1.3) [65].

Table 3.2 summarizes the initial choice of hyperparameters.

<b>MLP Structure</b>	Input Layer → 500 neurons [ReLU] 1 <sup>st</sup> Hidden Layer → 50 neurons [ReLU] 2 <sup>nd</sup> Hidden Layer → 15 neurons [ReLU] Output Layer → 3 neurons [softmax]
<b>Optimizer</b>	SGD (learning rate 0.01)
<b>Loss Function</b>	categorical_crossentropy

Table 3.2: Initial Choice of Hyperparameters

Everything is ready for the definition of the model in TensorFlow with the *Sequential* Keras API, since the flexibility it provides is sufficient for this simple application (each layer only features one input tensor and one output tensor). The model is created by defining the densely-connected layers with the `add` method from the `Sequential` class. It is then configured for training with the `compile` method [41]. One of the parameters to pass to `compile` is the list of metrics to evaluate during training and testing. The *accuracy*, defined as the ratio of correct predictions to the total number of predictions over a given set of data, is a natural metric to evaluate a classification model<sup>3</sup>, and indeed the requirements (section 3.1) are expressed in terms of accuracy. Instead, the mean absolute error (`'mae'`), which is also included here, is more suited to regression problems, where the output of the model is a continuous numerical value; it is nonetheless useful as a rough indicator of performance.

```
import tensorflow as tf

model = tf.keras.Sequential()
# define the model structure
model.add(tf.keras.layers.Dense(50, activation='relu')) # 1st hidden layer
model.add(tf.keras.layers.Dense(15, activation='relu')) # 2nd hidden layer
# output layer -> 3 classes = 3 neurons
model.add(tf.keras.layers.Dense(NUM_CLASSES, activation='softmax'))

model.compile(optimizer='sgd', loss='categorical_crossentropy',
              metrics=['mae', 'accuracy'])
```

### 3.3.3 Model Training, Performance Evaluation and Tuning

The method `fit` [41] enables training the model for a certain number of epochs, i.e. iterations over the complete dataset. 300 can be a reasonable starting value.

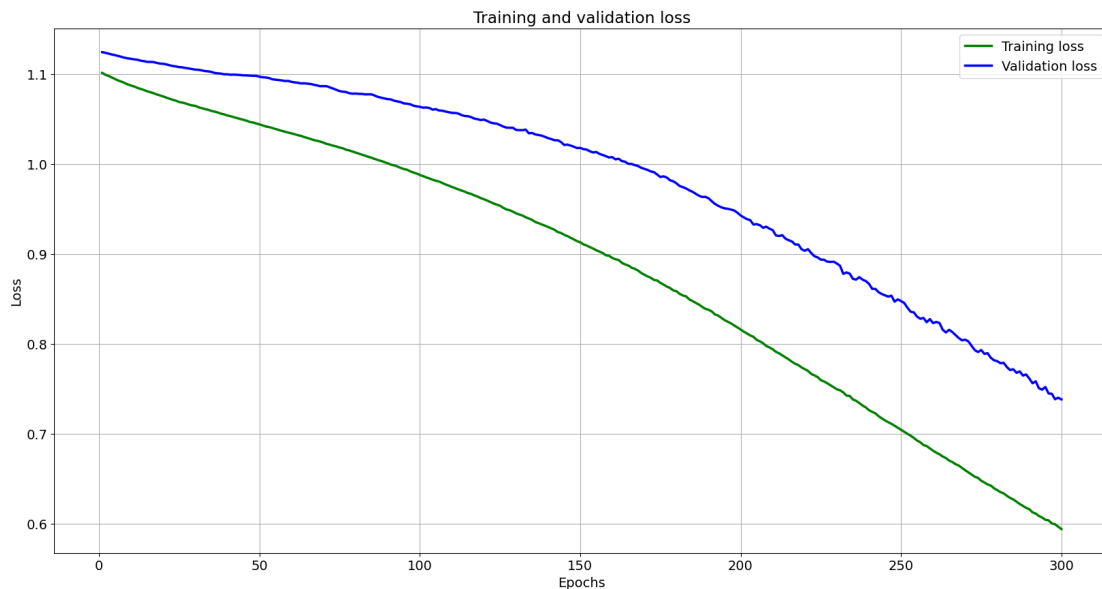
---

<sup>3</sup>Other common metrics for classification include *precision*, *recall*, the *F1 score*, the *area under the ROC curve*..., each with a slightly different focus [66].

Furthermore, the method allows for the specification of the number of training examples to approximate the gradient at each step (recall mini-batch stochastic gradient descent, discussed in Section 1.1.1). The default value is 32. As the `batch_size` increases, the accuracy of the approximation increases, but so does the computational cost: a trade-off is required to maximize training speed.

```
history = model.fit(inputs_train, outputs_train, epochs=300,  
                    batch_size=32, validation_data=(inputs_validate, outputs_validate))
```

Figure 3.9 shows how the basic stochastic gradient descent, with a learning rate of 0.01, takes too long to converge. One possible approach would be to increase the number of epochs, with the associated increase in computational cost. Instead, the more efficient solution that is now explored involves tuning the parameters of the optimization algorithm, for example introducing momentum to accelerate convergence, or choosing a different implementation altogether.



**Figure 3.9:** Training and Validation Loss - initial hyperparameters

### Adam Optimizer

A very efficient algorithm for stochastic optimization is Adam: «the method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters» [21]. Adam builds upon stochastic gradient descent, with different **adaptive** learning rates for

different parameters, computed on the basis of first and second order moments of the gradient.

Basic gradient descent does not take into account its past steps, as it updates the weights solely on the basis of the current gradient and learning rate. However, this may lead to oscillations and slow convergence, or even divergence if the learning rate is too large. To overcome this issue, it is convenient to update the weights with an exponentially-scaled moving average of the gradient, computed over the preceding steps, instead of the simple gradient. The update term becomes  $x_{k+1} = x_k + \beta(x_k - x_{k-1}) + \alpha v_k$ , where  $\beta \in (0,1]$  is the momentum constant: it determines how much importance to give to the previous values of the gradient.

This technique is called **gradient descent with momentum**: the weighted average of the gradients can be interpreted as a *momentum*, that pushes the algorithm in the direction of the minimum with larger steps, while reducing the learning rate (and thus damping out oscillations) towards other directions. Instead, basic gradient descent features a constant learning rate, resulting in steps in the "wrong" directions that are just as large as those towards the minimum.

Adam, which combines this approach with another adaptive algorithm, RMSProp<sup>4</sup>, is implemented in Keras with the Adam class.

```
model.compile(optimizer='adam', loss='categorical_crossentropy',
              metrics=['mae', 'accuracy'])

history = model.fit(inputs_train, outputs_train, epochs=300,
                    batch_size=32, validation_data=(inputs_validate, outputs_validate))
```

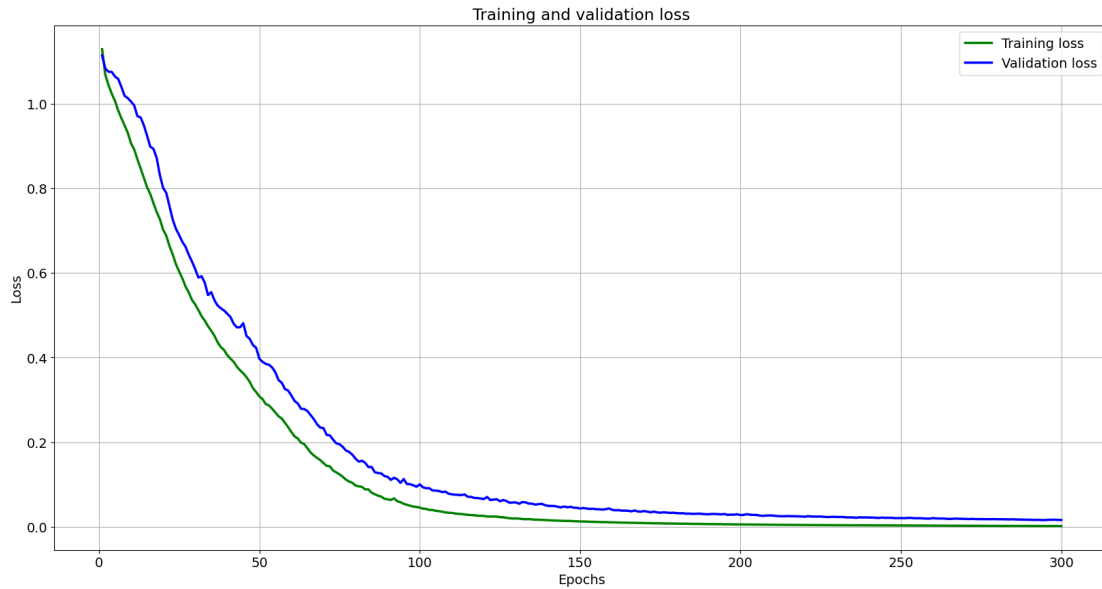
Figure 3.10 shows how the loss, calculated over both training and validation set, now converges in a much quicker fashion, demonstrating how Adam is much more powerful than vanilla gradient descent. Given the shape of the curves, it is reasonable to expect low bias and low variance, with very good overall performance of the trained model.

Table 3.3 summarizes the results of training and validation, with the two chosen metrics. It is normal that the metrics referring to the validation set display larger values with respect to those calculated on the training set, since the former is made of inputs never seen by the model. This is precisely the purpose of the validation set: evaluating the ability of the model to generalize to unknown inputs. Considering the minimal difference in loss and mean absolute error between the two datasets, along with the accuracy equal to 100% for both, it can be concluded

---

<sup>4</sup>*RMSProp* (Root-Mean-Square Prop) employs the square root of an exponentially-weighted average of the *squares* of the derivatives to damp out undesired oscillations, instead of *momentum*, while speeding up learning in the direction of the gradient [24].

that the model demonstrates strong generalization capabilities, i.e. it features a low variance.



**Figure 3.10:** Training and Validation Loss - Adam optimizer

	Training Set	Validation Set
loss	0.00189	0.01618
mae	0.00126	0.00985
accuracy	100%	100%

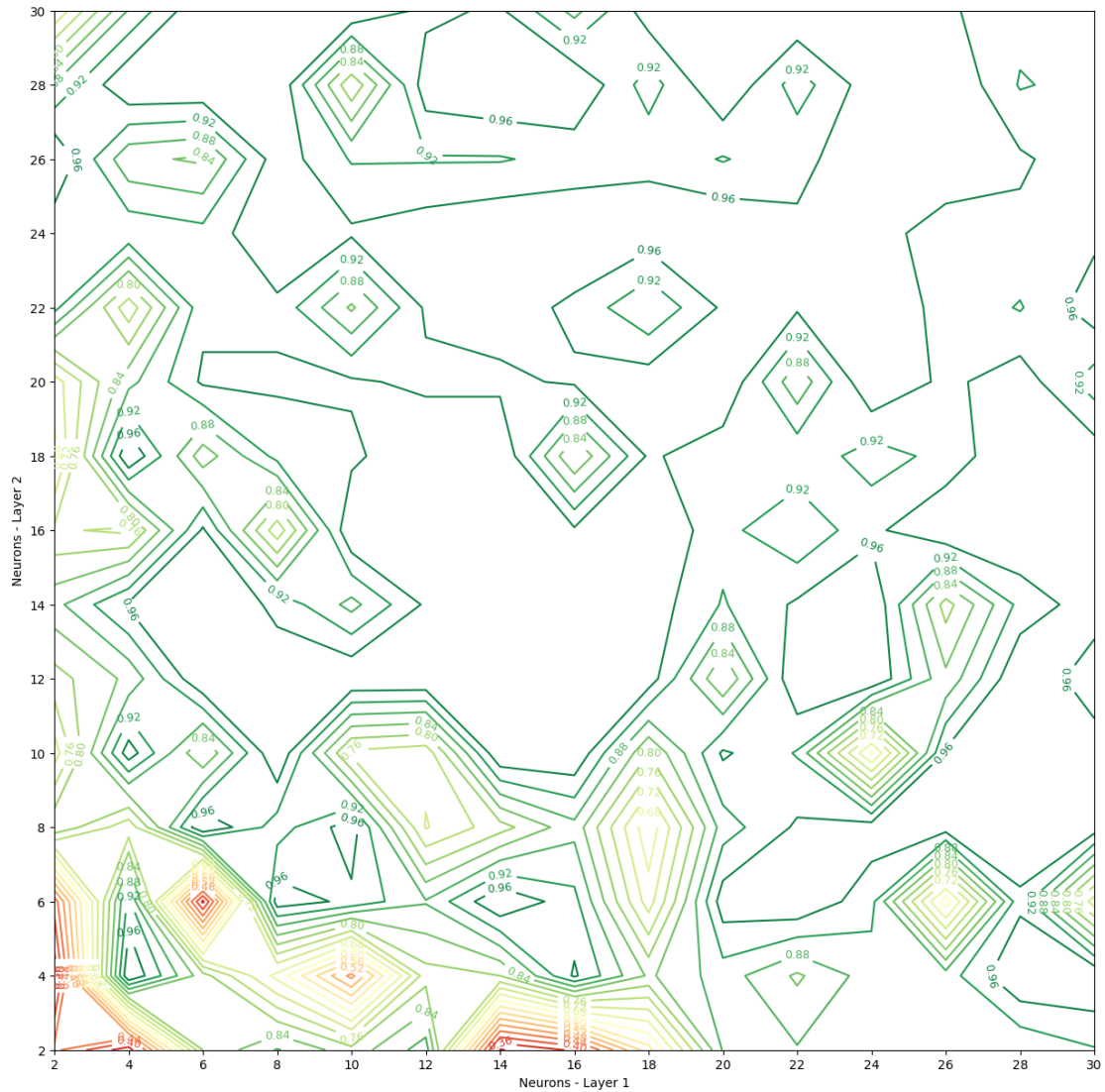
**Table 3.3:** Training and Validation Set metrics

### Adjusting the Model Structure

Judging by the metrics above, the model performs very well with both the training set and the validation set. Since the goal is deployment to a microcontroller, it is important to recall that size is a critical factor, that shall be reduced as much as possible. Therefore, it is interesting to evaluate how performance is affected by the reduction of the number of nodes in the two hidden layers.

Training was repeated for different combinations of number of neurons, with the accuracy contour lines, measured on the validation data set, plotted in Figure 3.11 for a rough estimate of the model performance function of size.





**Figure 3.11:** Validation Accuracy vs Number of Nodes

As expected, the model performance improves with size, since more nodes result in an enhanced ability to learn the features of the input data. Note, however, how the plot is very approximate, as it is highly sensitive to the local minima the optimization algorithm converges to, and it can thus vary widely as training is repeated. Nonetheless, a region of high accuracy ( $> 96\%$ ) can be identified around 6/8 nodes in hidden layer 1, and 10/14 nodes in hidden layer 2. This is an interesting area to explore as it would keep the model size limited, while guaranteeing an accuracy that meets the 90% requirement.

```
model = tf.keras.Sequential()
```

```

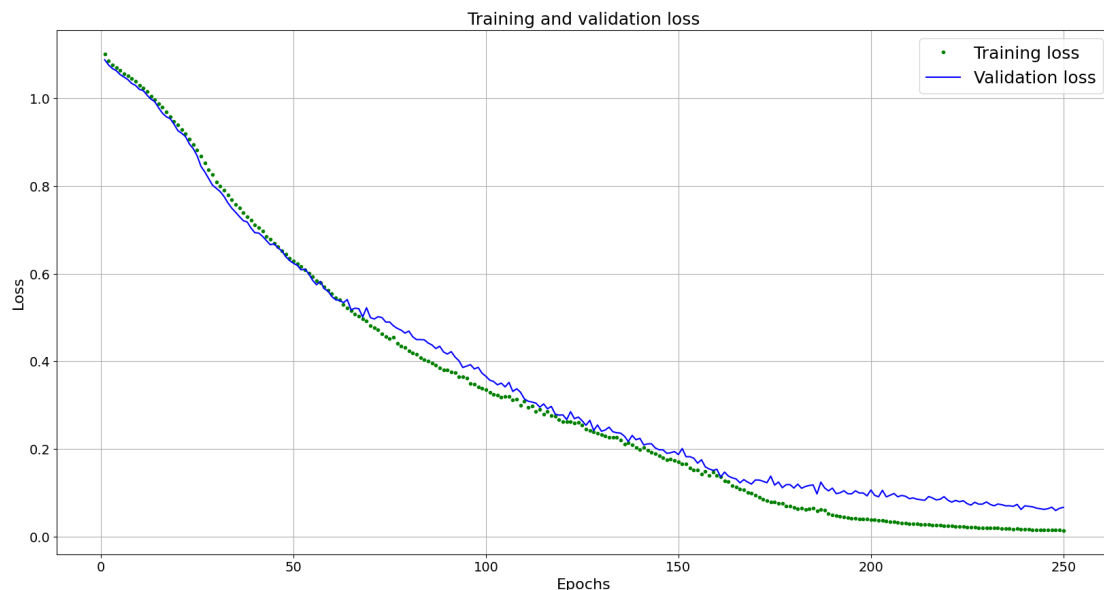
# redefine model structure with less neurons in the hidden layers
model.add(tf.keras.layers.Dense(8, activation='relu'))
model.add(tf.keras.layers.Dense(12, activation='relu'))
model.add(tf.keras.layers.Dense(NUM_CLASSES, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy',
              metrics=['mae', 'accuracy'])

# train the model
history = model.fit(inputs_train, outputs_train, epochs=250,
                   batch_size=32, validation_data=(inputs_validate, outputs_validate))

```

Figure 3.12 and Table 3.4 summarize the performance of a model with 8 neurons in layer 1, and 12 in layer 2.

While both final loss and mean absolute error are not on par with those obtained with the previous -larger- model, the validation accuracy is still calculated at 100%. This meets the 90% requirement, at least at this stage of the workflow, while also addressing the requirement about model size. Both final variance and bias of the classifier are small, despite the learning curve converging in a slightly slower fashion.



**Figure 3.12:** Training and Validation Loss - smaller model

Given the effectiveness of the Adam optimization algorithm, it was also decided to reduce the number of training epochs down to 250 to reduce the training time, a choice validated by Figure 3.12, since the loss appears to have converged.

	Training Set	Validation Set
loss	0.01210	0.03055
mae	0.00791	0.01850
accuracy	100%	100%

Table 3.4: Training and Validation Sets metrics - smaller model

<b>MLP Structure</b>	Input Layer → 500 neurons [ReLU] 1 <sup>st</sup> Hidden Layer → 8 neurons [ReLU] 2 <sup>nd</sup> Hidden Layer → 12 neurons [ReLU] Output Layer → 3 neurons [softmax]
<b>Optimizer</b>	Adam (epochs = 250, batch size = 32)
<b>Loss Function</b>	Categorical Crossentropy

Table 3.5: Final Choice of Hyperparameters

### 3.3.4 Final Model Testing

Before deploying the model to the Finder Opta™, a final testing phase is conducted: the model performance is evaluated with the test set, the remaining 20% of the data collected initially. This is necessary because, when tuning the model to accurately classify both training and validation data, it may overfit the features of those two datasets. Without a final unbiased testing phase, this would go undetected.

```
predictions = model.predict(inputs_test, batch_size=32, verbose=0)
```

The results of the test are summarized in a confusion matrix<sup>5</sup> (Fig. 3.13): the correct predictions are those on the diagonal, where the predicted label matches the true one, while any wrong prediction shows up as an off-diagonal entry. The accuracy achieved in the test is of **98.3%**, with only one wrong prediction out of the 60 test examples. This indicates that the model is not overfit, and that it has learned to generalize the input features.

The choice of hyperparameters is thus validated, and it is possible to move to the deployment phase.

---

<sup>5</sup>See Appendix B.3.3.

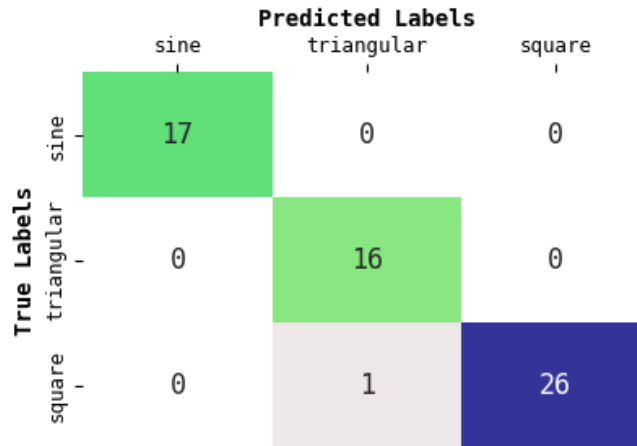


Figure 3.13: Confusion Matrix - Test Set

## 3.4 Model Deployment to the Finder Opta™

After successful testing, the model is ready to be deployed to the Finder Opta™. The first step involves its conversion to a TensorFlow Lite model, which can be embedded in a header file once encoded as a byte array (see section 1.2.2). This is then followed by the development of the Arduino application to sample the input signal, run inference with the model, and visualize the output.

### 3.4.1 TensorFlow Lite Conversion

Recall that the objective is to store the model as a *FlatBuffer*, to reduce its size for memory-constrained devices: the TensorFlow Lite Converter’s API facilitates this operation with high-level functions, and also supports further optimization, such as various levels of quantization (i.e. the representation of 32-bit floating-point numbers as integers) [67]. The model is exported both with and without quantization to analyze the process, and investigate the advantages and drawbacks.

```
# convert the model to TensorFlow Lite format - NO QUANTIZATION
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()
# save to a file
open("waveform_model.tflite", "wb").write(tflite_model)
```

*Post-training dynamic-range quantization* is the most straightforward choice as it requires no representative dataset for calibration, while resulting in little performance degradation. It can be obtained selecting the default optimization strategy,

`tf.lite.Optimize.DEFAULT`<sup>6</sup>, without further converter options. Additional hints about how quantization is implemented, and the various alternatives at one's disposal, are available in the dedicated section below.

```
# convert the model to TensorFlow Lite format - QUANTIZATION
converter = tf.lite.TFLiteConverter.from_keras_model(model)
# the default optimization includes quantization
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert() # convert

# save to a file
open("quantized_waveform_model.tflite", "wb").write(tflite_model)
```

The sizes of the two converted models are calculated to be 18.7 kB and 7.06 kB, respectively: the advantages of using *FlatBuffers* as a serialization library are evident from these figures, since the models can be stored in just a few kilobytes. The effect of quantization is also significant, as it reduces the model size to less than half, in addition to the improvements in inference speed due to the use of integers in place of floating-point calculations.

### Hints about quantization

The growing trend of deploying machine learning models to edge devices, with the associated constraints in terms of memory, power and connectivity, has sparked extensive research in the field of performance optimization.

While several approaches are available, like developing the model with a focus on efficiency from the ground up, a more straightforward technique involves lowering the precision requirements for the representation of weights and activations. This is called *quantization*, and it entails replacing the standard 32-bit floating-point representation of all parameters with 8-bit integers.

As summarized in a white paper by Krishnamoorthi (2018) [68], the many advantages of quantization include broad applicability, since already-existing floating-point models can be quantized without the need to retrain, smaller model size (by a factor of 4, from 32 to 8 bits) with negligible loss in accuracy, lower RAM/cache requirements, thereby freeing up resources for other processes and increasing stability, and faster computations, as integer operations are typically more efficient (especially when the microprocessor lacks a dedicated hardware FPU).

The TensorFlow Lite Optimization toolkit<sup>7</sup> supports various quantization techniques, as well as other optimization methods such as pruning or clustering.

---

<sup>6</sup>[https://www.tensorflow.org/api\\_docs/python/tf/lite/Optimize](https://www.tensorflow.org/api_docs/python/tf/lite/Optimize)

<sup>7</sup>[https://www.tensorflow.org/model\\_optimization](https://www.tensorflow.org/model_optimization)

This work focuses on the simplest and most immediate form, *post-training quantization*<sup>8</sup>, where an already trained model, that can take full advantage of all the operations supported in TensorFlow, is converted to a smaller footprint. Post-training quantization can be implemented with different methods, depending on the sought level of optimization, the target hardware, and the availability of a representative dataset:

- *Dynamic-Range Quantization*

Often the starting point, since it requires no representative dataset to calibrate the range of the input values to the model, it consists in the static conversion of the weights to 8-bit integers and in the dynamic quantization of the activations, while still representing the outputs as floats. This results in a latency that is slightly higher - but typically very close - to full-integer quantization, with little performance degradation.

- *Full-Integer Quantization*

To further improve latency and achieve compatibility with integer-only hardware, assuming that a representative dataset can be provided to the TFLite Converter.

- *Float-16 Quantization*

An intermediate solution, not capable of achieving a latency as low as with fixed-point operations, but useful in some specific applications.

If post-training quantization leads to unacceptable accuracy losses, *quantization-aware training*<sup>9</sup> may become necessary, with the associated changes to the model structure and training process.

In conclusion, it is best practice to explore model optimization when addressing microcontroller applications, as the gains in terms of performance usually far outweigh the degradation in accuracy.

## Conversion to a C file

As outlined in section 1.2.2, the final step before developing the Arduino application to run inference involves encoding the model as an array of bytes, and storing it in an Arduino header file that can be loaded directly into memory. The Unix tool `xxd`<sup>10</sup> can be used for this conversion.

---

<sup>8</sup>[https://www.tensorflow.org/lite/performance/post\\_training\\_quantization](https://www.tensorflow.org/lite/performance/post_training_quantization)

<sup>9</sup>[https://www.tensorflow.org/model\\_optimization/guide/quantization/training](https://www.tensorflow.org/model_optimization/guide/quantization/training)

<sup>10</sup><https://linux.die.net/man/1/xxd>

```
!echo "const unsigned char model[] = {" > /content/model.h
!cat waveform_model.tflite | xxd -i >> /content/model.h
!echo "};" >> /content/model.h
```

The header files generated for the regular model and its quantized version are 115.4 kB and 43.55 kB, respectively. The Finder Opta™ PLC offers sufficient memory (2 MB of Flash - Table 2.1) to store them alongside the necessary infrastructure, which is described in the next section.

The following snippet shows an example of model encoding as a byte array:

```
const unsigned char model[] = {
    0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,
    0x1c, 0x00, 0x18, 0x00, 0x14, 0x00, 0x10, 0x00, 0x0c, 0x00, 0x00, 0x00,
    0x08, 0x00, 0x04, 0x00, 0x14, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00,
    0x84, 0x00, 0x00, 0x00, 0xdc, 0x00, 0x00, 0x00, 0x84, 0x12, 0x00, 0x00,
    0x94, 0x12, 0x00, 0x00, 0xf4, 0x1a, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00,
    ...
    0x0c, 0x00, 0x00, 0x00, 0x72, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x72
};
```

### 3.4.2 Arduino Application

The embedded application running on the Opta™ is an Arduino sketch, built upon the example templates provided by the TensorFlow Lite library. Its core is the interpreter, that is responsible for running inference on the input data, employing the model encoded in the header file. For the interpreter to function correctly, a pre- and post-processing architecture must be implemented to transform the input into a format compatible with the model, and resolve the output to make decisions (in this case turn on/off some indicator lights). This section only provides a brief overview of the required software<sup>11</sup>, with a more in-depth analysis and detailed explanations available in the *TinyML* book by P. Warden and D. Situnayake [69].

#### TensorFlow Lite for Arduino Infrastructure

After including the model (`model.h`) and all the relevant dependencies from the TFLite library, it is time to set up the infrastructure for the interpreter to run.

```
#include "model.h"

namespace { // to avoid conflicts with variables in other files
```

---

<sup>11</sup>See Appendix B.4 for the full application code.

```

tflite::MicroErrorReporter tflErrorReporter;

const tflite::Model* tflModel = nullptr;
tflite::AllOpsResolver tflOpsResolver;
tflite::MicroInterpreter* tflInterpreter = nullptr;

TfLiteTensor* tflInputTensor = nullptr;
TfLiteTensor* tflOutputTensor = nullptr;

// allocate memory for the tensors
// (constexpr to evaluate the expression at compile time)
constexpr int tensorArenaSize = 6 * 1024;
byte tensorArena[tensorArenaSize] __attribute__((aligned(16)));
}

```

All the required objects are declared within a `namespace` to avoid potential conflicts (they are still accessible by functions in the main `.ino` file). These include an instance of `tflite::MicroErrorReporter`, which is only needed to construct the interpreter object, since errors will be handled directly via serial communication<sup>12</sup>, a pointer to the `tflite::Model` struct to reference the model, and an instance of the `tflite::AllOpsResolver` class to grant the interpreter access to the required operations. Two pointers to the input and output tensors are also defined, along with the allocation of the required memory for the execution of the interpreter. This step is critical since establishing the required amount beforehand is quite challenging, and it is important to keep memory usage as low as possible due to the RAM limitations in microcontroller applications. Through a process of trial and error, it was determined that 6144 ( $6 \times 1024$ ) bytes are sufficient.

As in all Arduino sketches, the function `setup()` is executed only once at the start of the program, and part of it is devoted to the configuration of the machine learning infrastructure.

```

void setup() {
    // map model into a usable data structure
    tflModel = tflite::GetModel(model);
    if (tflModel->version() != TFLITE_SCHEMA_VERSION) {
        // raise error if model version is incompatible
        Serial.print("ERROR: Model is schema version ");
    }
}

```

---

<sup>12</sup>In the official TensorFlow Lite library for Arduino, the `ErrorReporter` class logs data via the serial port, with a custom implementation of `micro\arduino\debug_log.cpp`. However, since this file is absent from the version employed in this work, it is preferable to bypass `ErrorReporter` entirely, and print the error messages directly.



```

Serial.print(tflModel->version());
Serial.print(", not equal to supported version ");
Serial.println(TFLITE_SCHEMA_VERSION);

while(1); // endless loop to stop execution
}

```

The model is initially mapped to the `tflite::Model` struct, using the `GetModel()` method, to be accessible by the interpreter. An error is raised if the model version and the library are incompatible.

```

// build the interpreter to run the model
static tflite::MicroInterpreter static_interpreter(tflModel,
    tflOpsResolver, tensorArena, tensorArenaSize, &tflErrorReporter);

```

The interpreter is then built passing all the previously declared objects to its constructor. Recall how `tflErrorReporter` was declared precisely with this purpose, since it is not used as an error logging mechanism for the rest of the code. Finally, the memory required by the tensors is allocated, with the two pointers declared at the start linked to the actual input and output tensors (only one input/output tensor is present, thus the index 0).

```

// allocate memory for the tensors
TfLiteStatus allocate_status = tflInterpreter->AllocateTensors();
if (allocate_status != kTfLiteOk) {
    // raise error if allocation fails
    Serial.print("ERROR: AllocateTensors() failed.");
    while(1); // endless loop to stop execution
}

// obtain pointers to input and output tensors
tflInputTensor = tflInterpreter->input(0);
tflOutputTensor = tflInterpreter->output(0);

...
}

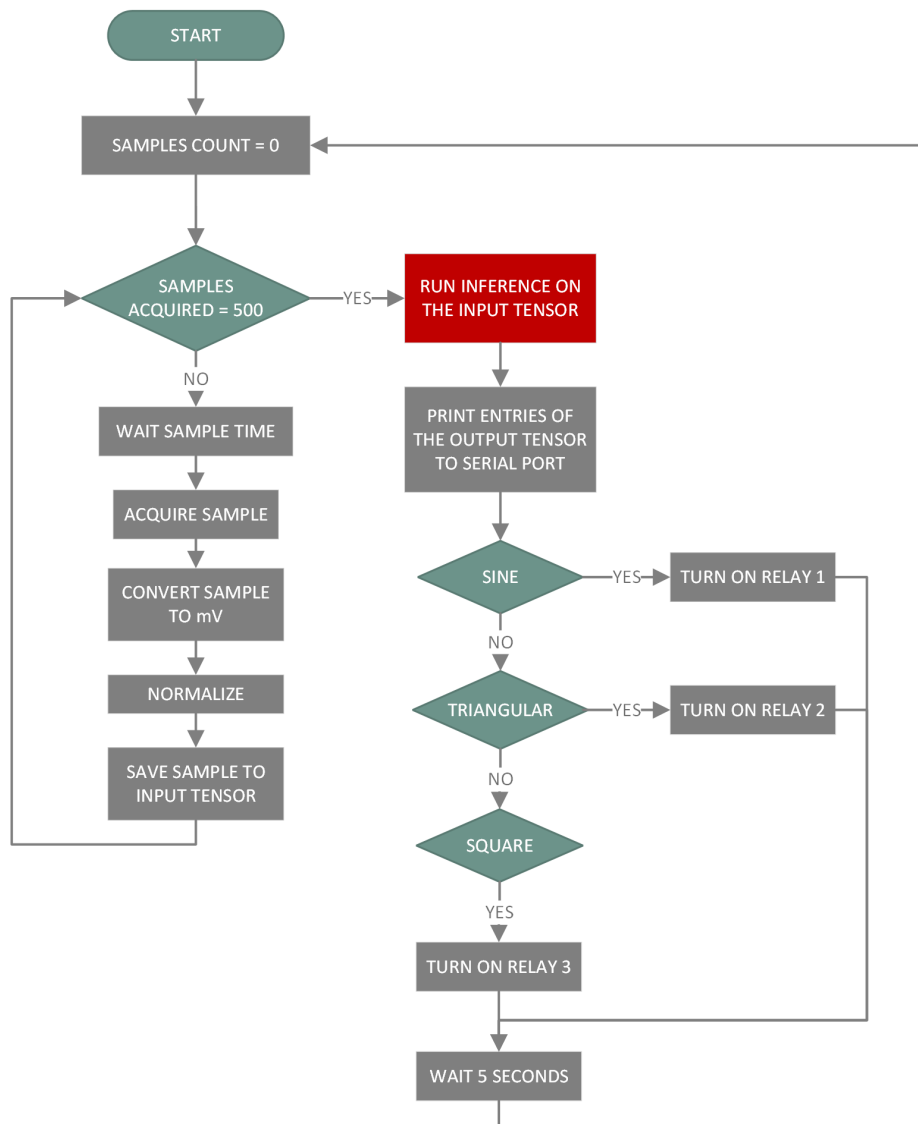
```

This concludes the setup for the TFLite library. It is now possible to store data in the input tensor, invoke the interpreter, and extract the output.

### Application Flowchart

Figure 3.14 presents the flowchart of the entire application, composed of the pre-processing flow for the input signal, the inference with the model, and the output handling to control the relays and associated indicator lights (see section 3.1.2).

- Input pre-processing:** this section of the algorithm must precisely replicate the logic described in section 3.3.1 to acquire each instance of the training dataset. In particular, 500 samples of the input signal are collected with a 5 ms period, converted to millivolts, and normalized to the range  $[-0.5, 0.5]$ , before being stored in the input tensor. It is essential that the input format matches the structure of the training examples, since this is what the MLP model has learned to recognize during training. Any deviations in sampling time, number of samples, or variable range could result in meaningless outputs.



**Figure 3.14:** Waveform Identification Application - Flowchart

- **Inference:** once the input tensor is full (500 entries), it is possible to run inference and classify the waveform.
- **Output handling:** the three output values, corresponding to the activations of the neurons in the output layer, are transmitted to the serial port for debugging purposes. Moreover, the greatest activation causes the associated relay and indicator light to be energized in the test bench (Fig. 3.1), providing a visual indication of the waveform type that was detected.

These three tasks are repeated in a loop every 5 seconds.

## Application Code

The description of the application code, initiated earlier with the machine learning infrastructure, can now be resumed.

In addition to defining all required global variables and constants, a `char` array `-WAVEFORMS[]` - is introduced to map the indexes of the output tensors to the corresponding classes.

```

/* ---- Configuration ---- */
unsigned long Ts = 5; // sample time [ms]
const uint8_t ANALOG_INPUT_RESOLUTION = 14;
// no. of samples to collect to run inference
const uint16_t NUM_SAMPLES = 500;
// time [in ms] to wait before new samples are collected
// for a new inference
const uint16_t delay_between_inferences_ms = 5000;
/* ----- */

// variable to store time
unsigned long chrono = 0;

// keep track of the number of samples acquired
uint16_t samples_count = 0;

// mapping of the output of the model to a waveform type
const char* WAVEFORMS[] = {
    "sine",
    "triangular",
    "square"
};
const uint8_t NUM_WAVEFORMS = sizeof(WAVEFORMS) / sizeof(WAVEFORMS[0]);

```

The configuration for the peripherals is then completed inside `setup()`.

The function `loop()` implements the logic outlined in the flowchart (Fig. 3.14).

```
void loop() {

    float sample; // to hold the current sample

    // check if enough samples have been collected to run inference
    // (at least 500)
    if (samples_count >= NUM_SAMPLES) {
        ...
    } else if (millis() - chrono >= Ts) { // if elapsed time since the
        // last recorded sample > sample time: record a new sample,
        // normalize it, and store it in the input tensor

        // acquire sample and convert to mV
        sample = digitalSampleToMillivolts(analogRead(A0));

        // normalize sample and save it in the input tensor
        tflInputTensor->data.f[samples_count] = (sample-5000.0)/10000.0;

        samples_count++; // record that a new sample has been collected
        chrono = millis(); // save the time
    }
}
```

The first action is to check whether the 500 samples of the input have been fully collected: if this is not the case, which means some samples still need acquiring to fill the input tensor, the program does so with built-in function `analogRead()`, assuming the sampling time has elapsed since the previous acquisition. Each sample is then converted to millivolts (`digitalSampleToMillivolts()`), normalized, and stored in the input tensor.

The last operation is accomplished with the variable `data` of the tensor structure, which is of type `TfLitePtrUnion`<sup>13</sup>. The pointer `.f` grants access to the allocated memory location, since the model operates with `float` quantities.

Instead, when the input tensor contains the full 500 samples, the program is ready to run inference by calling the `Invoke()` method of the interpreter. If this fails, an error is raised and the program halted.

---

<sup>13</sup>`TfLitePtrUnion` is declared in `src\tensorflow\lite\c\common.h`. As a union, it allows storing different data types at the same location in memory, a useful feature for tensors, as they can vary in type. The members of the union are pointers to the memory locations allocated during the declaration of the tensor [69].

Otherwise, the machine learning model executes correctly, filling the output tensor with the three output activations, that correspond to the confidence (or probability) of the network in each waveform class. These are printed to the serial port.

```

if (samples_count >= NUM_SAMPLES) {
  // invoke the interpreter to run inference
  TfLiteStatus invokeStatus = tflInterpreter->Invoke();
  if (invokeStatus != kTfLiteOk) {
    // raise an error if invoke fails
    Serial.println("ERROR: Invoke failed.");
    while (1); // endless loop to stop execution
  }
}

```

The index of the greatest activation, saved in `max_confidence_index`, represents the prediction of waveform type (0 → *sine*, 1 → *triangular*, 2 → *square*). As with the input tensor, each element of the output can be accessed with the variable `data`, of type `float`.

```

uint8_t max_confidence_index;
float max_confidence_value;

// loop through the output tensor
for (uint8_t i = 0; i < NUM_WAVEFORMS; i++) {
  // print the output values (probability of each class)
  Serial.print(WAVEFORMS[i]);
  Serial.print(": ");
  Serial.println(tflOutputTensor->data.f[i], 4);

  // record the maximum confidence value and its index
  if ((tflOutputTensor->data.f[i]) > max_confidence_value) {
    max_confidence_value = tflOutputTensor->data.f[i];
    max_confidence_index = i; // store the index
  }
}
Serial.println();

```

The predictions cause the respective relays to energize, controlling the three lights.

```

// turn on/off the output relays/leds depending on
// the class with max confidence
control_output_relays(max_confidence_index);

// built-in blue LED on when OPTA is paused
digitalWrite(LED_USER, HIGH);
delay(delay_between_inferences_ms); // pause for 5 seconds

```

```

digitalWrite(LED_USER, LOW);

// reset the counter for the number of samples
samples_count = 0;
chrono = millis(); // save the time
} else if (millis() - chrono >= Ts) {
    ...
}
}

```

During the development of the application code, no mention was made to the quantization (or lack thereof) of the model. The reason lies behind the method that was selected, as *dynamic-range quantization* converts weights and activations to perform fixed-point calculations during inference, but still uses a floating-point representation for inputs and outputs. No change to the application code is therefore required if the model is quantized, except for the header file to include.

```
#include "model_quantized.h"
```

### 3.4.3 Preliminary Application Unit Testing

Ahead of moving to the systematic testing of the deployed model (section 3.5), it is good practice to validate the application to ensure no software fault is present. However, since this case study has mainly a demonstration purpose, and robustness of the code is not a priority, it is sufficient to rapidly verify that the input processing and the output handling perform as expected, without developing a proper test set of all the functions and conditions.

To this end, a 1 Hz *sinusoidal* input signal of random amplitude and offset, within the 0 – 10 V range, is applied to pin I1. The serial port is then monitored to analyze the output of the model:

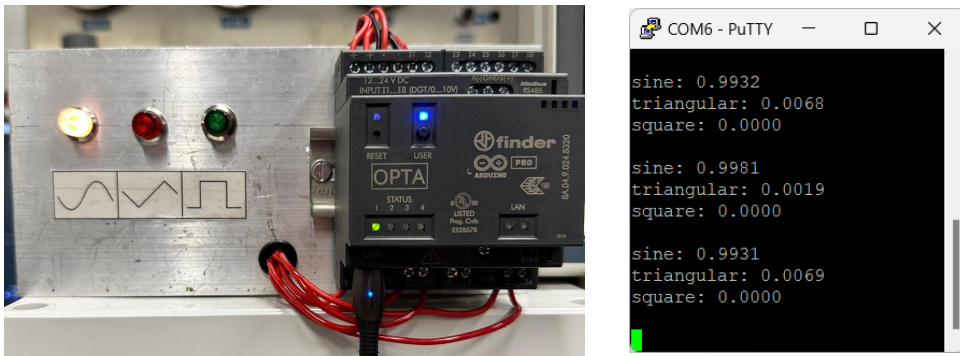
```

sine: 0.9932
triangular: 0.0068
square: 0.0000

```

The application appears to work as expected, given the 99.32% probability that the input is indeed sinusoidal. The output handling flow also shows the correct behavior, with relay D0 energized and the corresponding white indicator light illuminated (Fig. 3.15).

As soon as inference is complete, with the output printed on the serial port, the blue LED on the Opta™ turns on, indicating that the system has paused. After 5 seconds, as expected, the Opta™ resumes operations with another inference.

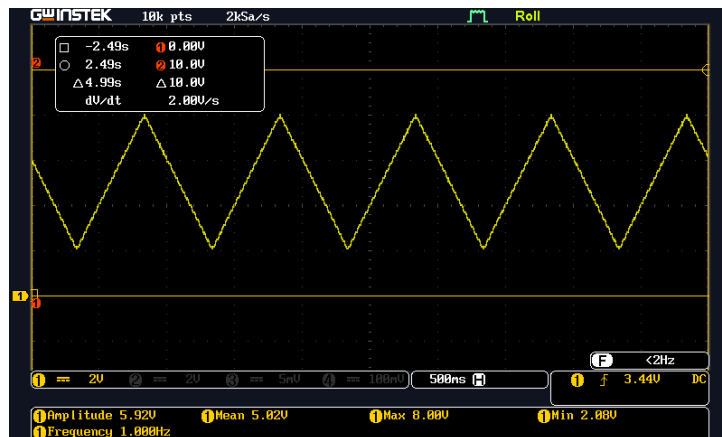


**Figure 3.15:** Application Preliminary Testing - Test Bench and Serial Port

The procedure above is then repeated for a triangular and a square input, with the model correctly identifying their types (99.66% and 91.9% confidence respectively). Despite this testing procedure being quite superficial, it rules out the presence of any major software fault.

### 3.5 Overall System Testing

Once verified that the application behaves as expected, it is possible to move to the last stage of the workflow: testing the performance of the deployed model, both in its regular and quantized version. A systematic test set, composed of 20 inputs per waveform type (total 60 - to mirror the test set employed during the model evaluation - section 3.3.4 - and be able to compare the results), is designed to cover the voltage range 0 – 10 V, with signals of different amplitudes and offsets.



**Figure 3.16:** Example of triangular input signal (amplitude = 6 V, offset = 5 V)

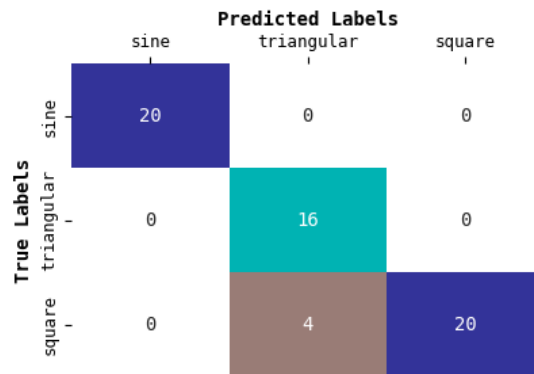
Table 3.6 summarizes the results of the testing activity, by reporting the confidence of the model in the correct labels, for both versions. Green values indicate correct predictions, whereas the red ones are incorrect.

Both the quantized and non-quantized models exhibit an accuracy of **93.3%**, with negligible differences between them in terms of probabilities (the confusion matrix in Fig. 3.17 is the same for both). This highlights the significance of quantization, which resulted in a 50% model size reduction and improved inference speed, without compromising the classification performance.

A lower accuracy with respect to the model testing phase directly in *Google Colab* (98.3%, section 3.3.4) is to be expected because of the conversion to the TFLite for Arduino format. However, it is interesting to note how wrong predictions consistently occur with low-amplitude square inputs (mistaken for triangular waveforms): this is a corner case that, most likely, the model has trouble treating due to an incomplete/limited training dataset, something not related to the deployment to the microcontroller. The implementation of a more systematic training data collection procedure is one of the most effective actions to mitigate issues like this, although the focus of this work is not on maximizing accuracy.

Recall also how, in section 3.3, the size of the MLP model was intentionally reduced, to the detriment of its ability to learn, and thus accuracy.

Finally, it is worth noting how all the requirements defined in section 3.1 were successfully met: the recorded accuracy of 93.3% is greater than the 90% minimum, with the size of the model and of the training dataset limited as much as possible.



**Figure 3.17:** Confusion Matrix  
Deployed Model Evaluation



		peak-to-peak amplitude [V]											
		2	3	4	5	6	9	2	3	4	5	6	9
offset [V]	+1.5	76.7	-	-	-	-	-	85.6	-	-	-	-	-
	+3.0	80.3	86.1	96.0	98.9	-	-	86.6	98.1	98.4	98.5	-	-
	+4.5	79.8	89.6	99.6	99.9	99.9	100	69.1	91.8	99.2	99.2	99.6	99.6
	+6.0	83.7	86.1	97.7	96.7	-	-	69.1	91.8	98.5	99.4	-	-
	+7.5	82.9	85.2	90.0	95.8	-	-	82.6	99.1	97.4	98.0	-	-
	+9.0	90.2	-	-	-	-	-	78.6	-	-	-	-	-

*non-quantized model*                      *quantized model*

(a) Sinusoidal Input Waveform - Confidence Percentage

		peak-to-peak amplitude [V]											
		2	3	4	5	6	9	2	3	4	5	6	9
offset [V]	+1.5	72.5	-	-	-	-	-	69.6	-	-	-	-	-
	+3.0	85.7	90.2	96.5	97.5	-	-	83.6	90.1	97.3	96.3	-	-
	+4.5	88.7	97.0	98.8	99.5	100	100	84.4	87.1	96.1	96.1	97.7	99.2
	+6.0	81.2	99.1	99.4	98.6	-	-	89.8	76.3	97.5	99.3	-	-
	+7.5	86.9	89.2	89.1	95.8	-	-	82.6	90.2	93.4	95.0	-	-
	+9.0	70.0	-	-	-	-	-	68.6	-	-	-	-	-

*non-quantized model*                      *quantized model*

(b) Triangular Input Waveform - Confidence Percentage

		peak-to-peak amplitude [V]											
		2	3	4	5	6	9	2	3	4	5	6	9
offset [V]	+1.5	22.0	-	-	-	-	-	3.1	-	-	-	-	-
	+3.0	8.4	89.2	95.6	98.5	-	-	2.7	80.3	91.8	92.0	-	-
	+4.5	75.6	80.3	88.9	96.8	91.9	99.7	90.2	92.1	99.6	99.6	99.6	99.6
	+6.0	62.3	87.9	88.0	94.5	-	-	77.3	76.3	97.5	99.3	-	-
	+7.5	23.5	89.2	89.1	95.8	-	-	5.5	90.2	93.4	95.0	-	-
	+9.0	2.1	-	-	-	-	-	0.3	-	-	-	-	-

*non-quantized model*                      *quantized model*

(c) Square Input Waveform - Confidence Percentage

**Table 3.6:** Waveform Identification - Deployed Model Testing

## 3.6 Conclusions

The case study analyzed in this chapter illustrates the general workflow of a machine learning application on the Finder Opta™, providing some insight into how a simple model can be configured, trained, tuned and deployed on a microcontroller, thereby enabling increasingly *intelligent* industrial processes.

While investigating a simple example of Arduino infrastructure to acquire the inputs, run inference with the model and handle its outputs, it was possible to show how reducing the model size is essential for resource-constrained devices, and how different techniques, such as the conversion to the TFLite format or quantization, can be put in place to enable AI applications, with small compromises in terms of accuracy.

All the requirements and goals defined at the start were successfully met, paving the way for more advanced machine learning implementations, that push the boundaries of the Finder Opta™ hardware, and provide the basis for innovative research in the field of predictive maintenance.



# Chapter 4

## Case Study II: Towards Predictive Maintenance

In recent years, efforts have been made towards the integration of traditional industrial processes with *smart*, digital environments, that allow the collection of huge amounts of data regarding all aspects of the manufacturing cycle, in what has been referred to as "The Fourth Industrial Revolution", or "Industry 4.0" [4].

In addition to enhancing the speed of information exchange between people, as investigated by Rauch et al. (2020) [70], the digital revolution has allowed access to invaluable data about the process dynamics, that, once analyzed with suitable analytical methods, provide advantages in terms of cost and fault reduction, smaller inventories of spare parts, lower downtime, production volume increases and improvements in operator safety (Sezer et al., 2018 [71]; Lee et al., 2006 [72]). One promising approach that is becoming more and more relevant, as well as familiar and accepted by the general public due to recent successes in AI-powered generative chatbots (Hyesun Choung and Ross, 2023 [73]; Rane, 2023 [74]), is machine learning. Thanks to the ability to handle complex, high-dimensional data, and extract hidden relationships from seemingly unrelated sources, machine learning can be an extremely useful prediction tool in this context, as investigated by Thorsten Wuest and Thoben (2016) [75].

However, as discussed at length throughout this work, machine learning is very costly in terms of computational resources, and typically requires a dedicated central infrastructure and powerful hardware, which makes integration with already-existing traditional production processes difficult, without significant investments or architectural changes.

Therefore, the possibility of deploying machine learning models (especially neural networks), albeit limited in size, directly to terminal devices like PLCs, which already manage process control and access to the various sensors, offers several advantages. These include eliminating the need for big centralized cloud infrastructure, enhancing data processing speed, ensuring better privacy and reducing costs,

in what is emerging as a new paradigm called "edge computing" [76].

This case study aims to explore a preliminary application, setting the basis for further research in this field. In particular, a machine learning model running on the Finder Opta™ will be employed to infer the rotational speed of two radial ball bearings, that support a rotating shaft mounted on a test bench. The input to the model is constituted solely by a short audio recording, collected with an inexpensive electret microphone. While it may seem pointless to employ such a convoluted approach for something as trivial as measuring a rotational speed, it is important to recall that the end goal is predictive maintenance in the context of the "Industry 4.0" framework, where anomalous sound detection is a well-known technique, subject of a large research effort, especially with the rise of deep learning [77]. Moreover, the type of microphone in question is extremely cheap, readily-available, and easy to install, where traditional speed sensors may be more challenging or costly to integrate.

The system under consideration was not originally developed for this application, but as a test bench for bearing lubrication (further details are available in section 4.3.1): this further exemplifies how such a monitoring methodology can be easily retrofitted into an already-existing production line, with minimal costs.

## 4.1 Machine Learning applied to Predictive Maintenance

The key idea behind the "Industry 4.0" transformation is that the «manufacturing industry needs to turn into predictive manufacturing», published by Lee et al. (2013) [3]: historic data combined with domain knowledge can predict trends and patterns in behavior, so as to improve any decision-making process.

One of the most impactful areas where to apply such methodologies is maintenance, which represents 15 to 70% of the total manufacturing cost (Thomas and Weiss, 2021) [78], since accurately predicting the Remaining Useful Life<sup>1</sup> (RUL) of an asset, or similar metrics, can translate into an optimal operating efficiency, minimizing downtime and associated costs.

Predictive maintenance is based on the continuous monitoring of a process, with the collection of a wide range of data to detect the earliest signs of failure, in order to perform maintenance only when needed. This represents an intermediate and optimal strategy between the two other common kinds of maintenance, in a classification proposed by Susto et al. (2012) [79]: "Run-To-Failure" (where maintenance is carried out only when the machine stops working, thus resulting in

---

<sup>1</sup>Operational lifespan before a machine requires maintenance or replacement.

unpredictable downtime) and "Preventive Maintenance" (purely based on time and not on the actual conditions of the machine).

In predictive maintenance, on the basis of the available data, the state of a machine can be estimated with statistical methods, which require a strong mathematical background, model-based methods, though it is not straightforward to build accurate representations of complex physical processes, or, more recently, machine learning and deep neural networks. As mentioned in the introduction, the latter can handle complex multidimensional chaotic information, extracting hidden relationships and exploiting readily available labeled data, that are common in this field, for supervised learning [75]. All of these characteristics make them an exceptionally promising tool, and it was estimated that machine learning strategies applied to predictive maintenance have the potential to achieve an overall equipment effectiveness<sup>2</sup> above 90% [80] [81].

In a literature review by Carvalho et al. (2019) [82], it was noted how among all machine learning methods, artificial neural networks (ANNs) stand out as the most common and applied strategy, as they do not require any domain-specific knowledge, they are robust against inconsistent data or in case of outliers, and they can work in real-time, without an architectural change for every update of the model. However, one significant disadvantage are the training and execution costs, both in terms of data and time, not to mention the computational resources that are required.

Therefore, investigating how to exploit the main advantages of ANNs, while reducing their footprint in terms of resource utilization, is a worthy endeavor to widen the applicability of predictive maintenance practices, and this is precisely the scope of this work.

In the literature, several applications of artificial neural networks to predictive maintenance can be found, with different methodologies and application fields. Some notable examples, relevant for this work, include a deep learning approach to estimate the RUL of rotating components (gears and bearings) by Deutsch and He (2018) [81], a hierarchical deep CNN application to bearing fault diagnosis by Guo et al. (2016) [83] and a novel "MI-YOLO" deep architecture for crack detection of wind turbine blades by Xiaoxun et al. (2022) [84], among others.

While all of these studies have demonstrated the feasibility of employing deep learning and ANNs in predictive maintenance, with satisfactory results in terms of accuracy, little focus is put on optimizing the model size for execution on resource-constrained hardware, such as microcontrollers or PLCs. Hopefully, this case study can represent a preliminary step in this direction.

---

<sup>2</sup>The overall performance of a production resource or set of resources, be they human or technical, during the time they are available to produce.

### 4.1.1 Anomalous Sound Detection

The data analyzed with machine learning techniques, such as vibration information from accelerometers, temperature measurements, electrical signals or historic failure rates, just to name a few examples, can belong to a wide range of different domains. This research focuses on acoustic data, sourced from an affordable electret microphone, within an area of predictive maintenance known as *anomalous sound detection* [77].

Anomalous sound detection aims at identifying whether the sound emitted by a machine is normal or abnormal, for early detection of malfunctions. Sound is often a telling indication of the state of a system, and skilled technicians have traditionally been able to detect anomalies by listening to the process noise [85]. However, since this is neither scalable nor systematic, and there is an increasing need to automate repetitive human tasks, a large research effort has concentrated on the issue in the last two decades [77].

Artificial intelligence has been promptly identified as one of the most promising tools in this area, especially because collecting sound data for normal and anomalous operations is relatively straightforward, and so is building large datasets for supervised learning.

For instance, Tagawa et al. (2021) [86] suggest that acoustic detection of failures is feasible with specialized deep learning techniques, even in noisy industrial environments, and Soni et al. (2023) [87] have successfully applied anomalous sound detection to predictive maintenance of vertical drilling machines.

Recent studies have also started investigating unsupervised models to recognize anomalies, without a-priori knowledge (Wu et al., 2020 [88]; Ikegami et al., 2024 [89]).

The points outlined above justify the choice of acoustic data for the inference of the speed. At present no malfunction is included in the experimental setup, but future developments do include anomalous sound detection for predictive maintenance.

## 4.2 Edge Computing and Machine Learning

The last key aspect worth emphasizing is the aim to deploy the ANN for sound recognition to a microcontroller-based device, such as the Finder Opta™ PLC.

Recent advances of the "Internet of Things" (IoT) have driven the development of *smart* devices that can sense their environment thanks to a wide range of onboard sensors, and transmit such data to a central cloud infrastructure for analysis and decision-making [6]. The same trend can also be observed in manufacturing [90], as discussed in section 4.1 within the context of Industry 4.0.

However, the reliance on a cloud infrastructure comes with several disadvantages in terms of latency and bandwidth, since huge amounts of data need to be transmitted

over long distances, energy consumption, and data privacy and security [8]. Technological improvements of microcontrollers, and machine learning optimization advances, are enabling new computing paradigms aimed at addressing the aforementioned issues, such as *edge computing*, which strives to bring data processing closer to the source. Deploying machine learning models directly to the devices that collect data and control the industrial process like PLCs, enables new real-time applications (for example analytics models to monitor the state of a machine, turning it off quickly in case of anomalies, thus preventing damage), that were made impossible by the high latency of cloud-based architectures, in addition to more efficient resource utilization and improved security [91].

Limitations in memory and power still represent a major challenge for the development of edge computing, especially when machine learning is involved. Many hybrid architectures are being investigated to circumvent the issue, such as training the model on the cloud before deploying it to the edge device, or "Federated Learning", proposed by Google in 2016 [92], where deep learning models are trained at the edge, with the cloud serving as model aggregator. The former approach is the most common and straightforward, and what will be employed in this work, though it does present some drawbacks, especially when the model needs updating, since all of the data need to be transferred to and from the cloud.

A final aspect, but just as important, is the cost-effectiveness of a fault diagnosis system of this nature: no expensive infrastructure is involved, making retrofitting into existing processes straightforward. This could foster adoption rates even in smaller plants, or where it would not make financial sense to employ high-end solutions, because of the type of technology (i.e. pneumatic systems, that are known for their reliability - fault detection is critical - but low overall costs).

## 4.3 Test Bench Setup and Objectives

### 4.3.1 Test Bench

The experimental test bench (Fig. 4.3), originally developed for bearing lubrication characterization, consists of a rotating shaft actuated by an asynchronous electric motor, and supported by two radial rolling bearings. The axial bearing under test, located at the leftmost end of the shaft and loaded thanks to a tie-rod, is temporarily released for it to rotate freely.

Speed management for the motor is in place thanks to an inverter controlled with a PID feedback strategy, and remotely commanded by an HMI, which is also in charge of the data acquisition system (several temperature and vibration sensors are mounted on the machine, but their output is neglected for this case study).

This setup was statically and dynamically tested for speeds in the range 100 – 1500 rpm, and though it was designed for faster speeds, 1500 rpm is set as the



upper limit in this application to avoid noisy heat dissipation fans, and focus on the sound emitted by the mechanical components.

The test bench is retrofitted with some application-specific instruments, as depicted in the scheme of Fig. 4.4: the Finder Opta™ PLC to manage the collection of the acoustic data for training, and later run inference with the machine learning model, an electret microphone to capture the sound emitted by the rolling bearings, and an inductive speed sensor to accurately label the training data.

**Electret Microphone** An electret microphone was deemed an appropriate choice for this application because of their wide availability on the market at low prices, and satisfactory performance.

Their working principle is based on the varying electrical capacity of a diaphragm, that is permanently charged thanks to an electret (a dielectric material exhibiting quasi-permanent electrical polarization, in analogy to a permanent magnet) [93]. A small output voltage (10 – 100 mV) is generated as sound waves deform the diaphragm, thus changing its capacity ( $\Delta V = Q/\Delta C$ ).

To obtain a usable signal, a pre-amplifier stage must be included between the microphone and the ADC of the Opta™: ideally, to minimize the quantization noise, the input signal to the ADC shall maximize its input dynamics, and thus be amplified to cover the 0–10 V range. However, many electret microphones on the market for prototyping applications include built-in amplification circuits up to 5/5.5 V: despite not maximizing the input dynamics of the Opta™ analog pin, thus reducing the sensitivity of the measurement, this would provide some amplification, while eliminating the need to design and prototype a custom circuit, an acceptable trade off.

The selected model (Table 4.1) features a MAX4466<sup>3</sup> operational amplifier, a version optimized to work as microphone pre-amplifier thanks to its optimal GBW product<sup>4</sup> to supply current, ability to work with low voltages, large PSRR<sup>5</sup> and rail-to-rail operations. The circuit is supplied with low-noise 5 V DC, close to the upper limit for  $V_{cc}$ : the output signal features the same range, 0 – 5 V, with the



**Figure 4.1:** Electret Microphone with Amplifier Circuit

<sup>3</sup><https://www.analog.com/media/en/technical-documentation/data-sheets/MAX4465-MAX4469.pdf>

<sup>4</sup>GBW Product: Gain-Bandwidth Product

<sup>5</sup>PSRR: Power Supply Rejection Ratio

gain set to the maximum of 125x (42 dB), and a DC bias of about 2.5 V ( $V_{cc}/2$ ).

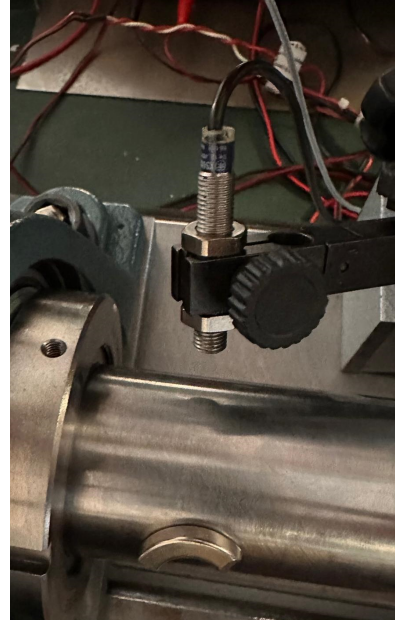
**Inductive Proximity Switch (Speed Sensor)** An inductive proximity sensor, supplied with 24 V DC, is deployed to detect the passage of a small magnet attached to the shaft, for a reliable measurement of the rotational speed. This kind of sensor works by detecting the disruption in its electromagnetic field caused by the presence of a foreign magnetic object [94].

By averaging the number of detected passages over a certain time interval  $\Delta t$ , it is possible to derive the mean rotational speed:

$$n_{\text{avg}} = \frac{\text{no. of detected passages}}{\Delta t}$$

The measurements of the average speed are employed as labels for the training data during supervised learning, and as reference during model validation.

A plexiglas enclosure protects the operator from the rotating machinery, while also serving as weak acoustic insulation from foreign noise. Note, however, that the test bench is located in a busy industrial laboratory, where noise generated by other machinery, and thus external to the system, is very prevalent. This makes it a suitable environment to evaluate the applicability of the machine learning model to a real industrial application.



**Figure 4.2:** Inductive Proximity Sensor mounted on the Test Bench

Instrument	Model	Qty.
PLC	Finder Opta™ Wi-Fi	1
Regulated DC Power Supply	Voltcraft FSP1243	1
Adjustable DC Power Supply	Voltcraft LPS1305	1
Electret Microphone	ARCELI GY-MAX4466	1
Inductive Proximity Switch	-	1

**Table 4.1:** Case Study II - List of Instruments

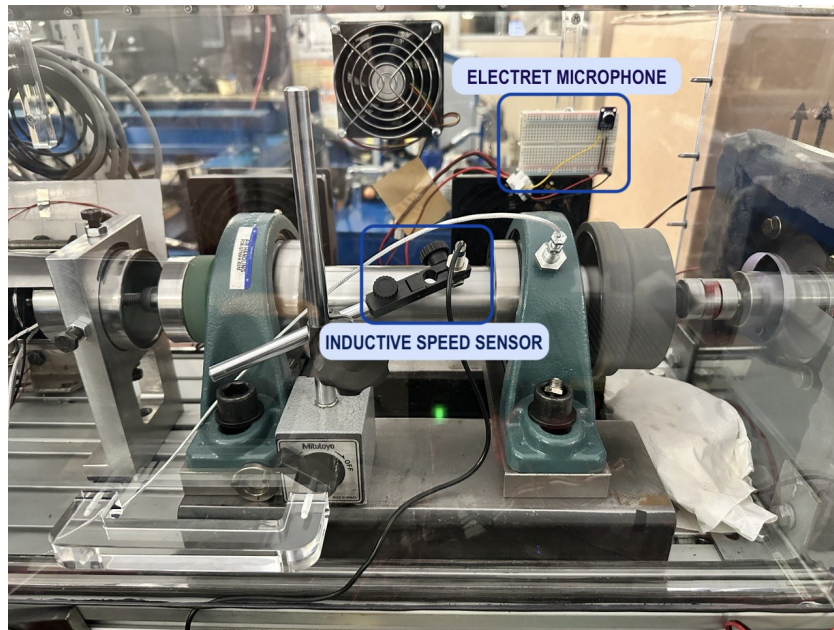


Figure 4.3: Test Bench

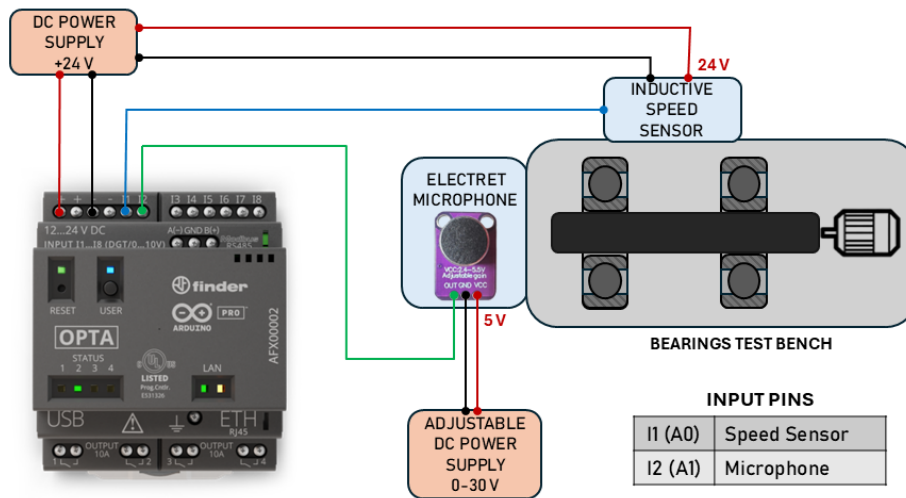


Figure 4.4: Test Bench Schematic

### 4.3.2 Requirements

The general requirements and settings for this case study were outlined in the introductory sections above. It is important, however, to quantify such requirements, to be able to evaluate performance down the line.

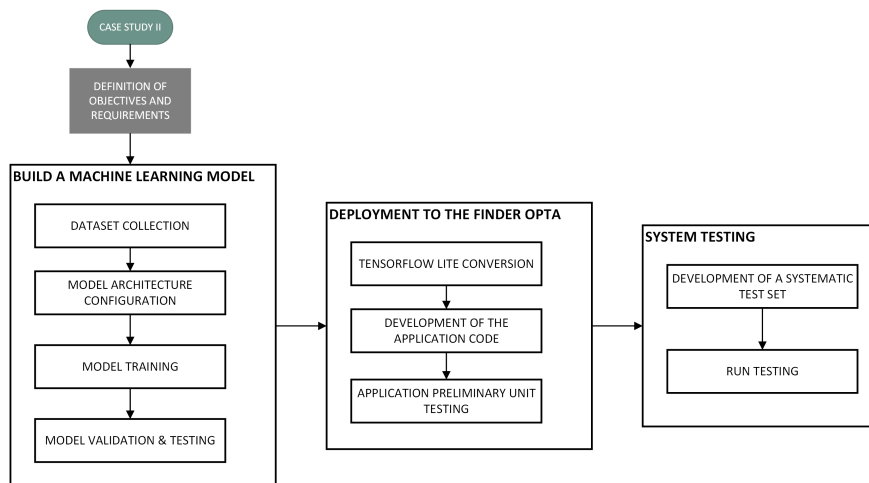
1. Infer the rotational speed of the system with a maximum average error of  $\pm 50$  rpm, within the range **200 – 1500 rpm**.
2. Validate the robustness of the system by ensuring that the maximum inference error does not increase beyond  $\pm 200$  rpm, when disturbance noise is present.

These requirements are quite mild, with relatively large errors (compared to traditional rotational speed sensors, whose average accuracy can vary, but it typically ranges between 1% and 5% [95]): the focus of this case study is not on the accuracy of the speed measurement itself, but on demonstrating the feasibility of the method, and as a building block towards predictive maintenance on microcontrollers.

**Note:** this case study involves a single final testing phase (section 4.7), with the presence of regular background noise and artificially-introduced disturbances, for validation of requirement 1. Instead, the systematic evaluation of the error due to specific disturbances of different types and intensity will be the subject of a later study.

## 4.4 Workflow

The general workflow that was detailed in section 3.2 can be applied, largely unchanged, to this case study as well, as it constitutes the backbone of any machine learning problem. Figure 4.5 recalls the main three stages: the creation of the neural network model in *Google Colab* with TensorFlow, the development of the embedded application for the Finder Opta™ and the final testing.



**Figure 4.5:** Workflow Overview

It is nevertheless worth investigating some peculiarities, that are specific to this case study, to have a clear picture of the entire architecture before moving forward.

**Training Dataset and Model Input** The input to the model is a short audio recording of the system during operations, captured via the electret microphone. While it would be conceivable to feed the model the raw voltage signal sampled by the PLC, extracting meaningful features would pose quite a big challenge for a small network. Instead, a common approach in audio recognition is providing a spectrogram as input, i.e. a 2D representation of the frequency spectrum of the signal over time, a higher-layer abstraction with the most useful information [96]. The training data collection is thus more involved than in Case Study I, with the need to compute the spectrogram of each training example directly on device.

**Machine Learning Model Architecture** A regular MLP model is not suitable for audio or image recognition because of its inability to work with multidimensional tensors and recognize the relationships between groups of adjacent pixels. Instead, as covered in section 1.1.2, *Convolutional Neural Networks* were developed precisely for such purpose, as they can learn how simple features of a multidimensional tensor fit together into more complex structures, aiding in the interpretation of the frequency information contained in the spectrogram. A CNN is therefore the most natural choice here [96], and Section 4.5.2 goes into details about its configuration. Another notable difference with respect to Case Study I is the type of problem that is addressed: the output is not a class probability, as in multi-class classification, but a continuous value representing a physical quantity, a velocity. These are referred to as *regression* problems.

**Robustness** The final testing needs to focus not only on the nominal functionality, but also on robustness of the inference, as foreign and unpredictable noise is to be expected on a plant floor. One of the key advantages of neural networks is their ability to reject outliers, but it is nonetheless important to quantitatively evaluate this aspect, and possibly include hard-coded strategies to improve on it even more, as will be discussed during the development of the output handling flow of the embedded application (section 4.6.2).

## 4.5 Building a Neural Network

### 4.5.1 Training Data Collection

As introduced in section 4.4, extracting useful features from a raw audio recording (Fig. 4.6) would be ambitious for a small neural network. A spectrogram representation is employed instead, a higher-level abstraction of the frequency information in the audio signal.

The training dataset must be comprised of a sufficient number of training examples (spectrograms), covering the entire range of interest (200 – 1500 rpm), and collected in different conditions (at different times of the day, and over several days, to ensure they are representative of all working conditions of the system).

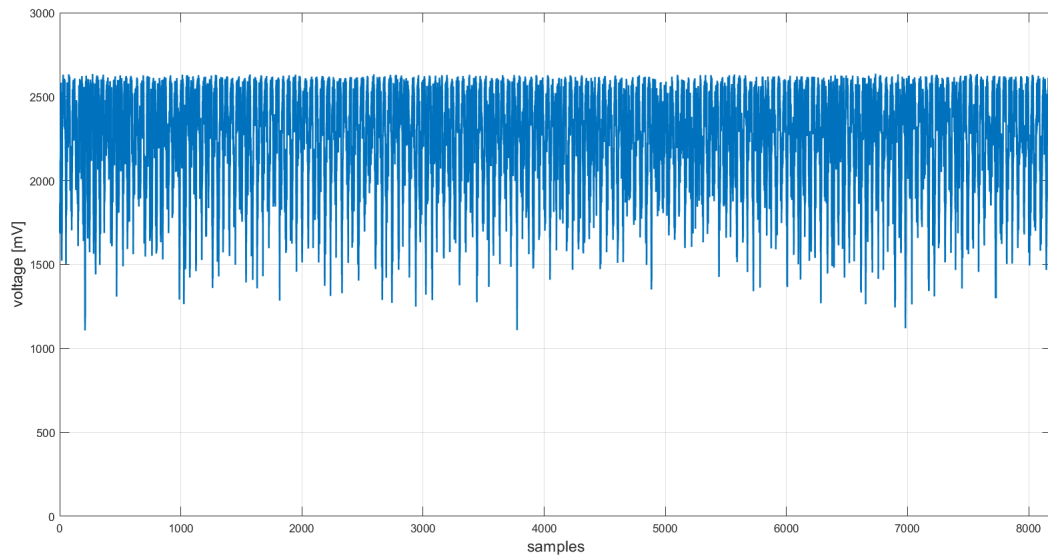
During the development process, it was found that subdividing the speed range at increments of 50 rpm, and collecting 50 spectrogram examples for each speed label, strikes a good balance between performance and dataset size (since larger datasets are associated with higher costs).

speed [rpm]

<b>200</b>	<b>250</b>	<b>300</b>	<b>350</b>	...	<b>1400</b>	<b>1450</b>	<b>1500</b>	total: 1350
50	50	50	50		50	50	50	

no. of training examples

Note, however, that the actual speed of each example is measured alongside the spectrogram with the inductive proximity sensor, and thus it may differ slightly from the nominal label.



**Figure 4.6:** Microphone Voltage Output

## Spectrogram

A spectrogram is a 2D representation of the frequency spectrum of a signal with respect to time. Fig 4.7 shows an example, depicted as a heatmap, with time on the ordinate axis and the frequency bins on the abscissa. The color intensity represents the amplitude of each frequency component at a certain time.

Starting from the time-domain sampled signal out of the PLC ADC, the spectrogram can be created using the discrete Fourier transform, computed on-device via the *Fast Fourier transform* (FFT) algorithm [97].

Before analyzing the application, a few design choices regarding the sampling frequency and the size of the spectrogram need to be addressed.

The hardware limitations of the Opta™ in terms of memory are highly restrictive, with the CNN model that will be developed approaching the limit of what can be handled. The dimension of the input spectrogram is one of the most critical factors in this regard, since it affects both the model size and the memory required to store the input tensor. With a careful trial and error procedure, it was determined that  $256 \times 32$  are optimal dimensions, that maximize the use of memory while resulting in models that could consistently be executed.

These dimensions constrain the audio signal size to  $256 \times 32 = 8192$  samples, with the frequency range of interest divided into 32 frequency bins.

The sampling frequency is the last item to design, and it determines how the audio signal is reconstructed and its duration in time. The audible audio spectrum is 20 Hz – 20 kHz, therefore, following the Nyquist sampling theorem to avoid aliasing, the most common audio sampling frequency is 44.1 kHz [98]. However, such a large frequency would result in sound recordings just  $8192/44.1 \text{ kHz} = 0.19 \text{ s}$  long, with the risk of not capturing the dynamics of the system, since the time constant is of the order of a few seconds. Moreover, the goal is not accurately reconstructing the recorded audio, but capturing the most fundamental frequencies associated to the

rotation of the bearings, so that the model can learn a correlation to the speed. This led to the selection of a sampling rate of **5000 Hz**, which yields audio recordings that are  $8192/5 \text{ kHz} = 1.64 \text{ s}$  long, and cover the fundamental frequencies of the

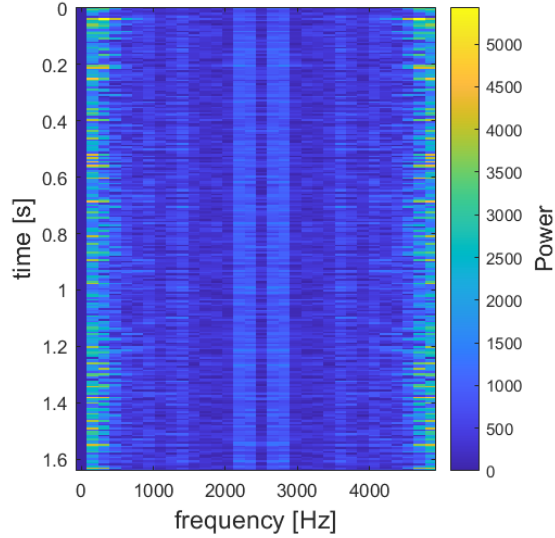


Figure 4.7: Spectrogram Example

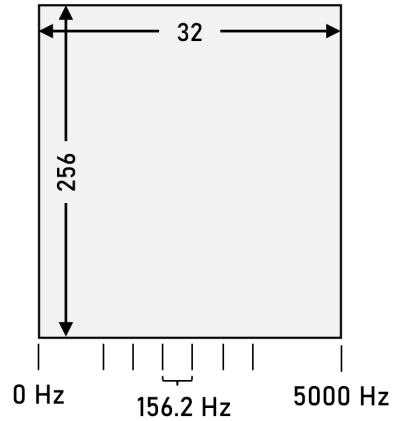


Figure 4.8: Spectrogram Dimensions

bearings of the test bench<sup>6</sup>.

### Arduino Application

Fig. 4.9 shows how the Arduino application employed to collect each spectrogram is split into three main tasks.

1. Measure the speed of the shaft with the inductive proximity sensor, by averaging the number of magnet detections over 10s, to obtain a label for the training example.

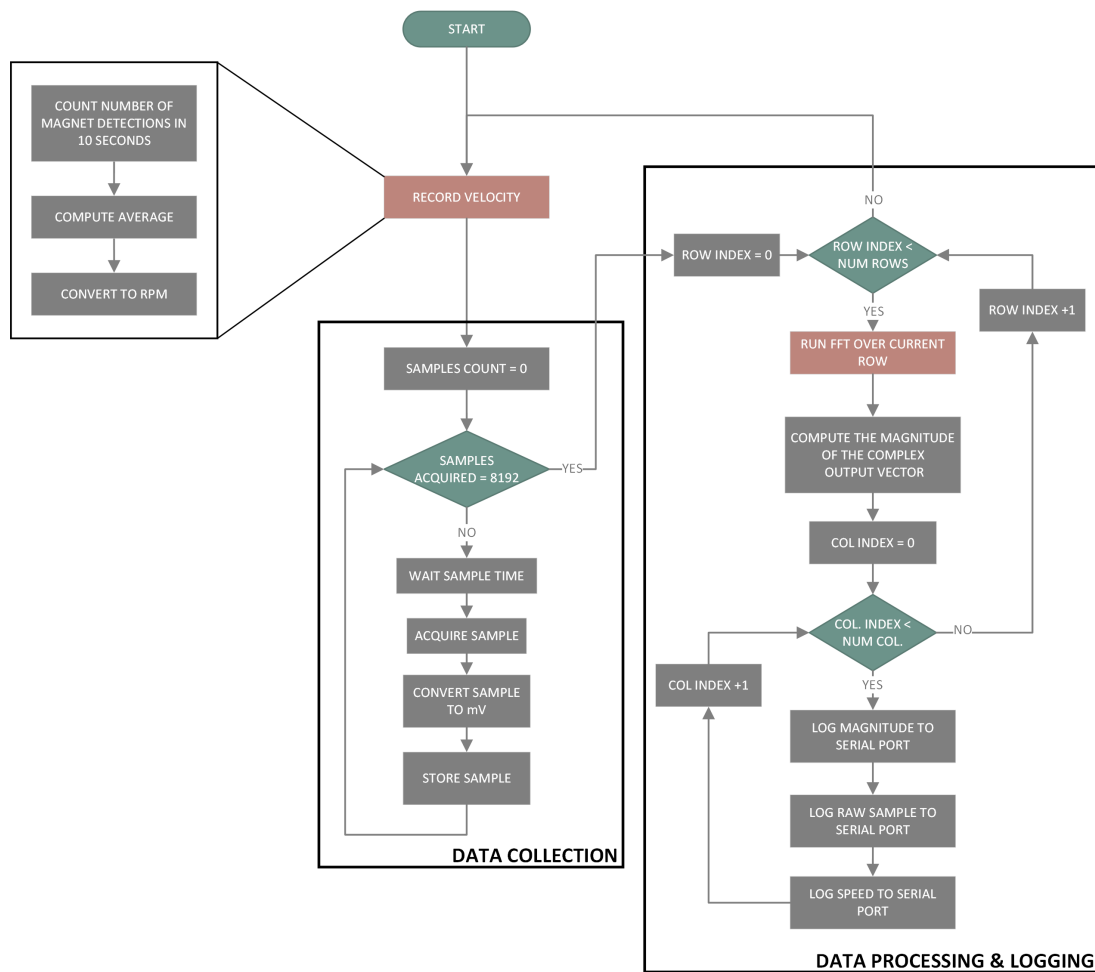


Figure 4.9: Training Data Collection - Flowchart

<sup>6</sup>Bearing Unit Dodge P2B-GTMAH-50M



2. Record the 8192 samples of the audio signal with a sampling rate of 5000 Hz, and convert the digital samples to millivolts.
3. Loop through recorded samples, computing the FFT over 32 samples at a time (one row of the final spectrogram), and log the magnitudes to the serial port, along with the raw voltage samples and the measured speed; repeat until the spectrogram is complete (256 rows processed).

The key sections of the Arduino application deployed to the Finder Opta™ are discussed below, with the entire code reported in the Appendix (C.5.1) for reference. The `setup()` function establishes serial communication for logging, and it configures the required input pins (A0 digital, A1 analog, 16-bit resolution). Moreover, it associates a hardware interrupt to the falling edge of pin A0.

```
void setup() {  
  
    ...  
  
    // configure interrupt (pin A0) for falling edge of velocity sensor  
    attachInterrupt(digitalPinToInterrupt(A0),  
                   count_magnet_detections_ISR, FALLING);  
}
```

The inductive proximity sensor is digital, with its output HIGH (24 V) when a magnetic object is detected, and LOW otherwise: an interrupt service routine (ISR), triggered when a falling edge of the signal is recognized (pin A0), updates a counter to keep track of the number of magnet detections, and thus compute the average speed. Note that the sensor signal is affected by bouncing<sup>7</sup>: a software debouncing strategy is implemented, whereby the program waits for 50 ms before any new update to the counter. This time period was empirically found to be sufficient for the transient of the switch to be over.

```
// counter for number of magnet detections  
// (volatile because updated inside ISR)  
volatile uint32_t magnet_detections = 0;  
  
/* Interrupt Service Routine (ISR) linked to falling edge of  
 * velocity sensor (magnetic) -> increase counter after  
 * debouncing */  
void count_magnet_detections_ISR() {  
    // debouncing
```

---

<sup>7</sup>A non-ideal behavior of real switches, that results in multiple transitions out of a single input.

```
    if (millis() - last_detection_time > kDebounceInterval_ms) {  
        magnet_detections++; // update counter  
        last_detection_time = millis(); // record current time  
    }  
}
```

The `loop()` function implements the three tasks outlined in the flowchart. First, the average velocity is calculated by dividing the number of magnet detections by the elapsed time ( $\approx 10$  s):

```
    // compute the average velocity [rpm]  
    velocity = magnet_detections*60000 / (millis()-chrono_velocity);
```

Then, the audio signal is sampled at 5000 Hz and converted to millivolts:

```
    // collect kNumCols*kNumRows=8192 samples of the microphone signal  
    while(i < (kNumCols * kNumRows)) {  
        if (micros() - chrono >= kSamplingPeriod_us) {  
            /* If a time > sample time has passed since the last sample  
             * that was recorded -> record a new sample  
             * and store it into the tempSamples vector,  
             * after the conversion to millivolts */  
            sample = digitalSampleToMillivolts(analogRead(A1));  
            tempSamples[i] = sample;  
  
            i++;  
            chrono = micros(); // record the time  
        }  
    }  
}
```

The final step involves the computation of the spectrogram with the FFT algorithm: an Arduino library, `arduinoFFT`<sup>8</sup>, provides ready-to-use functions for this purpose. The FFT is computed over 32 samples at a time: two vectors of length 32 are defined to store the real and imaginary parts of the output. Then, these are passed to the constructor of a `ArduinoFFT` object, as required by the library. All quantities are encoded with `double` precision (64-bit floating point numbers) for improved numerical accuracy in the FFT.

```
    // spectrogram dimensions  
    const uint32_t kNumCols = 32; // number of columns  
    const uint32_t kNumRows = 256; // number of rows
```

---

<sup>8</sup>`arduinoFFT` library documentation: <https://github.com/kosme/arduinoFFT>

```
// sampling rate of microphone signal
const double kSamplingFrequency_Hz = 5000.0;

double vReal[kNumCols];
double vImag[kNumCols];

// create FFT object
ArduinoFFT<double> FFT = ArduinoFFT<double>(vReal, vImag,
                                             kNumCols, kSamplingFrequency_Hz);
```

Inside `loop()`, the FFT is repeatedly computed over the 32 audio samples that are stored in the vector `vReal`, until the whole spectrogram is complete. The Hann window function is applied to avoid spectral leakage<sup>9</sup>, since it provides good overall performance [99]. Finally, the magnitudes of the complex output are computed to be logged to the serial port.

```
// run row-wise FFT and print the output to the serial port
for (uint16_t j = 0; j < kNumRows; j++) { // loop through the rows
    for (i = 0; i < kNumCols; i++) { // loop through the columns
        vReal[i] = tempSamples[j*kNumCols + i];
        vImag[i] = 0;
    }

    // run FFT
    FFT.dcRemoval(); // remove offset
    // Hann window function
    FFT.windowing(FFTWindow::Hann, FFTDirection::Forward);
    FFT.compute(FFTDirection::Forward);
    FFT.complexToMagnitude(); // convert real/imag part to magnitude

    ...
}
}
```

The training dataset of 1350 spectrograms, collected with the application code above by recording the audio of the machine in different working conditions, is stored in 27 `.csv` files, one for each speed label. The `double` precision values are stored with 4 decimal places.

---

<sup>9</sup>When the signal under analysis with the FFT is not an integer multiple of the period, the sharp transitions at the two ends result in discontinuities and thus in high-frequency components in the spectrum. To avoid this effect, the signal can be multiplied by a window function with an amplitude that smoothly reaches 0 at the edges, thus preventing the sharp discontinuity.

## Data Processing

Following the same procedure employed for the previous case study (section 3.3.1), the dataset is processed in *Google Colab* by parsing the `.csv` files, and building the "inputs" and "outputs" numpy arrays to feed to the model for training (see Appendix C.5.1).

In this application, the input to the model is a 2-dimensional tensor, a matrix, therefore "inputs" is a 3D tensor of dimensions  $1350 \times 256 \times 32$  (1350 training examples, each  $256 \times 32$ ), and "outputs" a 1D array of length 1350 containing the speed labels, this being a regression problem and not classification.

At last, the dataset is randomly split into a training, a validation and a test set, with a 70% / 15% / 15% ratio, as more data is available.

```

1 power,raw,speed
2 0.00000,2363.00000,1104.00000
3 4027.03810,2581.00000,1104.00000
4 2869.27510,2553.00000,1104.00000
5 1942.52190,2464.00000,1104.00000
6 957.93290,2263.00000,1104.00000
7 297.84880,1995.00000,1104.00000
8 419.50460,2101.00000,1104.00000
9 439.92190,1952.00000,1104.00000
10 198.55220,1856.00000,1104.00000
11 442.52260,1461.00000,1104.00000
12 378.14250,1652.00000,1104.00000
13 288.38540,1194.00000,1104.00000
14 537.92760,1518.00000,1104.00000
15 233.26700,1546.00000,1104.00000
16 891.81120,1732.00000,1104.00000
17 996.24990,1791.00000,1104.00000
18 226.99690,2162.00000,1104.00000

```

**Figure 4.10:** Example of csv dataset file

## 4.5.2 Machine Learning Model Configuration

As introduced in section 4.4, convolutional neural networks (CNNs) are the most appropriate to handle multidimensional data like spectrograms, while maintaining the spatial relationships between features.

Designing and tuning a neural network architecture is a highly empirical process, especially when hard performance and memory constraints impose that the model is not too large nor too complex, for deployment to a microcontroller-based device. The neural network described in this section is the result of a lengthy and careful design process, during which multiple architectures were compared, striving for the best trade-off between performance and complexity. The goal, however, is to lay the foundations for future research in the field of predictive maintenance, and not to maximize accuracy.

The CNN can be constructed with the **Sequential** Keras API, since only a single input and output tensor are required for each layer. The input to the model is a two-dimensional  $32 \times 256$  tensor, while the output layer comprises a single neuron, whose activation represents the speed prediction, as this is a regression problem.

As discussed in section 3.3.2, one of the most common activation functions nowadays is the ReLU, or rectified linear unit. This is especially true for CNNs, where its low computational complexity and cleanly-defined gradients provide significant advantages in terms of training speed and accuracy [100].

The ReLU is therefore the choice for all layers of the network, including the output, where it is particularly appropriate since it leaves positive speeds unaltered ( $y = x$ , for  $x \geq 0$ ), and brings negative predictions, which are unphysical, to zero.

```
import tensorflow as tf
from tensorflow import keras

# build the model and train it
model = keras.Sequential(
    [
        # maintain the input dimensions, without flattening
        keras.Input(shape=(NUM_ROWS, NUM_COLS, 1)),
        .
        .
        .
        keras.layers.Dense(1, activation='relu') # output layer
    ]
)
```

Three 2D convolutional layers are employed to extract the high-level features from the spectrograms. Each layer, defined with the `keras.layers.Conv2D()`<sup>10</sup> class, is composed of 32 filters for the convolution, striding one pixel at a time. The filter size (`kernel_size`) is decreased the deeper into the network for improved accuracy, and to maintain compatible dimensions with the layers as they are downsampled by three `MaxPooling2D()` operations, that halve the size of the input. The structure is completed with two hidden fully-connected layers of 64 and 32 nodes respectively, that predict the speed on the basis of the features extracted by the convolutional layers.

```
model = keras.Sequential(
    [
        # maintain the input dimensions, without flattening
        keras.Input(shape=(NUM_ROWS, NUM_COLS, 1)),
        keras.layers.Conv2D(32, kernel_size=(9,9), padding='same',
                             strides=1, activation='relu'),
        keras.layers.MaxPooling2D(),
        keras.layers.Conv2D(32, kernel_size=(5,5), padding='valid',
                             strides=1, activation='relu'),
        keras.layers.MaxPooling2D(),
        keras.layers.Conv2D(32, kernel_size=(3,3), padding='valid',
                             strides=1, activation='relu'),
        keras.layers.MaxPooling2D(),

        keras.layers.Flatten(), # flatten to 1D for the Dense layers
        keras.layers.Dense(64, activation='relu'),
```

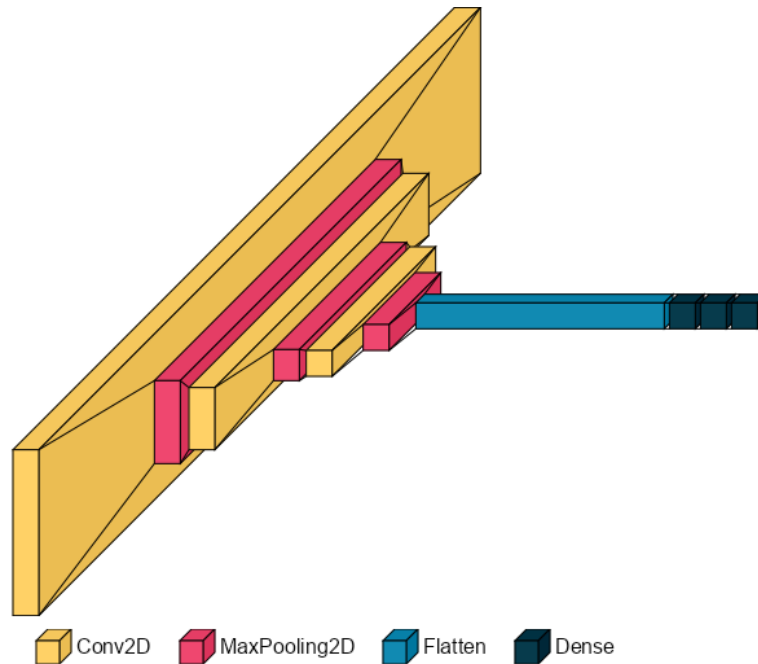
---

<sup>10</sup>Documentation for all Keras classes is available at <https://keras.io/api/>

```

keras.layers.Dense(32, activation='relu'),
keras.layers.Dense(1, activation='relu') # output layer
]
)

```



**Figure 4.11:** CNN Model Architecture<sup>11</sup>

After the definition of its structure, the model requires compiling by selecting the loss function for training and the optimization algorithm.

### Loss Function

In the context of regression problems, the most common loss functions are the *Mean Squared Error* (MSE) and the *Mean Absolute Error* (MAE) [102]:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad \text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

where  $n$  is the number of predictions,  $y_i$  the true values, and  $\hat{y}_i$  the predicted ones

<sup>11</sup>The illustration was created with Gavrikov's utility `visualkeras` (2020) [101].

The former is the preferred loss under the maximum likelihood statistical framework, when the random variable under consideration features a normal distribution. However, this assumption rarely holds in practice, and the squaring operation assigns more importance to larger errors, which could be the result of outliers in the dataset: due to the nature of the application, it is reasonable to expect a significant presence of outliers, as the microphone picks up noise external to the machine. For this reason, the second option of loss function, the Mean Absolute Error, is employed here, since the lack of a squaring operation makes it more robust to values that are far from the mean.

### Optimization Algorithm

The previous case study (section 3.3.2) highlighted the effectiveness of Adam [21] for stochastic gradient descent, thus making it the default choice when selecting the optimization algorithm for this application as well.

```
model.compile(loss=keras.losses.MeanAbsoluteError(),
              optimizer=keras.optimizers.Adam(), metrics=['mse', 'mae'])
```

### 4.5.3 Model Training and Performance Evaluation

Supervised training is carried out in *Google Colab* with a GPU runtime, optimizing the parameters of the CNN over the 950 randomly selected spectrograms that make up the training set.

Since models often benefit from a reduction in learning rate when one or more training metrics stagnate [103], the callback `ReduceLROnPlateau` is included. This monitors the MAE calculated over the validation set, and reduces the learning rate by a factor of 1.25 if no improvement is observed after five epochs, from the initial value of 0.001.

```
rlronp=keras.callbacks.ReduceLROnPlateau(monitor="val_mae",factor=0.8,
                                         patience=5, verbose=1)

history = model.fit(inputs_train, outputs_train, epochs=80,
                   batch_size=32, callbacks=[rlronp],
                   validation_data=(inputs_validate, outputs_validate))
```

The training and validation loss curves (Fig. 4.12) show a rapid decrease during the first 10/20 epochs, with the error stabilizing around 30 rpm on the validation dataset, and 10 rpm on the training set, after 60 epochs. Large oscillations can be observed in the initial stages, because of the higher learning rate.

This figure validates the CNN design as the final accuracy is well within the requirements set out in section 4.3 (maximum prediction error  $\pm 50$  rpm), and

the loss appears to have converged. Though no significant overfitting is present, training was also repeated with a  $L_2$  regularization term in the cost function, which, however, resulted in no appreciable improvement in terms of variance.

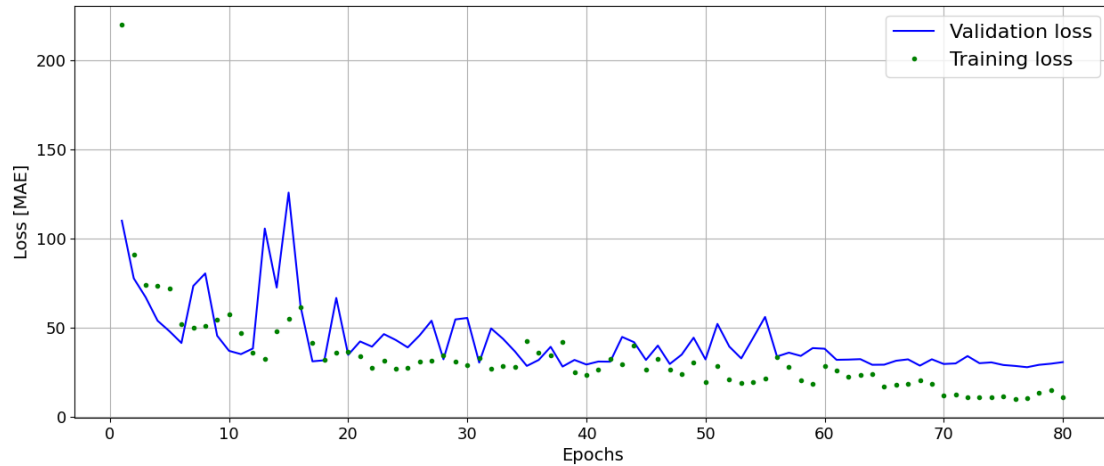


Figure 4.12: Training and Validation Loss

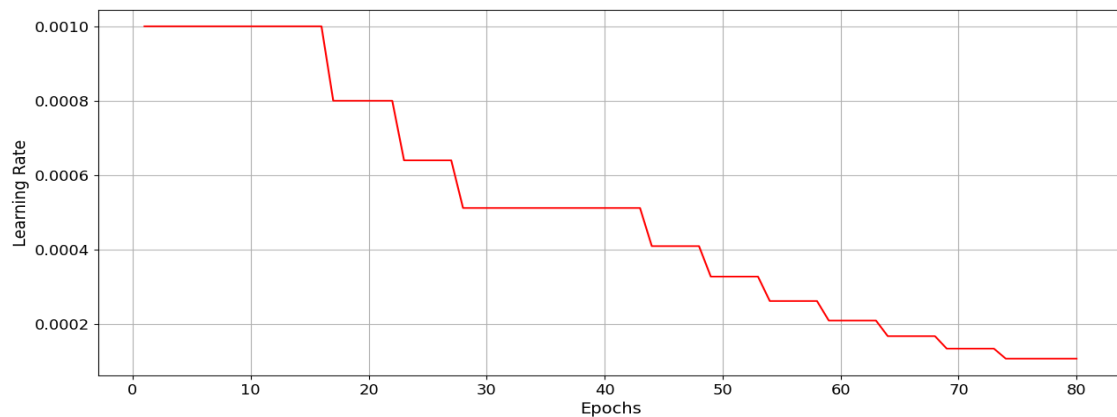


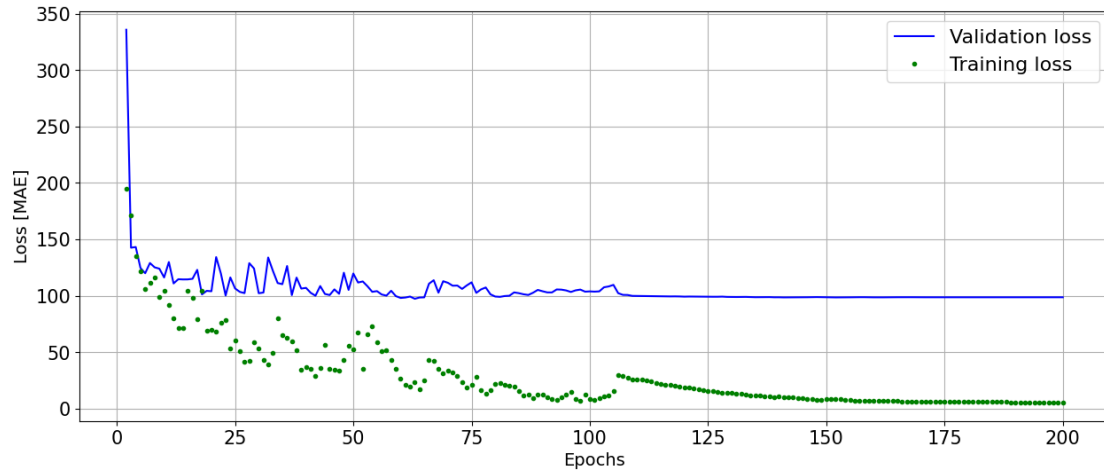
Figure 4.13: Learning Rate Decay

### Importance of the Spectrogram Representation

To highlight the importance of encoding the audio data as a spectrogram, despite the code overhead that this entails, training is repeated with a dataset composed of the raw voltage recordings. The CNN needs to be modified accordingly, substituting all `Conv2D` and `MaxPooling2D` layers with `Conv1D` and `MaxPooling1D`.

From figure 4.14 it is immediate to conclude that the generalization ability of the resulting model is much worse, with the MAE calculated over the validation set plateauing at 100 rpm, indicating a large variance.





**Figure 4.14:** Training and Validation Loss - Raw Voltage

However, thanks to the ability of convolutional neural networks to maintain the temporal relationships between samples, these results are not completely without merit, and it is likely that they could be improved with a deeper network.

### Final Model Testing

Moving back to the CNN trained to recognize the spectrogram of the audio signal, a final performance evaluation phase is conducted over the test dataset, the remaining 15% of the collected spectrograms, before deployment to the Finder Opta™. The resulting mean absolute error of **28.7 rpm** confirms the model's ability to generalize to unknown inputs, and ensures that no overfitting of the validation set was inadvertently introduced when tuning the hyperparameters.

	Training Set	Validation Set	Test Set
MAE [Loss]	10.8 rpm	30.5 rpm	28.7 rpm
MSE	208.3 rpm <sup>2</sup>	1627 rpm <sup>2</sup>	1575 rpm <sup>2</sup>

**Table 4.2:** Evaluation metrics

## 4.6 Model Deployment to the Finder Opta™

To deploy the trained convolutional neural network to the Finder Opta™ PLC, it requires conversion to the TensorFlow Lite format for size reduction and optimization (section 1.2.2). Alongside this process, the embedded application to acquire the audio samples, compute their spectrograms and run inference with the model

needs to be designed, building on the infrastructure discussed in section 3.4.2, and on the application for the training data collection in section 4.5.1.

### 4.6.1 TensorFlow Lite Conversion

Thanks to the TensorFlow Lite Converter’s Python API, the model can be encoded as a *FlatBuffer*, applying post-training full-integer quantization to achieve a smaller footprint in terms of memory and resource utilization. Unlike the model employed for case study I, the convolutional neural network for sound recognition approaches the limitations of the Finder Opta™ hardware, and deployment would not be feasible without converting all floating-point quantities to fixed-point integers.

```
# convert the model to the TensorFlow Lite format with QUANTIZATION
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
```

Full-integer quantization requires a representative dataset, to estimate the typical range of variation of the variables involved. To this end, it is convenient to provide the converter with the test dataset, defining function `representative_dataset_generator()`, as it contains a sufficient number of examples to completely characterize the input.

```
# provide the test set as representative dataset (15% of the data)
def representative_dataset_generator():
    for example in inputs_test:
        yield [np.array(example, dtype=np.float32,
                        ndmin=4).reshape((1, NUM_ROWS, NUM_COLS, 1))]

converter.representative_dataset = representative_dataset_generator
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS,
                                       tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
```

Lastly, `int8` (fixed-point 8-bit integer) is specified as the target datatype for all internal quantities, while `float32` are still used for the input and output tensors. This ensures better future compatibility with traditional applications, as `float32` is the most common type for tensors, and the Finder Opta™ is not limited to integer-only operations.

```
# full-integer quantization: only int8
converter.target_spec.supported_types = [tf.int8]
# set both input and output tensors to float32
converter.inference_input_type = tf.float32
converter.inference_output_type = tf.float32
```

The model can be finally converted with the specified optimization, and saved as a byte array to be embedded in an Arduino header file. The resulting size is 1 029 kB, which leaves just enough memory available for the rest of the code.

```
tflite_model = converter.convert() # perform conversion

# save to a file
open("model.tflite", "wb").write(tflite_model)

!echo "const unsigned char model[] = {" > /content/model.h
!cat model.tflite | xxd -i >> /content/model.h
!echo "};" >> /content/model.h
```

## 4.6.2 Arduino Application

Following the same steps taken in section 3.4.2, it is now time to develop the embedded Arduino application to run inference with the model. As usual, it must include the infrastructure required by the TensorFlow Lite library for the interpreter to execute, along with the pre-processing architecture to sample the audio signal and compute its spectrogram, and the post-processing flow to analyze the output of the model.

### TensorFlow Lite for Arduino Infrastructure

The discussion about the basic TensorFlow Lite for Arduino infrastructure is available in section 3.4.2, and thus is not repeated here. The entire code is reported in the Appendix C.6 for reference.

The only difference with respect to the previous case study is the amount of memory allocated for the tensor operations. The input, in particular, takes up a large amount of space, since it is made of 8192 floating-point values in a 2D array. By trial and error, it was determined that 358.4 kB are sufficient, with this amount being just short of the maximum memory available (recall how the choice of the spectrogram size in section 4.5.1 was made with such memory limitations in mind).

```
constexpr int tensorArenaSize = 350 * 1024; // 358.4 kB
byte tensorArena[tensorArenaSize] __attribute__((aligned(16)));
```

### Application Flowchart

Figure 4.15 reports the flowchart of the application, that includes the input pre-processing flow, matching the algorithm employed in the training data collection (Fig. 4.9), the inference with the model, and the output handling architecture.

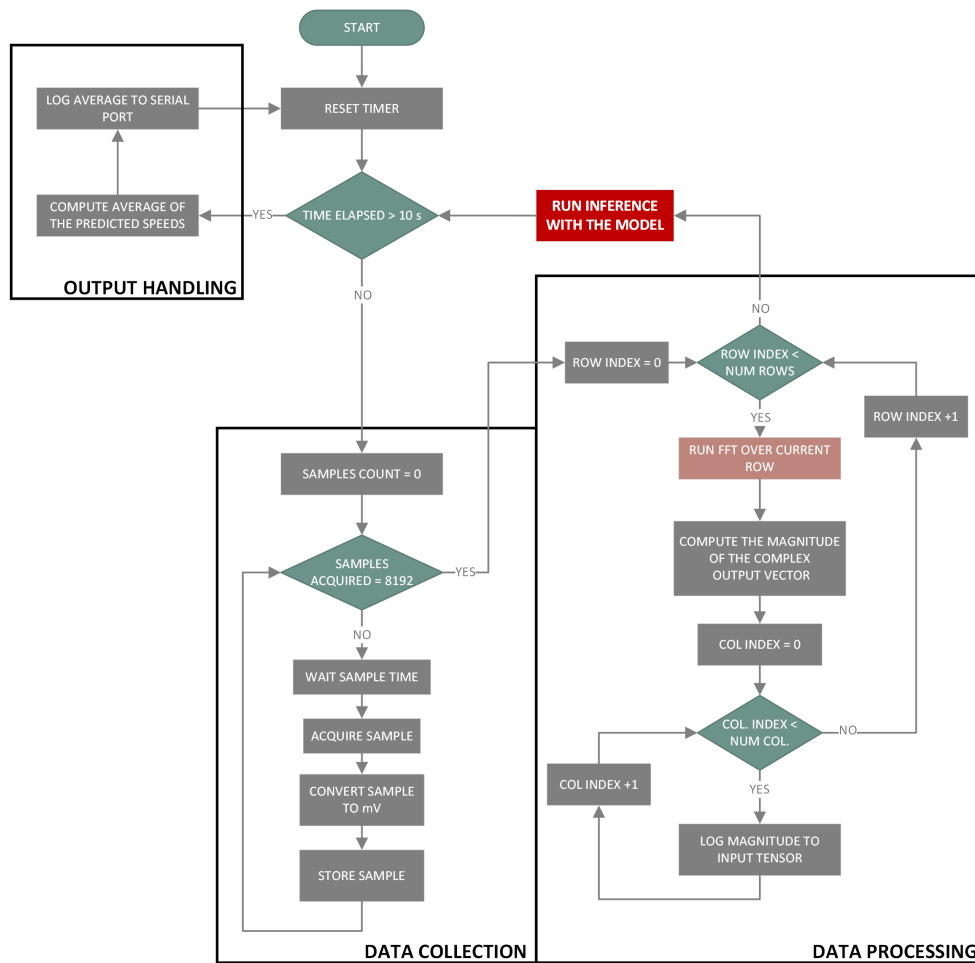


Figure 4.15: Arduino Application Flowchart

**Input Pre-Processing:** this part of the algorithm replicates exactly the procedure employed in the training data collection (section 4.5.1), sampling the audio signal and computing its spectrogram with the FFT to feed to the model. The same sampling frequency (5000 Hz) and spectrogram dimensions ( $32 \times 256$ ) of the training dataset are mandatory for coherent inferences.

Measuring the speed of the shaft with the magnetic sensor is no longer a part of the "official" flowchart, since it is the model's job to output the speed. However, the speed sensor remains a useful tool for debugging purposes and to estimate the final accuracy of this experiment.

**Output Handling:** the output tensor contains a single value, which represents the "instantaneous" speed inference in rpm. In principle, no other operation, other

than transmitting it to the serial port, is required. However, to improve robustness to temporary glitches in the input data, that could be the result of foreign noise on the plant floor, it is advantageous to calculate the average inference over a certain time frame (10 seconds), in order to smooth out any disturbance.

## Application Code

The code starts off with the same configuration of the training data collection, since the `ArduinoFFT` library is employed again to compute the spectrogram. The code to acquire the speed measurements from the inductive sensor is maintained, to later evaluate the performance of the neural network.

```

/* ---- CONFIGURATION ---- */
const uint8_t kAnalogReadResolution = 16; // resolution of A1

// time interval to average the speed measurement
const unsigned long kVelocityCalculationInterval_ms = 10000;
// debouncing time of the magnetic digital speed sensor in ms
const unsigned long kDebounceInterval_ms = 50;

// spectrogram dimensions
const uint32_t kNumCols = 32; // number of columns
const uint32_t kNumRows = 256; // number of rows

// sampling time of microphone signal
const double kSamplingFrequency_Hz = 5000.0;
unsigned int kSamplingPeriod_us = 200;

/* ----- */

// variables to store time
unsigned long chrono; // sample microphone signal
unsigned long chrono_velocity; // sample speed sensor signal
// for debouncing of speed sensor
unsigned long last_detection_time = 0;

// counter for number of magnet detections
// (volatile because updated inside ISR)
volatile uint32_t magnet_detections = 0;

// to perform the average of multiple inferences
double running_sum; // sum for the avg
uint32_t sum_elements_count; // number of entries for the avg

// vectors to store the raw microphone samples and the complex

```

```

// number after FFT (real/imag parts)
double tempSamples[kNumCols * kNumRows];
double vReal[kNumCols];
double vImag[kNumCols];

// create FFT object
ArduinoFFT<double> FFT = ArduinoFFT<double>(vReal, vImag,
      kNumCols, kSamplingFrequency);

```

As usual, the `setup()` function instantiates the neural network model stored as a byte array in `model.h`, builds the interpreter and allocates the memory for the tensor operations, before configuring the input pins A0 and A1, along with the interrupt to acquire the speed measurements from the inductive sensor.

```

void setup() {
  // initialize serial communication
  Serial.begin(115200);
  while(!Serial); // wait for serial port

  /* ---- Setup the Machine Learning Infrastructure ---- */
  // map model into a usable data structure
  tflModel = tflite::GetModel(model);
  if (tflModel->version() != TFLITE_SCHEMA_VERSION) {
    // raise error if model version is incompatible
    Serial.print("ERROR: Model is schema version ");
    Serial.print(tflModel->version());
    Serial.print(", not equal to supported version ");
    Serial.println(TFLITE_SCHEMA_VERSION);

    while(1); // endless loop to stop execution
  }

  // build the interpreter to run the model
  static tflite::MicroInterpreter static_interpreter(tflModel,
    tflOpsResolver, tensorArena, tensorArenaSize, &tflErrorReporter);
  tflInterpreter = &static_interpreter;

  // allocate memory for the tensors
  TfLiteStatus allocate_status = tflInterpreter->AllocateTensors();
  if (allocate_status != kTfLiteOk) {
    // raise error if allocation fails
    Serial.print("ERROR: AllocateTensors() failed.");
    while(1); // endless loop to stop execution
  }
}

```

```
// obtain pointers to input and output tensors
tflInputTensor = tflInterpreter->input(0);
tflOutputTensor = tflInterpreter->output(0);

/* ---- Setup the Arduino Peripherals ---- */
pinMode(A0, INPUT); // A0 -> digital input port for velocity sensor
pinMode(A1, INPUT); // A1 -> analog input port for microphone signal
analogReadResolution(kAnalogReadResolution);

// configure interrupt for pin A0 to detect
// falling edge of velocity sensor
attachInterrupt(digitalPinToInterrupt(A0),
                count_magnet_detections_ISR, FALLING);
}
```

The `loop()` function implements the three main tasks outlined in the flowchart: the input pre-processing, the inference, and the output handling.

At each iteration, the function first checks whether 10 seconds have elapsed since the last output to the serial port: if this is the case, it needs to compute a new average, by taking the ratio of the sum of the inferences in the last 10 seconds (`running_sum`) to how many of them have been output by the neural network (`sum_elements_count`). To evaluate the accuracy, the average speed measured by the inductive proximity sensor is calculated at the same time (thus also over 10s), and transmitted to the serial port.

```
void loop() {

    double sample;
    uint16_t speed;

    uint16_t i = 0; // counter for the number of microphone samples

    // if more than 10s have elapsed from last output: record the speed
    // measurement from the sensor for debugging, print the average of
    // the inferences over the 10s, and reset the average
    if (millis() - chrono_velocity > kVelocityCalculationInterval_ms) {
        // debugging: record the current speed
        speed = compute_velocity();

        // log actual and average predicted speed to serial port
        Serial.print("Average predicted speed: ");
        Serial.print(running_sum / sum_elements_count); // compute average
        Serial.println(" rpm");
        Serial.print("Measured speed: ");
        Serial.print(speed);
    }
}
```

```

Serial.println(" rpm");
Serial.println();

// initialize average
running_sum = 0;
sum_elements_count = 0;
}

```

The following section of code matches the input pre-processing implemented for the training data collection (section 4.5.1), with the sampling of the audio signal, and calculation of the FFT. However, instead of transmitting the resulting spectrogram to the serial port, it is saved, one row at a time, to the input tensor via its `data` variable, accessing the memory reserved for floating-point quantities (`.f`): recall how the model was quantized to use integer values internally, but the input and output tensor maintain a `float32` representation.

```

// collect kNumCols*kNumRows=8192 samples of the microphone signal
while(i < (kNumCols * kNumRows)) {
    if (micros() - chrono >= kSamplingPeriod_us) {
        /* If a time > sample time has passed since the last sample
        * that was recorded -> record a new sample
        * and store it into the tempSamples vector,
        * after the conversion to millivolts */
        sample = digitalSampleToMillivolts(analogRead(A1));
        tempSamples[i] = sample;

        i++;
        chrono = micros(); // record the time
    }
}

// run row-wise FFT
for (uint16_t j = 0; j < kNumRows; j++) { // loop through the rows
    for (i = 0; i < kNumCols; i++) { // loop through the columns
        vReal[i] = tempSamples[j*kNumCols + i];
        vImag[i] = 0;
    }

    // run FFT
    FFT.dcRemoval(); // remove offset
    // Hann window function
    FFT.windowing(FFTWindow::Hann, FFTDirection::Forward);
    FFT.compute(FFTDirection::Forward);
    FFT.complexToMagnitude(); // convert real/imag part to magnitude
}

```



```

for (uint16_t idx_col = 0; idx_col < kNumCols; idx_col++) {
    // save magnitude out of FFT into the input tensor
    tflInputTensor->data.f[j*kNumCols+idx_col] = vReal[idx_col];
}
}

```

As soon as the input tensor is filled with the entire spectrogram, it is possible to invoke the interpreter to obtain a speed inference, and add it to the running sum for the average.

```

// invoke the interpreter to run inference
TfLiteStatus invokeStatus = tflInterpreter->Invoke();
if (invokeStatus != kTfLiteOk) {
    // raise an error if invoke fails
    Serial.println("ERROR: Invoke failed.");
    while (1); // endless loop to stop execution
}

// add element to running sum for the average
running_sum += tflOutputTensor->data.f[0];
sum_elements_count++; // update the entries counter
}

```

### 4.6.3 Preliminary Application Unit Testing

A rapid testing procedure for the application is implemented by evaluating the behavior of its three main blocks - the input processing, the inference and the output handling - while the test bench operates at 200 rpm, 500 rpm, 900 rpm and 1400 rpm. The focus is on

ruling out any major issues in the application code, ensuring that all stages of the pipeline, from the sampling of the audio signal and the FFT to the calculation of the average inference, perform as expected. Instead, in the following section, 4.7, the objective will be to collect a systematic dataset to assess the final performance of the model, in real operating conditions.

A first high-level test involves setting the desired speeds in the control panel (Fig. 4.16), and monitoring the serial port for the average inference and the measurements

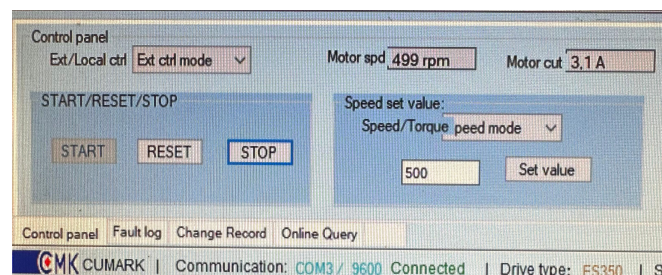
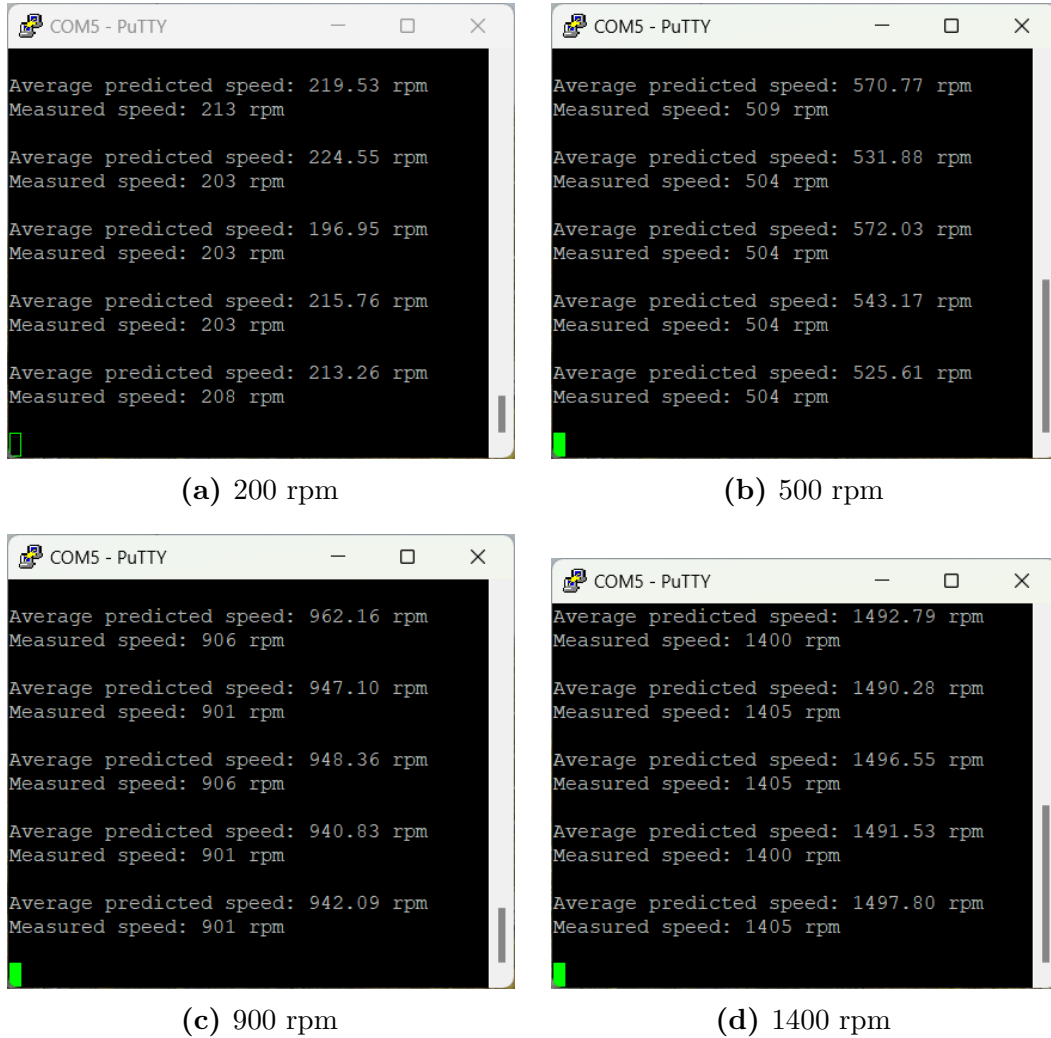


Figure 4.16: Test Bench Speed Control

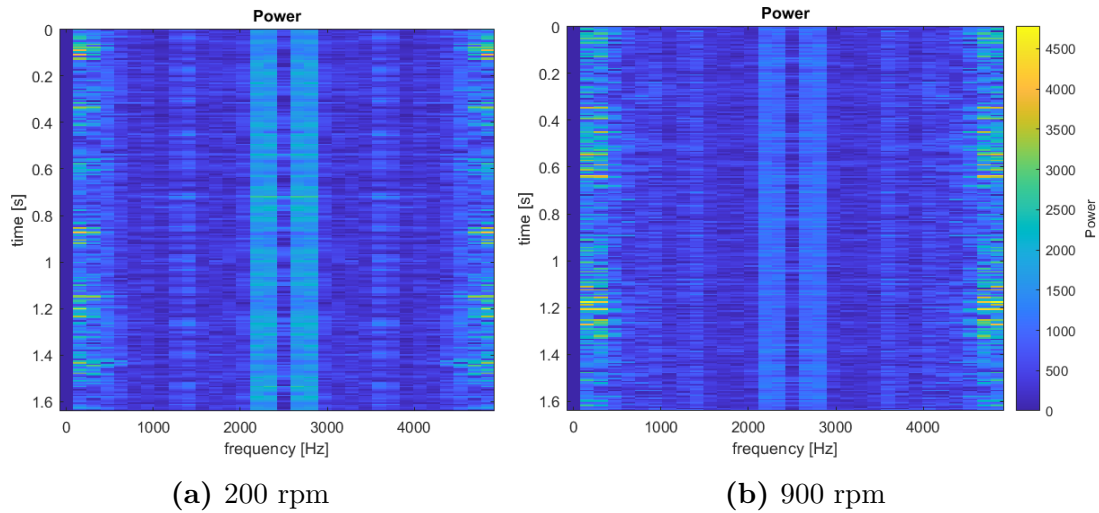
from the inductive sensor. As illustrated in Fig. 4.17, the results are reasonable and consistent with the selected values of speed.



**Figure 4.17:** CNN Application Unit Testing - Serial Port

A test of the pre-processing flow is conducted next, by transmitting the content that gets stored in the input tensor to a PC. The  $256 \times 32 = 8192$  samples are analyzed and processed with MATLAB, confirming that the tensor represents the spectrogram of the original raw audio recording (Fig. 4.18).

Finally, the output handling architecture is evaluated by broadcasting each single inference to the serial port, along with the average, to confirm the latter is calculated correctly, as shown in Fig. 4.17.

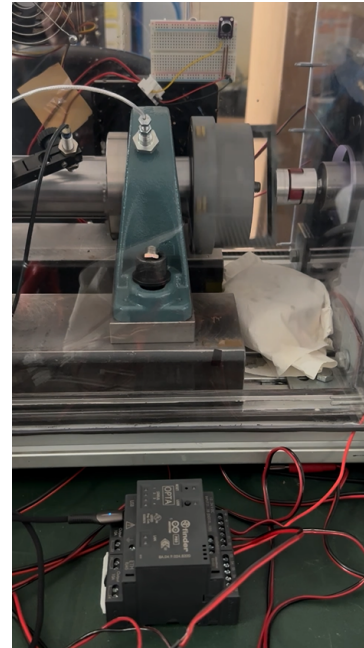


**Figure 4.18:** CNN Application Unit Testing - Input Tensor Content

## 4.7 Overall System Testing and Conclusions

After confirming the absence of major software faults in the application, it is possible to move forward with the last phase of the workflow: the systematic evaluation of the model performance once deployed in real operating conditions.

With the test bench running on the laboratory floor, ten inferences are acquired for each speed category used in the training dataset (see section 4.5.1 - totaling 270 inferences). For a representative test, the measurements are taken at different times over several days, to ensure an average level of background noise comparable to that encountered during training data collection. Additionally, a random subset of inferences is conducted with artificial disturbances, such as human speech close to the microphone or a siren going off in the background. The goal is to compute the Mean Absolute Error across the collected dataset, and verify it remains below the **50 rpm** threshold mandated by Requirement 1, in section 4.3. As mentioned in that section, this does not constitute a comprehensive investigation of the model's robustness to external noise,



**Figure 4.19:** Test Bench during Testing

which will be carried out in a future study to address Requirement 2.

Appendix C.7 reports the values of inference collected during the testing activity, along with the measured speed from the inductive sensor for reference. It can be observed that the predictions are consistently above the measured values (the mean errors reported next to each speed category are all positive), suggesting the presence of a systematic bias in the model. Finding the root cause can be the subject of a future investigation, perhaps while expanding the training dataset to attempt to resolve the issue.

The Mean *Absolute* Error (MAE) for each speed category is reported in Table 4.3. No apparent trend can be identified here, with consistent performance across the entire speed range 200 – 1500 rpm. The overall MAE is calculated at **42.1 rpm**, below the 50 rpm requirement. This value is only 47% larger than the MAE of 28.7 rpm displayed by the original CNN model before deployment (section 4.5.3). This demonstrates the effectiveness of the optimization techniques put in place for execution on the Opta™, and the ability of neural network models to reject outliers, in particular when specific countermeasures, such as taking the average over the outputs, are implemented.

<b>200 rpm</b>	56.1	<b>650 rpm</b>	29.5	<b>1100 rpm</b>	59.7
<b>250 rpm</b>	29.9	<b>700 rpm</b>	33.4	<b>1150 rpm</b>	42.2
<b>300 rpm</b>	35.8	<b>750 rpm</b>	36.3	<b>1200 rpm</b>	45.0
<b>350 rpm</b>	28.3	<b>800 rpm</b>	45.1	<b>1250 rpm</b>	59.1
<b>400 rpm</b>	53.2	<b>850 rpm</b>	32.2	<b>1300 rpm</b>	41.2
<b>450 rpm</b>	22.6	<b>900 rpm</b>	34.5	<b>1350 rpm</b>	42.0
<b>500 rpm</b>	36.1	<b>950 rpm</b>	62.1	<b>1400 rpm</b>	47.7
<b>550 rpm</b>	46.5	<b>1000 rpm</b>	36.5	<b>1450 rpm</b>	40.9
<b>600 rpm</b>	60.5	<b>1050 rpm</b>	44.0	<b>1500 rpm</b>	36.3

<b>Overall MAE</b>	<b>42.1 rpm</b>
--------------------	-----------------

**Table 4.3:** Overall System Testing - MAE [rpm] per Speed Category

This case study has successfully demonstrated the feasibility of deploying somewhat complex neural network models to industrial edge devices like PLCs, to enable more *intelligent* manufacturing and maintenance practices. Moreover, a system of this kind is straightforward to retrofit into already existing applications, lowering the bar to implement innovative solutions in industrial settings.

This work has illustrated extensively the effectiveness of representing the audio

signal as a spectrogram, and how the convolutional layers are the most appropriate to process image-like information, all with on-device tools thanks to the integration of the Opta™ with the Arduino ecosystem.

In addition to analyzing robustness systematically, and investigating the source of the bias on the speed predictions, future developments include steps in the field of predictive maintenance: by collecting a dataset for anomalous sound detection, it will be possible to monitor the rotating parts' health, and take the most efficient maintenance decisions with a relatively inexpensive setup.

# Chapter 5

## Case Study III: Application to a Wave Energy Generator

In addition to the novel applications in the field of machine learning and AI, the Finder Opta™ PLC can be an interesting tool in more traditional industrial control problems as well, given the flexibility provided by the Arduino ecosystem, its connectivity options and relatively low cost.

This case study explores an application of the Opta™ to an offshore wave energy generator, and the advantages it brings in terms of remote monitoring and control, predictive operational safety (exploiting weather forecasts fetched through the internet), and robustness to the harsh field conditions, all at a limited cost, a key factor in the prototyping stage.

### 5.1 Experimental Setup

Figure 5.1 shows a basic scheme of the prototype of wave energy generator to be controlled. What follows is a brief description of its components and working principle, with a special focus on the tasks that are going to be carried out on the Opta™ to understand how it fits into the final assembly, since the aim of this work is not to describe the inner workings of the prototype device itself.

Electrical energy is generated thanks to a floating device with an optimized shape (A), that is set in an alternating linear motion by the sea waves. A suitable mechanism (GEAR) transforms the linear motion into a regular one-way rotary motion, in order to actuate a 3-phase electrical generator.

During operations, the Finder Opta™ monitors the electrical power generation by acquiring the output of a power meter, along with the position of the float via two limit switches, a0 and a1. Because the system is designed to be deployed in a remote environment, this information shall be displayed on a virtual HMI, accessible via the internet. This is the first reason why the Opta™ is a suitable choice: the flexibility that it provides in terms of connectivity and programming

languages allows for a straightforward setup of remote communication.

The Opta™ is also in charge of the emergency operations: a small independent electrical motor, that actuates a pulley system, shall be controlled in order to raise or lower the float as necessary. The main threat to the device is when the energy of the sea waves exceeds the design limitations of the float, due to adverse sea and wind conditions. The connectivity capabilities of the Opta™ allow addressing such issue with a predictive approach: by connecting to the internet and periodically downloading the relevant marine weather forecast, it is possible to automatically lift the float when the height of the sea waves is predicted to rise above the emergency threshold, and lower it again when sea conditions return to normal, all without manual intervention, enhancing safety.

Moreover, the virtual HMI can also be designed to monitor the emergency state of the system, allowing for manual remote control, as discussed in more details in the dedicated sections below.

The outlined tasks constitute the backbone of the control logic of the device, to ensure a minimum level of operational safety during the prototyping and testing phase. More advanced functionalities could be incorporated in the design at a later stage, with the Opta™ Over-The-Air (OTA) feature even allowing for remote updates without a physical cable connection, a particularly useful capability given the working conditions of the generator. The aim of the case study is to showcase how a smart PLC can be integrated into an industrial application of this kind, and all the advantages it provides in terms of connectivity and information management, improving decision-making for more efficient operations.

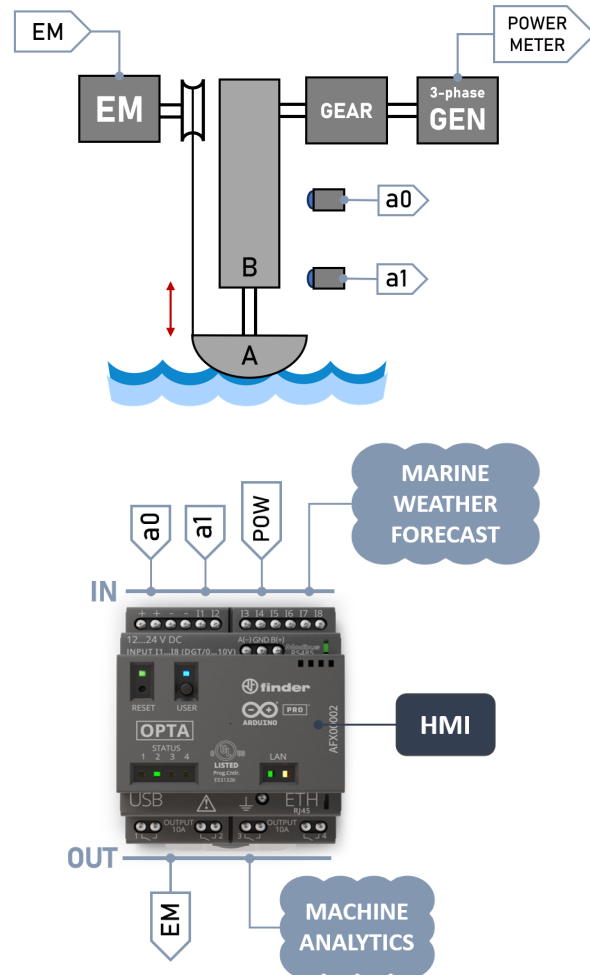


Figure 5.1: Wave Energy Generator Control Scheme

## 5.2 Internet Connectivity and Infrastructure

This section explores how the Finder Opta™ can connect to the internet, delving into the libraries and code that serve as infrastructure for the rest of the application. Throughout the case study, software development is carried out on the Arduino IDE, exploiting the Arduino programming language. This guarantees a greater degree of flexibility, despite the Opta™ also supporting the IEC 61131-3 languages, more suited to technicians familiar with traditional PLCs.

As the version name suggests, the **Opta™ Wi-Fi** features a 2.4 GHz Wi-Fi module (see Table 2.1), which is employed to connect to the internet. A basic example of how this can be achieved thanks to the Arduino `WiFi.h` library is available in chapter 2.

This case study involves advanced remote control and monitoring, which would require a complex web service and backend infrastructure for communication with the Opta™. Instead, at least in this initial stage, it is more convenient to exploit **Arduino Cloud**<sup>1</sup>, a tool in the Arduino ecosystem which offers an online framework to remotely monitor and control IoT devices, with a built-in HMI called **Dashboard**. The service provides a streamlined configuration process, taking care of communication with the device behind the scenes, thus enabling complex cloud solutions even for technicians with minimal coding expertise. Whereas relying on a third-party service for a safety-critical function is not ideal, it reduces complexity, allowing for the prototyping to move forward at a quicker pace, even when the limitations of the Arduino Cloud platform<sup>2</sup>, which mainly targets amateur projects, are taken into account. In the meanwhile, an in-house solution can be developed in the background, with functional safety in mind.

### 5.2.1 Workflow and Requirements

Below are the main steps and requirements for the internet infrastructure.

1. Configure the Arduino Cloud "Thing" for communication.
2. Implement the code required by Arduino Cloud in an Arduino sketch.
3. Implement a parallel thread on the Opta™ to manage a LED indicator for the connection status, exploiting the RTOS features of Mbed OS (see section 2.2):
  - built-in **GREEN** LED steady: connected to the internet

---

<sup>1</sup><https://cloud.arduino.cc/how-it-works/>

<sup>2</sup>Different subscription plans are available, with the free one characterized by several limitations in terms of number of devices, dashboard size, storage space, API access, etc.



- built-in **RED** LED steady: connection failed

4. Verify the connection, and perform unit testing of the application developed up to this point.

Point 3 mandates the use of a parallel thread to manage the indicator LEDs: this is to showcase the RTOS features that Mbed OS brings to the Finder Opta™, albeit not strictly necessary here.

Note how this and the following sections include snippets of the application code on the basis of the features being discussed, and thus not necessarily in the correct order. To avoid losing sight of the larger picture, Appendix D.1 reports the entire application code, as deployed to the Opta™.

## 5.2.2 Arduino Cloud Connection

Arduino Cloud features an online IDE, a cloud backend service to synchronize data from the Arduino board, and a graphical dashboard tool for remote control, accessible via desktop or mobile app.

After including the Opta™ in the list of devices on Arduino Cloud, the first item to address is the setup of a "Thing", i.e. a virtual twin of the board with the project configuration details. The Thing requires an associated device, and contains the network credentials, along with all the shared variables required for communication between the Opta™ and the cloud.

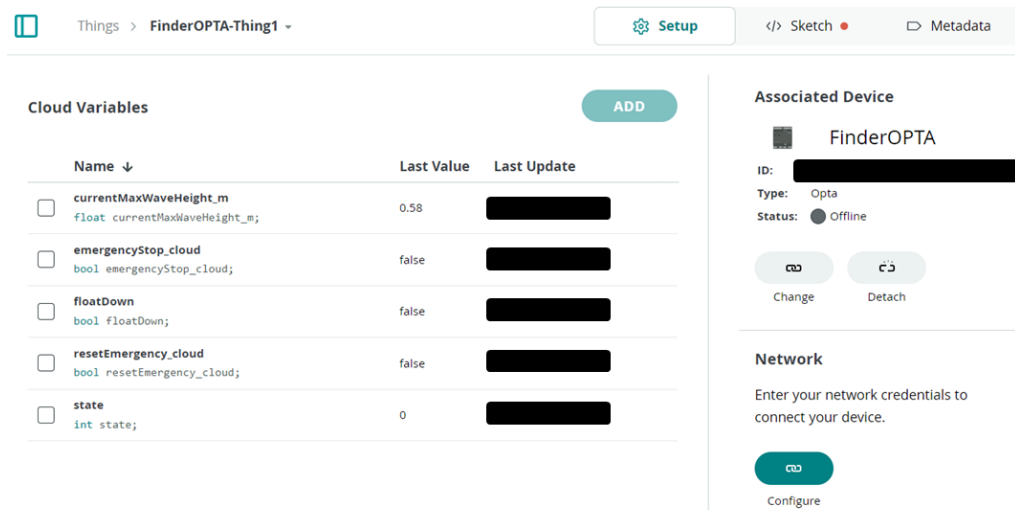


Figure 5.2: Arduino Cloud Thing

## Arduino Code

When creating a Thing, a sketch template is automatically generated with all the libraries and functions to handle the connection to the internet and the communication with the Arduino Cloud backend.

A function, `initProperties()`<sup>3</sup>, called right after the configuration of the serial port in `setup()`, initializes the shared variables: these can be read-only, when they do not need to be changed by the cloud, or read-write, if some dashboard elements need to control their values; when a read-write variable is declared, a callback function is automatically generated in the template, to specify what actions to execute when its value is modified by the cloud. The update frequency shall also be specified, choosing between `ON_CHANGE` and a time-based approach. More details on the actual shared variables in this case study are available in section 5.4.

The connection to the internet and Arduino Cloud is initialized by the `begin()` method of the `ArduinoIoTCloud` library, passing the configuration object with the Wi-Fi network details. The `update()` method called inside `loop()` maintains and updates the connection: it is important that `update()` is repeatedly called without delay, else the timeout of a watchdog<sup>4</sup> timer causes the connection to reset.

```
#include <ArduinoIoTCloud.h>
#include <Arduino_ConnectionHandler.h>

const char SSID[] = "****"; // Network SSID (name)
const char PASS[] = "****"; // Network password

void initProperties(){
    ArduinoCloud.addProperty(currentMaxWaveHeight_m, READ, ON_CHANGE, NULL);
    ArduinoCloud.addProperty(floatDown, READ, ON_CHANGE, NULL);
    ArduinoCloud.addProperty(state, READ, ON_CHANGE, NULL);
    ArduinoCloud.addProperty(emergencyStop_cloud, READWRITE, ON_CHANGE,
        onEmergencyStopCloudChange);
    ArduinoCloud.addProperty(resetEmergency_cloud, READWRITE, ON_CHANGE,
        onResetEmergencyCloudChange);
}

WiFiConnectionHandler ArduinoIoTPreferredConnection(SSID, PASS);
```

---

<sup>3</sup>Reference for all libraries: <https://www.arduino.cc/reference/en/libraries/>

<sup>4</sup>A watchdog timer is a common safety mechanism in microcontrollers to monitor the software execution and reset the processor in case of unexpected behaviors [104]. Since the Opta™ is a Mbed OS based board, watchdog timers can be implemented leveraging the `Watchdog` class. Instead, the watchdog mentioned above is automatically started by the `begin()` method of the `ArduinoIoTCloud` library, for the purposes of monitoring the connection.

```
void setup() {
  // Initialize serial port and wait for it to open:
  Serial.begin(115200);
  delay(1500);

  initProperties();

  // Connect to Arduino IoT Cloud
  ArduinoCloud.begin(ArduinoIoTPreferredConnection);

  setDebugMessageLevel(2);
  ArduinoCloud.printDebugInfo();
}

void loop() {
  ArduinoCloud.update();
}

/* emergencyStop_cloud is READ_WRITE variable:
 * onEmergencyStopCloudChange() is executed every time
 * a new value is received from IoT Cloud. */
void onEmergencyStopCloudChange() { }

/* resetEmergency_cloud is READ_WRITE variable:
 * onResetEmergencyCloudChange() is executed every time
 * a new value is received from IoT Cloud. */
void onResetEmergencyCloudChange() { }
```

### 5.2.3 Mbed OS Parallel Thread - LEDs Management

The last item to address in this section, before moving to the core functionalities of the application, is the parallel thread to manage the Opta™ LEDs.

It was decided to exploit the multithreading<sup>5</sup> ability of Mbed OS on the powerful dual-core Cortex-M processor to showcase this capability, and make the visual indications completely independent of the main application code: if, for example, the connection drops and the main `loop()` is stuck trying to reconnect to the internet, it would not be reflected promptly by the LEDs, but only after a certain amount of time, when the CPU finally frees up and executes the code responsible

---

<sup>5</sup>Multithreading is the ability of a CPU to run multiple parallel threads, i.e. sequences of instructions managed independently by the scheduler of the operating system [105].

for such changes. Instead, when the thread runs in parallel it is autonomous, and not subject to other blocking tasks.

Figure 5.3 reports the flowchart to implement in the parallel thread, with a steady green/red LED to indicate the connection status (section 5.2.1, requirement 3). This thread also manages the built-in blue LED, that serves as an indicator for the emergency state, when blinking with a 0.5 s period (listed as a requirement in section 5.3.1, dedicated to the emergency operations).

### Arduino Code

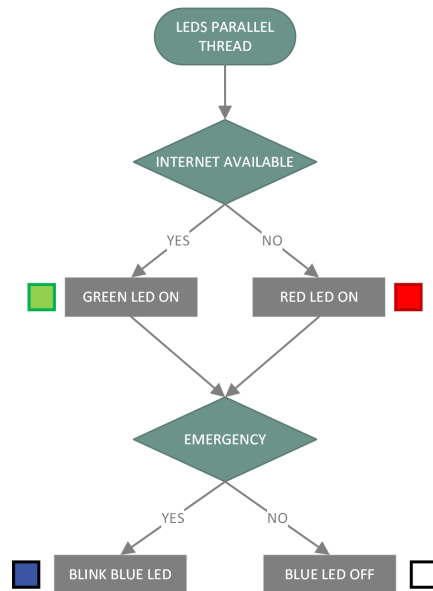
An instance of the `Thread`<sup>6</sup> class from the RTOS API is created to allocate the new thread to run concurrently to the main `loop()`: `ledsManagementThread`. This is then started inside `setup()` when the Opta™ is powered on, by executing the callback `manageOptaLeds()`. Its definition implements the instructions of the flowchart in an infinite loop: function `WiFiStatusLED()` controls the built-in green/red LED on the basis of the connection status, while `blinkLED()` actuates the blue LED when the emergency is active (indicated by the shared variable `state`, see section 5.3).

```
#include <mbed.h> // for parallel threads
#include <rtos.h>
```

```
// allocate parallel thread - LED management
static rtos::Thread ledsManagementThread;
```

```
void setup() {
```

```
    pinMode(LED_USER, OUTPUT); // blue LED, blinking when emergency active
    pinMode(LED_R, OUTPUT); // red LED, on when WiFi connection failed
    pinMode(LED_BUILTIN, OUTPUT); // green LED, on when connected to WiFi
```



**Figure 5.3:** MbedOS Parallel Thread LED Management Flowchart

<sup>6</sup>Documentation: <https://os.mbed.com/docs/mbed-os/v6.16/apis/thread.html>

```

// Start the parallel thread
ledsManagementThread.start(manageOptaLeds);
}

/* ----- PARALLEL THREAD ----- */
/* Callback to manage the built-in LEDs of the Opta */
void manageOptaLeds() {
  while (1) {
    /* To provide a visual indication of the Wi-Fi status:
     * - steady green LED (LED_BUILTIN): Wi-Fi connected
     * - steady red LED (LEDR): Wi-Fi connection failed */
    WiFiStatusLED();

    if (state == 1) { // if normal operations (no emergency)
      digitalWrite(LED_USER, LOW); // turn off the blue LED
    } else { // if emergency
      // blink the emergency blue LED with a period of 500 ms
      blinkLED(LED_USER, 500);
    }
  }
}
}

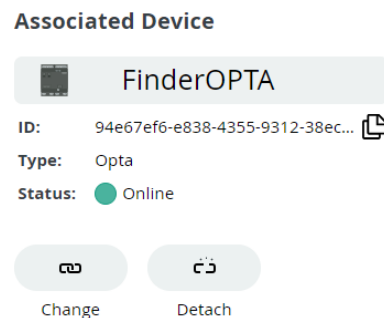
```

### 5.2.4 Unit Testing

It is always good practice to perform continuous testing during development, subdividing the application into smaller units to promptly identify any fault, instead of a single final testing phase. Hence, it is time to verify that the Opta™ can successfully connect to Arduino Cloud, and that the LEDs do reflect the connection status.

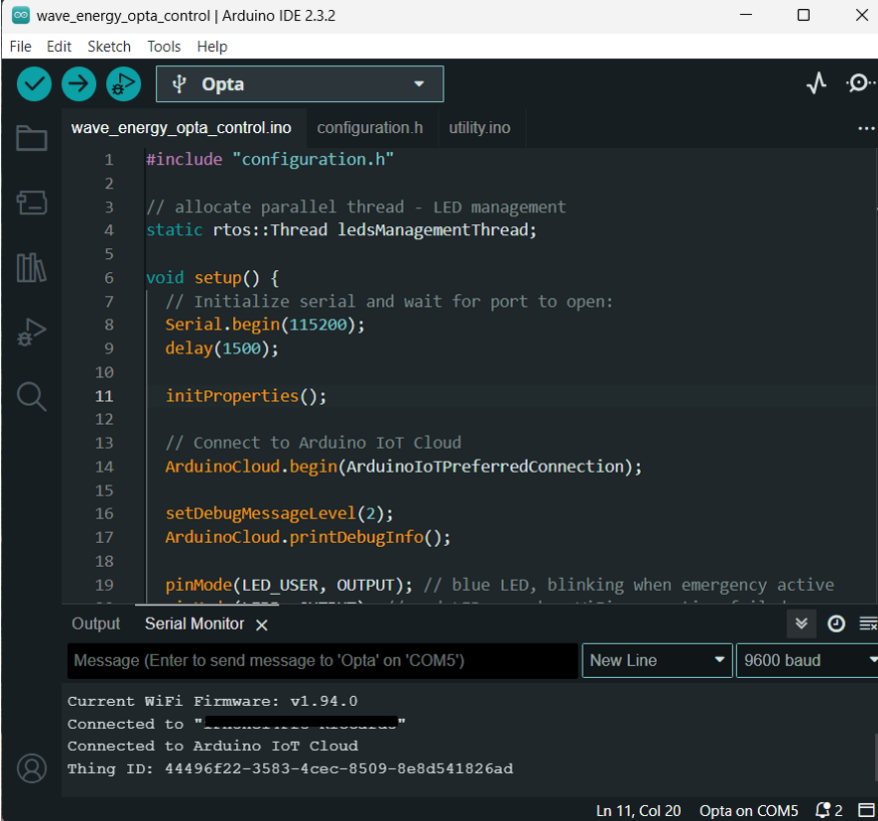
After compiling the code and deploying it to the Opta™, the device connected to the internet right away via the Wi-Fi network specified during configuration: this is indicated by the green light in the Arduino Cloud Thing page (Fig. 5.4), and by the diagnostic messages received through the serial port (Fig. 5.5).

As expected, the built-in green LED signals the presence of an internet connection (Fig. 5.7a). Note how the blue LED is also on, and blinking: as described in section 5.3, the shared integer variable `state` being equal to 0 denotes that the emergency is active; since no logic has been



**Figure 5.4:** Thing Connected

developed to manage `state` thus far, it is initialized to 0, simulating the presence of an emergency. The blue LED, therefore, displays the intended behavior.



The screenshot shows the Arduino IDE interface with the Serial Monitor open. The code in the background includes a setup function that initializes the serial port, connects to the Arduino IoT Cloud, and sets up a blue LED. The Serial Monitor output shows the following messages:

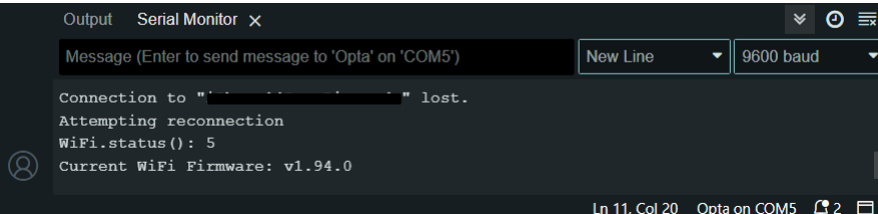
```

Current WiFi Firmware: v1.94.0
Connected to "XXXXXXXXXX"
Connected to Arduino IoT Cloud
Thing ID: 44496f22-3583-4cec-8509-8e8d541826ad

```

**Figure 5.5:** Arduino IDE Serial Monitor - Connection Successful

If the Wi-Fi network is disabled, the `update()` method in the main `loop()` fails to synchronize with the cloud and gets stuck (Fig. 5.6): the watchdog thus reboots the device repeatedly to attempt to reestablish a connection. Thanks to the parallel thread, the LEDs are not subject to any of these operations, with the red light promptly indicating that the connection is absent (Fig. 5.7b).



The screenshot shows the Serial Monitor output with the following messages:

```

Connection to "XXXXXXXXXX" lost.
Attempting reconnection
WiFi.status(): 5
Current WiFi Firmware: v1.94.0

```

**Figure 5.6:** Arduino IDE Serial Monitor - No Connection

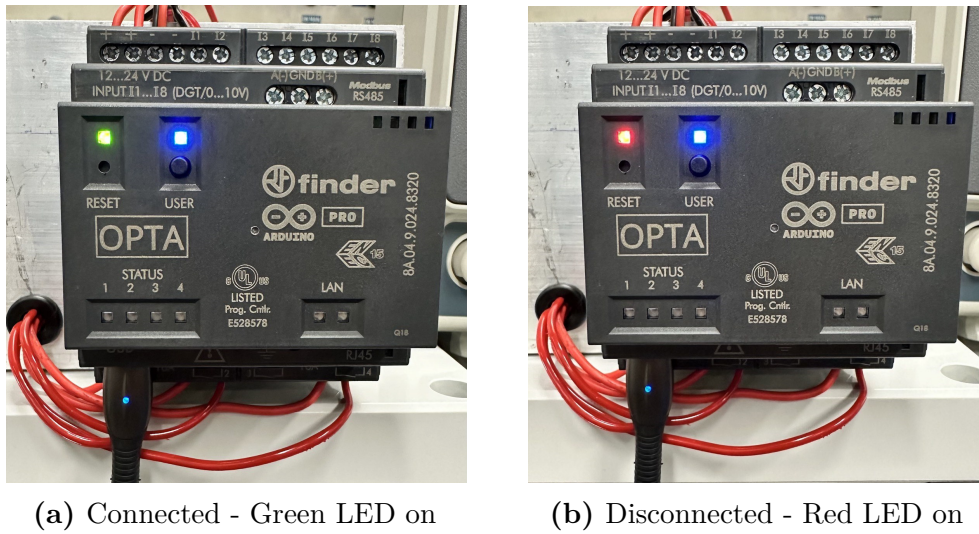


Figure 5.7: Opta™ LEDs Testing

Despite not constituting a comprehensive testing activity, with systematic verification of all statements and conditions in the code, it is sufficient to conclude that no major fault is present, and that the connection to the cloud functions as intended.

## 5.3 Emergency Operations

The core task of the Opta™ in this case study is to guarantee the system's safety, preventing damage due to excessive stress on the floating device. This is achieved by actuating the emergency motor via the integrated output relays, thus lifting the float out of the water when necessary, as illustrated in the scheme of Fig. 5.1.

The capabilities of the Opta™ in terms of internet connectivity allow adopting a predictive approach, relying on weather forecasts to make the optimal decision about the suspension of normal operations and the retraction of the floating device. In particular, the PLC shall periodically download the forecasted wave height for the operational site from a reputable web service, analyzing the data to actuate the emergency motor when a set threshold is surpassed.

In addition to these automatic safeguard measures, manual intervention shall also be provided for: the Arduino Cloud service proves convenient here, allowing for remote control and monitoring of the emergency operations.

### 5.3.1 Requirements

Below are summarized the key requirements to implement, to provide a broad overview before the detailed analysis of the following sections.

1. When entering the emergency state, the Opta™ shall activate the emergency motor by energizing relay D1, thus lifting the float.
2. A blinking **BLUE** LED shall indicate that the emergency is active.
3. The emergency state shall be entered if **any** of the following conditions is met:
  - the 12-hour forecast of the wave height indicates a value above the set threshold;
  - a 12-hour forecast of the wave height is not available;
  - the emergency is manually commanded by the operator (either via a physical pushbutton or remotely).
4. Normal operations shall be resumed if **all** of the following conditions are met:
  - the 12-hour forecast of the wave height does not indicate a value above the set threshold;
  - a 12-hour forecast of the wave height is available;
  - the *Reset Emergency* command is issued by the operator (either via a physical pushbutton or remotely), in case the emergency was manually activated.
5. When the emergency is reset, the Opta™ shall reverse the emergency motor by energizing relay D0, and lower the float back into the water.

### 5.3.2 Marine Weather Forecast

The internet infrastructure developed in section 5.2 allows for an easy connection process to any web service providing marine weather forecasts: `open-meteo.com` is selected among the several available options because of its high-resolution open-source API<sup>7</sup>, which is free for non-commercial purposes. By sending periodically a forecast request through the web service API (thus employing the HTTP<sup>8</sup> methods), and decoding the response, it is possible to monitor the evolution of the wave height at the operational site.

The flowchart (Fig. 5.8) outlines the main steps in order to meet the requirements laid out in section 5.3.1. The first item to address is the interval between two forecast update requests: one hour is deemed appropriate because of the slow variations that the sea wave height experiences.

---

<sup>7</sup>API ("Application Programming Interface"): software interface for different components and services to communicate, exchanging information.

<sup>8</sup>"Hypertext Transfer Protocol", the foundation of the World Wide Web, a protocol for information transfer between the networked devices [106].



Moreover, forecasts are downloaded for a time horizon of three days, despite the shorter 12-hour requirement: this provides the operator with more advance notice of future weather conditions, and improves robustness to temporary connection outages, as the system can rely on previously downloaded data instead of entering the emergency as soon as an update fails.

### API Request

The HTTP method to retrieve information from a service, without modifying it in any way, is `GET` [106].

Reported below is the complete `GET` request issued to the `marine-api.open-meteo.com` server, following the syntax specified in the documentation<sup>9</sup>: in this prototyping stage, it is formulated for a generic location with coordinates `44.3N 8.72E`; these can be easily modified as soon as the operating site is finalized. As per the design choices, the forecast consists of the hourly wave height (`hourly=wave_height`) for three days (`forecast_days=3`).

```
GET /v1/marine/?latitude=44.3&longitude=8.72&current=wave_height&hourly
    =wave_height&timezone=Europe%2FBerlin&forecast_days=3 HTTP/1.1;
Host: marine-api.open-meteo.com
Connection: close
```

The server returns the requested information formatted as a JSON<sup>10</sup> object that requires parsing: once again, the Arduino ecosystem offers an official pre-built library to process JSON (`Arduino_JSON.h`), streamlining the coding process. Below is a shortened response sample: the initial key/value pairs report some generic information about the request; then is a JSON array with the current time and wave height, followed by the array of height predictions (for example, it is

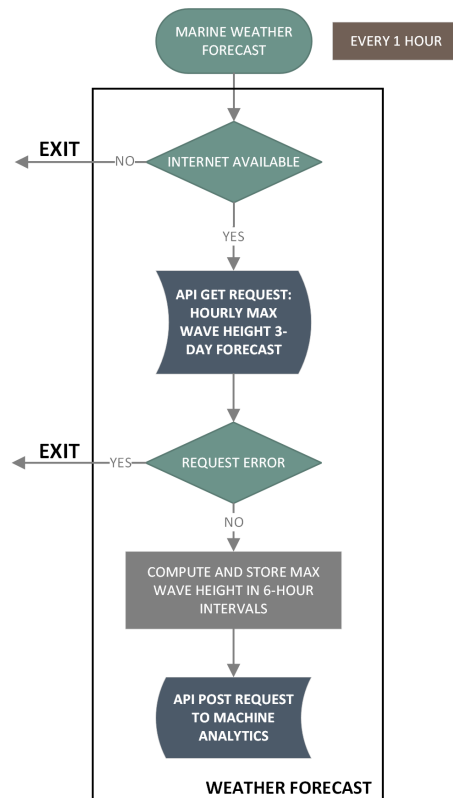


Figure 5.8: Marine Weather Forecast Flowchart

<sup>9</sup><https://open-meteo.com/en/docs/marine-weather-api>

<sup>10</sup>"JavaScript Object Notation", an open standard to exchange information formatted in a human-readable manner as key/value pairs, very common in web applications [107].

forecasted that at 22:00 local time, on September 15<sup>th</sup>, 2024, the wave height at 44.25N 8.75E is going to be 0.44 m).

```
{
  "latitude": 44.25,
  "longitude": 8.75,
  "generationtime_ms": 0.0929832458496094,
  "utc_offset_seconds": 7200,
  "timezone": "Europe/Berlin",
  "timezone_abbreviation": "CEST",
  "elevation": 0,
  "current_units": {
    "time": "iso8601",
    "interval": "seconds",
    "wave_height": "m" },
  "current": {
    "time": "2024-09-13T10:00",
    "interval": 3600,
    "wave_height": 0.46 },
  "hourly_units": {
    "time": "iso8601",
    "wave_height": "m" },
  "hourly": {
    "time": [
      "2024-09-13T00:00",
      "2024-09-13T01:00",
      ....
      "2024-09-15T22:00",
      "2024-09-15T23:00"
    ],
    "wave_height": [1.06, 1.01, ... , 0.44, 0.46] }
}
```

### Arduino Code

The Arduino main loop calls function `fetchWaveHeight()` every 60 minutes to update the weather forecasts, provided that an internet connection is present. This function returns `-1` if unsuccessful, or the API request time (in `hh` format) if no errors occur. The fetched forecasts are then logged to an online *Google* spreadsheet (Fig. 5.8) as part of the remote monitoring effort, with a new API request issued by function `logToSpreadsheet()`, and a custom Google Apps Script to interpret the data (see section 5.4.3 for further details).

```
void loop() {
  // Fetch an update to the wave height forecast every
```

```

// kWeatherUpdateInterval_min if connected to the internet
if (millis() - prevWeatherFetchAttempt > (unsigned long)
    (kWeatherUpdateInterval_min*60000) && WiFi.status()==WL_CONNECTED) {

    // fetch forecast and save it in max_wave_height_6h[]
    fetchWeatherReturn = fetchWaveHeight(max_wave_height_6h);

    if (fetchWeatherReturn >= 0) { // if no errors (-1 if error)
        fetchHour = fetchWeatherReturn; // update the fetch time (hour)
        // log the forecast to Google Sheet
        logToSpreadsheet(max_wave_height_6h,
            (24 / kIntervalLength_h * kForecastLength_days));

        // save the time of last SUCCESS in updating the forecast
        prevWeatherFetchSuccess = millis();
    }
    // save the time of last ATTEMPT at updating the forecast
    prevWeatherFetchAttempt = millis();
    ...
}
}

```

`fetchWaveHeight()` sends the GET request, parses the response, and saves the height forecasts in an array, passed by reference to the function.

This array (`maxWaveHeight []`) needs to contain 12 elements, as the hourly wave height data is processed by dividing the 3-day horizon into intervals of 6 hours ( $\frac{24\text{h} \times 3\text{ days}}{6\text{h}} = 12$ ), and saving the maximum forecast for each 6-hour period. Such intermediate step reduces the amount of information that needs storing - always a worthy endeavor in memory-constrained applications - and formats the forecasts in accordance to the requirements.

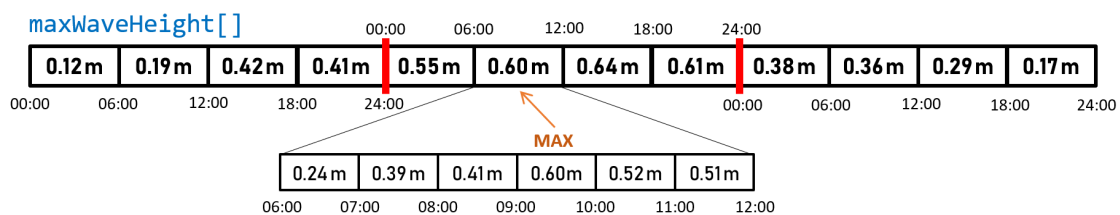


Figure 5.9: Forecast Data Processing in 6-hour Intervals

The connection to the server can be established thanks to `client`, an instance of the `WiFiSSLClient` class. As the name suggests, this class supports the SSL protocol, to guarantee privacy, security and message integrity, contrary to a plain HTTP connection.

```

WiFiSSLClient client; // client object to connect using SSL

int8_t fetchWaveHeight(double maxWaveHeight[]) {
    // initialize return value:
    // negative value -> ERROR in fetching the weather data
    // positive value -> the hour the weather data was fetched
    int8_t returnCurrentHour = -1;

    double coordinates[2] = {44.3, 8.72}; // desired coordinates
    double maxCoordinateError = 0.3;

    // open-meteo API server to fetch marine weather forecasts
    const char* openMeteoServer = "marine-api.open-meteo.com";
    // API endpoint path and query to get required data
    String path = "/v1/marine";
    String queryCoordinates = "?latitude=" + (String) coordinates[0] +
        "&longitude=" + (String) coordinates[1];
    String queryParameters = "&current=wave_height&hourly=wave_height"
        "&timezone=Europe%2FBerlin&forecast_days=3";
    // connect to server (port 443: SSL)
    if (client.connect(openMeteoServer, 443)) {
        // send HTTP GET request if connection successful
        client.print("GET " + path + queryCoordinates + queryParameters);
        client.println(" HTTP/1.1");
        client.print("Host: ");
        client.println(openMeteoServer);
        client.println("Connection: close");
        client.println();
    }
}

```

As mentioned, the `Arduino_JSON.h` library provides useful tools to process the JSON response, such as the `parse()` method or the `JSONVar` class, just to name a couple. After skipping the HTTP header, the first task is to check whether the forecast location is reasonably close to the desired site, and return an error otherwise. Then, the hourly forecasts are copied to a temporary array, and processed as detailed above, by saving only the maximum value for each 6-hour interval. Finally, the request hour is extracted from the JSON object, and returned by the function to signal a correct execution.

```

// skip HTTP header
char startOfJson = '{';
client.find(startOfJson);

// parse JSON response
JSONVar doc; // JSON object to store the response
String payload = "{" + client.readStringUntil('\n');

```

```
doc = JSON.parse(payload);

// check is parsing was successful
if (JSON.typeof(doc) == "undefined") {
    Serial.println("- Parsing failed (open-meteo)!");
    client.stop();

    returnCurrentHour = -1; // error: return -1
    return returnCurrentHour;
}

// parse latitude and logitude to check location
double latitude = doc["latitude"];
double longitude = doc["longitude"];
// check if coordinates of the fetched weather are reasonably
// close to desired location
if (abs(latitude - coordinates[0]) > maxCoordinateError ||
    abs(longitude - coordinates[1]) > maxCoordinateError) {
    Serial.println("- Coordinates error!");
    client.stop();

    returnCurrentHour = -1; // error: return -1
    return returnCurrentHour;
}

// parse JSON array containing the forecast for wave height
JSONVar hourly_wave_height = doc["hourly"]["wave_height"];
// declare an array to store the forecast and initialize it to 0
double wave_height_forecast[hourly_wave_height.length()];
memset(maxWaveHeight, 0, sizeof(maxWaveHeight));

// go through the hourly forecasts in batches of 6 hours, and
// save the max for each 6h period into maxWaveHeight
for (int i = 0; i < hourly_wave_height.length()/6; i++) {
    for (int j = 0; j < 6; j++) {
        // go through the hourly forecasts
        wave_height_forecast[i*6+j] = hourly_wave_height[i*6+j];
        // if the value is > than the max stored, update it
        if (wave_height_forecast[i*6+j] > maxWaveHeight[i]) {
            maxWaveHeight[i] = hourly_wave_height[i*6+j];
        }
    }
}

// parse the current time, format "2024-05-03T15:00",
```

```
// extracting the hour (-> "15")
JSONVar current = doc["current"];
const char* current_time = current["time"];
char current_time_only_hour[] = {current_time[11],
    current_time[12], '\0'};

// convert the time string to an integer
returnCurrentHour = atoi(current_time_only_hour);

} else {
    Serial.println("- Failed to connect to server (open-meteo)!");
    returnCurrentHour = -1; // error: return -1
}
client.stop();

return returnCurrentHour;
}
```

### 5.3.3 State Machine

Thanks to the logic implemented thus far, the Opta™ has access to the maximum wave height at its operational site, with a forecast horizon of three days. Based on this information, it needs to decide when to enter or exit emergency operations, and what actions to carry out depending on the system's state, following the requirements in section 5.4.1.

The system's behavior can be modeled as a basic finite state machine [108] with two states: **NORMAL** and **EMERGENCY**. The scheme in Fig. 5.10 reports the actions to be performed in each state, as only one can be active at any given time. The transition between them is governed by the wave height forecast and any manual intervention by the operator.

Every 60 minutes, the Opta™ shall check the current time, and verify whether a 12-hour forecast is available (for example, in Fig. 5.10, the Opta™ shall check whether a wave height value is available for the interval 18:00 – 24:00, and for 00:00 – 06:00 the following day): if this is not the case, because of prolonged internet outages or other faults in the connection process, the emergency state shall be entered, as safe operations can no longer be guaranteed. Instead, if a 12-hour forecast is available, the Opta™ shall check whether the two wave height values are below the safety threshold - set to 1 m at this stage - and enter the emergency otherwise. It is possible to exit the emergency at any time, provided the forecast falls below the threshold.

All of the above is carried out automatically, every 60 minutes, without the need for manual intervention. If, instead, user input is required, and the operator commands

the emergency state, the Opta™ shall comply regardless of the height of the waves. In this case, the operator manually commands the "Reset Emergency" to return to normal operations, in addition to the forecasts indicating a safe wave height.

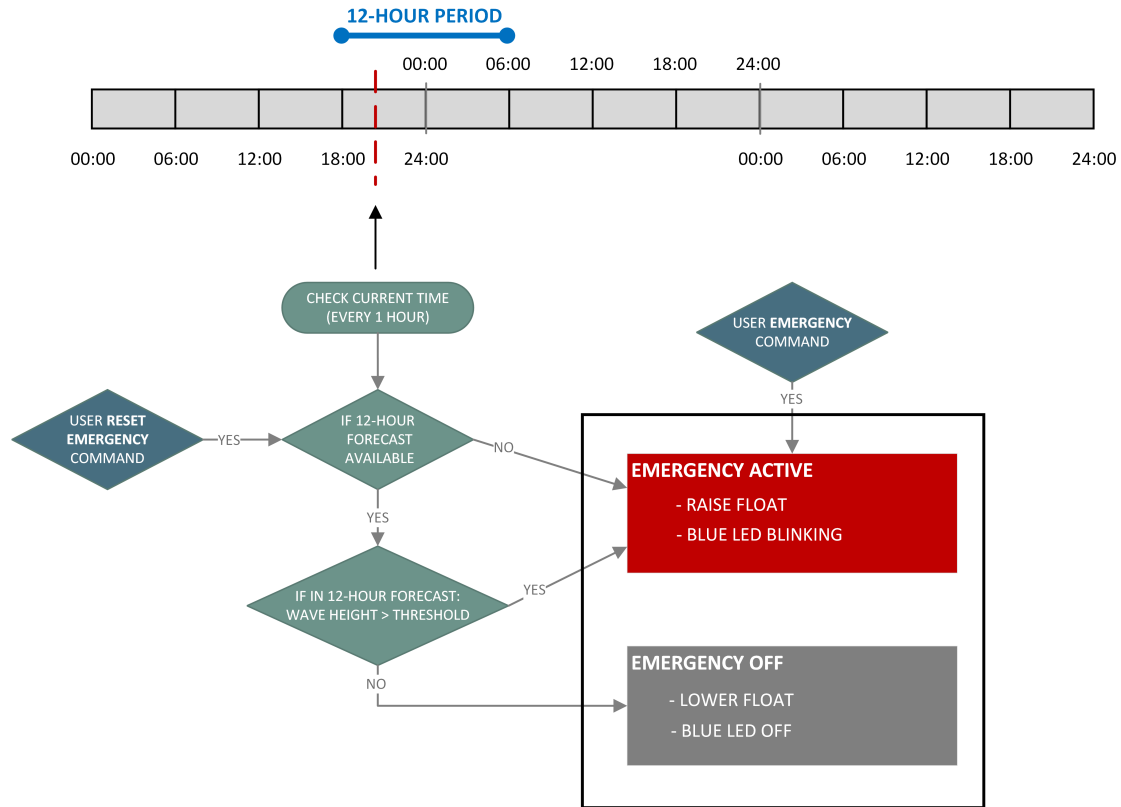


Figure 5.10: State Machine Scheme

## Arduino Code

The integer variable `state` is employed to keep track of the state the system:

`state = 1` → NORMAL  
`state = 0` → EMERGENCY

This is also an Arduino Cloud shared variable, configured as `READ_ONLY`, to display the state of the system on the remote dashboard (further details on the shared variables and the Arduino Cloud HMI are available in section 5.4).

The conditional statements to manage the value of `state` mainly rely on three flags, that indicate whether a transition condition has been met. These are:

- `emergencyWaveHeight`: equal to 1 when the wave height is above the threshold in the time frame of interest, and 0 otherwise;
- `emergencyPB` and `resetEmergencyPB`: equal to 1 when the operator commands the emergency state or the "Reset Emergency", respectively.

The code to control the last two flags is related to the remote monitoring of the system via Arduino Cloud, thus it is described in section 5.4.

```
// Flags to activate or reset emergency
// (volatile because handled by Interrupt Service Routine)
volatile uint8_t emergencyPB; // user command
volatile uint8_t resetEmergencyPB; // user command
uint8_t emergencyWaveHeight;
```

Per the state machine scheme (Fig. 5.10), the main loop extracts the current value of wave height and the prediction for the next 6 hours from the array `maxWaveHeight[]`, thanks to custom function `findWaveHeight()`. This process occurs every 60 minutes, concurrently with each update attempt (note how it is not necessary that the weather update succeeds: `findWaveHeight()` can rely on previously downloaded forecasts as long as they cover the time intervals of interest). If either value is found to be above the threshold, or the unphysical value of  $-1$  signals the lack of a prediction, flag `emergencyWaveHeight` is set to 1 (true).

```
void loop() {
  /* ----- WAVE HEIGHT FORECAST ----- */
  // Fetch an update to the wave height forecast every
  // kWeatherUpdateInterval_min if connected to the internet
  if (millis() - prevWeatherFetchAttempt > (unsigned long)
      (kWeatherUpdateInterval_min*60000) && WiFi.status()==WL_CONNECTED) {

    ...

    // extract the forecast for the current 6h interval and next 6h
    currentMaxWaveHeight_m = findWaveHeight(0);
    next6hMaxWaveHeight_m = findWaveHeight(1);
    // if wave height in current 6h period or in next 6h > threshold
    // (or < 0: ERROR) --> ACTIVATE EMERGENCY */
    if (currentMaxWaveHeight_m >= kEmThreshold_m ||
        next6hMaxWaveHeight_m >= kEmThreshold_m ||
        currentMaxWaveHeight_m < 0 || next6hMaxWaveHeight_m < 0) {
      emergencyWaveHeight = true;
    } else {
      emergencyWaveHeight = false;
    }
  }
}
```



The proper state machine actions and transitions are coded with a `switch` structure inside `loop()`. Depending on the active state, the position of the float is controlled by functions `raiseFloat` and `lowerFloat`, that manage the relays of the Opta™ as necessary. Each state also features a conditional statement to transition to the other state, if all requirements are met.

```
/* --- STATE MACHINE ACTIONS (EMERGENCY / NORMAL) --- */
switch(state) {
  /* Normal State */
  case 1:
    /* Check if the float is in the working position.
     * If NOT activate the relay to lower it */
    lowerFloat();

    /* Transition condition NORMAL -> EMERGENCY:
     * if the emergency is commanded by operator or
     * wave height > threshold --> ENTER EMERGENCY */
    if (emergencyPB || emergencyWaveHeight) {
      state = 0;
    }
    break;

  /* Emergency State */
  case 0:
    /* Check if the float is in the safe position.
     * If NOT activate the relay to raise it */
    raiseFloat();

    /* Transition condition EMERGENCY -> NORMAL:
     * - if the emergency pushbutton was pressed, check it is released
     *   and the "Reset Emergency" command was issued;
     * - check if wave height is below threshold
     * The emergency can only be reset if the float is in the SAFE
     * (limit switch A6 to signal if the float is raised) */
    if (!emergencyWaveHeight && (!emergencyPB || resetEmergencyPB)
        && digitalRead(A6)) {
      state = 1;
      resetEmergencyPB = false;
      emergencyPB = false;
    }
    break;

  default:
    /* If state NOT 0 or 1: enter emergency by setting it to 0 */
    state = 0; // enter emergency
}
```

```

    break;
}
}

```

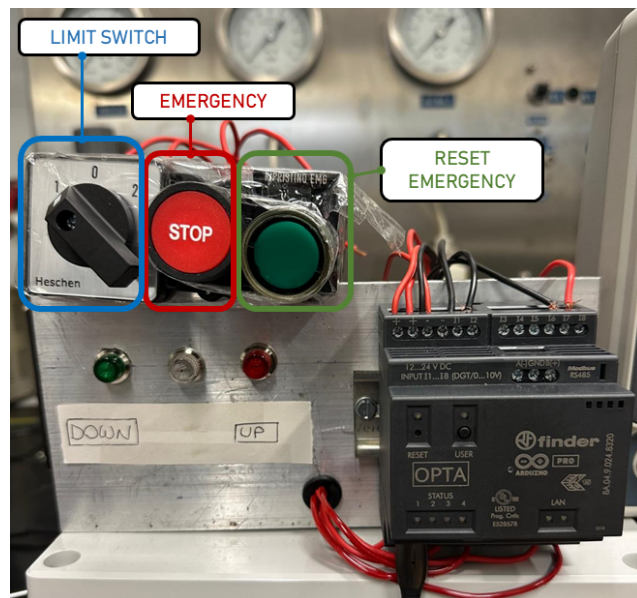
Any `state` different from 0 or 1 is invalid, and indicates a software error which triggers the activation of the emergency.

The use of an integer and of a flexible `switch` structure ensures code scalability, making the addition of new operational states quite straightforward, without extensive code modifications.

### 5.3.4 Unit Testing

A testing phase must now be conducted to verify that the implementation of the emergency logic complies with the requirements, with no major software fault.

Since the remote HMI has yet to be developed, a temporary test bench is arranged to test the state machine behavior (Fig. 5.11), repurposing the setup employed in chapter 3.1.2. Moreover, the marine weather forecasts are transmitted to the PC via serial port for debugging purposes.



The two physical pushbuttons are used to issue the manual commands ("Emergency Stop" or "Reset Emergency"), while the three-way selector simulates the limit switches of the floating device (position 1: float DOWN, position 2: float UP). To enhance the visual feedback, the position of the float is also indicated by two pilot lights: the green one signals that the float is lowered into the water, in normal operating conditions, while the red one denotes that the float is safe, raised out of the sea.

#### INPUT PINS

11 (A0)	Emergency Stop
12 (A1)	Reset Emergency
17 (A6)	Float UP
18 (A7)	Float DOWN

#### OUTPUT RELAYS

1 (D0)	Lower Float
2 (D1)	Raise Float

**Figure 5.11:** Emergency Test Setup and I/O

The input pins of the Opta™ are interrupt-enabled (see section 2.3), which represents the most reliable approach to acquiring the input from the pushbuttons: by issuing a hardware interrupt when a button press is detected, it is ensured that the associated actions are executed, even if the Opta is busy with different operations. Such actions are implemented in dedicated interrupt service routines (ISR) to manage the flags for the state transitions.

```
void setup () {
  // Initialize interrupts associated to the input pins
  attachInterrupt(digitalPinToInterrupt(A0), EmergencyPB_ISR, FALLING);
  attachInterrupt(digitalPinToInterrupt(A1),EmergencyResetPB_ISR,RISING);
}

/* ----- INTERRUPT SERVICE ROUTINES ----- */
/* ISR called at the FALLING EDGE of pin I1 (A0), connected to the
 * physical emergency pushbutton (normally closed) */
void EmergencyPB_ISR(){
  emergencyPB = true;
}

/* ISR called at the RISING EDGE of pin I2 (A1), connected to the
 * physical "Reset Emergency" pushbutton (normally open) */
void EmergencyResetPB_ISR(){
  // If "Reset Emergency" command from operator
  // -> set the resetEmergencyPB flag IF the emergency is active,
  // the wave height is below threshold, and the float is raised
  if (!emergencyWaveHeight && state == 0 && digitalRead(A6)) {
    resetEmergencyPB = true;
  }
}
}
```

The test activity begins by powering the Opta™, and waiting for the connection to the internet and Arduino Cloud. Once the connection is established, an API GET request for the wave height forecast is issued to `open-meteo`, followed by the processing of the JSON response, and the printing of the 12 values (three-day horizon, split into 6-hour intervals) to the serial port for debugging (Fig. 5.12). Since the hour of the request is 10 AM, the wave height is reported at 0.26 m, and it is predicted to raise to 0.34 m within the following six hours. Both values are below the emergency threshold of 1 m, so the system operates under the **NORMAL** state, with the float down into the water (Fig. 5.13a).

In order to test the state machine transitions, the emergency pushbutton is actuated, triggering the **EMERGENCY** state, as indicated by the blue LED blinking (Fig. 5.13b): since the float is simulated to be in the water (selector in position 1 - green pilot light on), relay D1 energizes to lift it out.

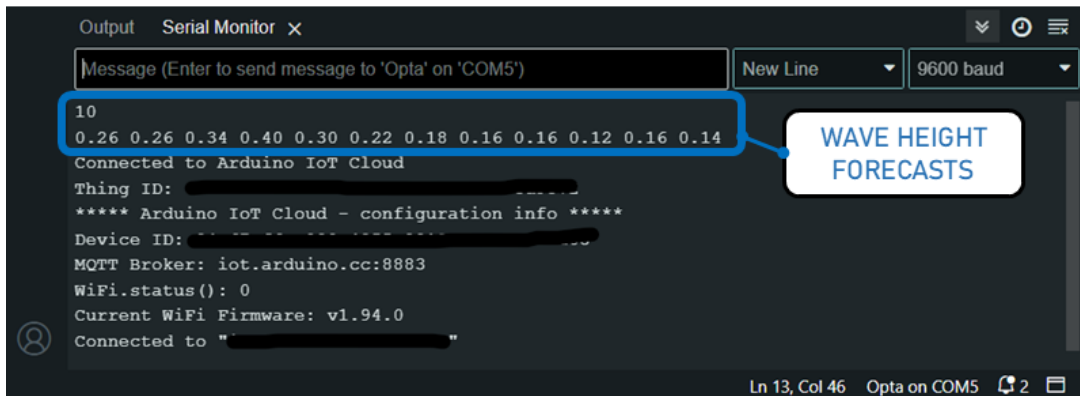


Figure 5.12: Connection Established - Serial Communication

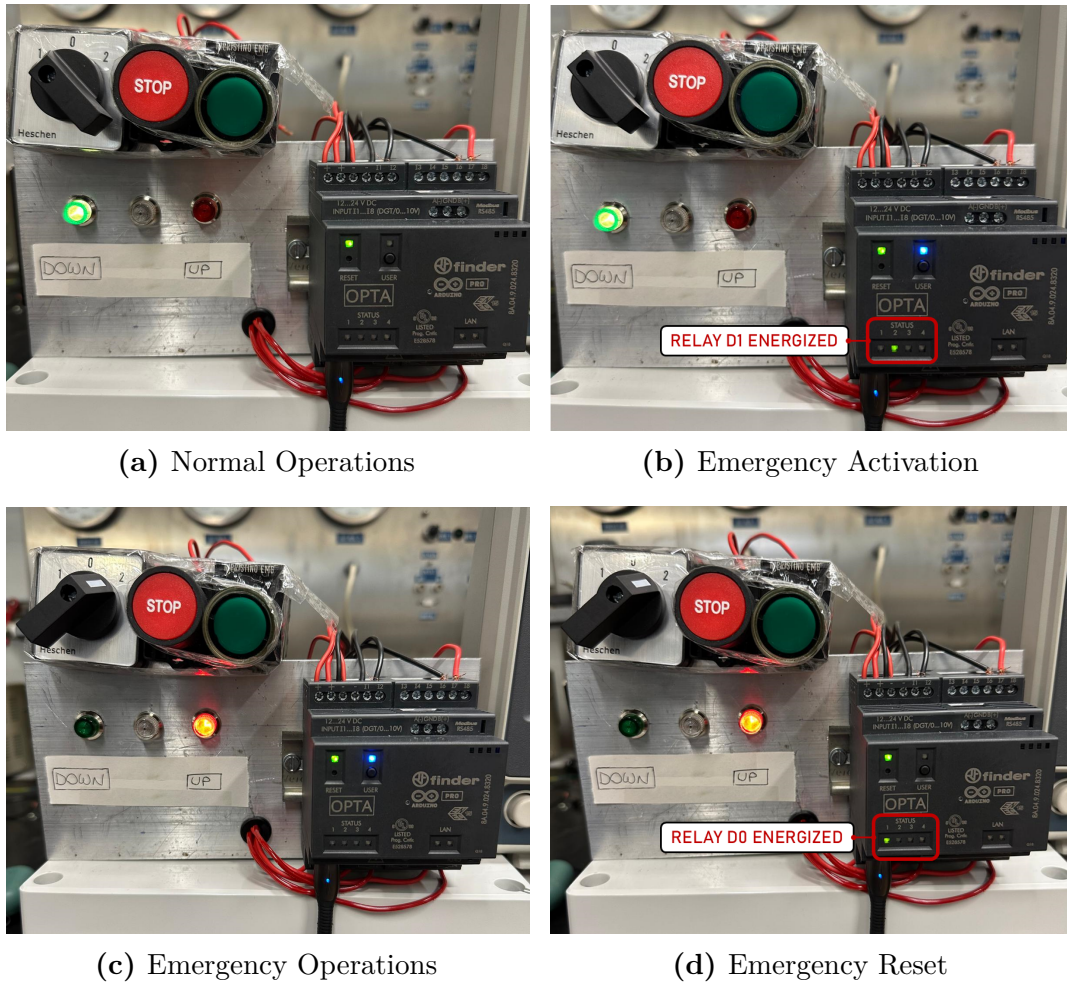
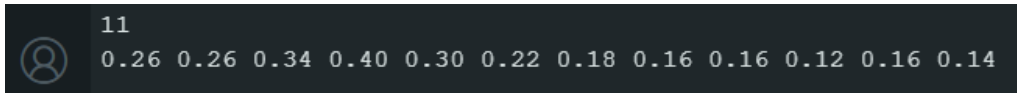


Figure 5.13: Emergency Operations - Unit Testing

As soon as the floating device is raised to safety (simulated by moving the three-way selector to 2 - red pilot light on), relay D1 is de-energized, halting the emergency motor (Fig. 5.13c). Since the EMERGENCY state was entered manually, it remains active until the "Reset Emergency" command, regardless of wave height. When the physical "Reset Emergency" pushbutton does get pressed, relay D0 closes to lower the float (Fig. 5.13d): in this case, the emergency can be successfully reset because the wave height is below the threshold.



**Figure 5.14:** Weather Update - Serial Monitor

The forecasts get an update every 60 minutes, as shown in Fig. 5.14 (11 AM).

To verify that the emergency state activates automatically when the wave height forecasts indicate to, the threshold is lowered to 0.2m. Upon the first weather update, the system correctly transitions to the emergency without operator intervention, repeating the actions described earlier (Fig. 5.13a-5.13c).

The remaining conditions laid out in the requirements are tested in a similar manner, verifying that the logic developed thus far behaves as expected, and allowing for development to continue.

## 5.4 Human-Machine Interface

The last remaining issue to address before the conclusion of the case study is the development of a Human-Machine Interface (HMI), leveraging Arduino Cloud and other web services for remote control and monitoring of the system.

As already highlighted, controlling the system remotely thanks to the IoT capabilities of the Opta™ platform and of the Arduino ecosystem offers a significant advantage in applications where the physical device is normally inaccessible.

The HMI shall continuously provide the operator with operational metrics such as the generator's electrical output, the state of the system (NORMAL or EMERGENCY), the position of the floating device (raised or lowered), and the wave height forecasts downloaded to the Opta™. This represents the minimum level of information to ensure safety, with more details and complexity to be included in the future.

In addition to the monitoring function, the HMI shall also feature the "Emergency Stop" and "Reset Emergency" remote commands, to force the system's emergency or to allow reentering the normal state, as discussed in section 5.3.3.

The HMI relies on two separate services, to showcase different options:

- **Arduino Cloud Dashboard:** to convey the most critical pieces of information such as the state of the system, and allow for remote control;
- **Google Sheets**<sup>11</sup>: an online spreadsheet service to log more detailed operational data, such as the generator output or the wave height forecasts, and serve as a short-term database.

For redundancy purposes, especially during the prototyping phase, it is convenient to retain the physical HMI employed in section 5.3.4 for the unit testing of the emergency logic.

### 5.4.1 Requirements

1. The Opta™ shall be monitored and controlled remotely via a virtual HMI, leveraging the Arduino Cloud infrastructure and other web services such as Google Sheets
2. The Arduino Cloud Dashboard shall contain:
  - (a) an indication of the current state of the system
  - (b) an indication of the current position of the float
  - (c) the current wave height at the operational site
  - (d) the "Emergency Stop" command
  - (e) the "Reset Emergency" command
3. The Google spreadsheet shall contain:
  - (a) a log of the wave height forecasts
  - (b) a log of the power output of the generator

However, it should be noted that, at the time of writing, the three power meters to measure the electrical output of the generator - one for each phase - are not available due to delays in procurement. Requirement 3b will therefore be neglected temporarily, leaving it as a future development for the project.

### 5.4.2 Arduino Cloud Dashboard

The Dashboard<sup>12</sup> represents the control center integrated into Arduino Cloud, where it is possible to visualize data coming from the physical devices with a number of widgets, and modify the value of specific variables to control Arduino boards

---

<sup>11</sup><https://workspace.google.com/products/sheets/>

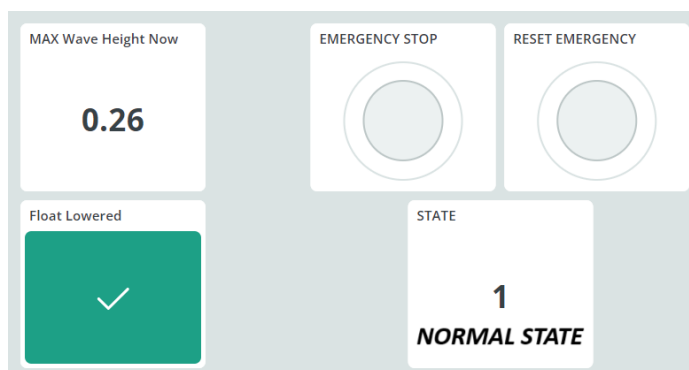
<sup>12</sup><https://cloud.arduino.cc/features-custom-dashboard/>

remotely. Dashboards are accessible via the internet both from desktop and mobile, ensuring high flexibility. They rely on the shared variables between the board and the cloud, configured as part of a Thing, allowing for the monitoring of those initialized as READ\_ONLY, and for modifications to the READ\_WRITE ones.

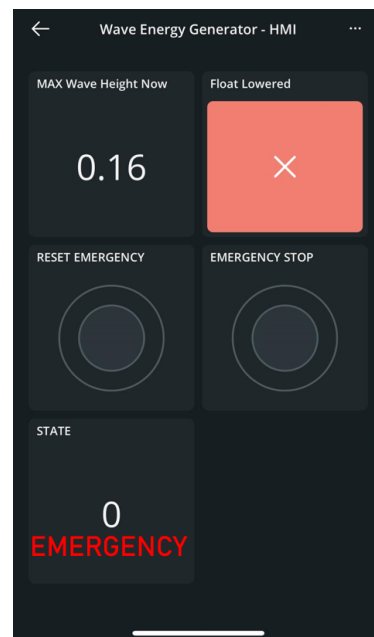
When the Thing for this application was first configured (Fig. 5.2), all the shared variables to meet requirements 2a-2e were included in anticipation of this section:

- **state**: READ\_ONLY integer to visualize the state of the system, NORMAL (1) or EMERGENCY (0);
- **currentMaxWaveHeight\_m**: READ\_ONLY float to display the wave height at the operational site;
- **floatDown**: READ\_ONLY boolean variable, true when the floating device is lowered in the operating position, and false when raised during an emergency;
- **emergencyStop\_cloud**: READ\_WRITE boolean variable that can be set to true by the operator to force the system into the emergency state;
- **resetEmergency\_cloud**: READ\_WRITE boolean variable, to reset a manually-triggered emergency.

Figures 5.15 and 5.16 show the Dashboard on both desktop and mobile, with all the widgets to fulfill the requirements, including a display of the current wave height (0.26 m and 0.16 m, respectively) and system's state (1 - normal- and 0 - emergency-), respectively), an indication of the position of the float, and the two pushbuttons for manual control.



**Figure 5.15:** Desktop Dashboard Normal Operations



**Figure 5.16:** Mobile Dashboard - Emergency Operations

## Arduino Code

The Arduino code is fairly straightforward as all `READ_ONLY` variables are automatically shared to the cloud after their initialization by `initProperties()` (section 5.2), without the need for additional software functions. Instead, the two `READ_WRITE` variables require a callback, i.e. a function to execute the desired actions when their values is modified remotely, similarly to an *interrupt* service routine. This simplicity and ease of use are precisely the main reasons behind the choice of Arduino Cloud, at least in the initial prototyping stage.

```
/* emergencyStop_cloud is READ_WRITE: onEmergencyStopCloudChange()
 * is executed every time a new value is received from IoT Cloud */
void onEmergencyStopCloudChange() {
  if (emergencyStop_cloud) // if emergency command from Arduino Cloud
    emergencyPB = true; //set the emergencyPB volatile flag
}

/* resetEmergency_cloud is READ_WRITE: onResetEmergencyCloudChange()
 * is executed every time a new value is received from IoT Cloud */
void onResetEmergencyCloudChange() {
  // If "Reset Emergency" command from Arduino Cloud dashboard
  // -> set the resetEmergencyPB flag IF the emergency is active,
  // the wave height is below threshold, and the float is raised
  if (resetEmergency_cloud && state == 0 && !emergencyWaveHeight
      && digitalRead(A6))
    resetEmergencyPB = true;
}
```

The two `onChange` functions manage the corresponding flags employed for the state transitions in accordance with the requirements, mirroring the ISR implemented for unit testing of the previous section (5.3.4). In particular, note how the "Reset Emergency" command from the cloud only has an effect if the wave height is below the threshold, to prevent damage to the float due to user error.

### 5.4.3 Data Logging to Google Sheets

The Arduino Cloud Dashboard is an effective interface to display glanceable information and control the board remotely, but lacks a structured data management system, such as a database or even a simple spreadsheet. Instead, it would be useful to log the wave height figures or the power output of the generator in an organized fashion, to monitor the operational trends of the system.

Thanks to the IoT capabilities of the Opta™, it is advantageous to employ an online service such as Google Sheets to address this requirement, a web-based



spreadsheet application for easy remote access. The choice of specific service is justified by the integration with "Apps Script"<sup>13</sup>, a JavaScript-based platform that allows for automation routines to be embedded into Google products. Not only does this enable the development of a web application to receive any requests from the Opta™ and parse the data to log, but also allows formatting the spreadsheet, and ensures seamless communication between all the parties involved.

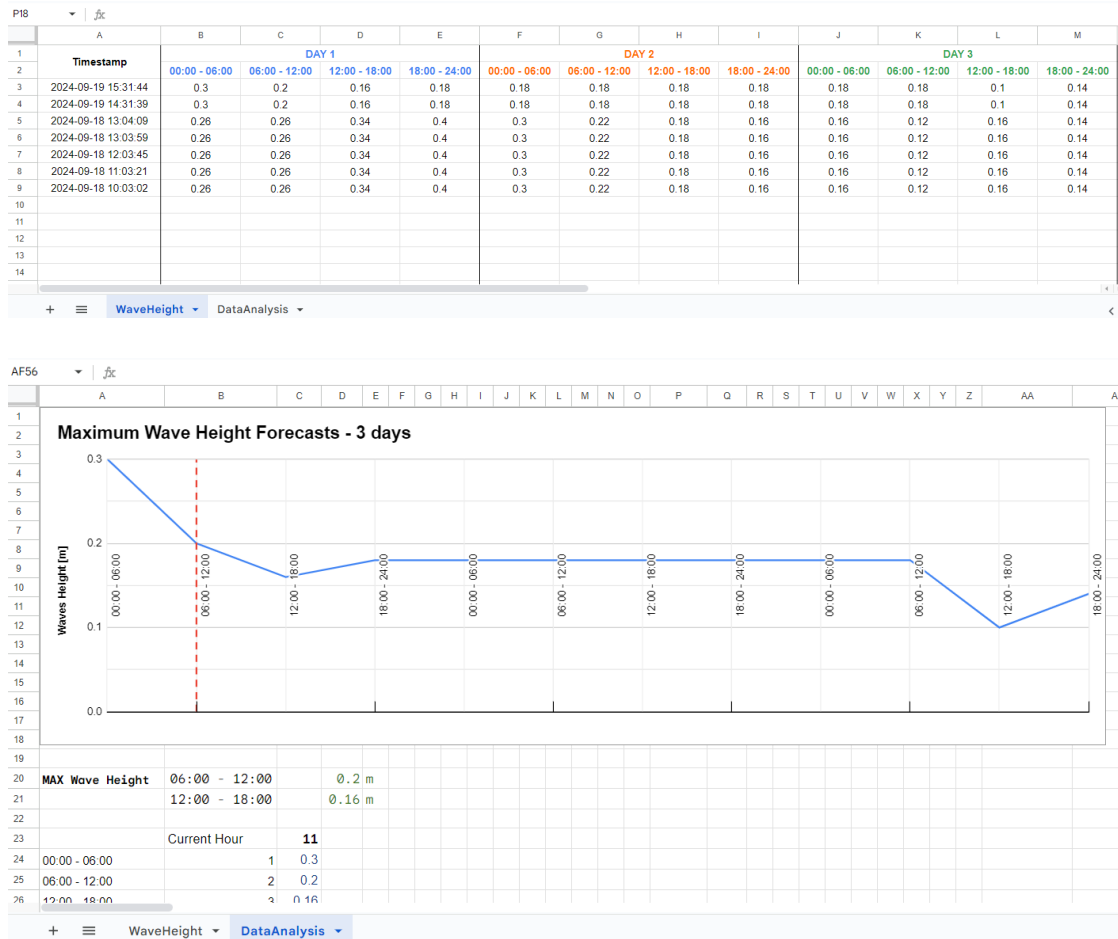


Figure 5.17: Google Sheets - Spreadsheet to log the Wave Height Data

As illustrated in Fig. 5.17, the Google spreadsheet developed for this application consists of two sheets: the first - titled "WaveHeight" - contains the raw figures resulting from each weather update (i.e. the forecasts for three days, split into intervals of six hours), while the second - "DataAnalysis" - plots the latest wave

<sup>13</sup><https://developers.google.com/apps-script>

height data, emphasizing the current value and the prediction for the following six hours, the two figures that the Opta™ analyzes for the state transitions. A similar approach can be adopted for the electrical output of the generator, pending the availability of the power meters.

## API Request and Google Apps Script

The wave height information is transmitted from the Opta™ to Google Sheets via the API of the web application created with "Apps Script". Reported below is an example of POST request - the HTTP method to send data to a server [106] - that must be formulated by the Opta™. The message, composed of the twelve forecast values, is encoded as a JSON-formatted `String` with a single field, the array `"wave_height"`.

```
POST /macros/s/xxxxxxxxxxx/exec HTTP/1.1;
Host: script.google.com
Connection: close
Content-Type: application/json
Content-Length: 85
'{"wave_height": [0.3, 0.2, 0.16, 0.18, 0.18, 0.18, 0.18,
                  0.18, 0.18, 0.18, 0.1, 0.14]}'
```

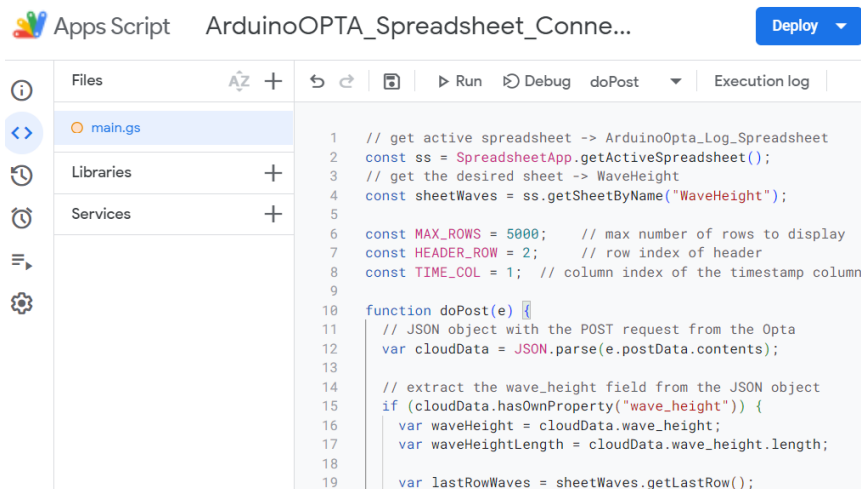


Figure 5.18: Google Apps Script Environment

The web application script (reported in full in the Appendix D.2) executes function `doPost()` every time it receives a POST request from the Opta™. Such function parses the twelve wave height values from the JSON object, and saves them inside sheet `WaveHeight`, adding a new row at the top with the appropriate formatting. Additionally, it prints the timestamp of the request, as shown in Fig. 5.17.

```
function doPost(e) {
  // JSON object with the POST request from the Opta
  var cloudData = JSON.parse(e.postData.contents);

  // extract the wave_height field from the JSON object
  if (cloudData.hasOwnProperty("wave_height")) {
    var waveHeight = cloudData.wave_height;
    var waveHeightLength = cloudData.wave_height.length;

    var lastRowWaves = sheetWaves.getLastRow();
    // delete last row to maintain constant the total number of rows
    if (lastRowWaves > MAX_ROWS + HEADER_ROW - 1) {
      sheetWaves.deleteRow(lastRowWaves);
    }
    // insert new row after deleting the last one
    sheetWaves.insertRowAfter(HEADER_ROW);

    var range = sheetWaves.getRange('A3:Z3');
    range.setFontColor('#000000'); // formatting options
    range.setFontSize(10);
    range.setFontWeight('normal');

    // write values in the respective columns
    for (var col=1+TIME_COL; col<=TIME_COL+waveHeightLength; col++) {
      sheetWaves.getRange(HEADER_ROW+1, col).setValue(
        waveHeight[col-1-TIME_COL]);
    }

    // write timestamp
    var timestampWaves = new Date();
    sheetWaves.getRange(HEADER_ROW+1, TIME_COL).setValue(
      timestampWaves).setNumberFormat("yyyy-MM-dd HH:mm:ss");
  }
}
```

## Arduino Code

Shifting focus back to the Arduino code on the Opta™, recall how function `logToSpreadsheet()` was included inside `loop()`, in section 5.3.2, precisely to transmit the weather data to Google Sheets, after each update to the forecasts:

```
void loop () {
  // log the forecast to Google Sheets
  logToSpreadsheet(max_wave_height_6h,
    (24 / kIntervalLength_h * kForecastLength_days));
}
```

The `logToSpreadsheet()` function processes the forecast values stored in `max_wave_height_6h[]` by organizing them into a JSON array, which is then converted to a `String` before sending the POST request. Library `Arduino_JSON.h` provides useful pre-built functions to carry out these operations.

The API call is issued by `sendPostRequest()`, following the syntax detailed previously, after establishing a connection to the `script.google.com` server.

```
uint8_t logToSpreadsheet(double variableValue[],
                        uint8_t numberOfEntries) {

    JSONVar object; // JSON object to share
    String jsonField = "wave_height"; // name of the JSON field

    // copy all entries of array variableValue into the JSON object
    for (uint8_t i = 0; i < numberOfEntries; i++)
        object[jsonField][i] = variableValue[i];

    if (JSON.typeof(object) == "undefined")
        return 1; // exit if error detected

    // JSON.stringify(myVar) to convert the JSONVar to a String
    String jsonString = JSON.stringify(object);
    if (sendPostRequest(jsonString) == 1) // send POST request
        return 1; // exit if error detected

    return 0;
}
```

#### 5.4.4 Unit Testing

The only aspect that still requires testing is remote control, specifically the two pushbuttons featured in the `Dashboard`. Instead, the monitoring functionality, assigned to the Arduino Cloud widgets and to the Google spreadsheet, was tested concurrently with its development yielding positive results, as demonstrated in Figures 5.15, 5.16 and 5.17.

To evaluate the efficacy of the remote pushbuttons, the same procedure of section 5.3.4 (unit testing of the emergency logic) is repeated. After the `Opta™` is powered on and enters normal operations, the remote "Emergency Stop" pushbutton is activated from the mobile app, with the system correctly entering the `EMERGENCY` state, and replicating the behavior observed in Fig. 5.13b and 5.13c (the three-way selector still simulates the signal from the limit switches of the float). Since the wave height is reported below the threshold, pressing the remote "Reset Emergency" pushbutton causes the system to exit `EMERGENCY`, as expected.

Once again, it is useful to remark how this is not a systematic testing activity, as the project is still at the implementation level, but rather a preliminary verification that no major problems can hinder subsequent development steps.

This also marks the end of the case study, as all the tasks assigned to the Opta™ for the prototyping phase of the energy generator were addressed successfully.

## 5.5 Conclusions and Future Developments

The main purpose of this last case study was to showcase how the IoT paradigm and a *smart* PLC such as the Finder Opta™ can fit into more "traditional" industrial applications, enhancing efficiency, guaranteeing a high level of operational safety, and enabling a more robust real-time decision-making process. By employing a predictive approach to safety, based on weather forecasts fetched from the internet, and by developing a virtual HMI to monitor operations and control the device remotely, this goal was successfully achieved.

Another key takeaway is that the integration with the Arduino ecosystem, which streamlines code development and offers many pre-built libraries and services such as Arduino Cloud, renders innovative technological solutions more accessible.

It is also worth highlighting how the application itself, the wave energy converter, though mostly in the background in this chapter, aims to provide an innovative contribution towards the increasingly relevant green energy transition. Any advances and improvements in efficiency, driven by a deep IoT integration and machine learning practices, can therefore have a meaningful impact on the spread and economic viability of renewable energy technologies.

Future short-term developments for the project include the integration of the already-mentioned power meters, enabling visualization of the instantaneous power output, the installation of strain gauges on the floating device to measure the input force of the waves, and be able to calculate the system's efficiency, and the development of an in-house solution to substitute Arduino Cloud and Google Sheets in the remote control and monitoring of the system. Once the mechanical and electrical design is finalized, it will be possible to construct the first scaled prototype, and enter the validation phase with extensive integration testing in a laboratory pool, eventually followed by deployment to the offshore site. Throughout the process, the Opta™ will continue in its coordination role, enabling the collection of vital operational metrics while guaranteeing safety, as demonstrated in the preliminary analysis of this chapter.

# Chapter 6

## Conclusions

The primary aim of the work conducted in the thesis was to showcase how machine learning and AI can be invaluable assets in an industrial setting, and how to deploy such *intelligence* directly on the edge devices in charge of process control.

The Programmable Logic Controller Finder Opta™ is a fitting and versatile tool for novel Industry 4.0 applications, thanks to the integration with the open-source Arduino ecosystem, the powerful Arm Cortex-M processor and the several connectivity options it features. As a cost-effective platform, it lowers the entry barriers, both in economic and technological terms, for smaller industrial realities to apply state-of-the-art data analysis and collection techniques, and obtain a deeper insight and better control over all aspects of the manufacturing process.

Through three case studies, the thesis investigated various enabling tools and technologies, such as TensorFlow with its high-level API Keras, TensorFlow Lite for Microcontrollers, Python and Google Colab, the Arduino variant of C++, Arduino Cloud, the FFT algorithm, etc., applying them successfully to the tasks at hand.

The main purpose of the Waveform Identification model developed in Case Study I (chapter 3) was to explore the machine learning workflow on the Finder Opta™. The application began with a discussion of the data collection process, including how to export and process such data with the Python library NumPy, before feeding it into the neural network for training. During the design of the MultiLayer Perceptron, exploiting some basic Keras structures, the constraints in terms of size and complexity for deployment on an embedded device were identified and addressed. A high-accuracy model was then successfully trained, after tuning some key hyperparameters, such as the number of neurons and the optimization algorithm. The model was subsequently converted to the TensorFlow Lite format, exploring different options in terms of optimization, and stored as a byte array to be loaded in the PLC memory. In designing the Arduino application to run inference with the model, the main tools offered by the TensorFlow Lite for Microcontrollers library were explored, including its data structures and memory requirements. Achieving a final accuracy of 93.3% in the testing phase with quite a small model

is a testament to a successful implementation, and demonstrates the potential of machine learning in industrial tasks.

As a preliminary application in the field of predictive maintenance of mechanical machinery, Case Study II (chapter 4) pushed the limits of the Opta's hardware with a somewhat complex Convolutional Neural Network model, demonstrating how more advanced tasks can be handled by industrial edge devices, thanks to machine learning. The goal of inferring the rotational speed of the test bench's rotating parts was successfully achieved with a final Mean Absolute Error of 42.1 rpm. A more systematic investigation of the model's robustness will be the subject of a future study, though this initial result has already been obtained with random artificial disturbances. Because of the choice of a short audio recording as input to the model, a more complex pre-processing pipeline needed to be implemented, including the calculation of the FFT to extract higher-level frequency information. Not only does this help the model abstract, as its size and complexity are still the main concern, but also demonstrates how the Opta™ platform can integrate different approaches for a reliable output. Future developments include training the CNN model to recognize anomalous sounds, and predict the health of the mechanical components. For enhanced accuracy, additional vibration and speed sensors could also be integrated into the system.

The final case study, Case Study III (chapter 5), shifted focus from machine learning to how the IoT integration can enhance traditional industrial applications, enabling remote monitoring and control practices. Moreover, employing a predictive approach to safety, based on weather forecasts, reduces the amount of human intervention required by the system, and, consequently, the chance of human errors.

All case studies were approached at high-level, with technologies and tools from different domains integrated together to achieve a certain goal. This system-level perspective reflects the typical role of Mechatronic Engineers, whose broad education allows them to identify novel multidisciplinary solutions, and liaise effectively with specialists from different fields. Nevertheless, this work represents a suitable baseline for further research in the fields of machine learning, predictive maintenance, edge computing, IoT, industrial automation, etc. — all highly relevant in today's technological landscape.

# Bibliography

- [1] Frank Emmert-Streib. «From the digital data revolution toward a digital society: Pervasiveness of artificial intelligence». In: *Machine Learning and Knowledge Extraction* 3.1 (2021), pp. 284–298 (cit. on p. 3).
- [2] Jay Lee et al. «Industrial ai». In: *Applications with sustainable performance* (2020) (cit. on p. 3).
- [3] Jay Lee et al. «Recent advances and trends in predictive manufacturing systems in big data environment». In: *Manufacturing Letters* 1.1 (2013), pp. 38–41. ISSN: 2213-8463. DOI: <https://doi.org/10.1016/j.mfglet.2013.09.005>. URL: <https://www.sciencedirect.com/science/article/pii/S2213846313000114> (cit. on pp. 3, 64).
- [4] Ercan Oztemel and Samet Gursev. «Literature review of Industry 4.0 and related technologies». In: *Journal of Intelligent Manufacturing* 31.1 (2020), pp. 127–182. ISSN: 1572-8145. DOI: 10.1007/s10845-018-1433-8. URL: <https://doi.org/10.1007/s10845-018-1433-8> (cit. on pp. 3, 63).
- [5] Neha Sharma et al. «The History, Present and Future with IoT». In: *Internet of Things and Big Data Analytics for Smart Generation*. Ed. by Valentina E. Balas et al. Cham: Springer International Publishing, 2019, pp. 27–51. ISBN: 978-3-030-04203-5. DOI: 10.1007/978-3-030-04203-5\_3. URL: [https://doi.org/10.1007/978-3-030-04203-5\\_3](https://doi.org/10.1007/978-3-030-04203-5_3) (cit. on p. 3).
- [6] Haochen Hua et al. «Edge Computing with Artificial Intelligence: A Machine Learning Perspective». In: *ACM Comput. Surv.* 55.9 (2023). ISSN: 0360-0300. DOI: 10.1145/3555802. URL: <https://doi.org/10.1145/3555802> (cit. on pp. 3, 66).
- [7] Nick Antonopoulos and Lee Gillam. *Cloud computing*. Vol. 51. 7. Springer, 2010 (cit. on p. 3).
- [8] Wei Yu et al. «A Survey on the Edge Computing for the Internet of Things». In: *IEEE Access* 6 (2018), pp. 6900–6919. DOI: 10.1109/ACCESS.2017.2778504 (cit. on pp. 4, 67).
- [9] Keyan Cao et al. «An Overview on Edge Computing Research». In: *IEEE Access* 8 (2020), pp. 85714–85728. DOI: 10.1109/ACCESS.2020.2991734 (cit. on p. 4).



- [10] Swapnil Sayan Saha et al. «Machine Learning for Microcontroller-Class Hardware: A Review». In: *IEEE Sensors Journal* 22.22 (2022), pp. 21362–21390. DOI: 10.1109/JSEN.2022.3210773 (cit. on p. 4).
- [11] K. Worden et al. «Artificial Neural Networks». In: *Machine Learning in Modeling and Simulation: Methods and Applications*. Ed. by Timon Rabczuk and Klaus-Jürgen Bathe. Cham: Springer International Publishing, 2023, pp. 85–119. ISBN: 978-3-031-36644-4. DOI: 10.1007/978-3-031-36644-4\_2. URL: [https://doi.org/10.1007/978-3-031-36644-4\\_2](https://doi.org/10.1007/978-3-031-36644-4_2) (cit. on p. 5).
- [12] F. Rosenblatt. «The perceptron: A probabilistic model for information storage and organization in the brain». In: *Psychological Review* 65.6 (1957), pp. 386–408 (cit. on p. 6).
- [13] Kurt Hornik. «Approximation capabilities of multilayer feedforward networks». In: *Neural networks* 4.2 (1991), pp. 251–257 (cit. on p. 7).
- [14] Charu C. Aggarwal. «An Introduction to Neural Networks». In: *Neural Networks and Deep Learning*. 2nd ed. Springer Cham, 2023. Chap. 1, pp. 1–25. ISBN: 978-3-031-29642-0. URL: <https://doi.org/10.1007/978-3-031-29642-0> (cit. on pp. 7, 8, 39).
- [15] G. James et al. «Logistic Regression». In: *An Introduction to Statistical Learning*. Springer International Publishing, 2023. Chap. 4, pp. 138–144. ISBN: 978-3-031-38747-0. URL: <https://doi.org/10.1007/978-3-031-38747-0> (cit. on p. 8).
- [16] Charu C. Aggarwal. «The Backpropagation Algorithm». In: *Neural Networks and Deep Learning*. 2nd ed. Springer Cham, 2023. Chap. 2, pp. 29–68. ISBN: 978-3-031-29642-0. URL: <https://doi.org/10.1007/978-3-031-29642-0> (cit. on p. 9).
- [17] Andreas Antoniou and Wu-Sheng Lu. «Basic Multidimensional Gradient Methods». In: *Practical Optimization*. Cham: Springer New York, NY, 2021, pp. 115–144. ISBN: 978-1-0716-0843-2. DOI: 10.1007/978-1-0716-0843-2. URL: <https://doi.org/10.1007/978-1-0716-0843-2> (cit. on p. 9).
- [18] Habib Izadkhah. «Backtracking Algorithms». In: *Problems on Algorithms: A Comprehensive Exercise Book for Students in Software Engineering*. Cham: Springer International Publishing, 2022, pp. 487–496. ISBN: 978-3-031-17043-0. DOI: 10.1007/978-3-031-17043-0\_14. URL: [https://doi.org/10.1007/978-3-031-17043-0\\_14](https://doi.org/10.1007/978-3-031-17043-0_14) (cit. on p. 10).
- [19] Jack Kiefer and Jacob Wolfowitz. «Stochastic estimation of the maximum of a regression function». In: *The Annals of Mathematical Statistics* (1952), pp. 462–466 (cit. on p. 11).

- [20] Xinyu Peng et al. «Accelerating Minibatch Stochastic Gradient Descent Using Typicality Sampling». In: *IEEE Transactions on Neural Networks and Learning Systems* 31.11 (2020), pp. 4649–4659. DOI: 10.1109/TNNLS.2019.2957003 (cit. on p. 11).
- [21] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017 (cit. on pp. 11, 41, 82).
- [22] Matthew D Zeiler. «ADADELTA: an adaptive learning rate method». In: *arXiv preprint arXiv:1212.5701* (2012) (cit. on p. 11).
- [23] John Duchi et al. «Adaptive subgradient methods for online learning and stochastic optimization.» In: *Journal of machine learning research* 12.7 (2011) (cit. on p. 11).
- [24] Geoffrey Hinton. «Overview of mini-batch gradient descent». 2012. URL: [https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture\\_slides\\_lec6.pdf](https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf) (cit. on pp. 11, 42).
- [25] Athanasios Voulodimos et al. «Deep Learning for Computer Vision: A Brief Review». In: *Computational Intelligence and Neuroscience* 2018.1 (2018), p. 7068349. DOI: <https://doi.org/10.1155/2018/7068349>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1155/2018/7068349> (cit. on p. 11).
- [26] Kunihiko Fukushima. «Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position». In: *Biological cybernetics* 36.4 (1980), pp. 193–202 (cit. on p. 12).
- [27] Yann LeCun et al. «Backpropagation applied to handwritten zip code recognition». In: *Neural computation* 1.4 (1989), pp. 541–551 (cit. on p. 12).
- [28] Alex Krizhevsky et al. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: [https://proceedings.neurips.cc/paper\\_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf) (cit. on p. 14).
- [29] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. URL: <https://arxiv.org/abs/1409.1556> (cit. on p. 14).
- [30] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. URL: <https://arxiv.org/abs/1704.04861> (cit. on p. 14).
- [31] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. 2nd Edition. MIT Press, 2017. ISBN: 978-0-262-53381-2 (cit. on p. 15).

- [32] Yanjiao Chen et al. «Deep Learning on Mobile and Embedded Devices: State-of-the-art, Challenges, and Future Directions». In: *ACM Comput. Surv.* 53.4 (Aug. 2020). ISSN: 0360-0300. DOI: 10.1145/3398209. URL: <https://doi.org/10.1145/3398209> (cit. on p. 15).
- [33] S. K. Islam and M. R. Haider. *Sensors and Low Power Signal Processing*. 1st Edition. Springer New York, NY, 2009. ISBN: 978-0-387-79392-4. DOI: 10.1007/978-0-387-79392-4. URL: <https://doi.org/10.1007/978-0-387-79392-4> (cit. on p. 15).
- [34] Pete Warden and Daniel Situnayake. «Introduction». In: *TinyML. Machine Learning with TensorFlow Lite on Arduino and Ultra-Low Power Microcontrollers*. 1st Edition. O'Reilly, 2019. Chap. 1, pp. 1–4. ISBN: 9781492051992 (cit. on p. 15).
- [35] Robert David et al. «TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems». In: *Proceedings of Machine Learning and Systems*. Ed. by A. Smola et al. Vol. 3. 2021, pp. 800–811. URL: [https://proceedings.mlsys.org/paper\\_files/paper/2021/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf](https://proceedings.mlsys.org/paper_files/paper/2021/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf) (cit. on p. 16).
- [36] Pete Warden and Daniel Situnayake. *TinyML. Machine Learning with TensorFlow Lite on Arduino and Ultra-Low Power Microcontrollers*. 1st Edition. O'Reilly, 2019. ISBN: 9781492051992 (cit. on p. 16).
- [37] Martin Abadi et al. «TensorFlow: A system for large-scale machine learning». In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: <https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf> (cit. on p. 16).
- [38] Tal Ben-Nun et al. «Stateful dataflow multigraphs: a data-centric model for performance portability on heterogeneous architectures». In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356173. URL: <https://doi.org/10.1145/3295500.3356173> (cit. on p. 16).
- [39] Jeff Heaton. *Applications of Deep Neural Networks with Keras*. 2020 (cit. on p. 16).
- [40] *Keras: The high-level API for TensorFlow*. URL: <https://www.tensorflow.org/guide/keras> (cit. on pp. 16, 17).
- [41] *Keras API: model training*. URL: [https://keras.io/api/models/model\\_training\\_apis/#model-training-apis](https://keras.io/api/models/model_training_apis/#model-training-apis) (cit. on pp. 17, 40).
- [42] Avid Farhoodfar. «Machine Learning for Mobile Developers: TensorFlow Lite Framework». In: (Apr. 2019) (cit. on p. 17).

- [43] Robert David et al. *TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems*. 2021 (cit. on p. 17).
- [44] Pete Warden and Daniel Situnayake. «TensorFlow Lite for Microcontrollers». In: *TinyML. Machine Learning with TensorFlow Lite on Arduino and Ultra-Low Power Microcontrollers*. 1st Edition. O'Reilly, 2019. Chap. 13, pp. 355–392. ISBN: 9781492051992 (cit. on pp. 17, 18).
- [45] *FlatBuffers white paper*. URL: [https://flatbuffers.dev/flatbuffers\\_white\\_paper.html](https://flatbuffers.dev/flatbuffers_white_paper.html) (cit. on p. 17).
- [46] Bernadette M Randles et al. «Using the Jupyter notebook as a tool for open science: An empirical study». In: *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE. 2017, pp. 1–2 (cit. on p. 18).
- [47] Ekaba Bisong. «Google Colaboratory». In: *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*. Berkeley, CA: Apress, 2019, pp. 59–64. ISBN: 978-1-4842-4470-8. DOI: 10.1007/978-1-4842-4470-8\_7. URL: [https://doi.org/10.1007/978-1-4842-4470-8\\_7](https://doi.org/10.1007/978-1-4842-4470-8_7) (cit. on p. 18).
- [48] Sandeep Mistry and Dominic Pajak. *Get Started With Machine Learning on Arduino*. URL: <https://docs.arduino.cc/tutorials/nano-33-ble-sense/get-started-with-machine-learning/> (cit. on p. 18).
- [49] *TensorFlow Lite Micro Library for Arduino - GitHub repository*. URL: <https://github.com/tensorflow/tflite-micro-arduino-examples> (cit. on p. 18).
- [50] *TensorFlow Lite Micro Library for Arduino Opta - GitHub repository*. URL: <https://github.com/riccardomennilli/tensorflow-lite-micro-arduino-opta> (cit. on p. 18).
- [51] Frank Petruzella. *Programmable Logic Controllers*. Ed. by McGraw Hill. 5th Edition. 2019. Chap. 1-5. ISBN: 978-0-07-337384-3 (cit. on p. 20).
- [52] Tom Antonsen. *PLC Controls with Structured Text (ST)*. 3rd Edition. ISBN: 978-8-74-301554-3 (cit. on p. 20).
- [53] *The making of Arduino*. 2011. URL: <https://spectrum.ieee.org/the-making-of-arduino> (cit. on p. 22).
- [54] *What is Arduino?* URL: <https://www.arduino.cc/en/Guide/Introduction> (cit. on p. 22).
- [55] *Important Update on Mbed*. URL: <https://os.mbed.com/blog/entry/Important-Update-on-Mbed/> (cit. on p. 23).
- [56] *Arduino Opta Collective Datasheet*. URL: <https://docs.arduino.cc/resources/datasheets/AFX00001-AFX00002-AFX00003-datasheet.pdf> (cit. on pp. 24, 25).

- [57] Katie Boeckl et al. *Considerations for managing Internet of Things (IoT) cybersecurity and privacy risks*. US Department of Commerce, National Institute of Standards and Technology . . . , 2019 (cit. on p. 24).
- [58] Kelsy Cabello-Solorzano et al. «The Impact of Data Normalization on the Accuracy of Machine Learning Algorithms: A Comparative Analysis». In: *18th International Conference on Soft Computing Models in Industrial and Environmental Applications (SOCO 2023)*. Ed. by Pablo García Bringas et al. Cham: Springer Nature Switzerland, 2023, pp. 344–353. ISBN: 978-3-031-42536-3 (cit. on p. 31).
- [59] Mohamed Aly. «Survey on multiclass classification methods». In: *Neural Netw* 19.1-9 (2005), p. 2 (cit. on p. 37).
- [60] Katarzyna Janocha and Wojciech Marian Czarnecki. *On Loss Functions for Deep Neural Networks in Classification*. 2017 (cit. on p. 38).
- [61] Anqi Mao et al. *Cross-Entropy Loss Functions: Theoretical Analysis and Applications*. 2023. URL: <https://arxiv.org/abs/2304.07288> (cit. on p. 38).
- [62] *Keras API: categorical\_crossentropy*. URL: [https://keras.io/api/losses/probabilistic\\_losses/#categorical\\_crossentropy-function](https://keras.io/api/losses/probabilistic_losses/#categorical_crossentropy-function) (cit. on p. 38).
- [63] *Keras API: sparse\_categorical\_crossentropy*. URL: [https://keras.io/api/losses/probabilistic\\_losses/#sparsecategorical\\_crossentropy-class](https://keras.io/api/losses/probabilistic_losses/#sparsecategorical_crossentropy-class) (cit. on p. 38).
- [64] *Keras API: stochastic gradient descent*. URL: <https://keras.io/api/optimizers/sgd/> (cit. on p. 38).
- [65] Elisa Oostwal et al. «Hidden unit specialization in layered neural networks: ReLU vs. sigmoidal activation». In: *Physica A: Statistical Mechanics and its Applications* 564 (2021), p. 125517. ISSN: 0378-4371. DOI: <https://doi.org/10.1016/j.physa.2020.125517>. URL: <https://www.sciencedirect.com/science/article/pii/S0378437120308153> (cit. on p. 39).
- [66] Margherita Grandini et al. *Metrics for Multi-Class Classification: Overview*. 2020. URL: <https://arxiv.org/abs/2008.05756> (cit. on p. 40).
- [67] *TensorFlow Documentation: Post-training quantization*. URL: [https://www.tensorflow.org/lite/performance/post\\_training\\_quantization](https://www.tensorflow.org/lite/performance/post_training_quantization) (cit. on p. 47).
- [68] Raghuraman Krishnamoorthi. *Quantizing deep convolutional networks for efficient inference: A whitepaper*. 2018 (cit. on p. 48).

- [69] Pete Warden and Daniel Situnayake. «The “Hello World” of TinyML: Building an Application». In: *TinyML. Machine Learning with TensorFlow Lite on Arduino and Ultra-Low Power Microcontrollers*. 1st Edition. O’Reilly, 2019. Chap. 5, pp. 67–93. ISBN: 9781492051992 (cit. on pp. 50, 55).
- [70] Erwin Rauch et al. «Anthropocentric perspective of production before and within Industry 4.0». In: *Computers & Industrial Engineering* 139 (2020), p. 105644. ISSN: 0360-8352. DOI: <https://doi.org/10.1016/j.cie.2019.01.018>. URL: <https://www.sciencedirect.com/science/article/pii/S0360835219300233> (cit. on p. 63).
- [71] Erim Sezer et al. «An Industry 4.0-Enabled Low Cost Predictive Maintenance Approach for SMEs». In: *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*. 2018, pp. 1–8. DOI: 10.1109/ICE.2018.8436307 (cit. on p. 63).
- [72] Jay Lee et al. «Intelligent prognostics tools and e-maintenance». In: *Computers in Industry* 57.6 (2006). E-maintenance Special Issue, pp. 476–489. ISSN: 0166-3615. DOI: <https://doi.org/10.1016/j.compind.2006.02.014>. URL: <https://www.sciencedirect.com/science/article/pii/S0166361506000522> (cit. on p. 63).
- [73] Prabu David Hyesun Choung and Arun Ross. «Trust in AI and Its Role in the Acceptance of AI Technologies». In: *International journal of Human-Computer Interaction* 39.9 (2023), pp. 1727–1739. DOI: 10.1080/10447318.2022.2050543. URL: <https://doi.org/10.1080/10447318.2022.2050543> (cit. on p. 63).
- [74] Nitin Liladhar Rane. «ChatGPT and Similar Generative Artificial Intelligence (AI) for Building and Construction Industry: Contribution, Opportunities and Challenges of Large Language Models for Industry 4.0, Industry 5.0, and Society 5.0.» In: (2023). DOI: 10.2139/ssrn.4603221. URL: <http://dx.doi.org/10.2139/ssrn.4603221> (cit. on p. 63).
- [75] Christopher Irgens Thorsten Wuest Daniel Weimer and Klaus-Dieter Thoben. «Machine learning in manufacturing: advantages, challenges, and applications». In: *Production & Manufacturing Research* 4.1 (2016), pp. 23–45. DOI: 10.1080/21693277.2016.1192517. URL: <https://doi.org/10.1080/21693277.2016.1192517> (cit. on pp. 63, 65).
- [76] Haochen Hua et al. «Edge Computing with Artificial Intelligence: A Machine Learning Perspective». In: *ACM Comput. Surv.* 55.9 (2023). ISSN: 0360-0300. DOI: 10.1145/3555802. URL: <https://doi.org/10.1145/3555802> (cit. on p. 64).

- [77] Eduardo C. Nunes. *Anomalous Sound Detection with Machine Learning: A Systematic Review*. 2021. URL: <https://arxiv.org/abs/2102.07820> (cit. on pp. 64, 66).
- [78] Douglas Thomas and Brian Weiss. «Maintenance Costs and Advanced Maintenance Techniques in Manufacturing Machinery: Survey and Analysis». en. In: *Int J Progn Health Manag* 12.1 (2021) (cit. on p. 64).
- [79] Gian Antonio Susto et al. «A predictive maintenance system for epitaxy processes based on filtering and prediction techniques». In: *IEEE Transactions on Semiconductor Manufacturing* 25.4 (2012). Cited by: 72, pp. 638–649. DOI: 10.1109/TSM.2012.2209131. URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-84877023031&doi=10.1109%2fTSM.2012.2209131&partnerID=40&md5=d7e2a20d404fdc4496724aad0c46e34b> (cit. on p. 64).
- [80] C Coleman et al. «Making maintenance smarter». In: *Deloitte University Press* (2017), pp. 1–21 (cit. on p. 65).
- [81] Jason Deutsch and David He. «Using deep learning-based approach to predict remaining useful life of rotating components». In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 48.1 (2018). Cited by: 358; All Open Access, Green Open Access, pp. 11–20. DOI: 10.1109/TSMC.2017.2697842. URL: <https://www.scopus.com/inward/record.uri?eid=2-s2.0-85018899311&doi=10.1109%2fTSMC.2017.2697842&partnerID=40&md5=91bc4a1352529a29c0fc7446e153b3c1> (cit. on p. 65).
- [82] Thyago P. Carvalho et al. «A systematic literature review of machine learning methods applied to predictive maintenance». In: *Computers & Industrial Engineering* 137 (2019), p. 106024. ISSN: 0360-8352. DOI: <https://doi.org/10.1016/j.cie.2019.106024>. URL: <https://www.sciencedirect.com/science/article/pii/S0360835219304838> (cit. on p. 65).
- [83] Xiaojie Guo et al. «Hierarchical adaptive deep convolution neural network and its application to bearing fault diagnosis». In: *Measurement* 93 (2016), pp. 490–502. ISSN: 0263-2241. DOI: <https://doi.org/10.1016/j.measurement.2016.07.054>. URL: <https://www.sciencedirect.com/science/article/pii/S0263224116304249> (cit. on p. 65).
- [84] Zhu Xiaoxun et al. «Research on crack detection method of wind turbine blade based on a deep learning method». In: *Applied Energy* 328 (2022), p. 120241. ISSN: 0306-2619. DOI: <https://doi.org/10.1016/j.apenergy.2022.120241>. URL: <https://www.sciencedirect.com/science/article/pii/S0306261922014982> (cit. on p. 65).

- [85] Keisuke Imoto and Yohei Kawaguchi. «Research Trends in Environmental Sound Analysis and Anomalous Sound Detection». In: *IEICE ESS Fundamentals Review* 15.4 (2022), pp. 268–280 (cit. on p. 66).
- [86] Yuki Tagawa et al. «Acoustic Anomaly Detection of Mechanical Failures in Noisy Real-Life Factory Environments». In: *Electronics* 10.19 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10192329. URL: <https://www.mdpi.com/2079-9292/10/19/2329> (cit. on p. 66).
- [87] Nachiket Soni et al. «Acoustic Analysis of Cutting Tool Vibrations of Machines for Anomaly Detection and Predictive Maintenance». In: *2023 IEEE 11th Region 10 Humanitarian Technology Conference (R10-HTC)*. 2023, pp. 43–46. DOI: 10.1109/R10-HTC57504.2023.10461855 (cit. on p. 66).
- [88] Jingyao Wu et al. «Fault-Attention Generative Probabilistic Adversarial Autoencoder for Machine Anomaly Detection». In: *IEEE Transactions on Industrial Informatics* 16.12 (2020), pp. 7479–7488. DOI: 10.1109/TII.2020.2976752 (cit. on p. 66).
- [89] Ryusei Ikegami et al. «Anomalous Sound Detection System in Manufacturing Industry Using Unsupervised Learning». In: *2024 IEEE International Conference on Consumer Electronics (ICCE)*. 2024, pp. 1–4. DOI: 10.1109/ICCE59016.2024.10444392 (cit. on p. 66).
- [90] Tahera Kalsoom et al. «Impact of IoT on Manufacturing Industry 4.0: A New Triangular Systematic Review». In: *Sustainability* 13.22 (2021). ISSN: 2071-1050. DOI: 10.3390/su132212506. URL: <https://www.mdpi.com/2071-1050/13/22/12506> (cit. on p. 66).
- [91] Taimur Hafeez et al. «Edge Intelligence for Data Handling and Predictive Maintenance in IIOT». In: *IEEE Access* 9 (2021), pp. 49355–49371. DOI: 10.1109/ACCESS.2021.3069137 (cit. on p. 67).
- [92] Jakub Konečný et al. «Federated Learning: Strategies for Improving Communication Efficiency». In: *NIPS Workshop on Private Multi-Party Machine Learning*. 2016. URL: <https://arxiv.org/abs/1610.05492> (cit. on p. 67).
- [93] G. M. Sessler and J. E. West. «Electret transducers: a review». In: *The Journal of the Acoustical Society of America* 53.6 (June 1973), pp. 1589–1600. ISSN: 0001-4966. DOI: 10.1121/1.1913507. URL: <https://doi.org/10.1121/1.1913507> (cit. on p. 68).
- [94] Sabrie Soloman. *Sensors Handbook*. en. 2nd Edition. New York: McGraw-Hill Education, 2010. Chap. 1, pp. 21–32. ISBN: 9780071605700. URL: <https://www.accessengineeringlibrary.com/content/book/9780071605700> (cit. on p. 69).



- [95] Zhijun Ren et al. «A Summary of Rotational Speed Measurement Techniques for Bearing and Rotor Systems—Part II». In: *Proceedings of the UNified Conference of DAMAS, IncoME and TEPEN Conferences (UNified 2023)*. Ed. by Andrew D. Ball et al. Cham: Springer Nature Switzerland, 2024, pp. 523–537. ISBN: 978-3-031-49421-5 (cit. on p. 71).
- [96] L. Wyse. *Audio Spectrogram Representations for Processing with Convolutional Neural Networks*. 2017. URL: <https://arxiv.org/abs/1706.09559> (cit. on p. 72).
- [97] Paul Heckbert. «Fourier transforms and the fast Fourier transform (FFT) algorithm». In: *Computer Graphics* 2.1995 (1995), pp. 15–463 (cit. on p. 74).
- [98] Udo Zolzer. *Digital audio signal processing*. John Wiley & Sons, 2022 (cit. on p. 74).
- [99] S. Braun. «WINDOWS». In: *Encyclopedia of Vibration*. Ed. by S. Braun. Oxford: Elsevier, 2001, pp. 1587–1595. ISBN: 978-0-12-227085-7. DOI: <https://doi.org/10.1006/rwvb.2001.0052>. URL: <https://www.sciencedirect.com/science/article/pii/B0122270851000527> (cit. on p. 78).
- [100] Abien Fred Agarap. *Deep Learning using Rectified Linear Units (ReLU)*. 2019. URL: <https://arxiv.org/abs/1803.08375> (cit. on p. 79).
- [101] Paul Gavrikov. *visualkerass*. <https://github.com/paulgavrikov/visualkerass>. 2020 (cit. on p. 81).
- [102] Juan Terven et al. «Loss Functions and Metrics in Deep Learning. A Review». In: (July 2023) (cit. on p. 81).
- [103] Kaichao You et al. *How Does Learning Rate Decay Help Modern Neural Networks?* 2019. URL: <https://arxiv.org/abs/1908.01878> (cit. on p. 82).
- [104] Nikola Zlatanov. «Architecture and Operation of a Watchdog Timer». In: (July 2014). DOI: 10.13140/RG.2.1.1149.1605 (cit. on p. 101).
- [105] Anthony Williams. *C++ Concurrency in Action*. Manning Publications, 2012. ISBN: 9781933988771 (cit. on p. 102).
- [106] David Gourley et al. *HTTP: The Definitive Guide*. O’Reilly Media, Inc., 2002. ISBN: 9781565925090 (cit. on pp. 107, 108, 125).
- [107] *JSON Official Documentation*. URL: <https://www.json.org/json-en.html> (cit. on p. 108).
- [108] Ferdinand Wagner et al. *Modeling Software with Finite State Machines: A Practical Approach*. Auerbach Publications, 2006. ISBN: 9780849380860 (cit. on p. 113).

# Appendix A

## Finder Opta™

### A.1 PLC IDE Programming Example

#### A.1.1 Arduino Sketch

```
#include <WiFi.h>
#include <Arduino_JSON.h>

// Wi-Fi network information
const char* ssid      = "*****";
const char* password = "*****";

// server address for time api
const char* server = "worldtimeapi.org";

// API endpoint for the date and time (Rome timezone)
String path = "/api/timezone/Europe/Rome";

WiFiClient client; // client instance to set up connection
JSONVar doc; // to store fetched data

void setup() {

  PLCIn.inConnectionStatus = 0; // initialize connection flag

  // establish internet connection
  WiFi.begin(ssid, password);
  delay(2000); // wait for connection

}

void loop() {
```

```
// if NOT connected wait and retry to establish connection
while (WiFi.status() != WL_CONNECTED) {
    PLCIIn.inConnectionStatus = 0;
    WiFi.begin(ssid, password);
    delay(2000);
}

// if connected set flag and fetch the time
PLCIIn.inConnectionStatus = 1;
fetchRomeTime();
}

/**
 * Fetch the time for the Rome timezone from a server with a GET
 * request, and parse the response to obtain the minutes
 */
void fetchRomeTime() {
    if (client.connect(server, 80)) {
        // send HTTP GET request
        client.print("GET ");
        client.print(path);
        client.println(" HTTP/1.1");
        client.print("Host: ");
        client.println(server);
        client.println("Connection: close");
        client.println();

        client.find("\r\n\r\n"); // skip HTTP headers of the response

        // read the response and parse the JSON
        String response = client.readStringUntil('\n');
        doc = JSON.parse(response);

        // if parsing unsuccessful exit the function
        if (JSON.typeof(doc) == "undefined")
            return;

        // extract the date and time from response
        String query_datetime = doc["datetime"];

        // extract the minutes as a string
        query_datetime = query_datetime.substring(14,16);
        // convert to int and send to PLC
        PLCIIn.inMinutes = query_datetime.toInt();
    }
}
```

# Appendix B

## Case Study I

### B.3 Building a Neural Network

#### B.3.1 Training Data Collection

##### Arduino Application

```
/**
 * Acquisition of a training dataset to develop a Neural Network
 * that identifies the kind of input waveform
 * Purpose: this sketch acquires a certain number of samples
 * from analog input A0; these are then transmitted to the serial port
 *
 * @author Riccardo Mennilli
 */

/* ---- CONFIGURATION ---- */

unsigned long Ts = 5; // sample time
const uint8_t ANALOG_INPUT_RESOLUTION = 14; // resolution of A0
// no. of samples to collect for each training example
const uint16_t NUM_SAMPLES = 500;

// time to wait before a new training example is collected (5 seconds)
const uint16_t delay_between_examples_ms = 5000;

/* ----- */

// to keep track of time
unsigned long chrono = 0;
// keep track of the number of samples acquired
uint16_t samples_count = 0;
```

```
void setup() {
  // initialize serial communication
  Serial.begin(115200);
  while (!Serial); // wait until the serial port is open

  /* ---- Setup the Arduino Peripherals ---- */
  pinMode(A0, INPUT); // A0 -> analog input port
  // set resolution to 14 bits
  analogReadResolution(ANALOG_INPUT_RESOLUTION);
}

void loop() {

  // check if the current training example is complete
  // (500 samples collected)
  if (samples_count >= NUM_SAMPLES) {
    // if so wait 5 seconds to allow the operator to change
    // the parameters of the input waveform
    delay(delay_between_examples_ms);

    samples_count = 0; // reset the counter for the number of samples
    chrono = millis(); // save the time

  } else if (millis() - chrono >= Ts) {
    // If > sample time has elapsed since the last recorded sample
    * -> record a new sample and transmit it to the serial port
    * after conversion to millivolts and normalization */
    Serial.println(digitalSampleToMillivolts(
      (analogRead(A0) - 5000.0) / 10000.0));

    samples_count++; // record that a new sample has been collected

    chrono = millis(); // record the time
  }
}

/* This function converts a digital sample into
 * an analog value in millivolts, with 10V reference
 * ADC_counts: digital sample read by the ADC
 */
float digitalSampleToMillivolts(unsigned short ADC_counts)
{
  // voltage reference -> 10 V = 10000 mV
  const unsigned long Vmax_mV = 10000;
```

```
unsigned long value_temp; // temporary value for calculations

value_temp = ADC_counts;
value_temp = value_temp * Vmax_mV;

// divide by 214 by shifting right of 14 bits, then
// convert to float
return (float) (value_temp >> ANALOG_INPUT_RESOLUTION);
}
```

## Data Processing

```
# This function randomizes the order of the dataset,
# and it splits it into training set, validation set and test set
# inputs: numpy matrix containing the dataset
# outputs: numpy matrix with the data labels
# ratio: vector to specify the relative size of the subsets
def split_dataset(inputs, outputs, ratio):
    num_inputs = len(inputs)
    random_indexes = np.arange(num_inputs)
    np.random.shuffle(random_indexes)

    # introduce randomized indexes
    inputs = inputs[random_indexes]
    outputs = outputs[random_indexes]

    # split the examples (group of samples) into three sets:
    # training, testing and validation
    TRAIN_SPLIT = int(ratio[0] * num_inputs)
    TEST_SPLIT = int(ratio[1] * num_inputs + TRAIN_SPLIT)

    inputs_train, inputs_test, inputs_validate = np.split(inputs,
        [TRAIN_SPLIT, TEST_SPLIT])
    inputs_train = inputs_train.astype('float32')
    inputs_validate = inputs_validate.astype('float32')
    inputs_test = inputs_test.astype('float32')
    outputs_train, outputs_test, outputs_validate = np.split(outputs,
        [TRAIN_SPLIT, TEST_SPLIT])

    return outputs_train, outputs_test, outputs_validate, inputs_train,
        inputs_test, inputs_validate
```

### B.3.3 Final Model Testing

The confusion matrix was plotted with `seaborn`<sup>1</sup>, a high-level `matplotlib`<sup>2</sup> API.

```
import matplotlib.pyplot as plt
import seaborn as sns # high-level matplotlib API for the heatmap
import numpy as np
import tensorflow as tf

# predictions and outputs_test are one-hot encoded arrays,
# for the confusion matrix we need a 1D vector of labels:
predictions_array = np.argmax(predictions, axis=1)
outputs_test_array = test=np.argmax(outputs_test, axis=1)
conf_matrix = tf.math.confusion_matrix(labels=outputs_test_array,
                                       predictions=predictions_array)

# print the confusion matrix
plt.rcParams['font.family'] = "monospace"
fig, ax = plt.subplot(figsize = (4.5,3))
sns.heatmap(conf_matrix, annot = True, cmap = 'terrain_r',
            xticklabels=WAVEFORMS, yticklabels=WAVEFORMS, cbar = False)
ax.set_xlabel('Predicted Labels', fontweight='bold', fontsize=10)
ax.set_ylabel('True Labels', fontweight='bold', fontsize=10)
ax.xaxis.set_label_position('top')
plt.yticks(rotation=90)
plt.show()
```

## B.4 Model Deployment to the Finder OPTA PLC

### B.4.2 Arduino Application

Reported below is the full application code, developed in the Arduino IDE:

```
#include <TensorFlowLite.h>
#include <tensorflow/lite/micro/all_ops_resolver.h>
#include <tensorflow/lite/micro/micro_error_reporter.h>
#include <tensorflow/lite/micro/micro_interpreter.h>
#include <tensorflow/lite/schema/schema_generated.h>
#include <tensorflow/lite/version.h>
```

---

<sup>1</sup><https://seaborn.pydata.org/>

<sup>2</sup><https://matplotlib.org/>

```

#include "model.h"

namespace { // to avoid conflicts with variables in other files
    tflite::MicroErrorReporter tflErrorReporter;

    const tflite::Model* tflModel = nullptr;
    tflite::AllOpsResolver tflOpsResolver;
    tflite::MicroInterpreter* tflInterpreter = nullptr;

    TfLiteTensor* tflInputTensor = nullptr;
    TfLiteTensor* tflOutputTensor = nullptr;

    // allocate memory for the tensors
    // (constexpr to evaluate the expression at compile time)
    constexpr int tensorArenaSize = 6 * 1024;
    byte tensorArena[tensorArenaSize] __attribute__((aligned(16)));
}

/* ---- Configuration ---- */
unsigned long Ts = 5; // sample time [ms]
const uint8_t ANALOG_INPUT_RESOLUTION = 14;
const uint16_t NUM_SAMPLES = 500; // no. of input samples

// time [in ms] to wait before new samples are collected
// for a new inference
const uint16_t delay_between_inferences_ms = 5000;
/* ----- */

// variable to store time
unsigned long chrono = 0;

// keep track of the number of samples acquired
uint16_t samples_count = 0;

// mapping of the output of the model to a waveform type
const char* WAVEFORMS[] = {
    "sine",
    "triangular",
    "square"
};
};
const uint8_t NUM_WAVEFORMS = sizeof(WAVEFORMS)/sizeof(WAVEFORMS[0]);

// arrays to store the identifiers of the relay/led outputs
const uint8_t RELAY_PIN[] = {D0, D1, D2};
const uint8_t RELAY_LED[] = {LED_D0, LED_D1, LED_D2};

```



```
const uint8_t NUM_OUTPUTS = sizeof(RELAY_PIN) / sizeof(RELAY_PIN[0]);

void setup() {
    // initialize serial communication
    Serial.begin(115200);
    while (!Serial);

    /* ---- Setup the Machine Learning Infrastructure ---- */
    // map model into a usable data structure
    tflModel = tflite::GetModel(model);
    if (tflModel->version() != TFLITE_SCHEMA_VERSION) {
        // raise error if model version is incompatible
        Serial.print("ERROR: Model is schema version ");
        Serial.print(tflModel->version());
        Serial.print(", not equal to supported version ");
        Serial.println(TFLITE_SCHEMA_VERSION);

        while(1); // endless loop to stop execution
    }

    // build the interpreter to run the model
    static tflite::MicroInterpreter static_interpreter(tflModel,
        tflOpsResolver, tensorArena, tensorArenaSize, &tflErrorReporter);
    tflInterpreter = &static_interpreter;

    // allocate memory for the tensors
    TfLiteStatus allocate_status = tflInterpreter->AllocateTensors();
    if (allocate_status != kTfLiteOk) {
        // raise error if allocation fails
        Serial.print("ERROR: AllocateTensors() failed.");
        while(1); // endless loop to stop execution
    }

    // obtain pointers to input and output tensors
    tflInputTensor = tflInterpreter->input(0);
    tflOutputTensor = tflInterpreter->output(0);

    /* ---- Setup the Arduino Peripherals ---- */
    pinMode(A0, INPUT); // A0 -> analog input port
    analogReadResolution(ANALOG_INPUT_RESOLUTION); // 14 bits

    // configure the output relays and related LEDs
    for (uint8_t i = 0; i < NUM_OUTPUTS; i++) {
        pinMode(RELAY_PIN[i], OUTPUT);
        pinMode(RELAY_LED[i], OUTPUT);
    }
}
```

```
    }  
    // configure the blue LED as output (ON when OPTA is paused)  
    pinMode(LED_USER, OUTPUT); // blue LED  
}  
  
void loop() {  
  
    float sample; // to hold the current sample  
  
    // check if enough samples have been collected to run inference  
    // (at least 500)  
    if (samples_count >= NUM_SAMPLES) {  
  
        // invoke the interpreter to run inference  
        TfLiteStatus invokeStatus = tflInterpreter->Invoke();  
        if (invokeStatus != kTfLiteOk) {  
            // raise an error if invoke fails  
            Serial.println("ERROR: Invoke failed.");  
            while (1); // endless loop to stop execution  
        }  
  
        // to store the index/value of the class with max confidence  
        uint8_t max_confidence_index;  
        float max_confidence_value;  
        // loop through the output tensor  
        for (uint8_t i = 0; i < NUM_WAVEFORMS; i++) {  
            // print the output values (probability of each class)  
            Serial.print(WAVEFORMS[i]);  
            Serial.print(": ");  
            Serial.println(tflOutputTensor->data.f[i], 4);  
            Serial.println();  
  
            // record the maximum confidence value and its index  
            if ((tflOutputTensor->data.f[i]) > max_confidence_value) {  
                max_confidence_value = tflOutputTensor->data.f[i];  
                max_confidence_index = i; // store the index  
            }  
        }  
    }  
  
    // turn on/off the output relays/leds depending on  
    // the class with max confidence  
    control_output_relays(max_confidence_index);  
  
    // built-in blue LED on when OPTA is paused
```

```
digitalWrite(LED_USER, HIGH);
delay(delay_between_inferences_ms); // pause for 5 seconds
digitalWrite(LED_USER, LOW);

samples_count = 0; // reset the counter
chrono = millis(); // save the time

} else if (millis() - chrono >= Ts) {
    // if elapsed time since the last recorded sample
    // > sample time: record a new sample, normalize it,
    // and store it in the input tensor

    // acquire sample and convert to mV
    sample = digitalSampleToMillivolts(analogRead(A0));

    // normalize sample and save it in the input tensor
    tflInputTensor->data.f[samples_count] =
        (sample - 5000.0) / 10000.0;

    samples_count++; // record that a new sample has been collected
    chrono = millis(); // save the time
}
}

/* This function converts a digital sample into
 * an analog value in millivolts, with 10V reference
 * ADC_counts: digital sample read by the ADC
 */
float digitalSampleToMillivolts(unsigned short ADC_counts)
{
    // voltage reference -> 10 V = 10000 mV
    const unsigned long Vmax_mV = 10000;

    unsigned long value_temp; // temporary value for calculations

    value_temp = ADC_counts;
    value_temp = value_temp * Vmax_mV;

    // divide by 214 by shifting right of 14 bits, then
    // convert to float
    return (float) (value_temp >> ANALOG_INPUT_RESOLUTION);
}
```

```
/* Control the output relays on the basis
 * of the class with max confidence
 * SINE      -> RELAY D0 (LED_D0)
 * TRIANGULAR -> RELAY D1 (LED_D1)
 * SQUARE    -> RELAY D2 (LED_D2)
 *
 * max_confidence_index: index of the predicted class
 */
void control_output_relays(uint8_t max_confidence_index) {
    // turn off all relays
    for (uint8_t i = 0; i < NUM_OUTPUTS; i++) {
        digitalWrite(RELAY_PIN[i], LOW);
        digitalWrite(RELAY_LED[i], LOW);
    }

    // turn on the relay/led corresponding to the
    // index with max confidence
    digitalWrite(RELAY_PIN[max_confidence_index], HIGH);
    digitalWrite(RELAY_LED[max_confidence_index], HIGH);
}
```



# Appendix C

## Case Study II

### C.5 Building a Neural Network

#### C.5.1 Training Data Collection

##### Arduino Application

```
/**
 * Acquisition of a training dataset to develop a Neural Network that
 * identifies the rotational speed of two rolling bearings from sound
 * Purpose: this sketch periodically acquires a certain number of
 * samples from analog input port A1; these are processed running a
 * FFT to obtain the spectrogram, then transmitted to the serial port.
 *
 * @author Riccardo Mennilli
 */

#include <arduinoFFT.h>

/* ---- CONFIGURATION ---- */
const uint8_t kAnalogReadResolution = 16; // resolution of A1

// time interval to average the speed measurement
const unsigned long kVelocityCalculationInterval_ms = 10000;
// debouncing time of the magnetic digital speed sensor in ms
const unsigned long kDebounceInterval_ms = 50;

// spectrogram dimensions
const uint32_t kNumCols = 32; // number of columns
const uint32_t kNumRows = 256; // number of rows

// sampling time of microphone signal
const double kSamplingFrequency_Hz = 5000.0;
```

```
unsigned int kSamplingPeriod_us = 200;

/* ----- */

// variables to store time
unsigned long chrono; // sample microphone signal
unsigned long chrono_velocity; // sample speed sensor signal
// for debouncing of speed sensor
unsigned long last_detection_time = 0;

// counter for number of magnet detections
// (volatile because updated inside ISR)
volatile uint32_t magnet_detections = 0;

// vectors to store the raw microphone samples and the complex
// number after FFT (real/imag parts)
double tempSamples[kNumCols * kNumRows];
double vReal[kNumCols];
double vImag[kNumCols];

// create FFT object
ArduinoFFT<double> FFT = ArduinoFFT<double>(vReal, vImag,
                                             kNumCols, kSamplingFrequency_Hz);

void setup() {
  // initialize serial communication
  Serial.begin(115200);
  while(!Serial); // wait for serial port

  /* ---- Setup the Arduino Peripherals ---- */
  pinMode(A0, INPUT); // A0 -> digital input port for velocity sensor
  pinMode(A1, INPUT); // A1 -> analog input port for microphone signal
  analogReadResolution(kAnalogReadResolution);

  // configure interrupt for pin A0 to detect
  // falling edge of velocity sensor
  attachInterrupt(digitalPinToInterrupt(A0),
                 count_magnet_detections_ISR, FALLING);
}

void loop() {

  double sample;
  uint16_t speed;
```

```

uint16_t i = 0; // counter for the number of microphone samples

speed = compute_velocity(); // record the current velocity

// collect kNumCols*kNumRows=8192 samples of the microphone signal
while(i < (kNumCols * kNumRows)) {
    if (micros() - chrono >= kSamplingPeriod_us) {
        /* If a time > sample time has passed since the last sample
         * that was recorded -> record a new sample
         * and store it into the tempSamples vector,
         * after the conversion to millivolts */
        sample = digitalSampleToMillivolts(analogRead(A1));
        tempSamples[i] = sample;

        i++;
        chrono = micros(); // record the time
    }
}

// run row-wise FFT and print the output to the serial port
for (uint16_t j = 0; j < kNumRows; j++) { // loop through the rows
    for (i = 0; i < kNumCols; i++) { // loop through the columns
        vReal[i] = tempSamples[j*kNumCols + i];
        vImag[i] = 0;
    }

    // run FFT
    FFT.dcRemoval(); // remove offset
    // Hann window function
    FFT.windowing(FFTWindow::Hann, FFTDirection::Forward);
    FFT.compute(FFTDirection::Forward);
    FFT.complexToMagnitude(); // convert real/imag part to magnitude

    // transmit the spectrogram entry and the measured speed of this
    // training example to the serial port
    for (uint16_t idx_col = 0; idx_col < kNumCols; idx_col++) {
        // magnitude of this specific entry of the spectrogram
        Serial.print(vReal[idx_col], 4);
        Serial.print(",");
        // raw sample in millivolts
        Serial.print(tempSamples[j*kNumCols + idx_col]);
        Serial.print(",");
        // measured speed
        Serial.println(speed);
    }
}

```



```
    }  
}  
  
/* This function converts a digital sample into  
 * an analog value in millivolts, with 10V reference  
 * ADC_counts: digital sample read by the ADC */  
double digitalSampleToMillivolts(unsigned short ADC_counts)  
{  
    // voltage reference -> 10 V = 10000 mV  
    const unsigned long Vmax_mV = 10000;  
  
    unsigned long value_temp; // temporary value for calculations  
  
    value_temp = ADC_counts;  
    value_temp = value_temp * Vmax_mV;  
  
    // divide by 216 by shifting right of 16 bits, then  
    // convert to double  
    return (double) (value_temp >> kAnalogReadResolution);  
}  
  
/* Compute the average velocity over approx. 10s, by counting the  
 * number of times the magnet is detected in that time period */  
uint16_t compute_velocity() {  
  
    uint32_t velocity;  
  
    magnet_detections = 0; // reset counter to start average  
  
    chrono_velocity = millis(); // record current time  
    // hold program while speed is recorded (10s)  
    while (millis() - chrono_velocity < kVelocityCalculationInterval_ms);  
  
    // compute the average velocity [rpm]  
    velocity = magnet_detections*60000 / (millis()-chrono_velocity);  
  
    return (uint16_t) velocity;  
}  
  
/* Interrupt Service Routine (ISR) linked to falling edge of  
 * velocity sensor (magnetic) -> increase counter after  
 * debouncing */
```

```
void count_magnet_detections_ISR() {  
    // debouncing  
    if (millis() - last_detection_time > kDebounceInterval_ms) {  
        magnet_detections++; // update counter  
        last_detection_time = millis(); // record current time  
    }  
}
```

## Data Processing

```
import numpy as np  
import pandas as pd # to parse csv dataset files  
  
# list of speed categories of the training examples  
SPEED_CATEGORIES = list(range(200,1550,50))  
  
NUM_COLS = 32 # number of columns of each spectrogram  
NUM_ROWS = 256 # number of rows of each spectrogram  
SAMPLES_PER_EXAMPLE = NUM_COLS * NUM_ROWS # 8192  
  
NUM_SPEEDS = len(SPEED_CATEGORIES) # 27  
  
inputs = []  
outputs = []  
  
# read all csv files and parse inputs and outputs  
for idx in range(NUM_SPEEDS):  
    example = SPEED_CATEGORIES[idx]  
  
    df = pd.read_csv("/content/" + str(example) + ".csv")  
    num_examples = int(df.shape[0] / SAMPLES_PER_EXAMPLE)  
  
    for i in range(num_examples):  
        tensor = []  
        for j in range(SAMPLES_PER_EXAMPLE):  
            index = i * SAMPLES_PER_EXAMPLE + j  
            tensor += [  
                ((df['power'][index]).astype('float32'))  
            ]  
        tensor = np.reshape(tensor, (NUM_ROWS, NUM_COLS))  
        output = (df['speed'][index])  
  
        inputs.append(tensor)  
        outputs.append(output)
```

```
# convert the lists to numpy matrices
inputs = np.array(inputs).astype('float32')
outputs = np.array(outputs).astype('float32')
```

## C.6 Model Deployment to the Finder OPTA

### C.6.2 Arduino Application

```
#include <TensorFlowLite.h>
#include <tensorflow/lite/micro/all_ops_resolver.h>
#include <tensorflow/lite/micro/micro_error_reporter.h>
#include <tensorflow/lite/micro/micro_interpreter.h>
#include <tensorflow/lite/schema/schema_generated.h>
#include <tensorflow/lite/version.h>

#include <arduinoFFT.h>

#include "model.h" // quantized model

// namespace to avoid conflicts with variables in other files
namespace {
    tflite::MicroErrorReporter tflErrorReporter;

    const tflite::Model* tflModel = nullptr;
    tflite::AllOpsResolver tflOpsResolver;
    tflite::MicroInterpreter* tflInterpreter = nullptr;

    TfLiteTensor* tflInputTensor = nullptr;
    TfLiteTensor* tflOutputTensor = nullptr;

    // allocate memory for the tensors
    // (constexpr to evaluate the expression at compile time)
    constexpr int tensorArenaSize = 350 * 1024; // 358.4 kB
    byte tensorArena[tensorArenaSize] __attribute__((aligned(16)));
}

/* ---- CONFIGURATION ---- */
const uint8_t kAnalogReadResolution = 16; // resolution of A1

// time interval to average the speed measurement
const unsigned long kVelocityCalculationInterval_ms = 10000;
// debouncing time of the magnetic digital speed sensor in ms
const unsigned long kDebounceInterval_ms = 50;
```

```
// spectrogram dimensions
const uint32_t kNumCols = 32; // number of columns
const uint32_t kNumRows = 256; // number of rows

// sampling time of microphone signal
const double kSamplingFrequency_Hz = 5000.0;
unsigned int kSamplingPeriod_us = 200;

/* ----- */

// variables to store time
unsigned long chrono; // sample microphone signal
unsigned long chrono_velocity; // sample speed sensor signal
// for debouncing of speed sensor
unsigned long last_detection_time = 0;

// counter for number of magnet detections
// (volatile because updated inside ISR)
volatile uint32_t magnet_detections = 0;

// to perform the average of multiple inferences
double running_sum; // sum for the avg
uint32_t sum_elements_count; // number of entries for the avg

// vectors to store the raw microphone samples and the complex
// number after FFT (real/imag parts)
double tempSamples[kNumCols * kNumRows];
double vReal[kNumCols];
double vImag[kNumCols];

// create FFT object
ArduinoFFT<double> FFT = ArduinoFFT<double>(vReal, vImag,
      kNumCols, kSamplingFrequency);

void setup() {
  // initialize serial communication
  Serial.begin(115200);
  while(!Serial); // wait for serial port

  /* ---- Setup the Machine Learning Infrastructure ---- */
  // map model into a usable data structure
  tflModel = tflite::GetModel(model);
  if (tflModel->version() != TFLITE_SCHEMA_VERSION) {
    // raise error if model version is incompatible
    Serial.print("ERROR: Model is schema version ");
  }
}
```

```

Serial.print(tflModel->version());
Serial.print(", not equal to supported version ");
Serial.println(TFLITE_SCHEMA_VERSION);

while(1); // endless loop to stop execution
}

// build the interpreter to run the model
static tflite::MicroInterpreter static_interpreter(tflModel,
    tflOpsResolver, tensorArena, tensorArenaSize, &tflErrorReporter);
tflInterpreter = &static_interpreter;

// allocate memory for the tensors
TfLiteStatus allocate_status = tflInterpreter->AllocateTensors();
if (allocate_status != kTfLiteOk) {
    // raise error if allocation fails
    Serial.print("ERROR: AllocateTensors() failed.");
    while(1); // endless loop to stop execution
}

// obtain pointers to input and output tensors
tflInputTensor = tflInterpreter->input(0);
tflOutputTensor = tflInterpreter->output(0);

/* ---- Setup the Arduino Peripherals ---- */
pinMode(A0, INPUT); // A0 -> digital input port for velocity sensor
pinMode(A1, INPUT); // A1 -> analog input port for microphone signal
analogReadResolution(kAnalogReadResolution);

// configure interrupt for pin A0 to detect
// falling edge of velocity sensor
attachInterrupt(digitalPinToInterrupt(A0),
    count_magnet_detections_ISR, FALLING);
}

void loop() {

    double sample;
    uint16_t speed;

    uint16_t i = 0; // counter for the number of microphone samples

    // if more than 10s have elapsed from last output: record the speed
    // measurement from the sensor for debugging, print the average of
    // the inferences over the 10s, and reset the average

```

```

if (millis() - chrono_velocity > kVelocityCalculationInterval_ms) {
    // debugging: record the current speed
    speed = compute_velocity();

    // log actual and average predicted speed to serial port
    Serial.print("Average predicted speed: ");
    Serial.print(running_sum / sum_elements_count); // compute average
    Serial.println(" rpm");
    Serial.print("Measured speed: ");
    Serial.print(speed);
    Serial.println(" rpm");
    Serial.println();

    // initialize average
    running_sum = 0;
    sum_elements_count = 0;
}

// collect kNumCols*kNumRows=8192 samples of the microphone signal
while(i < (kNumCols * kNumRows)) {
    if (micros() - chrono >= kSamplingPeriod_us) {
        /* If a time > sample time has passed since the last sample
         * that was recorded -> record a new sample
         * and store it into the tempSamples vector,
         * after the conversion to millivolts */
        sample = digitalSampleToMillivolts(analogRead(A1));
        tempSamples[i] = sample;

        i++;
        chrono = micros(); // record the time
    }
}

// run row-wise FFT
for (uint16_t j = 0; j < kNumRows; j++) { // loop through the rows
    for (i = 0; i < kNumCols; i++) { // loop through the columns
        vReal[i] = tempSamples[j*kNumCols + i];
        vImag[i] = 0;
    }

    // run FFT
    FFT.dcRemoval(); // remove offset
    // Hann window function
    FFT.windowing(FFTWindow::Hann, FFTDirection::Forward);
    FFT.compute(FFTDirection::Forward);
}

```

```

FFT.complexToMagnitude(); // convert real/imag part to magnitude

for (uint16_t idx_col = 0; idx_col < kNumCols; idx_col++) {
    // save magnitude out of FFT into the input tensor
    tflInputTensor->data.f[j*kNumCols+idx_col] = vReal[idx_col];
}
}

// invoke the interpreter to run inference
TfLiteStatus invokeStatus = tflInterpreter->Invoke();
if (invokeStatus != kTfLiteOk) {
    // raise an error if invoke fails
    Serial.println("ERROR: Invoke failed.");
    while (1); // endless loop to stop execution
}

// add element to running sum for the average
running_sum += tflOutputTensor->data.f[0];
sum_elements_count++; // update the entries counter
}

/* This function converts a digital sample into
 * an analog value in millivolts, with 10V reference
 * ADC_counts: digital sample read by the ADC */
double digitalSampleToMillivolts(unsigned short ADC_counts)
{
    // voltage reference -> 10 V = 10000 mV
    const unsigned long Vmax_mV = 10000;

    unsigned long value_temp; // temporary value for calculations

    value_temp = ADC_counts;
    value_temp = value_temp * Vmax_mV;

    // divide by 2^16 by shifting right of 16 bits, then
    // convert to double
    return (double) (value_temp >> kAnalogReadResolution);
}

/* Compute the average velocity, by counting the number of
 * times the magnet is detected in a given time period */
uint16_t compute_velocity() {

```

```
uint32_t velocity;

// compute the average velocity
velocity = magnet_detections*60000 / (millis()-chrono_velocity);

magnet_detections = 0; // reset counter to start average
chrono_velocity = millis(); // record current time

return (uint16_t) velocity;
}

/* Interrupt Service Routine linked to falling edge of velocity
 * sensor (magnetic) -> increase counter after debouncing */
void count_magnet_detections_ISR() {
    // debouncing
    if (millis() - last_detection_time > kDebounceInterval_ms) {
        magnet_detections++; // update counter
        last_detection_time = millis(); // record current time
    }
}
```



## C.7 Overall System Testing

**200 rpm** (Mean Error = +14.05 rpm)

<i>Predicted</i>	219.5	224.6	219.5	224.6	219.5	224.6	219.5	224.6	219.5	224.6
<i>Measured</i>	213.0	203.0	213.0	203.0	213.0	203.0	213.0	203.0	213.0	203.0

**250 rpm** (Mean Error = +23.84 rpm)

<i>Predicted</i>	254.3	286.3	275.4	270.8	237.7	321.4	278.3	310.9	296.1	236.2
<i>Measured</i>	252.0	254.0	254.0	252.0	252.0	252.0	253.0	254.0	254.0	252.0

**300 rpm** (Mean Error = +32.91 rpm)

<i>Predicted</i>	347.5	303.3	306.6	364.7	351.6	350.4	396.2	349.8	292.0	300.0
<i>Measured</i>	302.0	304.0	304.0	303.0	303.0	303.0	304.0	304.0	302.0	304.0

**350 rpm** (Mean Error = +19.01 rpm)

<i>Predicted</i>	365.4	406.8	436.2	355.2	387.6	343.6	345.9	337.7	339.5	405.2
<i>Measured</i>	352.0	352.0	354.0	354.0	353.0	352.0	353.0	355.0	353.0	355.0

**400 rpm** (Mean Error = +52.77 rpm)

<i>Predicted</i>	463.2	472.8	469.7	418.2	441.3	472.8	448.7	494.5	400.6	488.9
<i>Measured</i>	403.0	404.0	405.0	405.0	405.0	405.0	405.0	404.0	403.0	404.0

**450 rpm** (Mean Error = +15.63 rpm)

<i>Predicted</i>	454.0	449.6	434.4	462.7	457.6	473.0	446.1	549.8	491.6	476.5
<i>Measured</i>	455.0	453.0	455.0	454.0	452.0	454.0	456.0	454.0	454.0	452.0

**500 rpm** (Mean Error = +29.47 rpm)

<i>Predicted</i>	543.4	579.1	595.5	517.6	575.7	514.2	481.8	491.3	519.4	516.7
<i>Measured</i>	503.0	503.0	503.0	506.0	504.0	505.0	502.0	504.0	505.0	505.0

**550 rpm** (Mean Error = +46.54 rpm)

<i>Predicted</i>	585.4	638.6	576.4	602.4	597.2	631.5	564.2	609.6	602.3	608.8
<i>Measured</i>	552.0	552.0	554.0	556.0	556.0	556.0	554.0	557.0	557.0	557.0

**600 rpm** (Mean Error = +58.72 rpm)

<i>Predicted</i>	688.6	683.9	594.3	694.6	632.9	685.1	683.8	622.6	655.7	683.7
<i>Measured</i>	603.0	603.0	603.0	602.0	603.0	604.0	605.0	607.0	602.0	606.0

**650 rpm** (Mean Error = +26.21 rpm)

<i>Predicted</i>	735.1	720.9	657.6	673.1	673.6	662.1	670.5	640.4	684.2	684.6
<i>Measured</i>	652.0	652.0	654.0	653.0	652.0	657.0	654.0	657.0	653.0	656.0

**700 rpm** (Mean Error = +21.05 rpm)

<i>Predicted</i>	688.9	735.3	724.4	778.9	744.5	778.1	735.2	681.7	680.6	699.9
<i>Measured</i>	702.0	706.0	702.0	703.0	706.0	705.0	702.0	704.0	704.0	703.0

**750 rpm** (Mean Error = +34.29 rpm)

<i>Predicted</i>	836.5	777.2	811.1	760.3	844.0	805.0	754.8	743.2	797.9	762.9
<i>Measured</i>	754.0	757.0	753.0	754.0	757.0	754.0	754.0	753.0	755.0	759.0

**800 rpm** (Mean Error = +42.97 rpm)

<i>Predicted</i>	821.3	844.0	855.3	833.6	877.3	797.4	897.0	880.1	820.8	853.9
<i>Measured</i>	802.0	803.0	802.0	807.0	808.0	808.0	804.0	809.0	806.0	802.0

<b>850 rpm</b> (Mean Error = +25.77 rpm)										
<i>Predicted</i>	896.4	896.0	849.3	844.1	877.8	929.8	852.2	890.1	845.2	933.8
<i>Measured</i>	854.0	852.0	856.0	859.0	858.0	860.0	855.0	855.0	853.0	855.0
<b>900 rpm</b> (Mean Error = +29.54 rpm)										
<i>Predicted</i>	973.1	959.4	936.4	906.4	952.3	902.1	903.7	983.4	895.1	957.5
<i>Measured</i>	908.0	907.0	905.0	906.0	909.0	910.0	906.0	906.0	910.0	907.0
<b>950 rpm</b> (Mean Error = +61.66 rpm)										
<i>Predicted</i>	1049	1030	1025	1008	995.2	976.4	1028	1001	1023	1048
<i>Measured</i>	956.0	958.0	959.0	955.0	952.0	960.0	955.0	958.0	960.0	954.0
<b>1000 rpm</b> (Mean Error = +35.1 rpm)										
<i>Predicted</i>	1061	1075	1061	1058	1008	1037	1092	1008	1005	1012
<i>Measured</i>	1009	1010	1006	1010	1005	1004	1006	1002	1010	1004
<b>1050 rpm</b> (Mean Error = +39.7 rpm)										
<i>Predicted</i>	1101	1035	1081	1092	1130	1105	1060	1138	1122	1097
<i>Measured</i>	1060	1055	1058	1054	1052	1055	1054	1062	1062	1052
<b>1100 rpm</b> (Mean Error = +51.8 rpm)										
<i>Predicted</i>	1189	1178	1145	1081	1091	1141	1194	1181	1192	1196
<i>Measured</i>	1111	1106	1105	1108	1102	1106	1104	1110	1109	1109
<b>1150 rpm</b> (Mean Error = +31.8 rpm)										
<i>Predicted</i>	1226	1201	1141	1197	1135	1220	1209	1201	1157	1215
<i>Measured</i>	1156	1155	1161	1154	1163	1163	1156	1156	1160	1160
<b>1200 rpm</b> (Mean Error = +44.6 rpm)										
<i>Predicted</i>	1259	1224	1222	1296	1233	1208	1303	1268	1307	1213
<i>Measured</i>	1210	1206	1202	1212	1211	1208	1210	1211	1205	1212
<b>1250 rpm</b> (Mean Error = +58.7 rpm)										
<i>Predicted</i>	1341	1311	1322	1317	1314	1354	1261	1297	1341	1299
<i>Measured</i>	1262	1258	1259	1252	1259	1260	1258	1252	1258	1252
<b>1300 rpm</b> (Mean Error = +36.2 rpm)										
<i>Predicted</i>	1331	1293	1398	1346	1399	1329	1372	1292	1351	1329
<i>Measured</i>	1311	1304	1310	1313	1304	1307	1302	1304	1310	1313
<b>1350 rpm</b> (Mean Error = +39.9 rpm)										
<i>Predicted</i>	1440	1427	1353	1359	1375	1390	1403	1373	1439	1435
<i>Measured</i>	1356	1365	1362	1356	1360	1357	1360	1365	1360	1354
<b>1400 rpm</b> (Mean Error = +47.2 rpm)										
<i>Predicted</i>	1414	1476	1488	1501	1451	1413	1462	1413	1439	1492
<i>Measured</i>	1405	1410	1403	1411	1406	1411	1406	1411	1407	1407
<b>1450 rpm</b> (Mean Error = +40.3 rpm)										
<i>Predicted</i>	1468	1467	1482	1550	1500	1536	1482	1495	1554	1456
<i>Measured</i>	1458	1465	1457	1462	1460	1453	1458	1465	1452	1457
<b>1500 rpm</b> (Mean Error = +31.3 rpm)										
<i>Predicted</i>	1608	1542	1531	1495	1517	1559	1562	1520	1568	1497
<i>Measured</i>	1503	1512	1515	1503	1511	1504	1512	1511	1502	1513

Table C.1: Deployed CNN Model - Test Results



# Appendix D

## Case Study III

### D.1 Arduino Application Code

The application code is organized into three .ino files and one header file for improved clarity. Recall how one of the advantages of the Arduino IDE is the automatic merging of all project files during compilation, eliminating the need for function prototypes and a rigorous order for the function definitions (section 2.2).

#### **wave\_energy\_opta\_control.ino**

```
#include "configuration.h"

// allocate parallel thread - LED management
static rtos::Thread ledsManagementThread;

// Client object to connect using SSL, compatible with
// standard plain connections methods
WiFiSSLClient client;

// Last time a weather update was attempted (intialized so that
// the weather is fetched at the start of the program)
unsigned long prevWeatherFetchAttempt =
    60000 * kWeatherUpdateInterval_min;
// Last time a successful weather update was fetched
unsigned long prevWeatherFetchSuccess;

// Last time (hh format) a successful weather update was fetched
int8_t fetchHour = -1;

// Flags to activate or reset emergency
// (volatile because handled by Interrupt Service Routine)
volatile uint8_t emergencyPB; // user command
volatile uint8_t resetEmergencyPB; // user command
```

```

uint8_t emergencyWaveHeight;

// To store the wave height forecast for the next 3 days in intervals
// of 6 hours (24h/6h = 4 intervals in a day, 4*3 DAYS = 12 elements)
double max_wave_height_6h[24 / kIntervalLength_h*kForecastLength_days];

void setup() {
  // Initialize serial port and wait for it to open:
  Serial.begin(115200);
  delay(1500);

  // Defined in configuration.h -> for Arduino Cloud variables
  initProperties();

  // Connect to Arduino IoT Cloud
  // [second parameter = false -> disable Arduino Cloud watchdog]
  ArduinoCloud.begin(ArduinoIoTPreferredConnection);

  setDebugMessageLevel(2);
  ArduinoCloud.printDebugInfo();

  /* Configure the required DIGITAL Input/Output pins */
  const uint8_t DIGITAL_INPUT_PIN[] = {A0, A1, A6, A7};
  const uint8_t NUM_INPUTS_DIGITAL = 4;
  for (uint8_t i = 0; i < NUM_INPUTS_DIGITAL; i++) {
    pinMode(DIGITAL_INPUT_PIN[i], INPUT);
  }

  const uint8_t RELAY_PIN[] = {D0, D1};
  const uint8_t RELAY_LED[] = {LED_D0, LED_D1};
  const uint8_t NUM_OUTPUTS = 2;
  for (uint8_t i = 0; i < NUM_OUTPUTS; i++) {
    pinMode(RELAY_PIN[i], OUTPUT);
    pinMode(RELAY_LED[i], OUTPUT);
  }

  pinMode(LED_USER, OUTPUT); // blue LED, blinking when emergency active
  pinMode(LED_R, OUTPUT); // red LED, on when WiFi connection failed
  pinMode(LED_BUILTIN, OUTPUT); // green LED, on when connected to WiFi

  // Initialize interrupts associated to the input pins
  attachInterrupt(digitalPinToInterrupt(A0), EmergencyPB_ISR, FALLING);
  attachInterrupt(digitalPinToInterrupt(A1), EmergencyResetPB_ISR, RISING);

  // Start the parallel thread

```

```

ledsManagementThread.start(manageOptaLeds);

// Initialize the wave height forecast to -1 to prevent
// "emergency reset" when no data has been downloaded yet
for (uint8_t i = 0; i < (24 / kIntervallLength_h*kForecastLength_days);
    i++) {
    max_wave_height_6h[i] = -1.0;
}
// Initialize system with emergency active for safety
emergencyWaveHeight = true;
}

void loop() {

// To store the return value of function fetchWaveHeight()
// if negative -> error
// if positive -> currentHour
int8_t fetchWeatherReturn;

ArduinoCloud.update(); // maintain connection

/* ----- WAVE HEIGHT FORECAST ----- */
// Fetch an update to the wave height forecast every
// kWeatherUpdateInterval_min if connected to the internet
if (millis() - prevWeatherFetchAttempt > (unsigned long)
    (kWeatherUpdateInterval_min*60000) && WiFi.status()==WL_CONNECTED) {

// fetch forecast and save it in max_wave_height_6h[]
fetchWeatherReturn = fetchWaveHeight(max_wave_height_6h);

if (fetchWeatherReturn >= 0) { // if no errors (-1 if error)
    fetchHour = fetchWeatherReturn; // update the fetch time (hour)
    // log the forecast to Google Sheets
    logToSpreadsheet(max_wave_height_6h,
        (24 / kIntervallLength_h * kForecastLength_days));

    // save the time of last SUCCESS in updating the forecast
    prevWeatherFetchSuccess = millis();
}

// save the time of last ATTEMPT at updating the forecast
prevWeatherFetchAttempt = millis();

// extract the forecast for the current 6h interval and next 6h
currentMaxWaveHeight_m = findWaveHeight(0);

```

```

next6hMaxWaveHeight_m = findWaveHeight(1);
// if wave height in current 6h period or in next 6h > threshold
// (or < 0: ERROR) --> ACTIVATE EMERGENCY */
if (currentMaxWaveHeight_m >= kEmThreshold_m ||
    next6hMaxWaveHeight_m >= kEmThreshold_m ||
    currentMaxWaveHeight_m < 0 || next6hMaxWaveHeight_m < 0) {
    emergencyWaveHeight = true;
} else {
    emergencyWaveHeight = false;
}
}

/* --- STATE MACHINE ACTIONS (EMERGENCY / NORMAL) --- */
switch(state) {
    /* Normal State */
    case 1:
        /* Check if the float is in the working position.
         * If NOT activate the relay to lower it */
        lowerFloat();

        /* Transition condition NORMAL -> EMERGENCY:
         * if the emergency is commanded by operator or
         * wave height > threshold --> ENTER EMERGENCY */
        if (emergencyPB || emergencyWaveHeight) {
            state = 0;
        }
        break;

    /* Emergency State */
    case 0:
        /* Check if the float is in the safe position.
         * If NOT activate the relay to raise it */
        raiseFloat();

        /* Transition condition EMERGENCY -> NORMAL:
         * - if the emergency pushbutton was pressed, check it is released
         *   and the "Reset Emergency" command was issued;
         * - check if wave height is below threshold
         * The emergency can only be reset if the float is in the SAFE
         * (raised) position */
        if (!emergencyWaveHeight && (!emergencyPB || resetEmergencyPB)
            && digitalRead(A6)) {
            state = 1;
            resetEmergencyPB = false;
            emergencyPB = false;
        }
    }
}

```

```

    }
    break;

    default:
        /* If state NOT 0 or 1: enter emergency by setting it to 0 */
        state = 0; // enter emergency
        break;
    }

    // flag to display the status of the float in the HMI
    floatDown = digitalRead(A7);
}

/* ----- PARALLEL THREAD ----- */
/* Callback to manage the built-in LEDs of the Opta */
void manageOptaLeds() {
    while (1) {
        /* To provide a visual indication of the Wi-Fi status:
        * - steady green LED (LED_BUILTIN): Wi-Fi connected
        * - steady red LED (LEDR): Wi-Fi connection failed */
        WiFiStatusLED();

        if (state == 1) { // if normal operations (no emergency)
            digitalWrite(LED_USER, LOW); // turn off the blue LED
        } else { // if emergency
            // blink the emergency blue LED with a period of 500 ms
            blinkLED(LED_USER, 500);
        }
    }
}

/* ----- ARDUINO CLOUD ON_CHANGE UPDATE FUNCTIONS ----- */
/* emergencyStop_cloud is READ_WRITE: onEmergencyStopCloudChange()
* is executed every time a new value is received from IoT Cloud */
void onEmergencyStopCloudChange() {
    // If emergency command from Arduino Cloud dashboard
    // -> set the emergencyPB volatile flag
    if (emergencyStop_cloud) {
        emergencyPB = true;
    }
}

/* resetEmergency_cloud is READ_WRITE: onResetEmergencyCloudChange()
* is executed every time a new value is received from IoT Cloud */
void onResetEmergencyCloudChange() {

```



```

// If "Reset Emergency" command from Arduino Cloud dashboard
// -> set the resetEmergencyPB flag IF the emergency is active,
// the wave height is below threshold, and the float is raised
if (resetEmergency_cloud && state == 0 && !emergencyWaveHeight
    && digitalRead(A6)) {
    resetEmergencyPB = true;
}
}

/* ----- INTERRUPT SERVICE ROUTINES ----- */
/* ISR called at the FALLING EDGE of pin I1 (A0), connected to the
 * physical emergency pushbutton (normally closed) */
void EmergencyPB_ISR(){
    emergencyPB = true;
}

/* ISR called at the RISING EDGE of pin I2 (A1), connected to the
 * physical "Reset Emergency" pushbutton (normally open) */
void EmergencyResetPB_ISR(){
    // If "Reset Emergency" command from operator
    // -> set the resetEmergencyPB flag IF the emergency is active,
    // the wave height is below threshold, and the float is raised
    if (!emergencyWaveHeight && state == 0 && digitalRead(A6)) {
        resetEmergencyPB = true;
    }
}
}

```

#### utility.ino

```

/**
 * Collection of utility functions for the Opta
 *
 * @author Riccardo Mennilli
 * @version 0.3 24/05/2024
 */

// time variable to blink LEDs
unsigned long prevBlinkTime = 0;

/**
 * Blink a built-in LED of the Opta with a specific period in ms.
 * It can also be used to toggle any digital pin.
 *
 * @param LED Reference to the LED (pin in general) to toggle
 * @param blinkPeriod_ms Period in ms

```

```
*/
void blinkLED(uint8_t LED, unsigned long blinkPeriod_ms) {
  if (millis()-prevBlinkTime > blinkPeriod_ms) {
    // if time elapsed from previous toggle > period: toggle the pin
    digitalWrite(LED, !digitalRead(LED));
    prevBlinkTime = millis(); // record the time
  }
}

/**
 * To provide a visual indication of the Wi-Fi status:
 * - steady green LED (LED_BUILTIN): Wi-Fi connected
 * - steady red LED (LEDR): Wi-Fi connection failed
 */
void WiFiStatusLED() {
  if(WiFi.status() != WL_CONNECTED) { // if no Wi-Fi connection
    digitalWrite(LED_BUILTIN, LOW); // turn off the green LED
    digitalWrite(LEDR, HIGH); // turn on the red LED (steady)
  } else if (WiFi.status() == WL_CONNECTED) { // if Wi-Fi is connected
    digitalWrite(LEDR, LOW); // turn off the red LED
    digitalWrite(LED_BUILTIN, HIGH); // turn on the green LED (steady)
  }
}

/**
 * Lower the float into the water by energizing relay D0
 * D0      -> relay to lower the float
 * LED_D0 -> associated LED, to visualize its state
 * A7      -> limit switch indicating the float is DOWN
 */
void lowerFloat() {
  // De-energize the relay to raise the float (D1)
  digitalWrite(D1, LOW);
  digitalWrite(LED_D1, LOW);

  if (!digitalRead(A7)) { // if the float is NOT down
    digitalWrite(D0, HIGH); // energize relay D0 to lower the float
    digitalWrite(LED_D0, HIGH);
  } else if (digitalRead(D0)) { // if the float is DOWN
    digitalWrite(D0, LOW); // de-energize the relay
    digitalWrite(LED_D0, LOW);
  }
}

/**
```

```
* Raise the float out of the water by energizing relay D1
* D1      -> relay to raise the float
* LED_D1 -> associated LED, to visualize its state
* A6      -> limit switch indicating the float is UP
*/
void raiseFloat() {
    // De-energize the relay to lower the float (D0) if it is active
    digitalWrite(D0, LOW);
    digitalWrite(LED_D0, LOW);

    if (!digitalRead(A6)) { // if the float is NOT raised
        digitalWrite(D1, HIGH); // energize relay D1 to raise the float
        digitalWrite(LED_D1, HIGH);
    } else if (digitalRead(D1)) { // if the float is UP
        digitalWrite(D1, LOW); // de-energize the relay
        digitalWrite(LED_D1, LOW);
    }
}
```

#### api.ino

```
/**
* Collection of custom functions to control the required API calls:
* GET  -> from api.open-meteo.com to download wave height forecast
* POST -> to Google Sheets to log data
*
* @author Riccardo Mennilli
* @version 0.1 03/05/2024
*/

/**
* Fetches the maximum wave height forecast from open-meteo.com
* Forecast length = 3 days
* Location Coordinates: 44.3N, 8.72E
*
* @param maxWaveHeight array containing the max wave height in m,
* in intervals of 6h, for 3 days
* @return the hour the weather data was fetched
*/
int8_t fetchWaveHeight(double maxWaveHeight[]) {
    // initialize return value:
    // negative value -> ERROR in fetching the weather data
    // positive value -> the hour the weather data was fetched
    int8_t returnCurrentHour = -1;
}
```

```
double coordinates[2] = {44.3, 8.72}; // desired coordinates
double maxCoordinateError = 0.3;

// open-meteo API server to fetch marine weather forecasts
const char* openMeteoServer = "marine-api.open-meteo.com";
// API endpoint path and query to get required data
String path = "/v1/marine";
String queryCoordinates = "?latitude=" + (String) coordinates[0] +
    "&longitude=" + (String) coordinates[1];
String queryParameters = "&current=wave_height&hourly=wave_height"
    "&timezone=Europe%2FBerlin&forecast_days=3";

// connect to server (port 443: SSL)
if (client.connect(openMeteoServer, 443)) {
    // send HTTP GET request if connection successful
    client.print("GET " + path + queryCoordinates + queryParameters);
    client.println(" HTTP/1.1");
    client.print("Host: ");
    client.println(openMeteoServer);
    client.println("Connection: close");
    client.println();

    // skip HTTP header
    char startOfJson = '{';
    client.find(startOfJson);

    // parse JSON response
    JSONVar doc; // JSON object to store the response
    String payload = "{" + client.readStringUntil('\n');
    doc = JSON.parse(payload);

    // check is parsing was successful
    if (JSON.typeof(doc) == "undefined") {
        Serial.println("- Parsing failed (open-meteo)!");
        client.stop();

        returnCurrentHour = -1; // error: return -1
        return returnCurrentHour;
    }

    // parse latitude and longitude to check location
    double latitude = doc["latitude"];
    double longitude = doc["longitude"];
    // check if coordinates of the fetched weather are reasonably
    // close to desired location
}
```

```

if (abs(latitude - coordinates[0]) > maxCoordinateError ||
    abs(longitude - coordinates[1]) > maxCoordinateError) {
    Serial.println("- Coordinates error!");
    client.stop();

    returnCurrentHour = -1; // error: return -1
    return returnCurrentHour;
}

// parse JSON array containing the forecast for wave height
JSONVar hourly_wave_height = doc["hourly"]["wave_height"];
// declare an array to store the forecast and initialize it to 0
double wave_height_forecast[hourly_wave_height.length()];
memset(maxWaveHeight, 0, sizeof(maxWaveHeight));

// go through the hourly forecasts in batches of 6 hours, and
// save the max for each 6h period into maxWaveHeight
for (int i = 0; i < hourly_wave_height.length()/6; i++) {
    for (int j = 0; j < 6; j++) {
        // go through the hourly forecasts
        wave_height_forecast[i*6+j] = hourly_wave_height[i*6+j];
        // if the value is > than the max stored, update it
        if (wave_height_forecast[i*6+j] > maxWaveHeight[i]) {
            maxWaveHeight[i] = hourly_wave_height[i*6+j];
        }
    }
}

// parse the current time, format "2024-05-03T15:00",
// extracting the hour (-> "15")
JSONVar current = doc["current"];
const char* current_time = current["time"];
char current_time_only_hour[] = {current_time[11],
    current_time[12], '\0'};

// convert the time string to an integer
returnCurrentHour = atoi(current_time_only_hour);

} else {
    Serial.println("- Failed to connect to server (open-meteo)!");
    returnCurrentHour = -1; // error: return -1
}
client.stop();

return returnCurrentHour;

```

```
}

/**
 * Sends an array to Google Sheets for logging (HTTP POST request)
 *
 * @param variableValue array containing the values to share
 * @param numberOfEntries length of the array variableValue
 */
uint8_t logToSpreadsheet(double variableValue[],
                        uint8_t numberOfEntries) {

    JSONVar object; // JSON object to share

    String jsonField = "wave_height"; // name of the JSON field

    // copy all entries of array variableValue into the JSON object
    for (uint8_t i = 0; i < numberOfEntries; i++) {
        object[jsonField][i] = variableValue[i];
    }

    if (JSON.typeof(object) == "undefined") {
        return 1; // exit if error detected
    }

    // JSON.stringify(myVar) to convert the JSONVar to a String
    String jsonString = JSON.stringify(object);

    if (sendPostRequest(jsonString) == 1) { // send POST request
        return 1; // exit if error detected
    }

    return 0;
}

/**
 * Sends the HTTP POST request to log data to a Google spreadsheet
 *
 * @param postData String containing the information to send
 */
uint8_t sendPostRequest(String postData) {
    uint8_t requestStatus; // return value

    // Google Webapp url to log data to a spreadsheet
    const char* googleScriptServer = "script.google.com";
    String googleScriptPath = "/macros/s/";
```

```

String googleScriptID = "*****";

// connect to server (port 443: SSL)
if (client.connect(googleScriptServer, 443)) {
    // send HTTP POST request
    client.print("POST " + googleScriptPath + googleScriptID);
    client.println(" HTTP/1.1");
    client.print("Host: ");
    client.println(googleScriptServer);
    client.println("Connection: close");
    client.println("Content-Type: application/json");
    client.print("Content-Length: ");
    client.println(postData.length());
    client.println();
    client.println(postData);

    requestStatus = 0;
} else {
    Serial.println("- Failed to connect to server (Google)!");
    requestStatus = 1; // error
}

client.stop();

return requestStatus;
}

/**
 * To find the maximum wave height (interval of 6h)
 * within an array that stores the forecasts for 3 days
 *
 * @param hIndex [hour index] integer to indicate the desired 6h period:
 *     -1 -> previous 6h
 *     0  -> now
 *     +1 -> next 6h
 * @return waveHeight_m -> the requested max wave height value in meters
 *     (-1.0 if error)
 */
double findWaveHeight(uint8_t hIndex) {

    double waveHeight_m; // return value
    // num of hours elapsed from previous successful weather download
    int hoursFromPrevFetchSuccess = (millis() - prevWeatherFetchSuccess)
        / 360000;

```

```

// check for errors: time variables must be non-negative
if (fetchHour < 0 || hoursFromPrevFetchSuccess < 0) {
    waveHeight_m = -1.0; // exit code -1: error
    return waveHeight_m;
}

// if no forecast available for the next 6 hours: enter emergency
if ((fetchHour+hoursFromPrevFetchSuccess)/kIntervalLength_h + hIndex
    > sizeof(max_wave_height_6h)/sizeof(max_wave_height_6h[0])) {
    // if the desired 6h interval is outside of array bounds: error
    waveHeight_m = -1.0; // exit code -1: error
} else { // forecast available
    // extract the desired value from "max_wave_height_6h"
    waveHeight_m = max_wave_height_6h[(fetchHour +
        hoursFromPrevFetchSuccess) / kIntervalLength_h + hIndex];
}

return waveHeight_m;
}

```

#### configuration.h

```

#include <ArduinoIoTCloud.h>
#include <Arduino_ConnectionHandler.h>

#include <Arduino_JSON.h>

#include <mbed.h> // for parallel threads
#include <rtos.h>

/* ----- CONFIGURATION ----- */
const uint8_t kIntervalLength_h = 6; // data saved in 6-hour intervals
// forecasts are downloaded for three days
const uint8_t kForecastLength_days = 3;
// time in minutes between weather updates
const uint8_t kWeatherUpdateInterval_min = 60;
// max wave height (meters) before emergency is triggered
const double kEmThreshold_m = 1;

// network information
const char SSID[] = "****"; // SSID (name)
const char PASS[] = "****"; // password
/* ----- END CONFIGURATION ----- */

/* ----- ARDUINO CLOUD VARIABLES ----- */

```



```

// Max wave height in current 6h interval and in the next 6h
// initialized to -1 for safety (non-physical value)
float currentMaxWaveHeight_m = -1.0;
float next6hMaxWaveHeight_m = -1.0;

// emergency command from Arduino Cloud dashboard
bool emergencyStop_cloud;
// reset emergency command from Arduino Cloud dashboard
bool resetEmergency_cloud;
// state of the float, true when float in working position (lowered)
bool floatDown;

/* State of the device:
 * 1: NORMAL
 * 0: EMERGENCY */
int state;

void initProperties(){
  ArduinoCloud.addProperty(currentMaxWaveHeight_m,READ,ON_CHANGE,NULL);
  ArduinoCloud.addProperty(floatDown, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(state, READ, ON_CHANGE, NULL);
  ArduinoCloud.addProperty(emergencyStop_cloud, READWRITE, ON_CHANGE,
    onEmergencyStopCloudChange);
  ArduinoCloud.addProperty(resetEmergency_cloud, READWRITE, ON_CHANGE,
    onResetEmergencyCloudChange);
}

WiFiConnectionHandler ArduinoIoTPreferredConnection(SSID, PASS);

```

## D.2 Google Apps Script

```

// get active spreadsheet -> ArduinoOpta_Log_Spreadsheet
const ss = SpreadsheetApp.getActiveSpreadsheet();
// get the desired sheet -> WaveHeight
const sheetWaves = ss.getSheetByName("WaveHeight");

const MAX_ROWS = 5000; // max number of rows to display
const HEADER_ROW = 2; // row index of header
const TIME_COL = 1; // column index of the timestamp column

function doPost(e) {
  // JSON object with the POST request from the Opta
  var cloudData = JSON.parse(e.postData.contents);

```

```
// extract the wave_height field from the JSON object
if (cloudData.hasOwnProperty("wave_height")) {
  var waveHeight = cloudData.wave_height;
  var waveHeightLength = cloudData.wave_height.length;

  var lastRowWaves = sheetWaves.getLastRow();
  // delete last row to maintain constant the total number of rows
  if (lastRowWaves > MAX_ROWS + HEADER_ROW - 1) {
    sheetWaves.deleteRow(lastRowWaves);
  }
  // insert new row after deleting the last one
  sheetWaves.insertRowAfter(HEADER_ROW);

  var range = sheetWaves.getRange('A3:Z3');
  range.setFontColor('#000000'); // formatting options
  range.setFontSize(10);
  range.setFontWeight('normal');

  // write values in the respective columns
  for (var col=1+TIME_COL; col<=TIME_COL+waveHeightLength; col++) {
    sheetWaves.getRange(HEADER_ROW+1, col).setValue(
      waveHeight[col-1-TIME_COL]);
  }

  // write timestamp
  var timestampWaves = new Date();
  sheetWaves.getRange(HEADER_ROW+1, TIME_COL).setValue(
    timestampWaves).setNumberFormat("yyyy-MM-dd HH:mm:ss");
}
}
```