Università degli Studi di Messina

Department of Engineering

Doctoral Programme in Cyber Physical Systems
XXXIII Cycle

Deep Learning Techniques for Intelligent Cyber Physical Systems: Towards a New Generation of Smart and Autonomous Things

Student:

Fabrizio De Vita

Advisor:

Prof. Dario Bruneo

Co-Advisor:

Prof. Sajal K. Das

Academic Year 2019 - 2020

Contents

Abstract

We are living an era constantly surrounded by objects blended with the environment. During the last decade, the advent of the Internet of Things totally revolutionized the way we interact with these objects, and acted as a catalyzer for the creation of a wide variety of Cyber Physical Systems exposing services to support the human being under different aspects of his life. Today, the maximum expression of Cyber Physical Systems are smart environments whose task is to simplify (or automate) certain types of operations to help their "guests" during the daily activities. Cloud and Edge computing paradigms play a fundamental role for the realization of these systems; the first providing storage and high performance computing capabilities, the second allowing a pervasive monitoring and an early processing of the data gathered from sensors. In such a context, Artificial Intelligence is another very important player which gained a lot of interest during these years, thanks also to the advancement in ICT and the huge amount of available data. Leveraging the above mentioned technologies, it allows the implementation of a new type of intelligent systems (e.g., Intelligent Cyber Physical Systems) able to make "reasonings" and perform autonomous actions according to the context. This thesis work presents a deep study of all these technologies and proposes intelligent systems and algorithms solutions applied to several fields (i.e., smart home, smart city, smart industry, smart health, and smart agriculture) focusing also on the challenges emerged during the design and implementation processes. Experimental results demonstrate the feasibility of the proposed approaches and show the benefits derived from using them.

Introduction

The progresses made in ICT technology during the last decade allowed the construction of a network that enables the interaction with objects. We usually refer to this phenomenon with the term Internet of Things (IoT) to indicate the trend where common use objects are equipped with an "intelligence" and act as sensors and actuators [1]. The advent of IoT totally revolutionized the way we interact with the physical world, in such a context, objects become an active component of the environment, and because these systems usually expose an interface also with the cybernetic world, they are usually called Cyber Physical Systems (CPS) [2]. CPS are a new generation of devices exhibiting advanced functionalities. Thanks to their multiple ways to interact with the human being and their computation capabilities strictly related to the physical world, they represent an enabling technology for the future systems development [3].

Today, smart environments represent the maximum expression of CPS; here, fleets of objects are connected together acting as bridges to the physical world and exposing several services to support their inhabitants. Originally intended to ease the life of the human being, today the market related to this technology expanded very quickly producing specialized solutions. Smart industry, smart cities, smart health, etc. are just a few examples which demonstrate the wide range of available applications that allow to enhance the productivity and the effectiveness of a system. In such a context, the Cloud plays a key role providing storage and computing capabilities to perform complex operations (e.g., data analysis, execution of onerous algorithms, etc.). Moreover, the Cloud can

be seen as an "orchestrator" that manages and coordinates all the components of an environment. Although the Cloud paradigm is useful to realize these systems, the advent of CPS and IoT gave the birth to new types of services and applications which pose significant challenges that the Cloud is usually unable to handle. For example, when working with applications with low latency or real time requirements, the use of the Cloud paradigm becomes unsuitable. Analogue is the condition for those services containing sensible data that should not travel the Internet.

The above mentioned examples, are just two of the many conditions where the Cloud becomes ineffective, thus leading to a new paradigm where the computation is shifted "closer" to the data [4]. This paradigm is called Edge computing and indicates a model where the computation is done directly on the device (e.g., an embedded system, a smartphone, a smartwatch, etc.) which generated the data or very close to it. However, the constrained resources of Edge devices make them suitable to perform simple operations. In this sense, the shift of computation from the Cloud to the Edge is not easy and requires a careful design of applications and services that can effectively run of these devices. Nowadays, modern systems exploit both the paradigms building hybrid platforms, where data is usually collected and preprocessed on the Edge and then sent to the Cloud to store it and perform more complex analysis.

Artificial Intelligence (AI) is another important player for the implementation of intelligent applications. Born in the 80's, this promising technique has always faced the problem related to the data gathering which slowed down its growth. Today, this problem has been largely solved thanks to the IoT widespreading that acted as catalyzer for the generation of a huge amount of information. AI brings the applications and services to a whole new level providing an advanced data processing, and becoming an active element in the decision process during the execution of a task. Through machine learning and deep learning in fact, it is possible to build new types of systems equipped with a layer of intelligence

that helps them to make context aware decision autonomously [5], and for this reason called Intelligent Cyber Physical Systems (ICPS). Such systems (in combination with Cloud and Edge paradigms) represent the core elements of modern frameworks and are used as entities capable to deliver a "reasoned" support to the human being during his daily activities.

Leveraging the above described technologies, this thesis work presents the study, design, and implementation of machine and deep learning solutions for ICPS on several smart application contexts (i.e., home, city, industry, health and agriculture). Starting from the problem analysis, we present the literature overview to describe the current state of the art and compare it with our techniques. In such a context, we provide a detailed description of the proposed solutions putting in evidence the challenges we faced during their realization. We also prove the effectiveness and feasibility of the proposed systems through the experimental results, highlighting the benefits derived from their implementation.

The rest of the thesis is organized as follows. Chapter 1 provides a background on ICPS, machine and deep learning technologies which are at the base of this thesis work. Chapter 2 presents an indoor localization system that exploits deep learning and wireless fingerprints to locate the user inside a smart environment. Chapter 3 proposes two different solutions to improve the overall performance of applications executed in smart city scenarios. Chapter 4 focuses on the realization of smart industry solutions that allow to monitor, detect, and prevent the occurrence of faults in industrial plants. Chapter 5 presents a smart health technique that uses wearable computing and deep learning techniques to create a support system for the prevention of pressure ulcers formation. Chapter 6 describes an ICPS solution for smart agriculture running on the smart Edge to assess the health conditions of plants. Finally, Chapter 7 concludes this thesis work and gives a vision of the steps necessary to extend the AI to a human level.

Chapter 1

Background

ICPS totally revolutionized the way we interact with the physical world; together with machine learning, they paved the way for new types of intelligent applications which exhibit cognitive capabilities improving our quality of life. In this chapter, we provide a general background of these two technologies which are at the base of this thesis work.

1.1 Intelligent Cyber Physical Systems

ICPS represent a very important resource for the fourth industrial revolution. Wanting to give a definition, ICPS are computer systems (usually blended with the environment) that allow to integrate sensing, actuating, and computing functionalities into the physical world and exposing, among other things, Internet communication capabilities [3]. The combination of physical processes with the computation is not new. Embedded systems are a perfect example of this trend, however, they are usually "closed" towards the external world and do not expose their computing capabilities [2]. In this sense, thanks to their possibility to interact with the cyber and the physical worlds, ICPS provide a pervasive infrastructure that can support the human being in many different ways [6]. In the recent period, a growing interest has arisen towards this technology, dictated

by the desire to equip these systems with such an "intelligence" to make them autonomous, and capable of taking decisions. In such a context, AI plays a decisive role in bringing the intelligence on CPS. Thanks to machine learning in fact, we introduce a "cognition" layer that enables a system to learn a task exploiting its ability to sense the physical world. Figure 1.1 depicts a typical scenario where the ICPS act as a connecting bridge exposing a double interface to put in communication the cyber and the physical worlds. Here the sensing/actuating layer is the closest to objects and, as the name suggests, it is responsible to sense the environment and interact with it via a set of actuators (e.g., switches, motors, etc.). The computing layer on the other hand has the task to process the data coming from both the two worlds and works in synergy with the cognition layer added via AI techniques. In this sense, ICPS exhibit a context awareness of the surrounding environment that results fundamental for the delivery of tailored services.

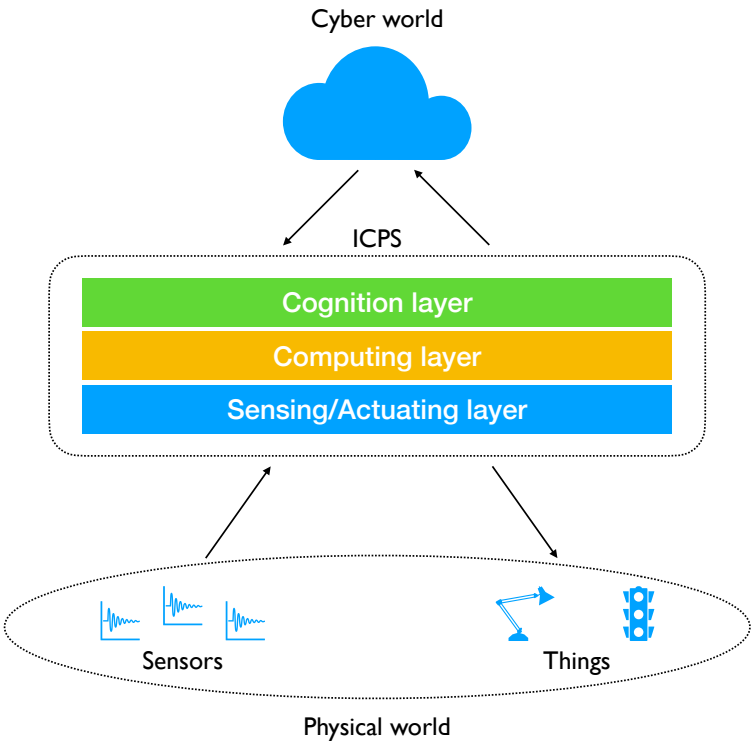


Figure 1.1: ICPS scenario.

ICPS completely revolutionized the way we interact with things, however they also pose significant challenges most of which related to the pervasive nature of this technology. In particular, the uncertainty of the environments requires the design of robust and fault tolerant systems in order to face unpredictable conditions that can compromise their operation [7]. Security should be another concern during the realization of an ICPS. Their capability to have a direct access to the "physical reality" imposes the implementation of mechanisms to prevent and contrast possible external attacks. A careful networking management is also necessary to provide a good quality of service, in this sense ICPS should exhibit self-organizing techniques to guarantee their autonomy [7]. Of course, some of these problems are not new, but they should be analyzed under a different perspective in order to propose effective solutions.

Despite the above mentioned challenges, today ICPS have a wide range of applications scenarios where their usage allows to improve the overall systems quality, productivity, and effectiveness.

1.2 Machine learning

In the introduction and in the previous paragraph, we mentioned that ICPS are equipped with an intelligence provided by machine learning, but we did not provide a proper definition of this technique. Machine learning is a branch of AI that gives, according to Arthur Samuel, to a computer or a device "*the ability to learn without being explicitly programmed*". The base idea of this technique consists in the definition of a mathematical model that is able to fit the output of a function which is not known a priori. Depending on the nature of this output, it is possible to split the machine learning problems into two categories, namely: *regression* and *classification* problems [8].

In general, we talk about regression problems when the output we want to predict is a continuous value (e.g., a temperature, a price, etc.). In such a context,

statistical modeling provides the so called regression analysis through which is possible to determine the relationship between the dependent variables (i.e., the outputs) and the independent variables (i.e., the inputs) called *features*. Given a regression problem, we can define the following quantities: the features vector X, the outputs vector Y, and the vector of parameters $\boldsymbol{\theta}$ which is not known. The relationship which binds these quantities can be defined as follows:

$$Y = f(X, \boldsymbol{\theta}), \tag{1.1}$$

where $f(.)$ is a function not known a priori. In such a context, the task of machine learning is to find the best set of parameters $\boldsymbol{\theta}$ such that the difference:

$$Y - \widehat{Y}, \tag{1.2}$$

(where $\widehat{Y}$ is the value estimated by the model) is minimum.

With respect to classification, we refer to this problem when the output to predict is a discrete value (e.g., digits from 0 to 9, cancer type, etc.). In this sense, the task of a machine learning classifier is to determine the most fitting category (i.e., the class) for a given feature vector X. Likewise for the regression, the algorithm has to find the best set of parameters $\boldsymbol{\theta}$ such that a *score function*:

$$score = f(X, \boldsymbol{\theta}), \tag{1.3}$$

returns the most probable class associated to X according to the score.

Depending on the data and the task to accomplish, machine learning problems can be further split into three categories: *supervised learning*, *unsupervised learning*, and *reinforcement learning* [8].

Supervised learning

In supervised learning problems the machine learning model is provided with a set of inputs and the corresponding outputs (i.e., the labels). The term supervised

comes in fact from the "supervision" of the human being, who provides to the model a feedback of the real world that is exploited to fit the data. In general, during the implementation of a supervised machine learning algorithm we can identify 5 main steps, namely: dataset construction, algorithm choice, model selection, model training, and model testing [8].

The first step is probably the most important one. During this phase in fact, the data is collected, organized, and the inputs and outputs of the problem are defined. The importance of this step comes also from the quantity and the quality of the collected data that determine the quality of the machine learning model. Once the data is collected, the next step is the choice of the most suitable algorithm. This process can be executed in different ways, but most of them imply the use of empirical trials. Since a lot of machine learning models require a manual setup, the model selection (or model validation) step is necessary to select the best set of input parameters (or *hyperparameters*). The fourth step is based on the training of the selected model. Of course this phase changes according to the algorithm, however, its goal is always to fit the input data while minimizing a cost function (usually denoted with J). In such a context, since we talk about supervised learning problems, the model can use the labels as an indicator that helps to understand if it is learning or not.

The last step is the model testing. This phase is necessary to test the "generalization" capabilities of the model on a new set of input data that it has never seen (i.e., not belonging to the training data). Unlike the training step, during this phase the model is only fed with the input without the corresponding labels that, in this case are used to test the learning. In this sense, if the model is able to correctly predict the output labels with a good level of accuracy, then this means it acquired the generalization capabilities necessary to "learn" the task.

Unsupervised learning

Unsupervised learning is a class of machine learning problems characterized by the absence of labeled data. Likewise the supervised, also in this case the implementation of an unsupervised algorithm is done performing the above described steps. However, this kind of learning is in general more difficult if compared to the supervised one because of the absence of a feedback. Nevertheless, this technique is very useful for a wide variety of machine learning tasks especially related to clustering problems, whose goal is to group inputs with similar patterns or structures. In general, unsupervised learning is very helpful also as a preprocessing step for the data organization or visualization before passing it to a supervised algorithm.

Reinforcement learning

Reinforcement Learning (RL) is a machine learning technique used to observe an environment and learn an optimal policy with respect to one or more performance indexes. From a structural point of view, RL is very similar to the supervised learning, with the only difference that, in this case, a reward is used instead of the labels. It adopts as base formalism a Markov Decision Process (MDP) that allows to model a decision process in a stochastic environment. From a mathematical point of view, a MDP is defined through a set of *states* $\mathcal{S}$ that the environment can assume, a set of *actions* $\mathcal{A}$ that can be performed in the environment, a *reward function* $R(s) : \mathcal{S} \to \mathbb{R}$ which defines the reward associated to an action, and a *discount factor* $\gamma \in [0, 1]$ stating the importance of future rewards. The objective of RL is to use an entity called *agent* to find an optimal policy that maximizes the reward on an environment. To do that, it builds a trial and error process by making actions while traversing the states inside the environment several times, and changing the learned policy according to the feedback received from the rewards. A typical RL approach is the Q-learning [9] a *model-free* technique based on the use of the Q-values $Q(s, a)$ to learn an optimal policy in

an environment. A Q-value is a real number representing the utility of doing the action a while the agent lies on state s; these values evolve as the agent takes action in the environment according to the following equation:

$$Q(s,a) = Q(s,a) + \alpha \cdot (R(s) + \gamma \cdot max_{a'} Q(s',a') - Q(s,a)), \tag{1.4}$$

where α is the *learning rate*, $R(s)$ is the reward function, γ is the discount factor, s' is the state towards which the environment will evolve by taking action a, and a' is the action to which corresponds the highest utility (i.e., the highest Q-value) when in state s'.

RL finds a large application especially in optimal control problems where an agent is trained to learn a policy on a given environment.

1.2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are a family of non-linear function approximation algorithms inspired by the biological brain [10]. From an architectural point of view, ANNs are systems of interconnected components (called *neurons*) which are able to process data [10]. Each connection is characterized by a *weight* (like in a weighted graph), and to each neuron is associated a *bias* (typically a constant value) whose task is to increase the model flexibility by shifting its output.

Figure 1.2 depicts the simplest possible ANN (consisting of one neuron) which takes the name of *perceptron*. From a mathematical point of view the operation executed by a neuron is the following:

$$y = f(\sum_{i=1}^{n} x_i w_i + b), \tag{1.5}$$

where y is the output of the neuron, x_i is the $i-th$ input, w_i is the weight associated to the $i-th$ input connection, and b is the neuron bias. Before generating the output, the result of the above defined summation passes through a function $f(.)$ called *activation function*. Such a function plays a very important role since

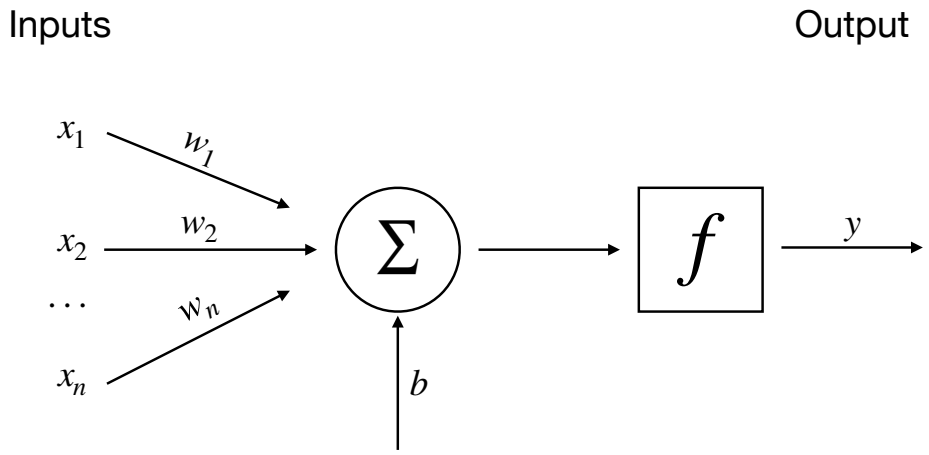


Figure 1.2: Structure of the perceptron.

it strongly affects the output of the neuron and enables ANNs to learn non-linear relationships between the input and the output. Table 1.1 shows some of the most popular activation functions used in modern ANNs applications i.e., the *Sigmoid*, *hyperbolic tangent* (tanh), *Rectified Linear Unit* (ReLU), *Leaky ReLU*, and *Exponential Linear Unit* (ELU). Each of these functions introduces a non-linearity to the ANNs which can produce very different results. In this sense, they require a careful usage and a deep understanding of the machine learning problem. Unfortunately, the single perceptron is not able to learn complex relationships between the input and the output, for this reason it is usually organized in structures called *layers* that group a set of neurons to perform advanced calculations.

Today, the most common topology in ANNs is the *multilayer perceptron* (or *feedforward ANN*) (showed in Figure 1.3) whose structure is characterized by the connection in cascade of multiple layers of perceptrons (called *hidden layers*) between the input and the output. In such a context, the main goal of the ANN is to find the best set of parameters for each network neuron (i.e., the weights) such that the error function J is minimized.

To do that, ANNs use a supervised technique which is a combination of two algorithms, namely: *backpropagation*, and an *optimizer* to iteratively update the weights values. The first one, it is responsible for the gradients computation of

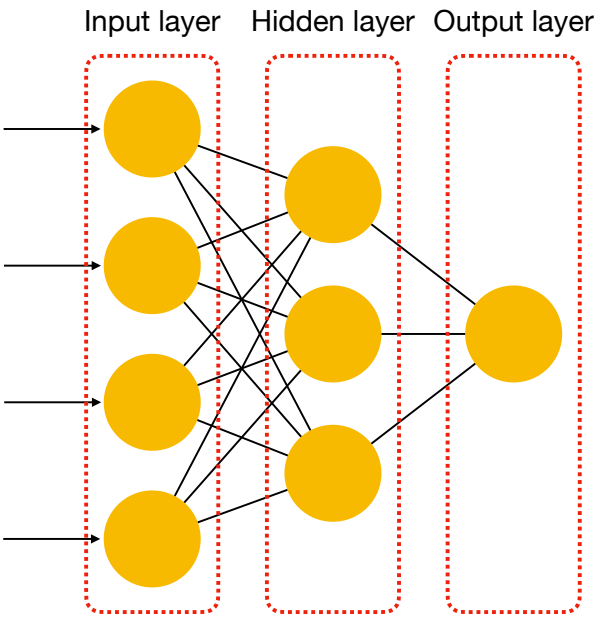


Figure 1.3: Example of a multilayer perceptron topology.

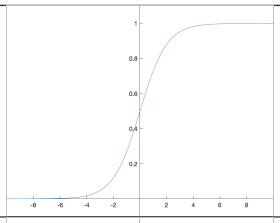
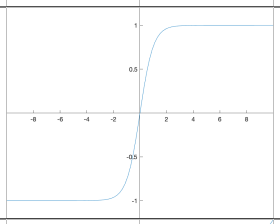
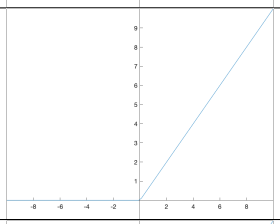
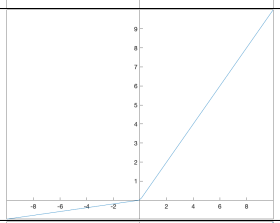
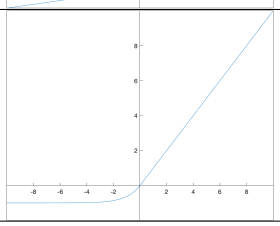
the error function with respect to the network parameters of the ANN (denoted with $\boldsymbol{\theta}$). In particular the term backpropagation derives from the fact that error gradients are back propagated from the output layer to the input one [11] via the *chain rule*, a technique used in calculus to compute the derivative of composite functions. In this sense, the computation of the error derivative returns useful information that can be exploited to have an understanding on how the network parameters should change in order to reduce the cost function effectively. On the other hand, the optimizer algorithm has the task to compute and update the ANN weights given the error gradients, thus performing the actual "learning" process [11]. For a better understanding in the following equation

$$w_{i,j}^{new} = w_{i,j}^{old} - \alpha \cdot \frac{\partial J}{\partial w_{i,j}}, \tag{1.6}$$

we report an example of a weight update where $w_{i,j}^{new}$ is the new generic weight computed by the optimizer, $w_{i,j}^{old}$ is the old weight at the previous iteration, $\frac{\partial J}{\partial w_{i,j}}$ is the error gradient with respect to the weight value computed via backpropagation,

and α (denoted sometimes also with η) is the *learning rate* defining the magnitude of the optimization step.

Table 1.1: Main activation functions.

Name	Graph	Function
Sigmoid		$y = \frac{1}{(1+e^{-x})}$
tanh		$y = \tanh(x)$
ReLU		$y = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$
Leaky ReLU		$y = \begin{cases} x & x \geq 0 \\ ax & x < 0 \end{cases}$
ELU		$y = \begin{cases} x & x \geq 0 \\ a(e^x - 1) & x < 0 \end{cases}$

Wanting to synthesize the learning process of a feedforward ANN, it consists of the following 4 steps: *forward pass*, *error computation*, *gradient computation*, and *parameters update*. In the first step, the data is forwarded from the input to the output layer for the ANN output computation. If we consider a traditional ANN, this task is accomplished by repeating multiple times eq. (1.5) according to the model topology. In the second step, the network computes the error; since

the algorithm is supervised, this can be done comparing the output produced by the ANN in the forward step and the corresponding *ground truth label*. In particular to make the comparison, ANNs use very specific cost functions from which is possible to effectively compute informative gradients. Functions like: *Mean Squared Error* (MSE), *categorical cross-entropy*, *binary loss*, etc. today are at the base of the learning process of many machine learning applications. Finally, the last two steps are executed via the above explained backpropagation and optimization algorithms. It is worth to mention that the learning (or training) of an ANN is an iterative process, which means that these steps are continuously repeated until a stop condition is satisfied.

1.2.2 Deep learning

Most of the core concepts of deep learning were defined between the 80's and the 90's. The idea that a machine can "think" is even older, however, only in the recent period we assisted to its wide spreading. The reasons for such a diffusion are twofold: *i.)* a massive availability of labeled data (thanks to IoT and CPS); *ii.)* the improvement of hardware capabilities that enabled the execution of onerous algorithms.

Deep learning is a subclass of machine learning algorithms that uses several layers (composed by non-linear processing units) connected in cascade, to discover intricate relationships between the input parameters (i.e., the features) and select the most useful by means of the so called *features extraction* process [12]. Such a process, given a set of input features is able to detect (and virtually remove) those ones which are redundant or not informative. Today, deep leaning and more specifically Deep Neural Networks (DNNs) find a lot of applications in different fields (e.g., computer vision, speech recognition, natural language processing, etc.). From a topological point of view, a DNN is an ANN with a larger number of hidden layers (in general more than two) between the input layer and the output one. DNNs are able to learn complex non-linear relationships between

the input and the output, however, they make harder the training phase because they require:

- a higher number of samples for the training;
- a careful tuning to avoid data overfitting;
- more computation time.

Typical DNNs are Recurrent Neural Networks and Convolutional Neural Networks that will be explained in the next paragraphs.

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a family of neural networks particularly suitable for the processing of sequences of data [11]. The base idea to pass from a traditional DNN to a RNN is the use of a self loop (as showed in Figure 1.4) that allows to maintain the information coming from the "past", thus enabling a memory mechanism known as *hidden state*.

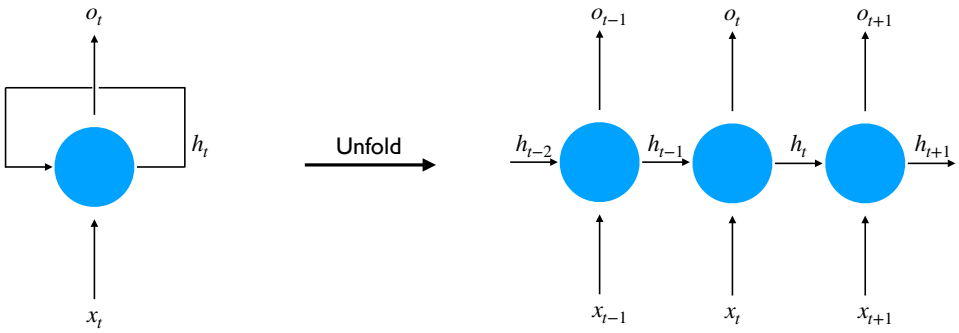


Figure 1.4: RNN neuron.

Considering a sequence x_t where t represents the time step at which is received, the hidden state of the RNN at the time step t is a function of the current input, the network parameters $\boldsymbol{\theta}$, and the previous hidden state at the $t-1$ time step:

$$h_t = f(x_t, h_{t-1}; \boldsymbol{\theta}), \tag{1.7}$$

where $f(.)$ is the activation function (see Table 1.1) [11]. In the same way, we can define the output at the t time step o_t as:

$$o_t = f(h_t), \tag{1.8}$$

where $f(.)$ is the activation function of the output layer. When working with RNNs we can think to "unfold" them over time to have a better vision of what is happening internally. Starting from eq. (1.7) and reorganizing it in order to make explicit the hidden state, we obtain:

$$h_t = f(x_t, f(x_{t-1}, f(x_{t-2}, ..., f(x_1, h_0; \boldsymbol{\theta}); \boldsymbol{\theta}); \boldsymbol{\theta}); \boldsymbol{\theta}), \tag{1.9}$$

where we can notice the inner relationship that binds the different time steps. In this sense, the output at the time instant t (see eq. (1.8)) is also "affected" by the past values of the state which explains why RNNs are suitable for machine learning tasks where the input order or time are crucial (e.g., in speech recognition).

Unfortunately, RNNs suffer from the so called *vanishing gradient* and *gradient exploding* problems especially when the number of unfolded time steps is large [13], [14]. In particular these conditions derive from eqs. (1.8), (1.9) where emerges a product of many factors when computing the error gradients. In this sense, the algorithm back propagates the gradients through the unfolded RNN navigating "back in time" and for this reason is called backpropagation through time (BPTT). In such a context, if we consider an unfolded RNN of t time steps (with large t), the gradient takes the form of a product of the same quantity:

$$\frac{\partial J}{\partial \boldsymbol{\theta}} = \boldsymbol{\theta} \cdot \boldsymbol{\theta} \cdot ... \cdot \boldsymbol{\theta}. \tag{1.10}$$

According to math, if $\boldsymbol{\theta}$ parameters are > 1, then the derivative value will tend to exponentially increase (as long as the RNN goes deep) reaching the infinity in certain cases, thus causing the gradient explosion. On the other hand,

if $\boldsymbol{\theta}$ parameters are small (i.e., < 1) the gradient value will tend to exponentially decrease as long as the time steps increase, until its vanishing. Both these problems pose significant challenges during the RNNs implementation and require a careful design of their topologies in order to avoid these issues during the training process.

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of DNNs mainly used in image and visual recognition tasks. From an architectural point of view, a CNN is very similar to a traditional feedforward DNN, in this sense, the main difference that characterizes this kind of networks are two specific layers i.e., the *convolution* and the *pooling* (as showed in Figure 1.5).

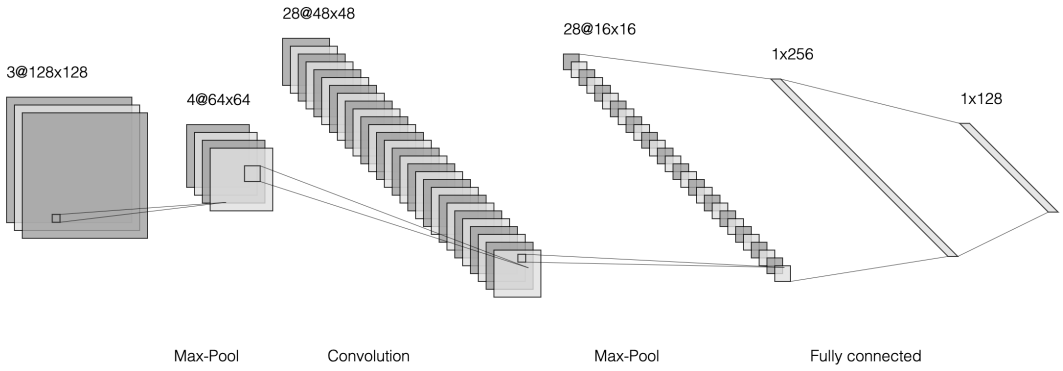


Figure 1.5: Example of a Convolutional Neural Network.

In particular, the purpose of the convolution layer is to perform a feature extraction process on the image passed as input, while maintaining the "spatial" relationship between the pixels. From a CNN point of view, the convolution step consists in a linear operation between two matrices (usually called *input* and *kernel*) that produces in output a *feature map* as depicted in Figure 1.6 [11]. In such a context, the kernel slides over the rows and columns of the input matrix and computes the feature map as follows:

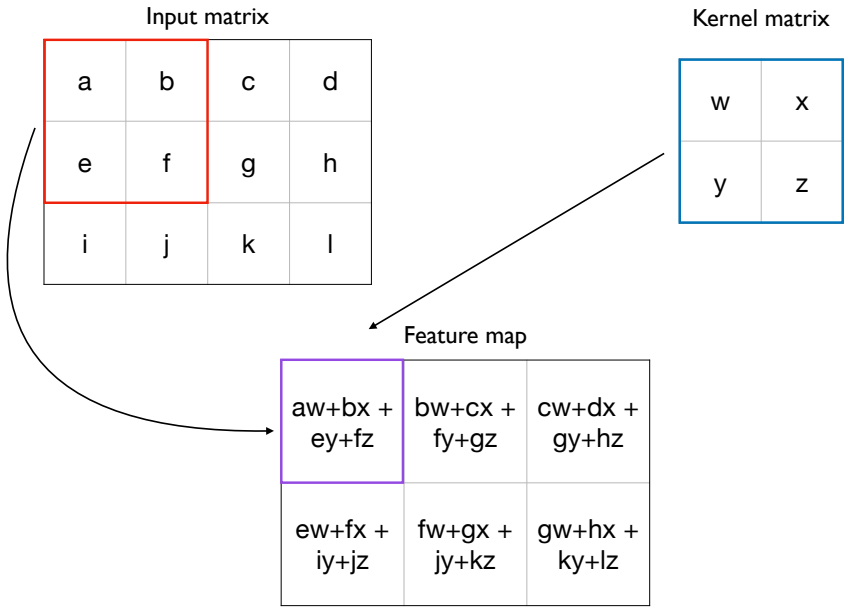


Figure 1.6: Convolution operation.

$$F_{i,j} = \sum_{h} \sum_{w} I(i+h, j+w) K(h, w), \tag{1.11}$$

where $F_{i,j}$ is the generic element of the feature map, h is the number of kernel rows (i.e., the height of the kernel), w is the number of kernel columns (i.e., the width of the kernel), i is the row index, and j is column index. The output of this operation is a new matrix (i.e., the feature map) whose dimensions can be computed starting from the ones of the input and kernel matrices [15]:

$$F_h = \frac{I_h - K_h + 2P}{S_h} + 1, \tag{1.12}$$

where F_h is the feature map height, I_h is the input matrix height (i.e., the number of rows), and K_h is the kernel height (i.e., the number of rows of the kernel). With respect to P, it is representative for the *padding* option which consists in adding a zero padding around the input matrix in order to preserve its dimensions after the convolution step. Finally, the S_h term defines the *stride* with respect to the height, i.e., how much the kernel shifts (in terms of rows) when sliding over the

input matrix during the convolution step. Likewise we did for the height, we can compute the width of the feature map as follows:

$$F_w = \frac{I_w - K_w + 2P}{S_w} + 1, \tag{1.13}$$

where the parameters are exactly the same we already described in eq. (1.12), but this time referred to the input and kernel widths and columns.

Typically in CNNs, after the convolution step, we apply a non-linear activation function that helps the network to learn possible non-linear relationships between the input and the output. Such an operation is then followed by a pooling step whose task is to further modify the output of the convolution [11]. Pooling operations play a very important role in CNNs since they allow to reduce the network feature space, thus simplifying the overall model complexity. Specifically, this operation replaces the output of a CNN in a certain spatial location with a value derived from the "nearby" outputs [11]. Examples of typical pooling operations are: *max-pooling* and *average-pooling* where the values of the feature map falling inside a rectangular area are respectively replaced with the maximum and the average values. For a better understanding, in Figure 1.7 we report an example of max-pooling where we considered a pooling window with height and weight set to 2.

Likewise the convolution, the pooling step is performed by sliding the pooling window through the feature map rows and columns and generating a new matrix whose dimensions are computed in the same way we did in eqs. 1.12, 1.13. In all cases, the pooling operation allows to make the feature map invariant to small variations of the input. Such a functionality is very helpful for example, when we are interested in knowing the presence of a feature inside an image instead of its precise location and provides a higher "elasticity" to the CNN model during the prediction task [11].

By looking at their architectures, CNNs topologies consist of two parts: a first one where a sequence of convolution and pooling operations are performed to

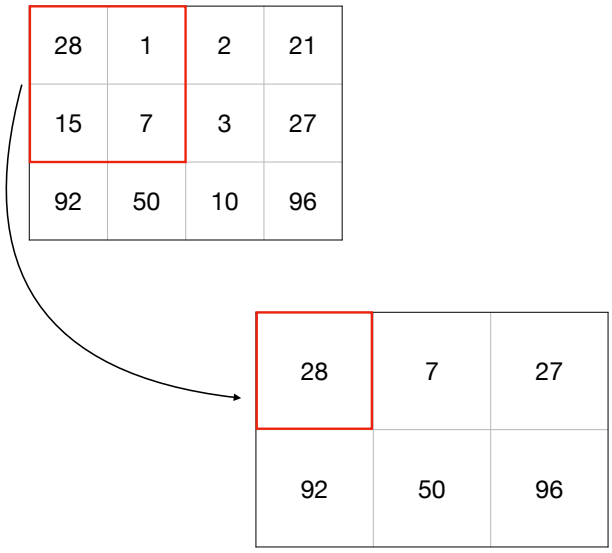


Figure 1.7: Max-Pooling operation.

extract informative patterns from the input images through the feature extraction process we described. The second one is a multilayer perceptron network that receives in input the features preprocessed by the first part of the network and performs the actual visual recognition task.

Chapter 2

Smart Home

Smart home is the most common example of smart environment. Thanks to a wide variety of consumer products available in the market, today the home automation has become a reality accessible to everyone. In such a context constellated by objects emerges the necessity to locate the user to deliver a better and more precise interaction with the environment. Unfortunately, the use of the GPS technology is not possible for indoor localization due to strong signals attenuation. In this chapter, we face this problem through the exploitation of deep learning techniques and radio fingerprints coming from a set of wireless access points [16].

2.1 Indoor user localization

Wireless technology wide spreading together with mobile computing devices and IoT generated the interest in the development of Location Based Services (LSBs) whose market is expanding [17]. Such services, are becoming a very important part of our lives and represent the core elements for the implementation of smart applications which are strictly related to the position of the user in an environment.

The main task of a smart environment is to support its "guests" by performing

a set of operations which can be partially or fully autonomous; to do that, a smart environment should exhibit a context-awareness capability in order to understand its current state (by means of several sets of sensors) and determine which action should be executed. IoT is an enabling technology to make smart environments a reality, by constructing a network that allows objects to interact between them and the users [18], [19].

Today smart environments tend to be very large and constellated by objects. In such a context, we tackled the problem of localizing the user inside a closed environment in order to allow a better and more precise interaction with it. The possibility to locate the user in an environment enables also the deployment of more efficient smart applications, that can exploit this information to deliver tailored services according to the position.

When talking about localization, two possible scenarios should be considered: outdoor and indoor. In outdoor scenarios, the Global Positioning System (GPS) is the standard *de-facto*; it uses a minimum of 24 artificial satellites for the data transmission and reaches a precision in the order of a meter. However, GPS cannot be used in indoor environments where the signals used by this technology are strongly attenuated by the building walls and materials, thus causing a significant loss in terms of accuracy. In indoor LBSs the main problem consists in determining the user position; due to the complex nature of indoor environments, the implementation of a user localization technique has to address a set of challenges which are in most part caused by the presence of physical obstacles (e.g., walls, doors, human beings, etc.), that affect the propagation of electromagnetic waves. Today, there are several methods to implement an indoor user localization technique and in general it is possible to classify them into three categories: *proximity detection*, *triangulation*, and *scene analysis*.

Proximity detection

Compared to the others, this is the simplest method to implement, and uses the Cell of Origin (COO) technique to localize the user [20]. In such a context, the user position is associated to the position of the base station to which is attached. If more base stations detect the presence of a mobile client, it simply forwards the position information of the beacon from which it is receiving the strongest signal. Since the position of the user is associated to the position of the base station, this technique is not very accurate and does not allow to implement a granular positioning system, however, it continues to be used in GSM, Bluetooth, and radio services.

Triangulation

It uses geometric properties to localize a user and can be divided (depending on the geometric approach) into two categories: *angulation* and *lateration* [20]. Angulation techniques measure the angle of incidence of a radio signal to determine the position of an object in the space. Lateration methods, on the other hand, are able to determine the position of an object by measuring the distance between it and several reference points (whose position in the space is known).

Angle of Arrival (AoA) belongs to the first category and uses a set of directional antennas (or antennas array) to detect the direction of an incident signal. Although this technique allows a precise localization, the hardware cost together with multipath and reflection problems make this method unsuitable for indoor localization applications.

Time of Arrival (ToA) adopts lateration to detect the position of an object in the space. Specifically, by using a group of beacons (whose position is known) the distance between the target and the beacon is computed considering the transmission delay and the speed of the signal. Despite, this technique solves the problems of AoA, also in this case the hardware required to synchronize the beacon signals (fundamental to get an accurate positioning) is very expensive.

Moreover, ToA technique is based on the strong assumption such that signals travel the distance between the beacon and the target according to the Line Of Sight path (LOS). If this assumption is not respected (e.g., for the presence of an obstacle), then the signals will travel a longer path thus causing a wrong localization [21]. In an indoor environment where this problem could be very frequent makes this technique unsuitable to localize the users effectively.

Scene analysis

Nowadays, most of the localization approaches adopt the concept of "fingerprint matching" for the location estimation. This method consists in collecting features from the scene (i.e., the fingerprints) and build a fingerprint dataset for each area of interest that is intended to be localized [20]. A fingerprint can contain different types of information (e.g., Bluetooth, Wi-Fi, RFID, etc.) however, Wi-Fi fingerprints are the most used, because they do not require additional hardware (i.e., they use Wi-Fi access points already installed in the environment) and, unlike the previous approaches, this localization technique can be implemented totally via software. Scene analysis approach can be divided into two phases:

- offline phase during which the fingerprints are collected or computed;
- online phase during which the localization process is performed by matching the perceived fingerprint with the closest contained inside the dataset.

Because the accuracy of this method is strongly affected by the stored fingerprints, the dataset update rate depends on the dynamics of the environment. The major drawback of this technique is in fact given by the laborious and time consuming fingerprint collection process that should be periodically repeated to update the dataset, thus addressing the problem related to adding or removing access points.

Our proposed solution belongs to this category since it relies on a Wi-Fi fingerprint radio map. Moreover, the software implementation of this technique

matches with the low cost requirements we posed during the design process of our user localization system. However, the use of such a technique becomes very challenging when the number of signals to determine the user position is large. To address this problem, our indoor localization system exploits deep learning techniques to learn the complex relationship between a wireless fingerprint passed as input and the most probable location associated to it. In the next paragraph, we describe how we realized the fingerprint dataset (i.e., the radio map) necessary to perform the scene analysis.

2.1.1 Fingerprint dataset

Since we planned to tackle the indoor localization problem as a supervised machine learning approach, the first step consists in creating a labeled dataset to train and test the model. The literature proposes different datasets like in [22], however, we preferred to build a new one by collecting wireless fingerprints in one floor of our University department. In this way, it has been possible to test the performance of our indoor localization system during real experiments on the field.

The data for the dataset construction was collected at the $7-th$ floor of the Engineering Department of the University of Messina where six different rooms and the main corridor which connects them have been considered as the set of locations $\mathcal{L}$ as depicted in Figure 2.1.

Each datapoint contains the information of all the Wi-Fi signals perceived during the scanning; in this sense, the dataset has been constructed by performing multiple Wi-Fi scans (by means of a set of smartphones) at different time instants and for each location $l \in \mathcal{L}$. Data received from Wi-Fi scans consists of the Service Set Identifier (SSID), the MAC address (MAC), the Wi-Fi signal strength measured using the Received Signal Strength Indicator (RSSI), and the location l where the scan has been performed.

In order to address the problem of those devices exposing Wi-Fi hotspots, that

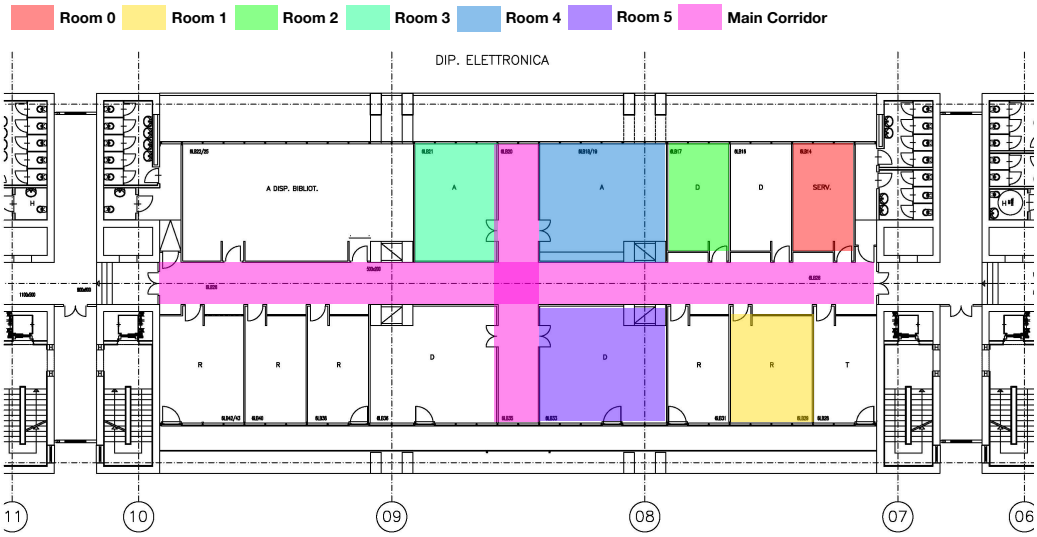


Figure 2.1: The floor map with the locations that have been monitored.

can produce distorted location information (e.g, printers, smartphones, portable modems, etc.), we used a filtering process based on the SSID in order to remove all those access points signals containing default keywords (e.g., iPhone, Android, Huawei, printer, etc.) in their names. Of course such a procedure is unable to remove hotspots with customized SSIDs, but from our empirical tests we found that it is sufficient to remove the majority of undesired signals.

Let us define a labeled Wi-Fi fingerprint $w_t^{(l)}$ as the t-uple composed of the $(MAC, RSSI)$ couples perceived in location l at timestamp t:

$$w_t^{(l)} =< (MAC_1, RSSI), (MAC_2, RSSI), ..., (MAC_q, RSSI) > . \tag{2.1}$$

Then, with $\mathcal{T}$ we indicate the ordered set of timestamps at which the Wi-Fi scans have been performed. Every MAC present in $w_t^{(l)}$ corresponds to a single *feature* of the dataset while the RSSI is the value of the corresponding feature in the $t-th$ sample. Since the total number of MAC addresses in an environment can not be known a priori, the final set of features $\mathcal{F}$ is obtained as the union of all the distinct MAC addresses found during the multiple scans and it is defined as follows:

$$\mathcal{F} = \bigcup_{t \in \mathcal{T}} \{MAC_x\} : (MAC_x, RSSI) \in w_t^{(l)}. \tag{2.2}$$

In this way, as long as the Wi-Fi scans are performed, the dataset will increase both "horizontally" by adding new features (i.e., new MAC address that have not been perceived til that moment) and "vertically" by adding new datapoints (i.e, new labeled fingerprints).

A machine learning model assumes that the input size once is fixed should remain the same, however, as we mentioned above, the dataset increases also horizontally over time which means that the "older" datapoints can contain a lower number of (MAC,RSSI) couples than the new ones. To address this problem, we introduce a generalized Wi-Fi fingerprint $\hat{w}_t^{(l)}$ containing all the features contained in $\mathcal{F}$ and where for each couple $(MAC_x, -) \notin \hat{w}_t^{(l)}$ the RSSI is set to -200. The reason why we choose -200 and not 0 to codify the absence of a MAC address during the scan comes from the meaning of the RSSI. Such an indicator usually ranges between 0 and -100 dBm where a signal close to 0 corresponds to a very strong signal whereas -100 is representative for a signal with a very low strength. In this sense, -200 is a value that codifies a MAC address which is virtually absent, thus representing all those MAC addresses not perceived during the Wi-Fi scans.

Table 2.1 shows a possible view of the constructed dataset where each row corresponds to the RSSI values of the wireless fingerprint $\hat{w}_t^{(l)}$, the columns are the MAC addresses contained in the features set $\mathcal{F}$ and the last column corresponds to the location $l \in \mathcal{L}$ where the fingerprint is collected.

Once we obtained a consistent number of datapoints (collected at different hours of the day for one week), we started a data analysis process to have a better understanding of the collected data. To see how the RSSI values change, we made several scatter plots while the user moved around the locations. Figure 2.2 shows an example of a scatter plot for a specific MAC address with respect to the locations $l \in \mathcal{L}$. In general, the RSSI tends to be well distributed among

MAC_1	MAC_2	MAC_3	...	$MAC_{\|\mathcal{F}\|}$	$\mathcal{L}$
$RSSI$	-200	$RSSI$	...	$RSSI$	$Room0$
...	...	...	...	...	...
-200	-200	-200	...	$RSSI$	$Room0$
-200	$RSSI$	$RSSI$	...	-200	$Room1$
...	...	...	...	...	...
-200	-200	$RSSI$	...	$RSSI$	$MainCorridor$

Table 2.1: Labeled Wi-Fi fingerprint dataset structure.

different average values when changing the location, in this sense, we exploited this data variability to localize the user inside the environment.

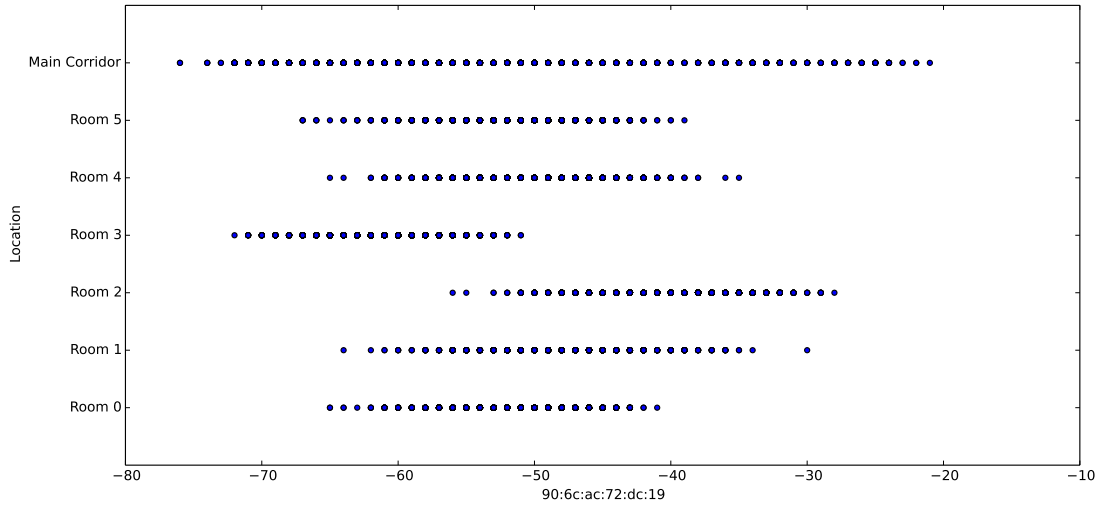


Figure 2.2: Scatter plot showing the distribution of RSSI values related to a specific MAC address with respect to different locations.

While analyzing the dataset, we also discovered that a set of MAC addresses was present for a limited period and then disappeared (probably due to those hotspots that the filtering process has not been able to catch). For this reason, a second filtering process has been applied to the dataset in order to remove the features (i.e, the MAC addresses) related to those access points that were perceived only a few times during the scanning, and potentially dangerous for the learning process. Specifically, a MAC_i is removed from the dataset according

to the following equation:

$$n_l^{MAC_i} \leq n_l * \delta, \forall l \in \mathcal{L}, \tag{2.3}$$

where $n_l^{MAC_i}$ is the number of labeled fingerprints associated to location l where the MAC address MAC_i has a RSSI different to -200, n_l is the total number of fingerprints collected in the location l, and δ is a threshold value with $0 \leq \delta \leq 1$ which represents the percentage of data relative to a location l that must be exceeded to maintain the MAC address MAC_i. In our empirical test, we found that $\delta = 0.2$ (i.e., the 20%) was sufficient to remove all the unwanted signals. After the second filtering process, we were able to filter 61 MAC addresses thus removing noisy data while reducing the model complexity. The final dataset after this data preprocessing consisted of $\|\mathcal{F}\|$ equal to 93, a number of locations $\|\mathcal{L}\|$ equal to 7, and a number of generalized labeled fingerprints $\|\mathcal{T}\|$ equal to $11,524$.

2.1.2 Deep learning approach for indoor localization

As we mentioned in the previous paragraph, the problem of detecting the user location in an indoor environment given a Wi-Fi fingerprint is a supervised multi-class classification problem. The literature proposes several approaches to tackle this problem. Authors in [22] propose the use of a basic machine learning model based on K-Nearest Neighbors (KNN), which is not powerful enough to localize the user in a highly mutable environment where the signals exhibit a wide variability. The approach described in [23] compares 20 machine learning algorithms and focuses on two of them, namely: the K* algorithm which adopts an entropy based distance, and a Radial Basis Fuction (RBF) regressor. In this work, authors considered a single location that they split into several sublocations and used the above mentioned algorithms to locate the user. In [24] Principal Component Analysis (PCA) is used to extract the most important features from a pre-defined wireless radio map. Then, these features are passed to traditional machine learn-

ing algorithms i.e., Random Forest (RF), Decision Tree (DT) and Support Vector Machines (SVMs) to perform the actual user localization. Authors obtained very good results, however, they considered a single location composed by only six Wi-Fi access points. The environment that we considered to test our approach is characterized by a quite large number of access points which exhibit a significant variability during the days (as depicted in Figure 2.2). Moreover, the use of the PCA by considering only the principal components would cause a high loss in terms of information, which can lead to a wrong predictions especially in those environments where the "contribution" of each access point is crucial to locate the user. These two reasons motivated us to implement an indoor localization system based on deep learning techniques. In a context where multiple signals have such a huge variability, we believe that deep learning can be a valid solution to build a model capable to learn the complex relationship between a Wi-Fi fingerprint and the user location.

From a DNN perspective, a multi-class classification problem can be seen as finding the best set of parameters $\boldsymbol{\theta}$ for a parametric family of probability distributions $p(\mathbf{y}|\mathbf{x};\boldsymbol{\theta})$ [11] where $\mathbf{x} \in \mathbb{R}^n$ is the input features vector (with $n = \|\mathcal{F}\|$) and $\mathbf{y} \in [0,1]^p$ is a vector representative for the locations (with $p = \|\mathcal{L}\|$) codified through one-hot-encoding (e.g., the vector $\mathbf{y} = [0,0,0,1,0,0,0]^T$ will correspond to *Room*3).

Starting from the dataset previously described in Paragraph 2.1.1 we randomly split it into three sets, namely: training, validation, and test representing the 65%, 15%, and 20% respectively of the entire dataset.

The proposed DNN consists of a feed forward fully connected network with 93 input units and 7 output units. The DNN hyperparameters have been tuned by running multiple tests in order to find the best configuration. In general, a good design choice is to start with a number of neurons in between the input and output units, for this reason we decided to use 80 neurons for each hidden layer. On the other hand, to determine the number of layers, we adopted a progressive

training approach which consists in starting with a single hidden layer and then progressively increase the number of layers until we reached a satisfying result in terms of accuracy and training time. After this procedure, we found that 6 hidden layers were sufficient to obtain a good model which was able to properly detect the user position inside the environment. The DNN architecture is showed in Figure 2.3.

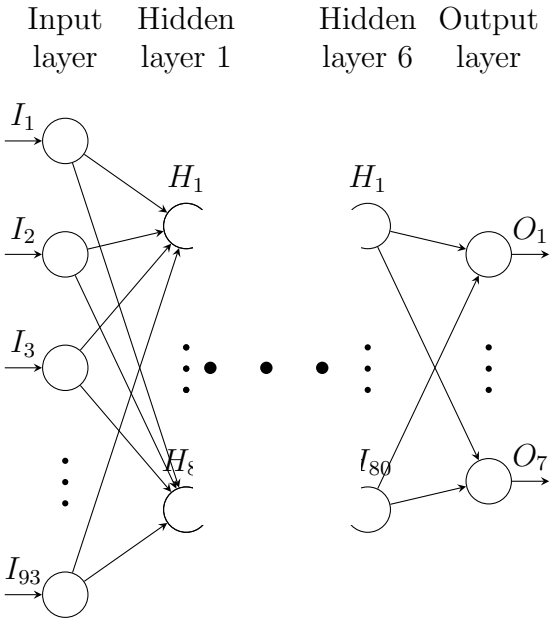


Figure 2.3: DNN architecture.

We selected the ReLU as activation function which resulted to be the best in terms of training time since it required a lower number of training iterations if compared with other activation functions like the Sigmoid (see Table 1.1). Then, we set the cross-entropy as cost function J as it is particularly suitable for classification problems, especially when used in combination with the *softmax* function. Softmax is a function that given an input vector $\mathbf{z} \in \mathbb{R}^p$, it produces as output a vector $\tilde{\mathbf{y}} \in \mathbb{R}^p$ whose values add up to 1 and it is defined as follows:

$$\tilde{y_j} = Softmax(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{p} e^{z_k}}, j = 1, ..., p, \tag{2.4}$$

where $\tilde{\mathbf{y}}$ is the DNN output vector and $\tilde{y_j}$ is the probability to be in location j

given a fingerprint $\mathbf{x}$. The cost function J is defined as:

$$J = -\sum \mathbf{y} \cdot log(\tilde{\mathbf{y}}), \tag{2.5}$$

where the sum is extended to all the training samples. Cross-entropy represents a Maximum Likelihood Estimator (MLE) for the DNN hyperparameters vector $\boldsymbol{\theta}$. In other words, minimizing the cost function J corresponds to find the best set of parameters such that the parametric probability distribution learned by the model is the closest to the real data distribution (which is not known a priori). In particular, the cost function is minimized through the training process during which the weights of the DNN are iteratively adjusted to better fit the input data. To do this, we used adaptive momentum estimation (known as *Adam* optimizer) that maintains a different learning rate for each network weight and adapts it during the training process. Finally to prevent the problem of overfitting, we used a L2 regularization by adding a weighted penalization term:

$$penalization = \lambda \cdot \sum_{L=1}^{N} W_L^2, \tag{2.6}$$

where N is the DNN number of layers, W_L^2 are the squared weights of the $L-th$ layer, and λ is the regularization rate (i.e., how much the penalization impacts the loss function). By adding this term to the loss function, we apply a penalization to the DNN weights thus reducing model complexity. DNN hyperparameters are summarized in Table 2.2.

Table 2.2: DNN hyperparameters.

DNN configuration	
# Hidden Layers	6
# Neurons per layer	80
Activation function	*ReLU*
Learning rate	0.001
Regularization rate	0.01
Training epochs	25000

The DNN implementation has been done using TensorFlow[1] an open source framework developed by Google for the design of powerful machine and deep learning models [25].

Figure 2.4 shows how the cost function J decreases at each iteration (i.e., training epoch). The trend of the curve demonstrates that the DNN is able to correctly learn the relationship between a Wi-Fi fingerprint and a location inside the indoor environment. This is also confirmed by the 99.6% of accuracy reached by the network on the test set which represents a very good result in terms of generalization.

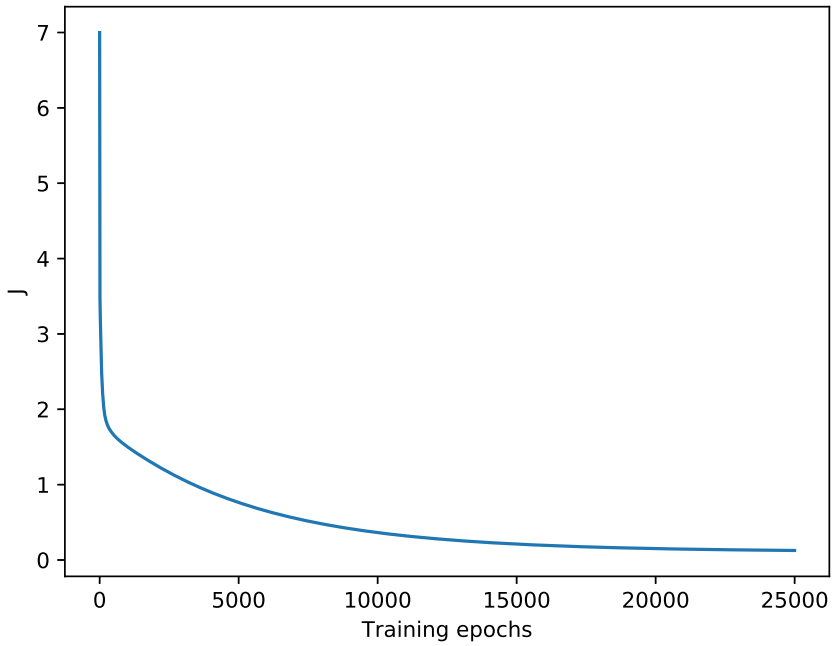


Figure 2.4: The value of the cost function J with respect to the training epochs.

[1]Available at https://www.tensorflow.org/, last accessed September 2020

```
int predict(float[] input){
     float[] result = new float[OUTPUT_SIZE];
     inferenceInterface.feed(INPUT_NODE,input,INPUT_SIZE);
     inferenceInterface.run(OUTPUT_NODES);
     inferenceInterface.fetch(OUTPUT_NODE,result);
     return argmax(result);
}
```

Listing 2.1: Prediction Method.

2.1.3 Indoor localization system

To implement the system and test it in a real scenario, we realized two software sub-systems: *i.)* a Python server hosted in the Cloud to train the DNN; *ii.)* an Android application to create the Wi-Fi fingerprint dataset and to localize the user. For the dataset creation, we installed the app on several mobile devices to collect a large number of fingerprints by means of crowdsensing. The collected data is sent from the smartphone to the Cloud where the DNN is trained. Once the training phase is finished, the trained DNN model is sent to the Android clients in order to start the localization phase. In this sense, one of the main advantages of using TensorFlow consists in the possibility to store a trained model in a binary file that can be loaded into an Android environment. By doing this, the heavy computation for the DNN training is done on the Cloud, while the smartphone is only responsible for the inference task (i.e., the prediction) which runs locally without needing an Internet connection to localize the user.

Our trained model occupies just $148KB$ which is a very good result if we consider the complex DNN topology, and it is perfectly suitable to be loaded into the RAM of a mobile device. Thanks to the Application Programming Interfaces (APIs) provided by TensorFlow, the Android application is able to interact with the model by feeding it and retrieving the results.

With reference to the code showed in Listing 2.1 used for the prediction, the *inferenceInterface* object acts as a communication bridge between the app and the model, and exposes three methods:

- `feed()`: to feed the DNN with new input data; it takes as input three parameters, namely: the name of the input layer, the input data, and the input size.

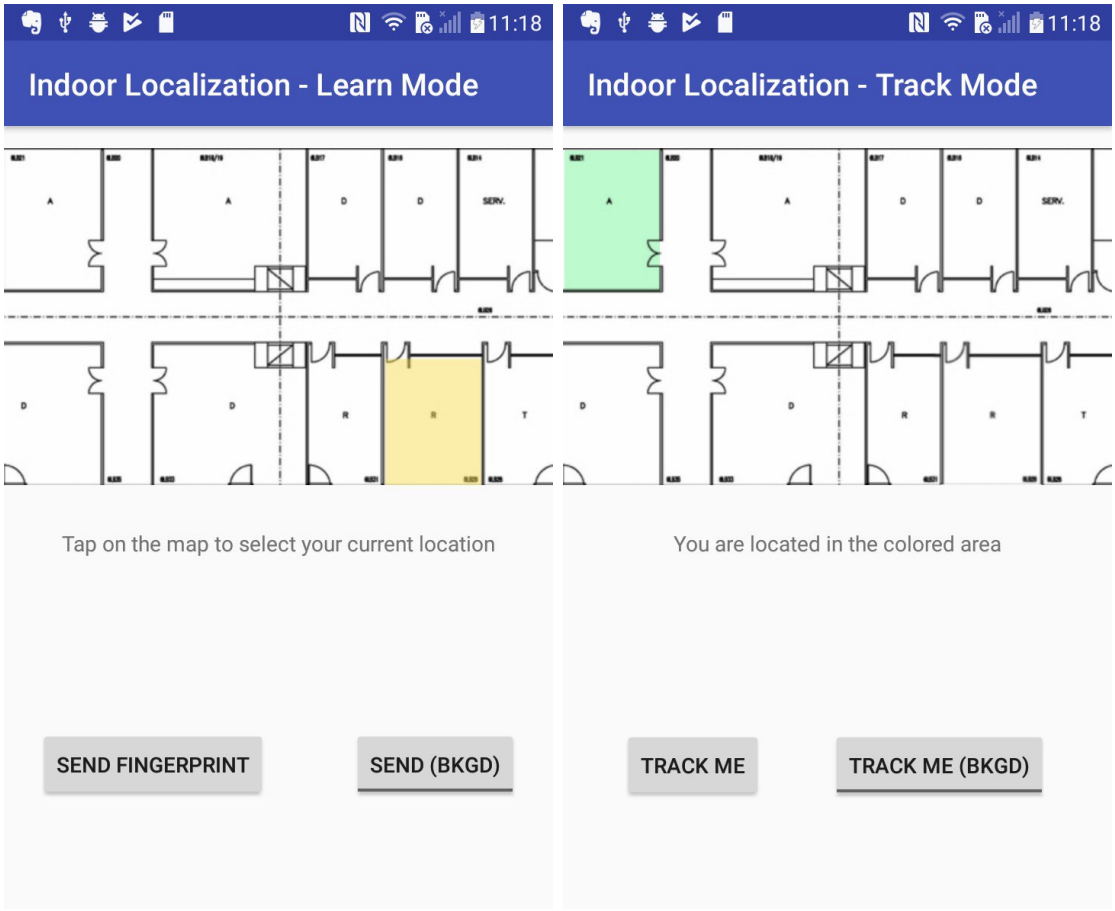
- `run()`: to run the model and generate the output; it takes as parameter only the name of the output layer.

- `fetch()`: to fetch the results from the output layer passed as input; the results are stored in a float array containing (for this application) the probabilities of each class (i.e., the locations).

After calling the aforementioned methods, the prediction result is finally passed to the *argmax* function to return the most probable location where the user is located according to the input fingerprint.

From a structural point of view, we designed the Android app by splitting it into two parts: one for the learning and another one for the tracking of the user. With respect to the learning part, it is responsible to collect the labeled wireless fingerprints via multiple Wi-Fi scans. This is accomplished using the method *getScanResults()* provided by the *WifiManager* class of Android. During this phase, the user communicates the room where he is located by just tapping on the map showed on screen as depicted in Figure 2.5a.

When in "learn mode", the user can decide to send one fingerprint through the button "SEND FINGERPRINT" or to send them continuously in background by tapping on "SEND (BKGD)". Each time a scan finishes, all the data and the corresponding labels are stored on a json file and then sent to the server. The tracking part is responsible to localize the user inside the indoor environment and it is showed in Figure 2.5b. Like the previous case, also in "Track mode" the user can decide to be tracked only once or continuously by pressing the button "TRACK ME" or "TRACK ME (BKGD)" respectively. During this phase, the DNN stored internally the application is fed with the MAC addresses and the corresponding RSSIs detected by the device to get a prediction of the most



(a) Learn Screen (b) Track Screen

Figure 2.5: Android application screenshots.

probable user location. Sometimes, it can happen that the app perceives new MAC address for which the DNN is not trained; since is not possible to change the neural network topology of a trained model, the application simply drops all those MAC addresses which are not contained in the features set $\mathcal{F}$.

2.1.4 Localization results

We conducted a set of experiments to test how the DNN performs in a real indoor scenario to measure the accuracy and system response time while predicting the user location. In this experiment, a user is asked to follow a specific path and to notify the entrance in a room using a button displayed on the application. In

particular, this signal has been used as *ground truth* to be compared with the output of the trained DNN. At fixed time intervals T, a Wi-Fi scan is performed and passed to the DNN to predict the user location. For this experiment we set the time intervals T to 1 second. Such a choice resulted a good trade-off in terms of device energy consumption and prediction accuracy, considering a user moving at a normal walking speed (about 1.4 m/s).

Once we established the path to follow, we also fixed a residence time of 1 minute for each location (except for the $MainCorridor$) during which the user is free to move inside the room without any type of constraint. By recalling Figure 2.1 and considering $Room0$ as starting point, the path traveled in this experiment is the following: $Room0 \rightarrow MainCorridor \rightarrow Room1 \rightarrow MainCorridor \rightarrow Room2 \rightarrow MainCorridor \rightarrow Room3 \rightarrow MainCorridor \rightarrow Room4 \rightarrow MainCorridor \rightarrow Room5 \rightarrow MainCorridor$.

This in particular represents one of a large set of experiments we performed in the $7-th$ floor of the engineering department. However, according to the empirical test we made, the path followed by the user did not affect the overall accuracy of the localization system which remained always in the same range.

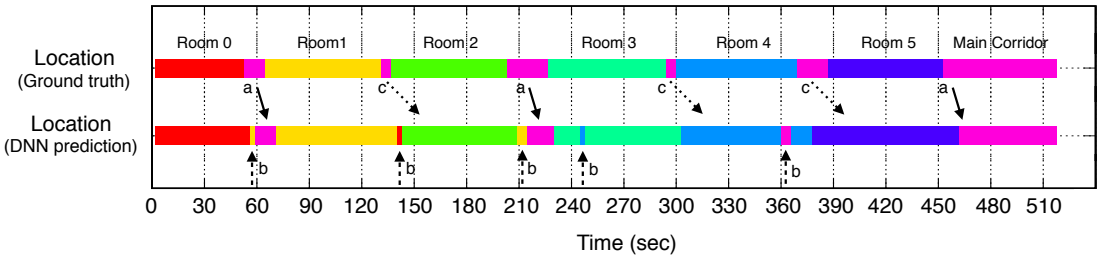


Figure 2.6: DNN prediction of the user location during a specific path taken as ground truth.

Figure 2.6 shows the comparison between the ground truth and the DNN prediction over time. The x-axis represents the time of the experiment (which lasted 520 seconds) while the colored bars represent the ground truth and the predicted user location. In general, the DNN was able to correctly detect the user position, however it generated some errors which we highlighted in Figure

2.6 with arrows labeled as a, b, and c representing three different error categories.

The first error (category a) is generated by a localization prediction delay and indicated in the figure with a solid arrow. It is mainly caused by the fact that the ground truth trace is instantaneously updated by the user when he enters in a new location. On the other hand, the DNN trace depends on the time intervals T at which the scans are performed, on the time required to retrieve the results from the Wi-Fi scan, and on the time necessary to run the DNN and get the prediction. Even considering these aspects, the delay introduced by the system is acceptable and does degrade the quality of the localization.

The second error category (category b) is indicated by dashed arrows and represents the DNN glitches during the inference phase. These errors, are caused by the DNN which makes a wrong prediction given a fingerprint as input. However, such a behavior is acceptable since the DNN is able to correctly detect the user for most part of the experiment.

Finally, the last category represented with dotted arrows labeled with c refers to a condition where the DNN is unable to localize the user when crossing the $MainCorridor$. In particular when the user moves between two very near locations e.g., $Room1$ and $Room2$ or $Room3$ and $Room4$ (as showed in Figure 2.1), the time necessary to move from a room to another is the order of a few seconds. The DNN is unable to catch such quick movements, thus resulting unable to detect the user position when he remains in the main corridor for a short period. Of course, this is a problem that happens only when the user moves between two near locations, in fact, by focusing on the last part of the experimental results, we can observe that the DNN is able to correctly localize the user in the $MainCorridor$ when remaining for a longer time.

At the end of the experiment, the accuracy reached by the DNN on a "dynamic" scenario was equal to 83.6% which is lower than the one obtained in the "static" test where we just fed the model with a test set. Nevertheless, the results obtained during this experiment are still good considering that the above men-

tioned errors become negligible in an application context where the user moves in an indoor environment remaining for long periods in the same location, and accessing services provided by smart objects inside of it.

Chapter 3

Smart Cities

In the recent period, smart cities have become an enabling technology for the development of infrastructures where applications can run. The Cloud paradigm plays a very important role for the realization of smart city scenarios, however, when working with applications which exhibit strict requirements in terms of latency and privacy, it becomes ineffective. In this chapter, we present two solutions that exploit AI and Edge computing technologies to improve applications performance in smart city contexts [26], [27].

3.1 Application relocation in Multi-access Edge Computing

Multi-access Edge Computing (MEC) is an emerging paradigm which shifts data and computational resources near mobile users, with the goal of reducing applications latency and improve resource utilization. Nowadays, smart services are becoming more resource demanding, requiring sometimes strict latencies that Cloud-based applications are unable to meet. In such a context, MEC standardized by the European Telecommunications Standards Institute (ETSI) can be considered a valid solution to address this problem [28]. MEC allows to move the Cloud capabilities at the edge of a mobile network, close to the users. Through

this paradigm, we are able to strongly reduce the communication latency, while enabling the development of more effective services that can benefit their proximity to the user to extract context information (e.g., user position, real-time awareness of the radio-access network, etc.), by means of secure and controlled interfaces. Initially designed to be a part of the 4G architecture, MEC will play a key role for the upcoming 5G network where it will be implemented as part of the Network Functions Virtualisation architecture [29].

One of the main challenges in MEC consists in the application relocation. To exhibit a low latency, applications must follow the mobile users by relocating from a MEC server to another one [30]. Although the MEC exposes a set of functionalities that allow to relocate an application (or service) in a seamless way to the user, suitable *policies* have to be designed to decide if, when, and where an application should be relocated. A policy should also take into account several parameters e.g., the availability and the load of the destination server, the communication latency, the overhead caused by the relocation, etc., with the goal of optimizing the application Quality of Service (QoS). Moreover, if we consider that these policies should work not just for a single user, but on global scale and for a wide range of applications, it is more than evident how challenging is this problem.

3.1.1 MEC architecture

The core elements of the MEC architecture are the Mobile Edge (ME) Hosts [31]: nodes equipped with storage and computing capabilities placed very close to radio stations, thus allowing low latency and context-aware services to the User Equipments (UEs). Figure 3.1 (left) depicts the internal structure of a ME Host. It provides an environment to run ME Applications (MEApp) as Virtual Machines (VMs) instances on top of a Virtualisation Infrastructure (VI). The ME Platform (MEP) is responsible to instantiate and terminate the execution of MEApps and provides the access to useful services by means of APIs. In

this context, two relevant services are: the Location Service (LS) and the Radio Network Information Service (RNIS). With respect to the first one, it provides location information of the UEs inside the network in terms of geolocalization indicating the radio station which is serving a UE. The RNIS provides information about the state of the radio network (e.g., the available bandwidth and UEs channel conditions). The overall structure of a ME Host is managed by the ME Orchestrator (MEO) whose task is to collect the information of the entire system in terms of topology (i.e., where the ME Host are deployed), the amount of resources available for each ME Host, and which types of services they provide. A possible deployment is shown in Figure 3.1 (right) where the ME Hosts are co-located with radio base stations (called eNodeBs, in the LTE lexicon). In this case, since the ME Host is directly connected to the eNodeB (eNB) the data traffic for MEApps only traverses the radio-access network, thus minimizing the latencies. However, in a mobility context such a deployment requires the relocation of MEApps every times a UE performs a handover (i.e., it enters in the coverage area of a new eNB). Not doing this, in fact, would cause a large number of inter-cell messages that could cause the network collapse.

As an alternative, the ME Hosts can be deployed close to a set of eNBs in order to cover a larger geographic area (instead of a single one like in the previous case). However, this will result in a higher MEApp latency and network traffic.

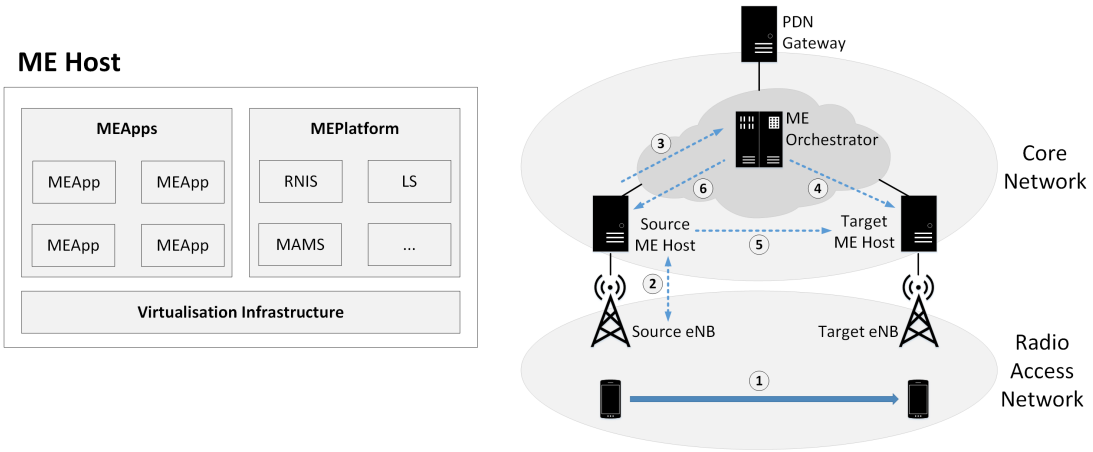


Figure 3.1: ME Host architecture (left) and relocation procedure (right).

An example of application relocation is depicted in Figure 3.1 (right). When an UE changes its serving radio station (step 1), the RNIS at the source ME Host collects the information (step 2). The ME Application Mobility Service (MAMS) running inside the MEP of the ME Host saves the state of the MEApp and forwards a relocation request to the MEO (step 3). When the MEO receives the request, it identifies a possible target ME Host, verifies if it has enough resources, if it supports the MEApp, and if it is able to satisfy the QoS requirements. If all the conditions are satisfied, then the MEO instantiates a new MEApp into the target ME Host (step 4) and transfers the UE context (step 5), if any. Finally, the MEO sends a termination message to the source ME Host to terminate the execution of the MEApp (step 6).

3.1.2 Deep reinforcement learning for application relocation

When the number of states describing an environment is too large (e.g., in the order of 10^{20} or more), RL becomes not very effective as it would require too much time to explore the entire environment. If we consider for example a Q-learning agent (described in Chapter 1), it has to traverse several times the entire environment to learn a policy, thus making unfeasible the use of this approach. States clustering (or state quantization) was initially considered a way to address this problem, however, it inevitably causes the learning of an imprecise policy. In such a context, Deep Reinforcement Learning (Deep RL) is a valid solution that adopts DNNs as function approximators to estimate the Q-values for those state that have not been explored. Originally pioneered by DeepMind [32], Deep RL provides a new way to represent the dynamics of complex environments with a very large number of states and actions.

The problem of relocation is not new in the literature and it has been tackled using different approaches. Authors in [33] used a traditional machine learning approach to predict user mobility in order to implement a proactive relocation

strategy in order to minimize system downtime. In [34] is presented a multi-agent RL scheme, where agents compete among themselves to find the best offloading policy. Unfortunately, in a real scenario where the number of states is too large to be modeled, the use of simple RL techniques in unfeasible. The approach discussed in [35] uses Deep RL to deploy an optimal strategy for the data offloading between mobile devices and cellular base stations. The technique presented in [36] uses a Deep RL framework to learn a binary offloading policy from the experience. Even though both the aforementioned works enforce Deep RL techniques, they are mainly related to general mobile systems, not specifically addressing 5G/LTE features. In this sense, our proposed approach is more general exploiting system status information to implement complex policies for the relocation of applications to a specific location.

Scenario

A MEC-enhanced LTE-A network (showed in Figure 3.1 (right)) is considered as scenario for our approach. Here, ME Hosts are placed close to the eNBs and run the MEApps requested by the UEs. The core element is represented by the Deep RL engine (run by the MEO) which learns a sub-optimal policy through a trial and error process. Given a set of MEApps and UEs, the data for the Deep RL application is gathered through the MEP using the MAMS. In particular, the MAMS obtains from the RNIS the number of UEs attached to the corresponding eNB (i.e., the eNB to which is attached the ME Host), while from the LS it obtains the geolocalization for each UE connected to the MEApps running on the ME Host. This information is sent from each ME Host to the MEO (acting as a data collector) and then passed to the Deep RL algorithm to make an action (i.e., to decide which MEApp should be relocated and where). Once a decision is made, the result is returned to the MEO that initiates the relocation procedure (already described in Paragraph 3.1.1) communicating with the involved ME Hosts (i.e., the source and target one) by means of the MEC frameworks APIs.

Problem formulation

The key idea at the base of the Deep RL consists in the use of two separate DNNs: the *Main DNN* and the *Target DNN*. The Main DNN is used as function approximator predicting the Q-values for a given input state s. On the other hand, the Target DNN is used to generate the target Q-values necessary to train the Main DNN. The reason why we need a Target DNN comes from the fact that a DNN should not be trained on the output produced by itself, as it could generate errors in the train loop and cause a wrong learning. Using two networks, with different set of parameters results in a more stable learning which is not prone to oscillations.

According to the scenario described in Paragraph 3.1.2, we assume that each eNB has its own ME Host and UEs are free to move around the environment and change the serving eNB while moving through handover procedures. Each UE can run applications that directly connect to the MEApps contained in the ME Hosts. Said $\mathcal{A} =< (a_1, a_2, a_3, ..., a_{\mathcal{Z}}) >$, the set containing the actions that the Deep RL can make in the environment, each one represents a MEApp relocation from a ME Host to another one. Then, let define $\mathcal{S}$ as the set of states tuples s of cardinality $\|\mathcal{T}\|$, containing the UEs positions, the apps they are running, and which MEApps is running a ME Host.

$$\begin{aligned} s =< (& < UE^i > \;\; i = 1, ..., M, \\ & < UE^i_k > \;\; i = 1, ..., M, \;\; k = 1, ..., N, \\ & < ME\_Host^i_k > \;\; i = 1, ..., M, \;\; k = 1, ..., N) >, \end{aligned} \tag{3.1}$$

where UE^i is the number of UEs connected to the *i-th* eNB, UE^i_k represents the number of UEs running the *k-th* application in the *i-th* eNB, and $ME\_Host^i_k$ is a value that is 1 if the *k-th* MEApp is in the *i-th* ME Host, 0 otherwise.

3.1.3 Proposed Deep RL algorithm

The proposed Deep RL algorithm is showed in Algorithm 1. Line 1 sets up the replay memory E, a data structure containing the previous experiences accumulated by the agent. During the initialization phase the Main and Target DNNs weights θ and $\widehat{\theta}$ are set to the same values (lines 2-3). The discount factor γ has been already described (line 4). The batch size B represents the amount of samples fetched from the replay memory for the Main DNN training (line 5). The update period defines the number of execution steps after which the weights of the two networks are synchronized (line 6). The exploration rate $\epsilon \in [0, 1]$ is a parameter that adds a probabilistic component to the algorithm allowing to explore new Q-values by taking random actions (line 7). The exploration rate decays at each iteration at a rate d to favor convergence (line 8). At each iteration loop the algorithm observes the current state s_j (line 10) and depending to the exploration rate it decides to take a random action or to select the one with the highest Q-value according to the policy learned until that moment (lines 11-16). After that an action is selected, the corresponding relocation is performed (line 17). Then, the agent observes again the state it reached (line 18), the corresponding reward (line 19), and stores this experience (consisting of the state s_j, the action taken a, the state reached s_{j+1} and the reward r) in the replay memory E (line 20). The core of the Deep RL algorithm is between lines 21-27. First the Main DNN samples a batch of experiences B from E (line 21) and for each state s_j in B the Main DNN predicts the corresponding Q-values (line 23). Then, the Target DNN computes the target Q-values (line 24) and updates the Q-value associated to the action contained in the experience (leaving the other untouched) (line 25). Finally, the Main DNN is trained with the updated Q-values (line 26) and if U execution steps have been executed, the Target DNN weights are set equal to the ones of the Main DNN (line 28).

Algorithm 1: *Deep RL for MEApp relocation in MEC*

initialize experience replay memory E to $\{\emptyset\}$
random initialize main DNN network weights θ
set target DNN network weights $\widehat{\theta}$ equal to θ
set discount factor γ
set batch size B
set update period U
set exploration rate ϵ
set decay rate d
for episode = 1 to *end*:
  observe state s_j containing current UEs positions and applications distribution among the ME Hosts
  p = *random([0,1])*
  if $\epsilon > p$:
    action = *random([1,$\mathcal{Z}$])*
  else:
    action = *argmax*(Q(s_j,θ))
  end if
  relocate the application to the destination ME Host as indicated by *action*
  observe the new state s_{j+1} which contains the UEs positions and the new application distribution after the relocation process
  observe the reward r
  store the tuple (s_j,*action*,s_{j+1},r) in E
  sample a *batch* from E
  foreach s_j in B:
    y = Q(s_j,θ)
    $y_{target} = \widehat{Q}(s_{j+1},\widehat{\theta})$
    y_{action} = r + $\gamma \cdot max(y_{target})$
    execute one training step on Main DNN network
  end foreach
  every U steps set $\widehat{\theta} = \theta$
end for

3.1.4 Simulation environment

The scheme of our Deep RL framework is showed in Figure 3.2 and consists of two components: a MEC LTE environment and a Deep RL engine.

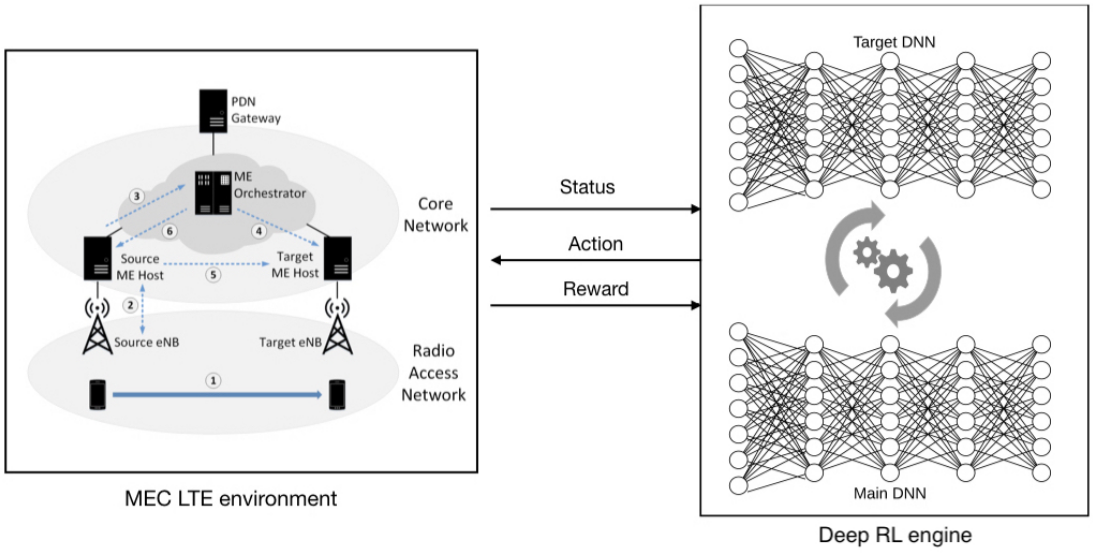


Figure 3.2: Deep RL framework.

With respect to the first one, it provides the input states for the Deep RL engine, receives the actions, and generates the corresponding rewards. This environment is simulated using two popular frameworks of the OMNeT++ [37] discrete event simulator, namely: SimuLTE [38] and INET[1]. The INET framework allows to simulate application level communications using all the layers protocols of the TCP/IP stack. Figure 3.3 shows the internal architecture of the main communication nodes. Both the UE and eNB are equipped with a Network Interface Card (NIC) providing the connectivity to the radio-access network and implements the complete LTE protocol stack. A UE can be configured to run one or multiple TCP/UDP applications which can communicate with one or multiple ME Hosts [39]. In addition, the UE node is also equipped with a mobility module that allows to move according to a way-point mobility where the points can be linearly or randomly generated. The eNB is connected to UEs by means of the

[1]Available at "https://inet.omnetpp.org/", last accessed September 2020

LTE NIC and to the Evolved Packet Core (EPC) (i.e., the core-network of the LTE) through the GPRS Tunneling Protocol (GTP). Finally, the ME Hosts can be placed at arbitrary points of the EPC and instantiate TCP/UDP MEApps when the UE requests it.

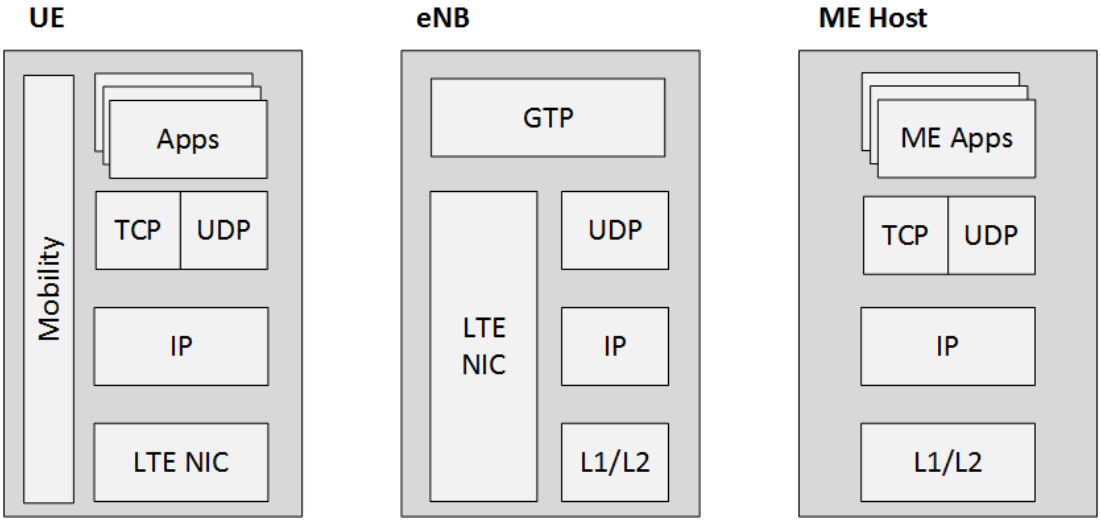


Figure 3.3: High-level view of the simulator main nodes and their layering.

The Deep RL engine has been implemented using Keras [40], an open source library running on top of TensorFlow (and many other frameworks) for the implementation of complex neural networks. Using Keras we realized a feed forward DNN with 4 hidden layers between the input and output layers with cardinality $\|\mathcal{T}\|$ and $\|\mathcal{Z}\|$ respectively. Table 3.1 reports the Main DNN and Target DNN hyperparameters setup we used in the proposed Deep RL algorithm.

Table 3.1: Main and Target DNNs hyperparameters.

DNN configuration	
# Hidden Layers	4
# Neurons per layer	$[21, 15, 15, 15]$
Activation function	$ReLU$
Learning rate	0.001
Regularization rate	0.01

In particular, we set the number of neurons for each hidden layer to 21 for the first layer and 15 for the remaining ones. As activation function we chose

the ReLU since it allows to speed up the training process. With respect to the optimizer, we used *Adam* with a *learning rate* of 0.001. Finally, to prevent the network overfitting, we used a L2 regularization with a regularization rate of 0.01.

The Deep RL engine has been written in Python and communicates with the OMNeT++ simulator via *syscalls*. When the data of the current state is ready, the OMNeT++ simulator makes a syscall to launch the script of the Deep RL solver. Once is called, the solver runs Algorithm 1 generating as output a text file with the action to enforce in the simulator. On OMNeT++, when the file is available, the simulator reads the file and changes the ME Host destination for the application indicated by the action, thus emulating the relocation. After that the action has been performed and the simulator evolved towards a new state, the Deep RL solver observes again the reward computed using several indices provided by OMNeT++ (e.g., the application response time, the load of the server, etc.), and compares it with the one observed before taking the action into the simulator. This information is used as feedback for the Deep RL agent to understand if the action performed can be considered a valid choice for that specific state or not.

3.1.5 Simulation results

To demonstrate the benefits produced by the proposed Deep RL agent, we created a Proof-of-Concept (PoC) case study consisting in an urban scenario where a group of stationary users access a video streaming application while another group accesses a different client-server application as it traverses the city highway.
Figure 3.4 depicts the above described scenario. Here, we deployed two different applications, namely: $MEApp_A$ which is always run by the ME_Host3, and $MEApp_B$ that can be relocated in any ME Host of the network. Each UE sends a request every $50ms$ and the ME Host response time depends on the number of UEs it is serving at the moment when the request arrives. In this urban scenario, we considered three eNBs positioned at 230 meters apart from each other. With

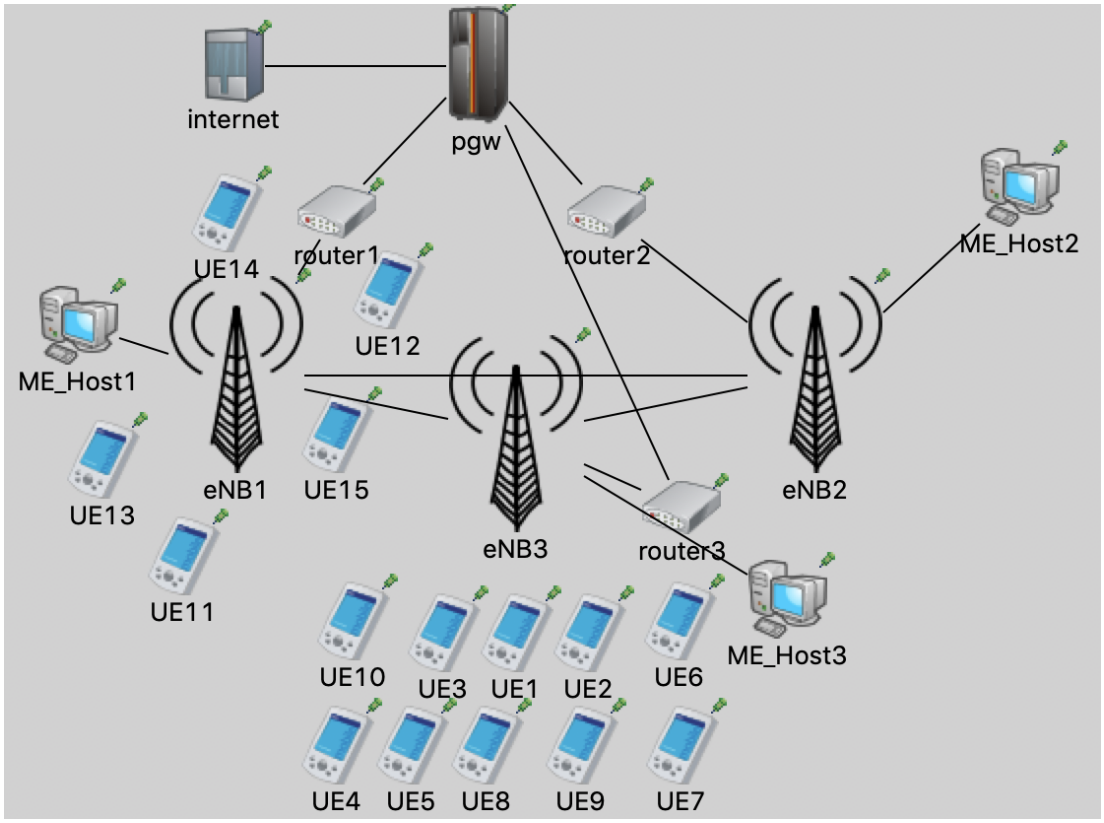


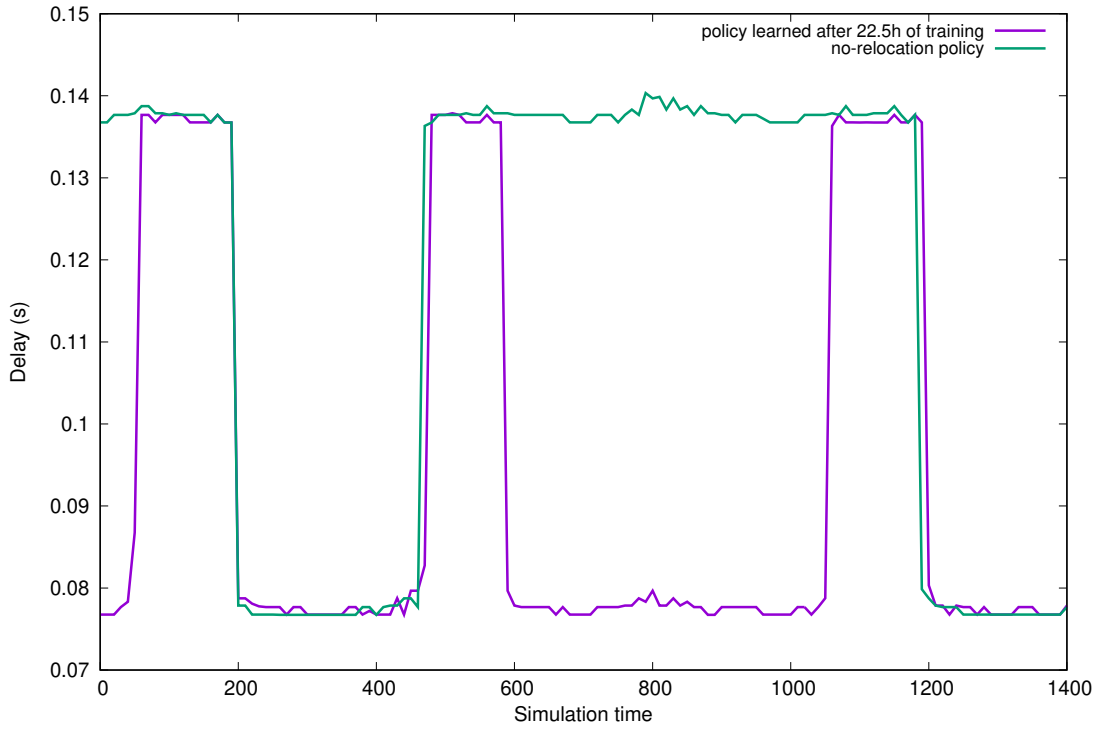
Figure 3.4: Urban scenario simulated on the OMNeT++ environment.

respect to the two groups of UEs, the first one consisting of ten users (numbered from 1 to 10) is static (i.e., the users do not move) and each UE uses the $MEApp_A$ attaching to the eNB3. On the other hand, the second group composed by five UEs (numbered from 11 to 15) and initially attached to eNB1 runs the $MEApp_B$. In this case, the users are able to move at a speed of about $50km/h$ (representative for the speed of a generic car) approaching eNB3 and eNB2. These last users while moving perform handover according to the signal strength received from eNBs. This allowed us to compare our algorithm with three straightforward heuristics i.e., a "no relocation" policy where the $MEApp_B$ always remains in the ME Host in which is initially deployed, a "load" policy where the $MEApp_B$ is relocated to the least loaded ME Host (i.e., the ME Host with the lowest number of UEs attached to it), and a "majority" policy where the $MEApp_B$ is relocated towards the ME Host where the majority of UEs is attached to.

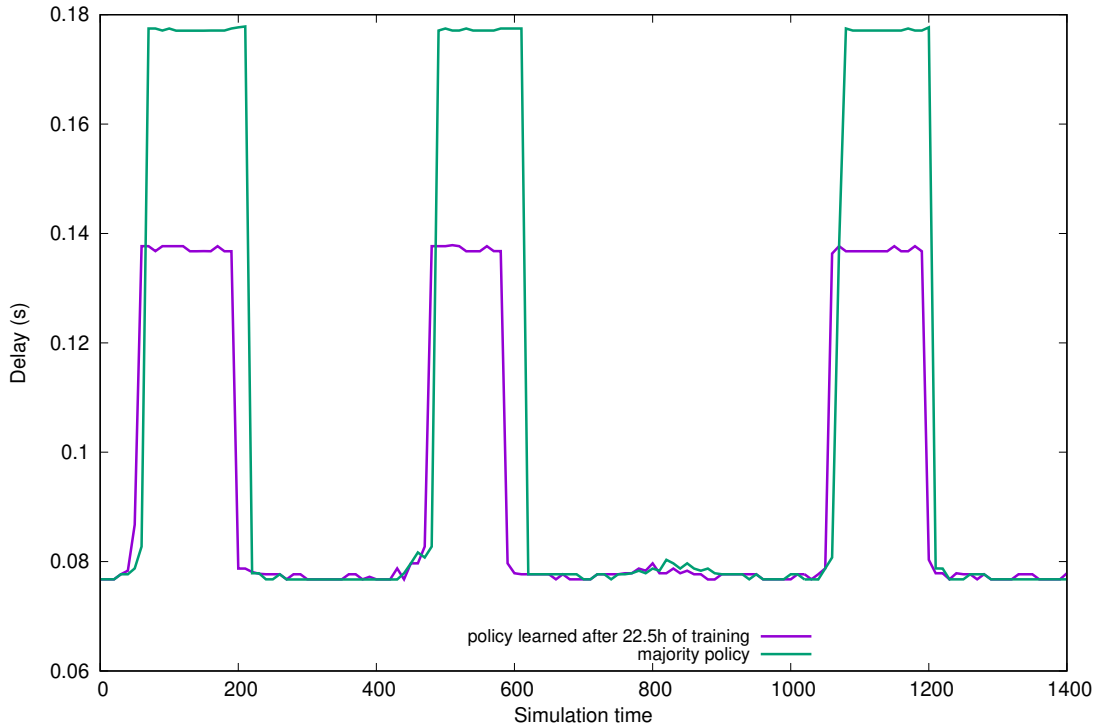
We compared the application level delay for each of the above described heuristics, computed every 10 second intervals and obtained as the sum of two terms: the Round Trip Time (RTT) which measures the time to traverse the network of a packet (also influenced by the position of the UE and the serving ME Host), and the computation time of the ME Host depending on its load. To test our system, we trained the Deep RL engine inside the simulator in order to let it make experience and learn a sub-optimal policy for the scenario we previously described. Figure 3.5 shows the comparison between the policy learned by our system and the policies above described, together with a histogram depicting how the mean applications level delay decreases as the training time (and so the experience) increases.

Figure 3.5a shows a comparison between the policy learned by the Deep RL agent after $22.5h$ of training time and a "no-relocation" policy where we consider the $MEApp_B$ deployed on ME_Host2. In such a context, the agent has been able to lean an effective policy that allows to reduce the application delay for the most part of the simulation. It is worth to mention that those intervals where the two policies coincide correspond to a condition where the "no-relocation" policy is optimal (i.e., when the UEs are served by the ME_Host2). At the beginning of the simulation, and in the interval $[600s; 1100s]$ our system is able to improve the performance of the application reducing the delay of about 45%.

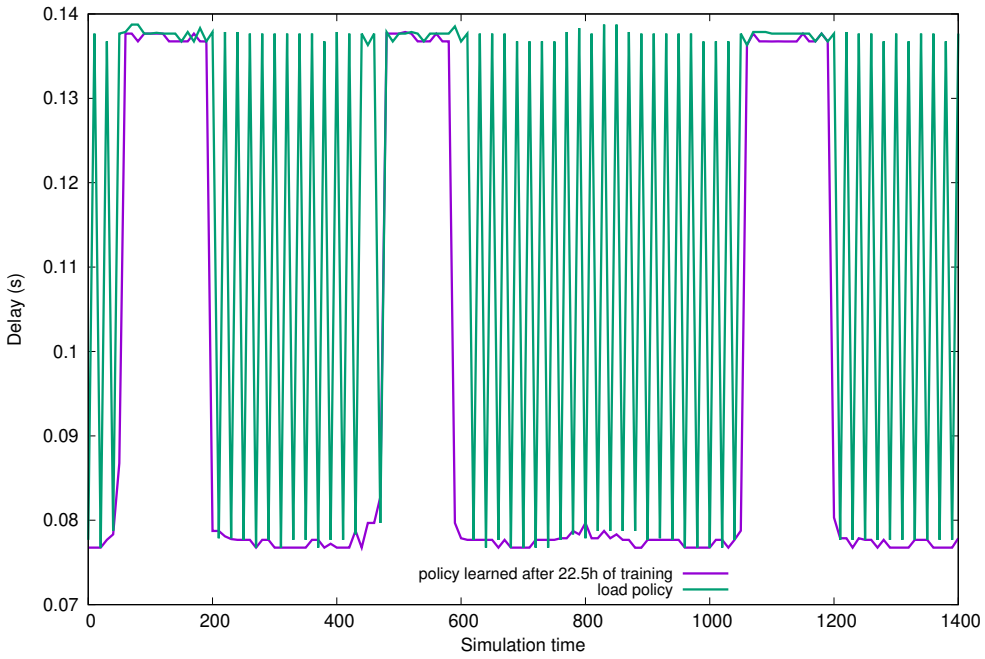
Figure 3.5b compares the policy learned by the agent with a "majority" policy. Also in this case, it is more than evident that the proposed policy outperforms the other for almost the entire duration of the simulation. In particular, during the time intervals $[100s; 200s]$, $[500s; 600s]$, and $[1100s; 1200s]$ which correspond to a condition where the majority of the users is attached to eNB3 together with ten stationary users connected to ME_Host3 which is running the $MEApp_A$ (as shown in Figure 3.4), the Deep RL agent is able to understand that the best strategy is to relocate the $MEApp_B$ into the ME_Host1 or ME_Host2 (depending on which is the closest) instead of further increase the load of ME_Host3. By



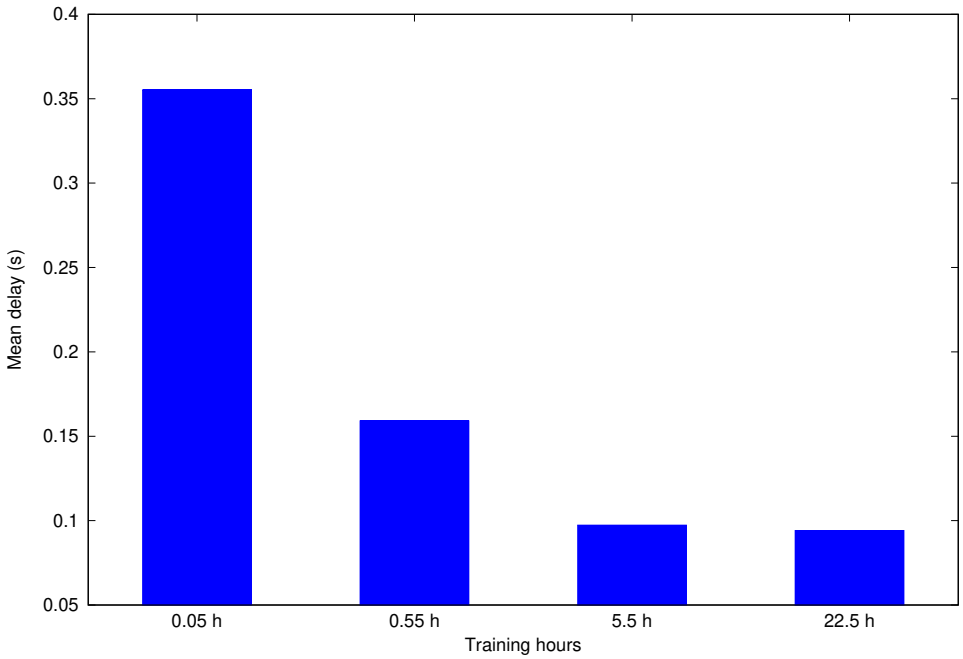
(a) Deep RL agent Vs. "no-relocation" policy.



(b) Deep RL agent Vs. "majority" policy



(c) Deep RL agent Vs. "load" policy



(d) Mean delay reached by the Deep RL agent at different training phases

Figure 3.5: Policy comparisons and impact of the training time.

doing so our policy allows to reduce the delay of about 20%.

Figure 3.5c depicts a comparison between our policy and a "load" one which

deploys the application towards the least loaded ME_Host without caring its distance to the UE. Since the ME_Host2 always serves ten user for the entire duration of the simulation, such a policy will never choose it for the relocation. In this sense, this heuristic relocates the $MEApp_B$ between the ME_Host1 and ME_Host2. As a result of this behaviour, the delay tends to continuously oscillate except for the above mentioned intervals during which the majority of users is connected to the eNB3. Similarly to the previous case, the policy learned by the Deep RL agent outperforms the other one for the entire simulation, thus proving that the agent has been able to learn the network dynamics of the proposed scenario and to deploy a sub-optimal policy for the application relocation.

Finally, Figure 3.5d shows the mean delays obtained by averaging the delays returned by an entire simulation and computed at different training phases. As we would expect, at the beginning of the training phase, the mean delay is still high. In particular, this is caused by the exploration phase of the agent which has not learned a proper policy, but performs random actions making mistakes and causing a noticeable increase of the delay. After $0.55h$ of training, the Deep RL agent has already learned a very good policy which allowed to reduce the delay of about 55%. Once the agent arrived at $5.5h$ of training, it has learned a sub-optimal policy further reducing the mean delay, and continuing to reduce it as long as the training time increases. In general, we can see that as the training progresses, the policy learned by agent improves, thus keeping the QoS as high as possible.

The results obtained from this simulation demonstrate that the use of smart relocation techniques based on AI can improve the overall QoS of applications and encourage the investigation of new techniques to be used as support technologies for the improvement of the next generation of mobile cellular networks.

3.2 Exploiting federated learning in smart city scenarios

In the previous paragraph, we presented a Deep RL technique for the application relocation in MEC. In such a context, we tackled the problem using a centralized approach where all the data is concentrated in a single point (i.e., the MEO) to perform the training and inference processes. Unfortunately, such a solution is not always applicable and results to be unsuitable for certain types of applications. In the last decade, we assisted to the IoT wide spreading that catalyzed the Big Data phenomenon [2]; in such a context the AI can benefit from this abundance of data allowing the implementation of more sophisticated applications that can make "reasonings" according to the context [5].

As result of this condition machine and deep learning applications are becoming "data hungry" requiring an increasing amount of data [41]. However, larger and more complex data requires also the use of deep learning algorithms whose training procedures can be unaffordable for an Edge device. To solve this problem, it could be possible to use a hybrid approach where the training is done on a powerful machine (e.g, the Cloud) and then perform the inference process on the Edge, but resulting in bad scalability and fault tolerance.

Federated Learning (FL) is an emerging technique that introduces a new way to perform the training process of machine and deep learning algorithms [42]. The concept at the base of FL is the collaboration; a set of clients (called participants) is trained under the coordination of a central entity to learn a new model which synthesizes the "knowledge" of each contributor. Moreover, FL allows also to preserve the data privacy since the data used for the training process is not shared among the participants, but it is kept local [43]. Recent approaches propose the use of FL in smart cities scenarios where a huge number of devices (i.e., ICPS) are blended with the environment generating data. In such a context, hundreds of clients collaboratively participate to create a global model to improve

its performance in doing a specific task [44].

If on the one hand FL solves the problems related to computing, scalability, and privacy, on the other it introduces a set of challenges related to the use of a distributed approach. Problems like sudden disconnections, limited bandwidth (just to name a few) have to be carefully managed through the implementation of effective frameworks. Moreover, if we consider the clients heterogeneity, the implementation is even more difficult. In such a context, we propose an extension of *Stack4Things* (S4T), a cloud based platform developed in our Engineering department that allows to orchestrate IoT devices. Exploiting its functionalities we implemented a FL framework to enable a distributed training over heterogeneous devices without caring the location, network configuration, and technology [45].

The literature proposes several solutions for the realization of a FL approach. Authors in [46] propose a work that exploits FL for Chinese text-recognition and compare two popular frameworks such as: TensorFlow Federated (TFF) and PySift[2] which are still under a development stage. TFF for example exposes a set of APIs that ease the implementation of FL algorithms. However, the framework does not provide any support in terms of communications between the clients and Cloud. On the other hand, S4T has been designed to effectively manage the communication among devices, even if behind NAT or firewalls [45].

With respect to PySift, it is a FL framework which uses cryptography techniques to deploy a secure training process, but only for Python libraries like TensorFlow, Keras, and PyTorch. In this sense, S4T works also with other machine learning tools since its base functionalities are independent from the application code running on top of it.

In [47] authors present a promising framework called "Flower" for the fast deployment of FL schemes on heterogeneous clients. In this work authors also demonstrate the effectiveness of their approach when compared with others (e.g., TFF and PySift), however Flower is only able to deploy FL schemes. Thanks

[2]https://www.openmined.org/, accessed October 2020

to the modularity of S4T, we can use it not only to accomplish FL tasks, but also to implement training procedures on the Edge, the Cloud, or using hybrid approaches.

In this thesis work, we present a PoC case study where we analyze a smart city urban scenario, wherein we think that the use of a FL can be beneficial. In such a context characterized by a huge amount of field deployed heterogeneous ICPS, where smart services are becoming more resource and data demanding while imposing strict requirements (e.g., latency and privacy), the use of a distributed approach can be considered a promising solution to improve the performance of smart applications through the cooperation of a set of clients for the accomplishment of a common task.

3.2.1 Stack4Things architecture

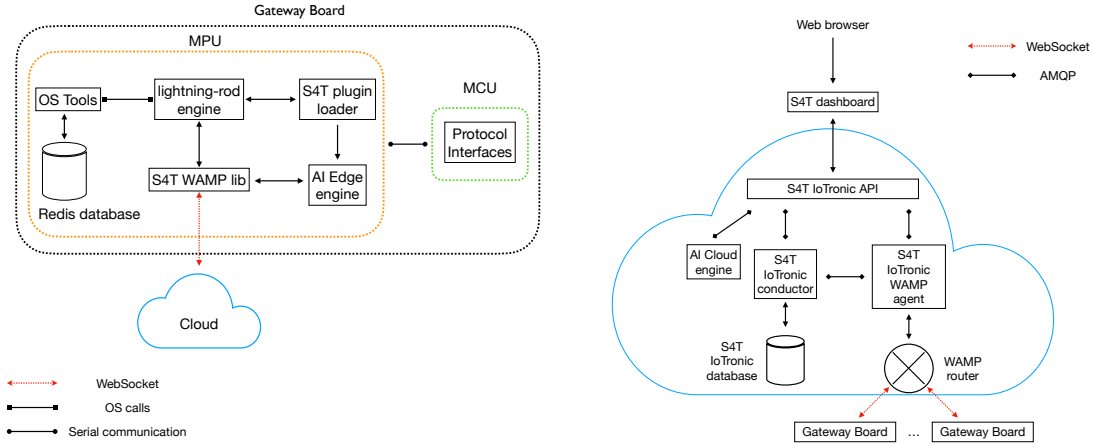
S4T is an open-source platform that has been developed with the goal to integrate the Cloud technologies with the IoT infrastructure [48]. Based on the OpenStack framework, S4T has been designed to meet several requirements like the scalability, registration mechanisms, and the possibility to work with heterogeneous systems. Given the above features, the platform performs the data acquisition and visualization tasks of samples coming from "things", it provides sensing-actuating capabilities to fleet of IoT devices, and allows their orchestration and control.

From an architectural point of view, S4T is split into two parts (showed in Figure 3.6, namely: the gateway side running on the Edge (i.e., the ICPS) and the Cloud side.

Gateway side

Figure 3.6a depicts the software architecture on the gateway side where we put in evidence the main building blocks. In such a context, the core element is represented by the *lightning-rod* engine that communicates with the Cloud through

the Web Application Messaging Protocol[3] (WAMP) router using the Web Socket (WS) protocol. All the aspects related to the communication are implemented in the *S4T WAMP lib* that exposes a set of APIs to ease the sending and receiving of data. The engine also implements a plugin loader (i.e., *S4T plugin loader*) that allows to remotely inject code into the board in real time. Such a functionality has been implemented providing the access to the Operating System (OS) tools (i.e., hardware peripherals, filesystem, package manager, etc.), thus resulting very useful for maintenance and system upload purposes.



(a) Gateway side (b) Cloud side

Figure 3.6: S4T software architecture of gateway and Cloud sides.

The main extension we implemented in S4T is the AI engine that enables the training and inference processes directly on the board. In particular, the engine is able to retrieve the data in two ways (depending on the application context): from the Cloud by means of the S4T WAMP lib (via WS) and from the Micro Controlling Unit (MCU) via several protocol interfaces (e.g., Bluetooth Low Energy (BLE), Wi-Fi, MODBUS, etc.). Specifically, the Micro Processing Unit (MPU) and MCU communication is implemented using the shared key/values database called *Redis*. When a new chunk of data arrives at the MCU, it uses the serial communication to write the information in the database. On the other hand, the MPU accesses Redis and forwards the data to the AI engine to train or

[3]https://wamp-proto.org, accessed October 2020

make the inference on it. The AI engine communicates also with the S4T plugin loader whose task is to inject machine learning models, on top of which the engine can perform inference tasks or the training process.

Cloud side

With respect to the Cloud side, Figure 3.6b shows its architecture. Likewise the gateway side, also in this case we can identify a core element represented by *IoTronic* which can be considered the Cloud counterpart of the lightning-rod engine. Specifically, IoTronic consists of three components, namely: *S4T IoTronic WAMP agent*, *S4T IoTronic conductor*, and *S4T IoTronic database*.

The S4T IoTronic WAMP agent can be considered the entry point for the Cloud. When a message is sent/received from/to the Cloud, the WAMP router interacts with the agent whose task is to convert the WAMP message into an Advanced Messages Queueing Protocol[4] (AMQP) one (and vice versa). The choice to convert the messages comes from the fact that S4T is an OpenStack[5] extension module, and for a better integration, its components should "talk" using the AMQP protocol.

The S4T IoTronic conductor is a controller to perform operations on the S4T IoTronic database through the execution of Remote Procedure Calls (RPCs). The presence of a conductor "entity" has several advantages in terms of security since it provides a communication interface with the database that prevents to directly access it, thus avoiding potentially dangerous operations. Finally, the S4T IoTronic database stores the information of registered boards (e.g., board identifiers, boards locations, which types of services they are running, etc.), and keeps track of the boards connected to the platform.

Like the gateway side, in this extended version of S4T, also the Cloud has been equipped with an AI engine to accomplish machine learning tasks. The engine communicates with the rest of the system by means of the *S4T IoTronic API*

[4]https://www.amqp.org, accessed October 2020
[5]https://www.openstack.org/, accessed November 2020

that enables it to perform training and inference processes as well as to deploy machine learning models to the gateway boards exploiting the S4T plugin loader. The very last component of the architecture is the S4T dashboard that allows to access all the framework functionalities via a web browser using the S4T IoTronic API.

3.2.2 Federated learning approach implementation

In the previous paragraph, we provided a detailed description of the S4T software organization putting in evidence how its internal components interact. In this paragraph we present the FL approach and how we implemented it in the proposed architecture. FL is an emerging machine learning technique that enables a new training paradigm based on the collaboration. The key aspect of this technique is that the participants do not reveal their data, as a result, FL allows to reduce the overall training time while preserving the privacy [49], [50].

In such a context, the clients (typically passive entities) become an active part of the training procedure, actively participating to the "knowledge building" process. The motivation of FL derives from the necessity to abate the training time on large datasets exploiting the workload distribution capabilities of this approach, thus improving the scalability, fault tolerance, and privacy [47]. Figure 3.7 shows a possible FL deployment scheme.

Here, we can define the four main steps iterated by a FL algorithm: *i.)* global model distribution, *ii.)* local training, *iii.)* weights aggregation, and *iv.)* global model update. During the first step, the Cloud contacts each participant and sends in parallel the global model (initialized with random values at the first iteration). When the global model is received by all the clients, they perform a fixed number of training steps (agreed with the central entity) using their local data and send back the trained model to the Cloud. Once each client returned the trained model to the server, it aggregates the weights of these models to create an updated version of the global one. FL proposes several strategies to merge

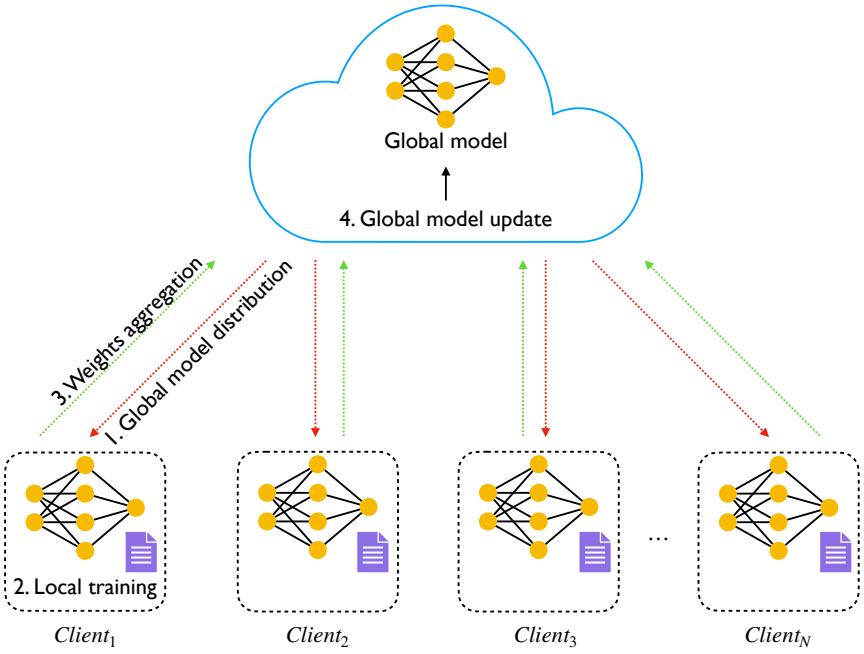


Figure 3.7: Deployment of a FL scheme.

the client models, but most of them involve the use of the *FederatedAveraging* (FedAvg) algorithm [47] that computes the weights average according to the following equation:

$$W_g^L = \frac{1}{N_c} \cdot \sum_{c=1}^{N_c} W_c^L, \tag{3.2}$$

where W_g^L and W_c^L are the weight matrices of the $L-th$ layer of the global and client models respectively, and N_c is the the number of clients participating to the training.

Algorithm 2 shows an implementation of FedAvg. Steps from line 1 to line 4 are used for the parameter initialization. At the beginning a shared global model is created with random weights (line 1), and the numbers of training epochs E, clients N_c, and training rounds T_r are set (lines 2-4). The rest of the algorithms consists in the iteration of the above explained steps when we introduced the FL scheme. The Cloud first sends (in parallel) the global model to all the clients c that want to participate to the training process (line 7). Then, each client

executes T_r training rounds on the global model received and updates the local weights W_c using its local data (line 9). After this step, the server waits for the trained client models (line 12), aggregates them using eq. (3.2) (line 13) and updates the global model weights W_g (line 14). In particular steps from lines 6 to 14 are repeated until the total number of training epochs E is reached.

Algorithm 2: *FedAVG algorithm*

```
1  build a global model and initializes the weights W_g to random values
2  set the number of training epochs E
3  set the number of clients N_c that will participate to the training
4  set the number of training rounds T_r to be executed on the client side
5  for epoch = 1 to E:
6    do in parallel
7     send global model to each client c participating to the training
8     for round = 1 to T_r:
9       update local model weights W_c
10    end for
11   end
12   server receives client models
13   aggregate the weights using eq. (3.2)
14   update global model weights W_g
15 end for
```

For a better understanding, Figure 3.8 depicts the entire workflow that implements the FL approach for a given client. By taking a closer look to the communication scheme, we can think to split it into two phases, namely: the registration, and the federated learning. If we consider a condition where S4T is installed from scratch, before interacting with the platform, it is necessary to make a registration with the Cloud in order to access all the framework functionalities. The registration starts with a request via WS through the lightning-rod engine (step 1). When the request arrives on the Cloud side, the WAMP router forwards it to the S4T IoTronic WAMP agent that converts the request into an AMQP message and contacts the S4T IoTronic conductor. The registration is accomplished using a suitable SQL query (step 2) that adds an entry into the S4T IoTronic database (step 3) that will persist as long as a de-registration request

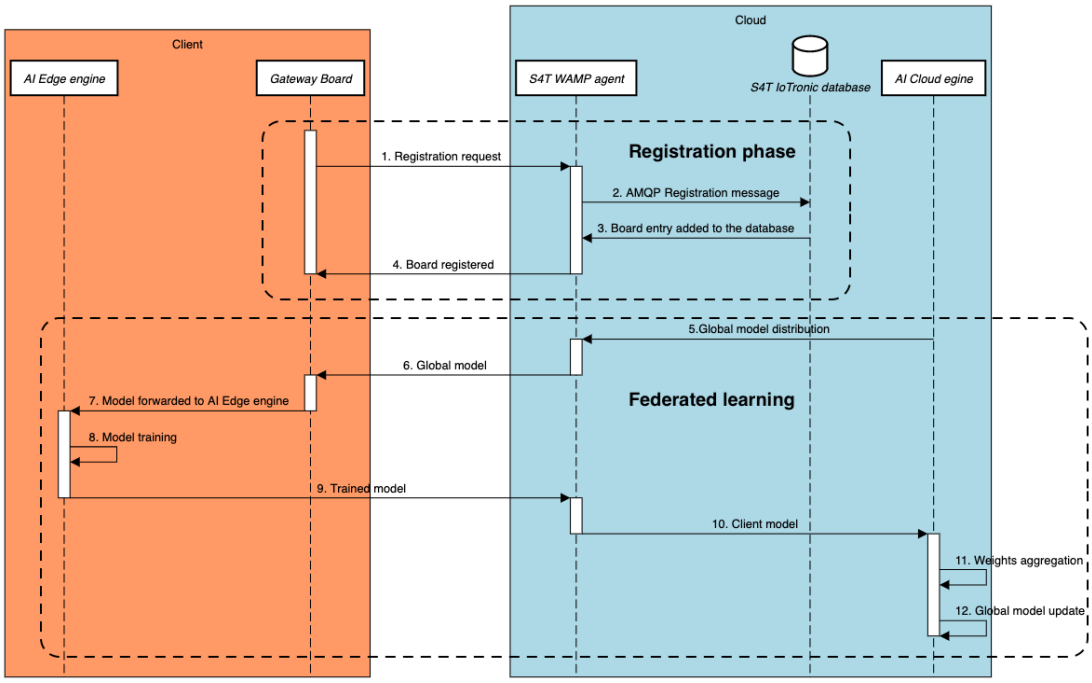


Figure 3.8: Client registration and FL training in S4T.

is performed. Finally, a message confirming the board registration is sent to the client gateway board (step 4).

The second phase of the workflow is the actual FL algorithm. Once each client confirmed its participation to the training process, the Cloud sends in parallel an initial global model with randomly initialized weights. In S4T this is accomplished by the AI engine of the Cloud which forwards the model to the S4T IoTronic WAMP agent (step 5). Then, the agent distributes the model in parallel to all the participants (step 6). When the gateway receives the model, it is passed to the AI Edge engine via the S4T plugin loader (step 7) to enable the training process (step 8). Once the client finishes the training, it sends the model to the S4T IoTronic WAMP agent (step 9) which then forwards it to the AI Cloud engine (step 10). After receiving the models from each client, the engine executes the weights aggregation to merge them into a single global model (step 11). Finally, the global model is updated (step 12). The above mentioned steps are iteratively repeated until the number of training epochs is reached (as

described in Algorithm 2).

3.2.3 Application scenario

To prove the feasibility and the effectiveness of the FL approach, we compared it with a centralized one to put in evidence its befits. In our experimental setup, we built a distributed scenario where the Cloud instance of S4T is represented by a public server available in our University department. With respect to the gateway side, in order to demonstrate the capabilities of the framework to work with heterogeneous hardware and software, we considered five clients: a laptop, three Raspberry Pi 3, and a NVIDIA Jetson Nano. Moreover, we positioned the gateways in different buildings of the University campus such that they do no lay on the same network, thus emulating a distributed deployment as showed in Figure 3.9. For a better understanding, in Table 3.2 we report the clients and server hardware configuration in terms of CPU, GPU, and RAM.

Table 3.2: Server and clients hardware configuration.

Hardware configurations			
Hardware	*CPU*	*GPU*	*RAM*
Server	*Intel Xeon* @ 2.13 *GHz*	–	16 *GB*
Raspberry Pi 3	*ARM A*53 @ 1.4 *GHz*	–	1 *GB*
Jetson Nano	*ARM A*57 @ 1.43 *GHz*	*NVIDIA Maxwell*	4 *GB*
Laptop	*Intel Core i*7 @ 2.6 *GHz*	*ATI Radeon Pro*	16 *GB*

In such a context, we considered a PoC case study which can benefit from the use of a FL approach. In particular, we selected a smart city application scenario where a set of traffic cameras is deployed at road junctions collecting images of vehicles (e.g., cars, trucks, buses, etc.) as showed in Figure 3.10. Each camera monitors only a part of the road and is equipped with a hardware that allows it to make computation and connect to the Internet. However, for privacy reasons the images captured by the cameras are private and can not be shared with external "entities". In this smart city scenario, we assumed to have a S4T

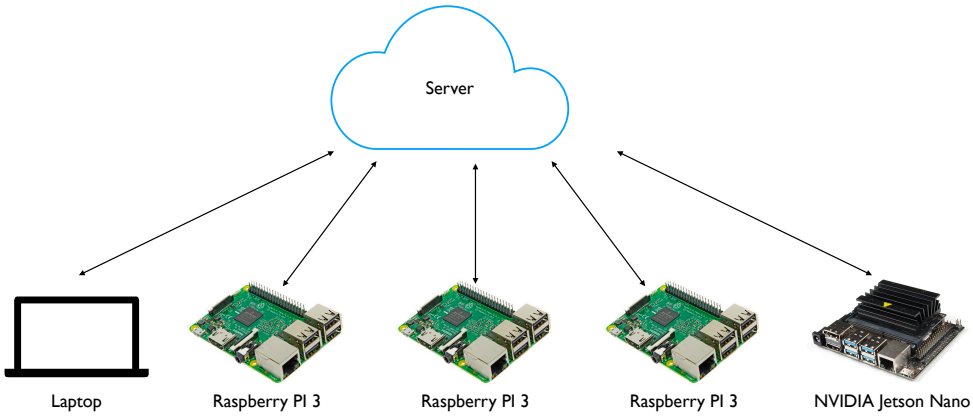


Figure 3.9: Federated learning training scenario consisting of five clients

deployment where the traffic cameras act as gateways that communicate with the Cloud using the framework functionalities we explained in Paragraph 3.2.1. Here, we consider a deep learning based smart city application that is able to classify which types of vehicles are crossing the road and uses this information for traffic analysis purposes. It is known that computer vision applications usually require a huge amount of data and training time to obtain a good level of accuracy. In such a context, this application could exploit a FL approach in order to enable a cooperation with the other traffic cameras with the aim to learn a shared model which synthesizes the information coming from each client preserving the privacy and abating the training time.

To measure the performance of the proposed FL architecture, we made a comparison with a centralized approach. The above described smart city scenario has been simulated using the Miovision traffic camera dataset (MIO-TCD) which contains the largest number of images for traffic analysis [51]. The dataset is composed by $786,702$ labelled images belonging to 11 classes i.e., articulated truck, background, bicycle, bus, car, motorcycle, non-motorized vehicle, pedestrian, pickup truck, single unit truck, and work van. In our application example, we selected a subset of four classes (i.e., articulated truck, bus, car, and motorcycle) representing the most popular vehicles in an urban traffic scene. This choice derives also from our intention to show the effectiveness of the FL approach even

Figure 3.10: Smart city scenario.

when working with clients that exhibit low power capabilities.

3.2.4 Comparison with a centralized approach

To prove the effectiveness of FL over a centralized approach, we conducted two sets of experiments: in the first one we considered a scenario where the number of labeled samples changes. In the second case, we changed the number of participants (i.e., clients) to the training process while keeping fixed the dataset. In such a context, our goal is to observe how FL reacts when varying the training conditions. Moreover, in both the experiments we used the same test set consisting of 800 samples (200 for each class) in order to measure the performance of the FL and centralized approaches.

We tackled problem of classifying which type of vehicle is crossing the road as a supervised deep learning approach using a CNN whose structure is showed

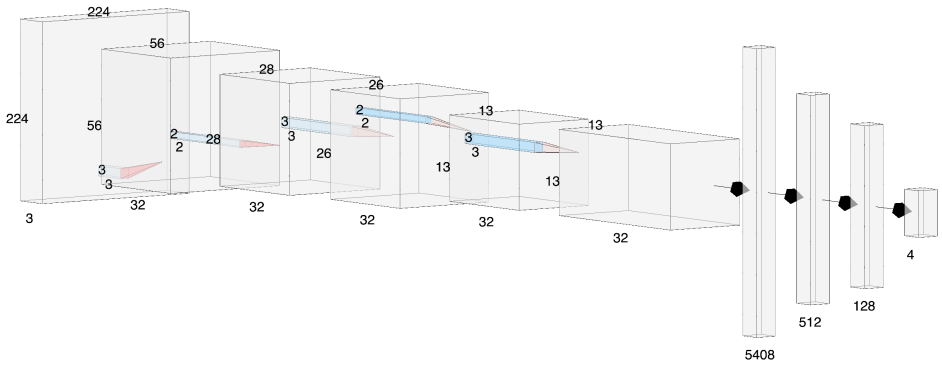


Figure 3.11: CNN architecture.

is Figure 3.11. Table 3.3 depicts the hyperparameters setting we adopted in our model. The CNN is composed by 7 hidden layers: the first is a convolution layer with 32 filters, a *kernel size* set to 3, and strides fixed to 4. The second one is a max-pooling layer with a pool size of 2. In the third and fourth layers we used again a combination of a convolution and max-pooling, with the only difference that we set the convolution strides to 1. Excluding the flatten layer (which is not a real hidden layer), the rest of the CNN consists of 3 fully connected layers with 512, 128, and 4 neurons which are responsible to perform the actual vehicle image classification. For each layer we used the ReLU activation function representing a sort of "standard" for these model architectures. Moreover, to avoid overfitting, we adopted a *dropout* technique which consists in "switching off" at each training iteration a random number of neurons in a layer (depending on the dropout rate) by cutting their connections. As a result, of this operation during the backpropagation process the weights of these neurons are not updated thus avoiding an overall network over training that could cause the model overfitting. Specifically, we set a drop rate of 0.2 and 0.5 for the first two fully connected layers leaving untouched the last one. Finally, we used *Adam* optimizer with a *learning rate* of 0.001 and set the maximum number of training epochs to 100.

Figure 3.12 depicts the cost function J of the federated model when varying the number of samples in the dataset and computed as the mean of the losses of

each of the five clients local models:

$$J = -\frac{1}{N_c} \cdot \sum_{c=1}^{N_c} \sum_{i=1}^{M} \sum_{j=1}^{C} y_{i,j} \cdot log(\hat{y}_{i,j}), \tag{3.3}$$

where M is the number of training samples, C is the number of classes, $y_{i,j}$ is the ground truth, and $\hat{y}_{i,j}$ is the value predicted by the CNN model.

From Figure 3.12 we can observe that the FL approach allows to successfully train a machine learning model. In fact, for all the number of samples the loss function has a decreasing trend that reaches a lower value as long as the dataset size increases. From the plot we can also notice that the number of training epochs (set to 100) stops to 25. This is due to an early stopping technique that we use to stop in advance the training if the model is not able to reduce the cost function for a consecutive number of training epochs, thus avoiding again the CNN over training.

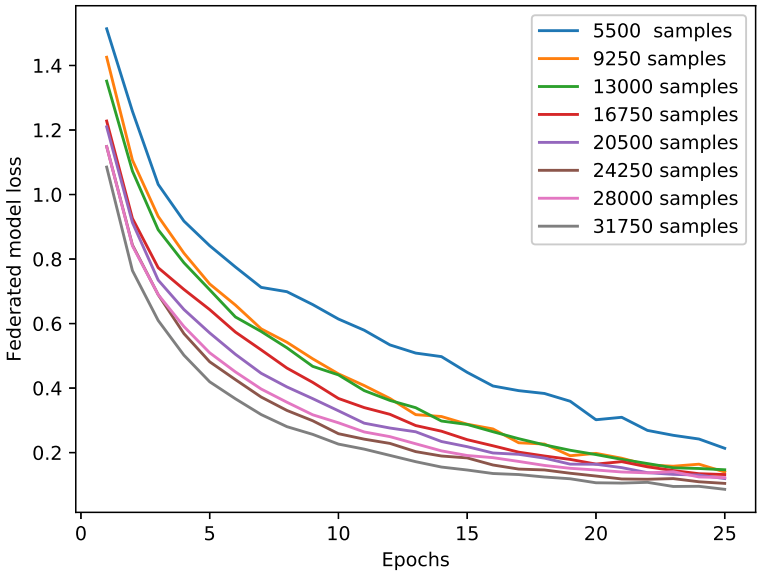


Figure 3.12: Loss of the federated model loss when varying the number of samples in the dataset.

Figure 3.13 shows a comparison between a FL and a centralized approach where we considered a deployment with the five clients we introduced in the

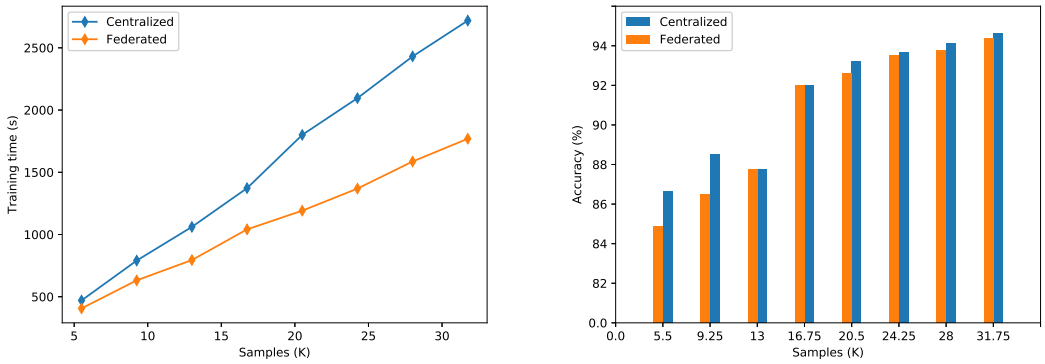
Table 3.3: CNN parameters.

CNN model architecture			
No. of layers	7	**Optimizer**	*Adam*
Activation function	*ReLU*	**Learning rate**	0.001
Pooling method	*Max-pooling*	**Dropout rate**	[0.2, 0.5]
Kernel size	3	**Max training epochs**	100

previous paragraph. In particular, Figure 3.13a depicts a comparison in terms of training time when varying the number of samples such that the numbers reported into the x-axis have to be considered as the sum of all the clients training samples. As one could expect, both the approaches exhibit an almost linear time increment as the number of samples increases. Such a trend depends on the number of the clients as well as on the hardware involved during the FL process. In this specific case, the majority of clients participating to the training process consist in three Raspberries whose hardware is less powerful than the one available on the laptop and the NVIDIA Jetson Nano, thus resulting in a lower time reduction. However, the FL approach is able to exploit the workload distribution among the participants, significantly reducing the training time. Moreover, taking a closer look to the plot we can notice that the time inference between the two approaches tends to increase as the dataset dimension becomes larger, thus suggesting that FL becomes even more effective.

Figure 3.13b presents a comparison in terms of accuracy when varying the number of training samples. In general, as long as the samples increase, the accuracy increments accordingly in both the approaches. Considering the largest dataset, the federated model reached an overall accuracy of 94.375% which is perfectly comparable with the one reached by the centralized approach of 94.625%. In some cases (i.e., $13K$ and $16.75K$ samples) both the techniques reached the exact same values, thus proving the effectiveness of FL in reducing the training time while maintaining comparable performance.

Figure 3.14 shows a comparison between the two approaches, this time varying



(a) Time (b) Accuracy

Figure 3.13: Time and accuracy comparison between a centralized approach (blue) and a federated one (orange) when varying the number of samples in the dataset.

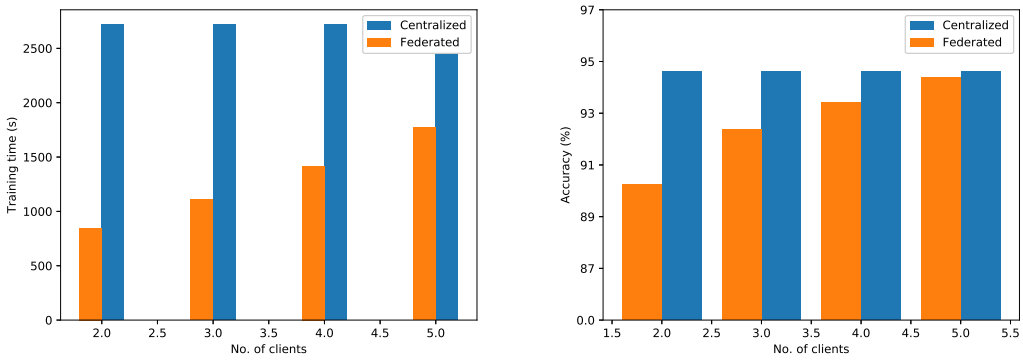
the clients that participate to the training process and keeping fixed the number of samples in the dataset. For this experiment, we considered a dataset with $31.75K$ samples for the centralized approach, while for the FL we gradually increased the dataset dimensions by $6.35K$ such that for five clients the number of samples coincides with the centralized (i.e., $5 \cdot 6.25K = 31.75K$). In Figure 3.14a, we report a comparison in terms of time. The plot shows that the training time increases accordingly with the number of participants, but always remaining below the training time required by the centralized approach. Such a result should not be misleading, in fact, when working with distributed approaches, we have the advantage to split the workload among the clients, however they also require an overhead due to the communication and coordination with the server. Moreover, in a FL scheme the larger is the number of clients, the higher is the time required to perform the model weights aggregation.

Another aspect that should be taken into account is related to the client heterogeneity. In particular, when working with FL approaches, a very challenging problem is the bottleneck caused by the slowest client performing the training [52]. Because of the centralized nature of the aggregation process, it requires the models of *every* client participating to the training; as a result of this condition,

the FL performance are affected by the hardware of each client. A possible solution to mitigate this effect could involve the use of timeout mechanisms during the training in order cut the connections with those clients that would cause the inevitable bottleneck. Other promising approaches propose the use of quantization or compression techniques to reduce the model complexity of a machine learning model, thus making their training process suitable, even for those clients with hardware constraints while reducing also the bottleneck time [53].

Figure 3.14b depicts a comparison between the two approaches in terms of accuracy varying also in this case the number of clients. In general, the accuracy raises accordingly with the number of participants and this is due because the increment of FL participants is equivalent to have a "virtually" larger training dataset. More precisely, even if the clients do not share their local data with the others, the trained models are "affected" by the data. In this sense, when the Cloud performs the aggregation, the new shared model reflects the contribution of each participant and their local data. As result of this condition, when the number of participants to the training increases, the shared model improves its generalization capabilities, thus increasing the accuracy performance on the test set. In particular, the accuracy starts with an initial value of 90.25 with two clients, and reaches a final value of 94.375% with five which results also to be the closest value to the one reached by the centralized approach.

The experiments put in evidence in which conditions the use of a FL can be beneficial outperforming a centralized one. According to the obtained results (showed in Figures 3.13 and 3.14), FL is not very effective when working with a low number of training samples. However, when working with large datasets, the use of a FL approach becomes more convenient since it allows to strongly reduce the training time while maintaining comparable accuracy performance and better preserving the data privacy. Of course, the use of more clients introduces an overhead due to the coordination with the server and to the weights aggregation. Nevertheless, the training time in FL remains always below the centralized one,



(a) Time (b) Accuracy

Figure 3.14: Time and accuracy comparison between a centralized approach (blue) and a federated one (orange) when varying the number of clients.

thus making the use of this distributed approach a promising solution for a new type of deep learning applications based on the cooperation to perform a common task.

Chapter 4

Smart Industry

In smart industry, aspects related to prognostics and diagnostics are gaining a lot of interest in the recent period. Deliver an effective maintenance schedule became a crucial problem especially in those sectors where the fault of a component can have catastrophic consequences. However, to do that there is the need to develop new techniques that should not only be able to warn the operator, but to timely prevent the occurrence of fault or an anomaly. In this chapter, we tackle this problem analyzing the possible solutions to address it and the related challenges [54]–[56].

4.1 Remaining useful life estimation using Long Short Term Memories

The estimation of equipment conditions gained a lot of interest in the recent years. If we consider an industrial system consisting of a set of components that cooperate to fulfill one or more tasks, it is easy to understand that the "health" of such a system is strictly related to the working conditions of its components. Because the failure of a component can cause the total unavailability of a system and in some contexts produce several problems which may include the loss of human lives in the worst scenario, it is evident the necessity to perform

a continuous monitoring to avoid such a condition.

Today, most of the systems available in the market monitor the data coming from a set of senors (e.g., temperature, noise, vibration, etc.) and send alarms only when one or more core components have a fault or do not work properly. In this sense, the main problem of these systems is their limited ability to warn the user when it is too late, instead of *prevent* the occurrence of a fault.

In smart industries, predictive maintenance is one of the most used approaches to address the above mentioned problem. Such a technique allows to estimate the conditions of a system component providing indexes that can be used to predict the maintenance *before* the occurrence of a failure [57]; by doing so, it is possible to prevent unplanned maintenances which would cause long offline periods with a consequent loss of time and money. Through the use of sensors mounted on the system, it is possible to produce a real time monitoring to be compared with the standard working condition described in the datasheets of each component.

In such a context, where the number of measurements coming from an industrial system can be quite large emerges the necessity to define an index to merge all the information into a single one synthesising the "health" state. The Remaining Useful Life (RUL) is one of these indices and defines the remaining useful period during which the system will properly work [58]. RUL prediction can be done in many different ways and there is no a predominant technique. However, when working with complex systems, the use of machine learning approaches can be very useful to find possible hidden correlations between the sensors in order to create a model for an accurate RUL estimation.

In this work, we tackled the RUL prediction problem via the use of deep learning techniques. Specifically, given a set of sensor time series, we realized a RNN model that could be executed on a Cloud/Edge platform to perform a real time monitoring of an industrial system with the goal of estimating its remaining "lifetime", and deliver on-time maintenances. Moreover, we provide an analysis showing how the hyperparameters setup impacts the performance of a machine

learning model, with the goal of defining a methodology that can help during the design process of a neural network.

4.1.1 Predictive maintenance approach

The main goal of predictive maintenance is to make an accurate estimation of the RUL of a system. Unfortunately, the RUL is not known a priori and it is strictly related to the system life history [59]; from a statistical point of view we can define it as $f(x_t|Y_t)$ where x_t is a random variable associated to the RUL at time t and Y_t represents the history of the system up to time t [59].

The literature proposes several approaches that aim to extract "fault patterns" which can be used to deliver optimal maintenance strategies and reduce long offline periods [60]. For example authors in [61] propose a comparison between SVMs, DT, RF, and many others algorithms to show which technique is the best in predicting the RUL on a dataset of turbofan engines. In [60] is presented an approach based on Long Short Term Memories (LSTMs) that is proven to outperform other techniques like Naive Bayesian regression. The technique presented in [62] is based on Auto Regressive Integrated Moving Average (ARIMA) forecasting on time series generated by a set of IoT sensors. However, the use of such a technique requires a careful manual tuning and it is not suitable to learn long time dependencies.

LSTM network

A LSTM network is a special neural network model belonging to the category of the RNNs which uses an efficient gradient based algorithm to keep bounded the error, thus avoiding its explosion or vanishing [13], [14]. Unlike the RNNs, LSTM networks are able to store long time dependencies between the inputs. In this sense, one of the main drawbacks of the RNNs is their ability to track only recent time dependencies because of the gradient exponential decay or explosion problems that can happen during the execution of the BPTT (see Chapter 1)

[13].

Figure 4.1 represents the internal structure of a LSTM cell. The cell has three gates, namely: the input gate, the output gate, and the forget gate which are ruled by the following equations:

$$f_t = \sigma(W_f * [h_{t-1}, x_t] + b_f), \tag{4.1}$$

$$i_t = \sigma(W_i * [h_{t-1}, x_t] + b_i), \tag{4.2}$$

$$\widetilde{C}_t = tanh(W_C * [h_{t-1}, x_t] + b_C), \tag{4.3}$$

$$C_t = f_t \circ C_{t-1} + i_t \circ \widetilde{C}_t, \tag{4.4}$$

$$o_t = \sigma(W_o * [h_{t-1}, x_t] + b_o), \tag{4.5}$$

$$h_t = o_t \circ tanh(C_t), \tag{4.6}$$

where $\circ$ is the symbol for the element wise product, f_t,i_t,o_t are the *forget gate*, the *input gate*, and the *output gate* respectively, and W_f, W_i, W_C, W_o are the weight matrices of the forget gate, input gate, cell state, and output gate.

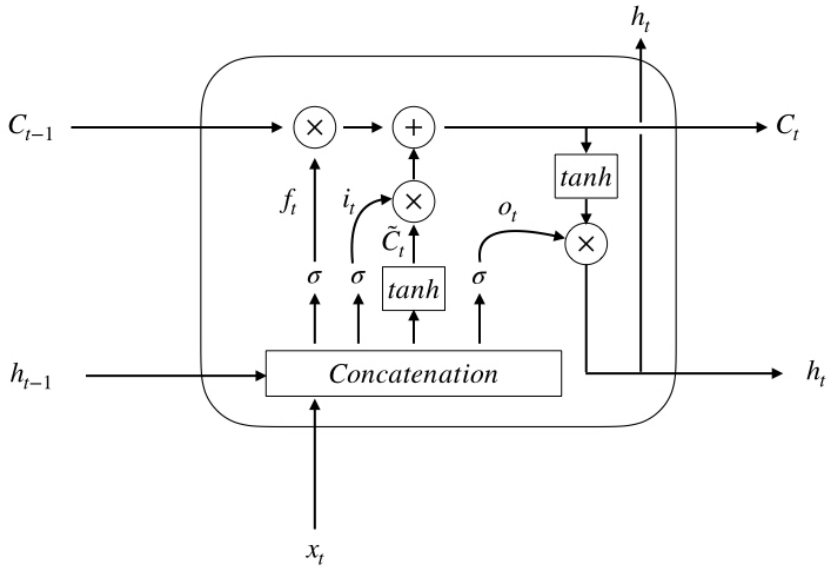


Figure 4.1: LSTM cell.

When the input flows inside the LSTM cell, the first step consists in establishing the information to keep through the eq. (4.1) which takes in input the data x_t at time t, together with the output h_{t-1} coming from the previous time instant $t-1$. In such a context, the input x_t is an array containing the data measurements coming from sensors. The results returned by the above equation is then passed to the Sigmoid activation function σ which returns a value between 0 and 1 for each element contained in the cell state C_{t-1}; here 0 indicates an element that can be forgotten, while 1 means that a value should be kept over time. In the second step, the input gate is used to decide which new information should be stored. Specifically, this is done via two sub steps: the first during which the new information is passed to the Sigmoid function (eq. (4.2) to establish which values to update, and the second one to compute the new cell state by means of eq. (4.3). In this last step, the LSTM adopts the hyperbolic tangent *tanh*, an activation function used for two reasons: *i.)* it normalizes the input between -1 and 1; *ii.)* it better distributes the gradients, thus allowing the cell state to be propagated for longer periods, thus addressing the vanishing and exploding gradient problems affecting the RNNs. After these sub steps the LSTM cell state is updated using eq. (4.4). In particular, the state is a combination of the new values received in the current time step, and the values coming from the "past". The very last step performed by the LSTM consists in generating the output through eq. (4.5), while eq. (4.6) is used to decide which part of the new LSTM should be passed in the next time step.

4.1.2 NASA dataset

We tackled the problem of predicting the RUL of a system given a set of sensors measurements as a supervised regression problem. Of course, the use of a supervised approach requires a labeled dataset containing the entire "history" of a system until its failure. However, the impossibility to reproduce such a complex scenario consisting of a large number of systems that run until their failure, we

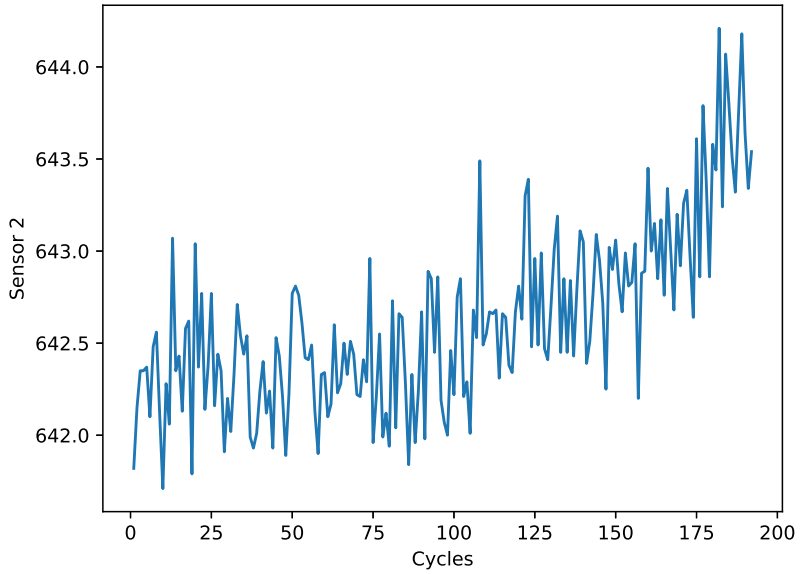
Table 4.1: NASA Dataset.

Turbofan Engine Degradation Simulation dataset							
ID	*Cycle*	*Setting* 1	*Setting* 2	*Setting* 3	*Sensor* 1	...	*Sensor* 21
1	1	0.0023	0.0003	100	518.67	...	23.3735
1	2	−0.0027	−0.0003	100	518.67	...	23.3916
...	...	...	...	...	...	...	...
1	31	−0.006	0.0004	100	518.67	...	23.3552
...	...	...	...	...	...	...	...
100	198	0.0013	0.0003	100	518.67	...	23.1855

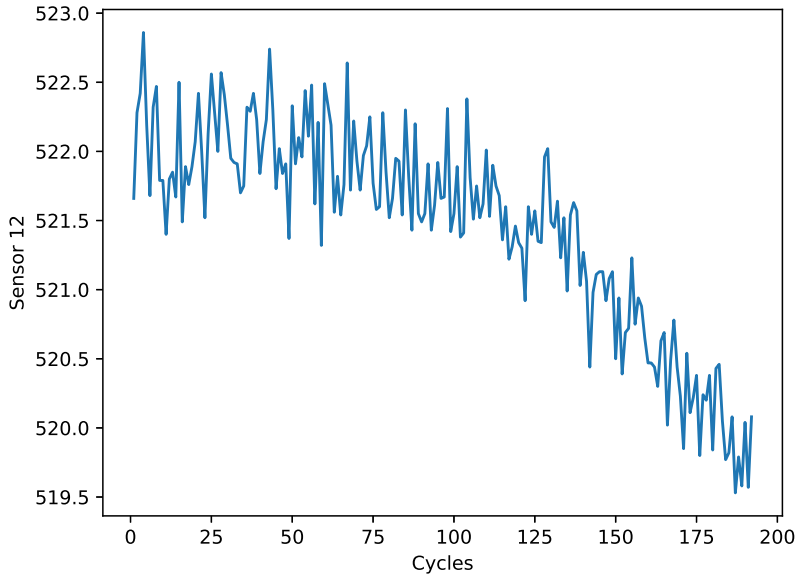
opted to use a dataset provided by NASA and generated using the C-MAPSS tool which simulates the degradation of a set of engines [63]. Over the years, this dataset has become a standard to benchmark predictive maintenance algorithms and consists of a multivariate time series where three settings variables together with a set of 21 sensors describe the operational conditions of the engines. Moreover, each dataset row is provided with *id* and *cycle* values which describe for how long each engine is running.

Figure 4.2 shows two typical "fault patterns" of an engine with respect to two sensors. By taking a closer look to both the plots, Figures 4.2a and 4.2b depict two evident trends that cause the engine failure. In this sense, our goal is to perform a timely detection of these patterns in order to deliver the maintenance before the occurrence of a fault.

Table 4.1 provides a view of the dataset organization where each row represents a working time cycle of an engine (identified by its ID) with the associated sensors data readings. NASA provides four different versions of this dataset (which differ for the engine operational setting) already split in train and test sets [57]. In particular, we selected the first one containing the run to failure history of 100 engines and consisting of $20,631$ and $13,096$ samples for the train and test sets respectively. The datasets come only with the test labels without the training ones, for this reason we manually computed them in order to provide the couples (input, label) necessary for the supervised training process. To



(a) Sensor 1



(b) Sensor 12

Figure 4.2: Sensors readings of engine 1 until its fault.

do that, since the dataset contains the entire history for each engine until the failure, we can exploit the information of the *Cycle* column to compute the RUL. Specifically, the RUL can be defined as the number of remaining working cycles

Cycle
1
2
3
4
5
…
98
99
100

RUL
99
98
97
96
95
…
2
1
0

Figure 4.3: RUL computation given the Cycle feature.

before the failure of a system, in this sense to compute it, we extracted the last value of the *Cycle* feature for each engine, and we created a RUL column vector whose $i-th$ element is obtained as follows:

$$RUL_{i^{th}} = last\_cycle - Cycle_{i^{th}}, \tag{4.7}$$

where $last\_cycle$ is the last value taken from the *Cycle* feature and $Cycle_{i^{th}}$ is its value in the $i-th$ row. For a better understanding, in Figure 4.3 we present an example of a generic engine where we assumed that the failure occurs after 100 cycles (i.e., $last\_cycle = 100$).

A closer look to Table 4.1 shows that the features lie in very different ranges, when this happens a good practice is to normalize the dataset. Not doing so in fact would result in giving more importance to those features with larger values, thus causing wrong assumptions that can cause the increment of the model error. To avoid such a condition, we normalized the dataset using the *min max normalization* with values between -1 and 1.

After the normalization step, we performed a manual feature extraction pro-

cess in order to remove those not informative features. After an extensive analysis we decided to keep all the features except for the *Engine ID* column since it is just an identifier not informative for the RUL prediction. At the end of this process, the dataset was ready to be passed to the LSTM network for the training.

4.1.3 LSTM hyperparameters tuning

When working with LSTMs the hyperparameters tuning process is in general more complicated than the one used for "traditional" neural network and this is due in part to its internal structure we already explained. From our experience, we noticed that the obtained results can be very different when varying the model parameters, in this sense, our idea is to perform an analysis for different hyperparameters configurations to measure the model performance in terms of the Root Mean Squared Error (RMSE) defined as follows:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(\widehat{y_i} - y_i)^2}, \tag{4.8}$$

where $\widehat{y_i}$ and y_i are respectively the value predicted by the model and the ground truth provided by the test set of the dataset of the generic $i-th$ input sample, and n is the number of samples of the test set. The motivation behind this analysis comes also from our intention to provide useful insights about the hyperparameters setting that can be used as a starting point to design a model with a good level of performance. Of course, because the hyperparameters space is too large, it would be impossible to analyze all of them and for this reason we focused on subset.

LSTMs are RNNs this means that we can "unfold" them and see the time steps they store internally. Figure 4.4 depicts an unfolded LSTM taking as input a set of measurements x at different time steps (i.e., a sequence), except for $t = 0$ where the state is initialized to zero. Let us define the three hyperparameters we selected for our analysis i.e., the number of layers l, the window size w and

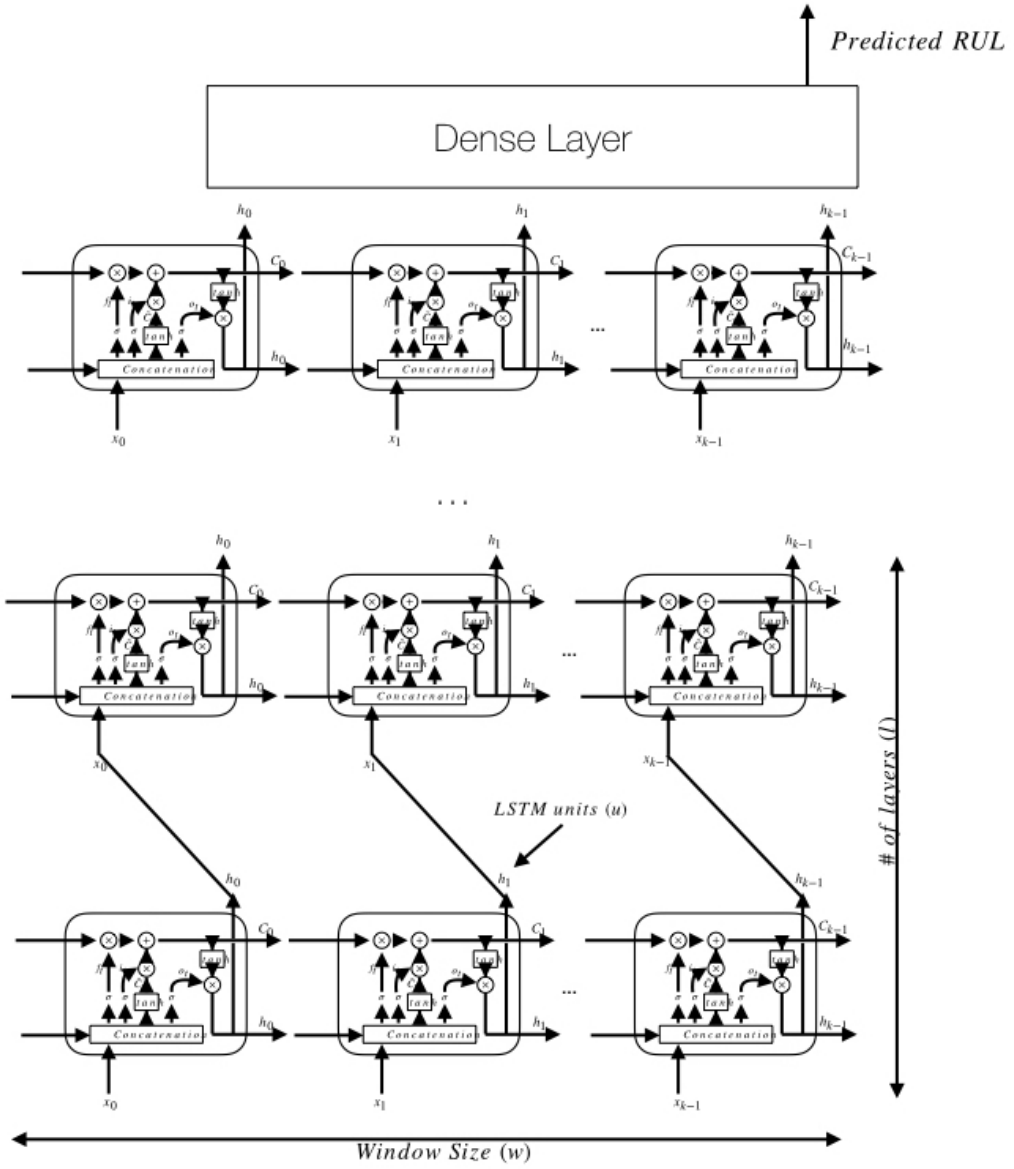


Figure 4.4: Model architecture.

the number of internal units u, where the first one defines the "depth" of the model such that a deeper model is able to extract more complex features. The second one defines the number of time steps kept in memory by the LSTM, so that the larger is the window the longer is the considered time dependency (i.e., the LSTM will look more into the past). Finally, the number of internal units can be considered as the number of neurons in a hidden layer, in this sense they define the output dimension for the LSTM and according to their number they affect

the learning power of the layer (i.e., a larger number of units increases the number of parameters that can be learned by the model). When a LSTM is composed by more layers connected in cascade, if we unfold both of them (as shown in Figure 4.4), for each time step t, the $l-th+1$ layer receives as input the hidden vector (i.e, the output) computed in the previous layer through eqs. (4.5) and (4.6) where the weight matrix W_o dimensions depend on the "width" (i.e., window size) of the unfolded LSTM and on the number of its internal units. Finally, the model topology terminates with a Dense (i.e., a fully connected) layer with one neuron that generates the predicted RUL given a sequence of sensor measurements.

To find the best tuning, we chose a set of values for each of the selected hyperparameters. Then, we applied a *grid search* process where we iteratively trained several models keeping fixed one hyperparameter and changing the other two according to the value contained in the above mentioned sets. By doing this procedure for each hyperparameter we ended up obtaining a number of trained models equal to the cardinality of the Carthesian product of the three sets. The result of this procedure allowed us to perform an in dept analysis of each hyperparameter and to better understand how they affect the model performance.

4.1.4 Hyperparameters analysis and LSTM results

Table 4.2 depicts the parameters configuration we created to train the LSTM models. Specifically, we chose seven values for each hyperparameter and during the training we applied the methodology explained in the previous paragraph. The total number of models obtained is computed as: $\|w\| \cdot \|u\| \cdot \|l\|$ which in this case is equal to 343. To train such a huge amount of models, we installed Keras on a cluster of machines available in our department and launched the training processes in parallel with different configurations according to Table 4.2. After this procedure, we created a set of 3D plots showing the RMSE trend by fixing a hyperparameter and varying the other two. Because this procedure generated a huge number of plots, for sake of simplicity, we show only those ones exhibiting

Table 4.2: LSTM network parameters.

LSTM parameters		
Window size (w)	*No. of units* (u)	*No. of layers* (l)
30	10	1
40	20	2
50	30	3
60	40	4
70	50	5
80	60	6
90	70	7
Learning rate		0.001
Training epochs		500
Dropout		$[0.2, 0.5]$

a noticeable trend and therefore more informative.

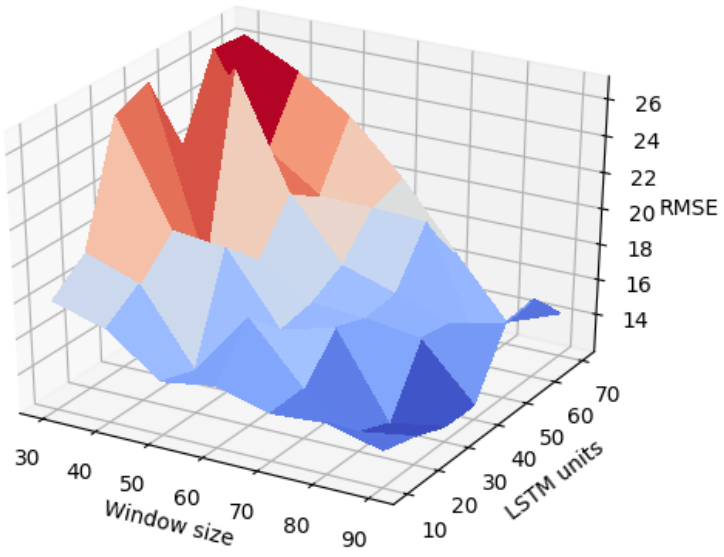


Figure 4.5: RMSE plot obtained by fixing the number of layer to 2.

Figure 4.5 shows the RMSE trend obtained by fixing the number of layers to two. In general, we can notice a descending trend as the window size increases, and this is valid for each value of LSTM units that has been selected. Such a

result is due to the fact that a larger window size enables the LSTM to store longer time steps sequences and learn more complex time correlations with a consequent reduction of the error.

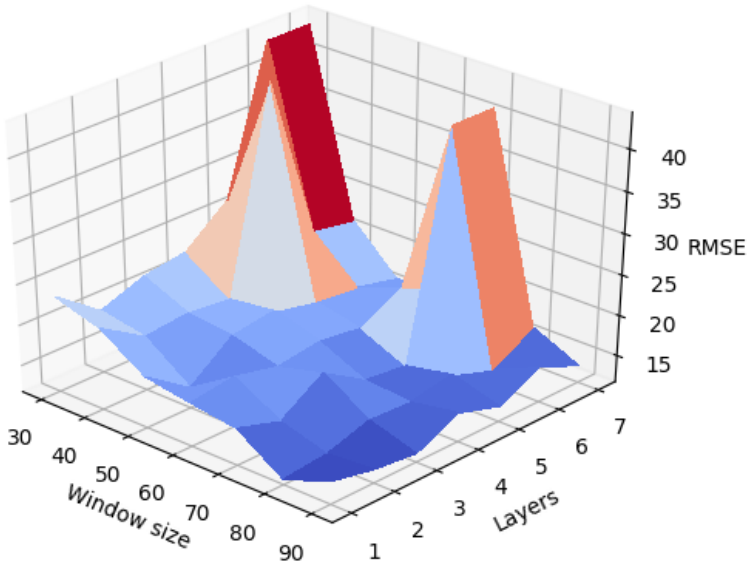


Figure 4.6: RMSE plot obtained by fixing the number of units to 20.

With reference to Figure 4.6, it depicts how RMSE varies when the window size and the LSTM layers change while keeping fixed the number of units to 20. Except for some peaks, which are evident case of model overfitting since they correspond to model topologies with a large number of layers (i.e., 5, 6,and 7), the overall trend is similar to the one showed in the previous figure. Likewise the previous case, the lowest value is obtained using the largest window size (i.e., 90) with a number of layers set to 3.

From Figure 4.7, we can notice that the RMSE reaches the lowest value by setting the layers between 2 and 3 (as showed in Figure 4.6) giving us a hint of the possible number of layers to use. Merging all the useful information derived from these plots, our goal was to design a new model to further reduce the error.

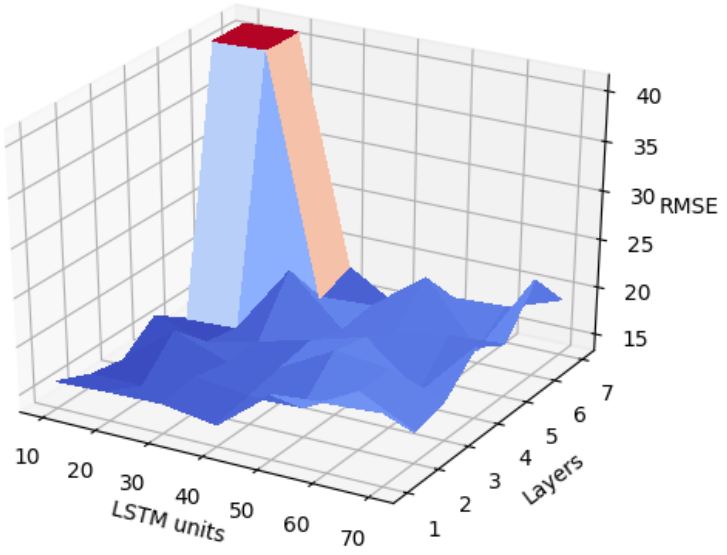


Figure 4.7: RMSE plot obtained by fixing the window size to 80.

In particular, from this analysis we learned that a larger window size reduces the error which reaches the lowest value by setting the layers between 2 and 3.

Table 4.3 represents the LSTM configuration we designed by merging all the information derived from the analysis. Specifically, we set the window size to 90 and tried several configurations using 2 or 3 layers, however the model with 3 layers has always outperformed the other one. With respect to the number of units, we decided to change it for each layer instead of keeping it fixed. In particular, we adopted 40 units for the first layer and 15 for the second and third. This model topology where the number of units decreases as the number of layers increases is a very common practice in deep learning. We used the *RMSProp* optimizer with a *learning rate* of 0.001 (which is a standard value for this optimizer) and trained the model for 500 epochs. Finally, to prevent overfitting, we adopted a dropout technique. In [64], it has been proven that the use of a dropout rate close to 0.1 or 0.2 for the first layer, and 0.5 for the others

Table 4.3: LSTM network configuration.

LSTM model architecture	
Window size	90
No. of layers	3
No. of units	$[40, 15, 15]$
Optimizer	*RMSProp*
Learning rate	0.001
Training epochs	500
Dropout	$[0.2, 0.5, 0.5]$
RMSE	11.42
Error%	0.12

is a valid choice to reduce the possibility of a model to overfit.

In such a context, let us introduce a new error index that express the mean percentage error of the model:

$$Error\% = \frac{1}{n}\sum_{i=0}^{n}\frac{|\widehat{y_i} - y_i|}{y_i}, \tag{4.9}$$

where $\widehat{y_i}$, y_i, and n are the values predicted by the model, the ground truth, and the number of samples respectively (as already defined in eq. (4.8)). Unlike the RMSE which measures the "distance" between the model prediction and the ground truth, the percentage error evaluates the impact of the error made by the model on each sample. Both these indices has been used to test the performance of the proposed LSTM model and their values has been reported in Table 4.3. In particular, with a RMSE of 11.42 and a percentage error of 0.12, this model resulted to be best among the 343 we created during the hyperparameters tuning.

Figure 4.8 depicts a comparison between the proposed LSTM model and the ground truth coming from the test set where on the x-axis we report the *ID* of the engine, while in the y-axis we report the real and predicted values of the RUL. The plots shows that for the most part of the engines, our model is able to correctly predict the RUL, thus demonstrating the success of the training process.

For a better understanding we report in Figure 4.9 another plot where in this

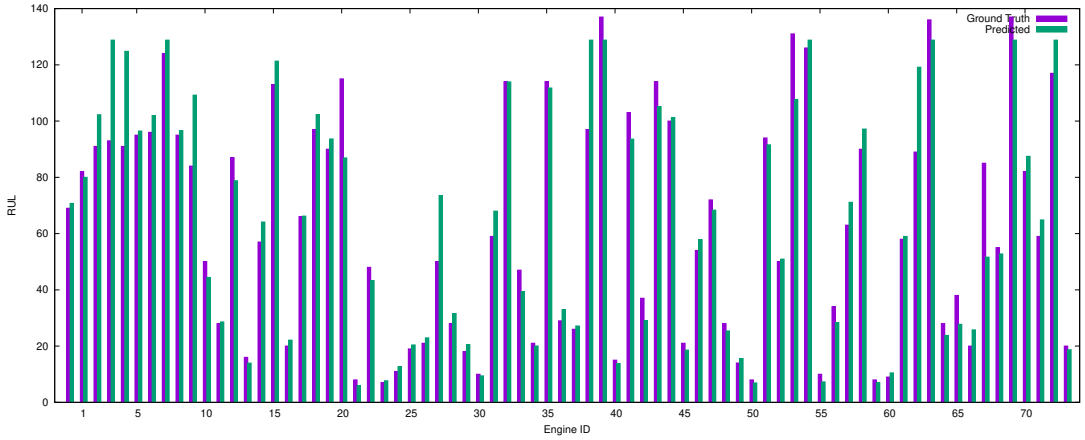


Figure 4.8: Plot which shows the RUL predicted by the LSTM and the correspondent ground truth.

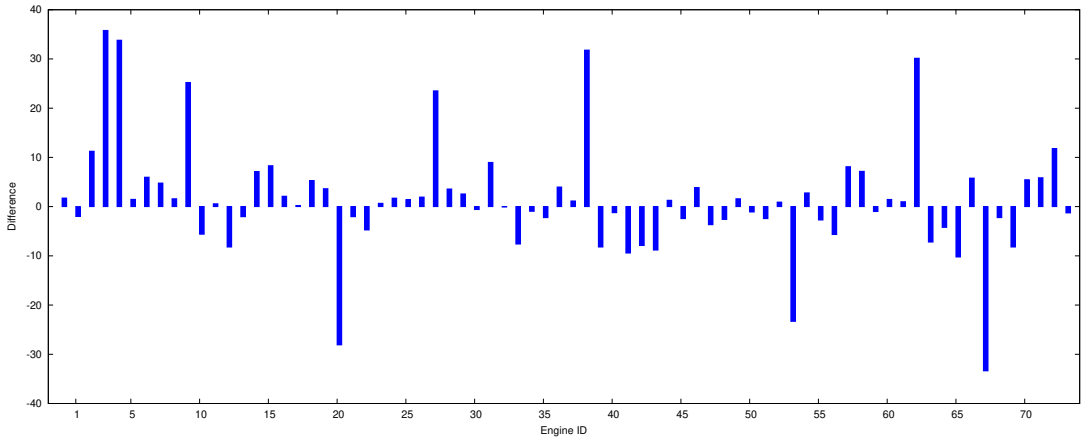


Figure 4.9: Plot which shows the difference between the predicted values and the real ones.

case the y-axis depicts the difference between the value predicted by the LSTM model and the ground truth computed as follows:

$$d_i = \widehat{y_i} - y_i, \tag{4.10}$$

where d_i is the difference for the $i-th$ sample, while $\widehat{y_i}$ and y_i have been already defined in eqs. (4.8) and (4.9). Here, a value higher than zero indicates an optimistic prediction made by the model, while a value lower than zero a pessimistic one. Nevertheless, most of the values exhibit a difference very close to zero, and only a small part of them presents a large difference higher than zero meaning

that the model most of the times makes a correct or a pessimistic prediction that in both cases do not cause the complete system failure.

As we can observe, in both the two plots given a test set of 100 engines, we were able to test only 74 of them. This is due to the selection of the window size of 90 time steps that obliges to select only those engines that have a longer life "history".

4.1.5 Comparison with other approaches

To further demonstrate the effectiveness of the proposed LSTM approach, we compared it with other two machine/deep learning techniques, namely: a DNN and a SVMs model. With respect to the first one, we designed a DNN with five hidden layers implementing also in this case a topology where the number of neurons decreases as the network goes deep. We used 25 neurons for the first layer and 15 for the remaining ones. ReLU has been used as activation function for each layer. To avoid overfitting, we used a L2 regularization (as showed in eq. (2.6)) with a *regularization rate* of 0.01, and an early stopping technique to avoid an over training. Finally, we set 500 training epochs and *Adam* as optimizer with a *learning rate* of 0.001.

The SVMs has been tuned with a RBF kernel (which is one of the most used), a penalty term C (used for the regularization) set to 1.0 and the *gamma* factor (affecting the spread of the decision boundary) to "scale", a special term provided by the *Scikit-learn* framework that allows to adapt this parameter according to the following equation:

$$gamma = \frac{1}{var(X) \cdot \|features\|}, \tag{4.11}$$

where $var(X)$ is the variance computed from the entire training set and $\|features\|$ is the cardinality of the features. Finally we set the number of training epochs to "auto", meaning that the algorithm stops only when the solver finds a solution.

For a better view, we report the DNN and SVMs hyperparameters configurations in Tables 4.4 and 4.5.

Table 4.4: DNN network configuration.

DNN model architecture	
No. of layers	5
No. of units	[25, 15, 15, 15, 15]
Activation function	*ReLU*
Regularization rate	0.01
Training epochs	500
Optimizer	*Adam*
Learning rate	0.001
RMSE	41.77
Error%	1.43

Table 4.5: SVMs hypeperameters setup.

SVMs configuration	
Kernel	*RBF*
Penalty term (C)	1.0
gamma	*scale*
Maximum training epochs	*auto*
RMSE	51.54
Error%	2.18

If we observe Figure 4.10, in both the cases the proposed LSTM approach outperformed the DNN and SVMs which reached a RMSE of 41.77 and 515.54 respectively, and a percentage error of 1.43 and 2.18. Unlike LSTM networks, the DNN and SVMs analyze the samples one by one (i.e., they do not consider the time correlation between them), in this sense, these techniques fail in detecting those "fault patterns" that indicate the occurrence of a fault. On the other hand, LSTMs are able to store long time series that allow them to capture possible time correlations and to effectively predict the RUL. If compared with the works presented in [60] and in [61], our approach improves the RMSE by reducing it of about 35% and 54% respectively.

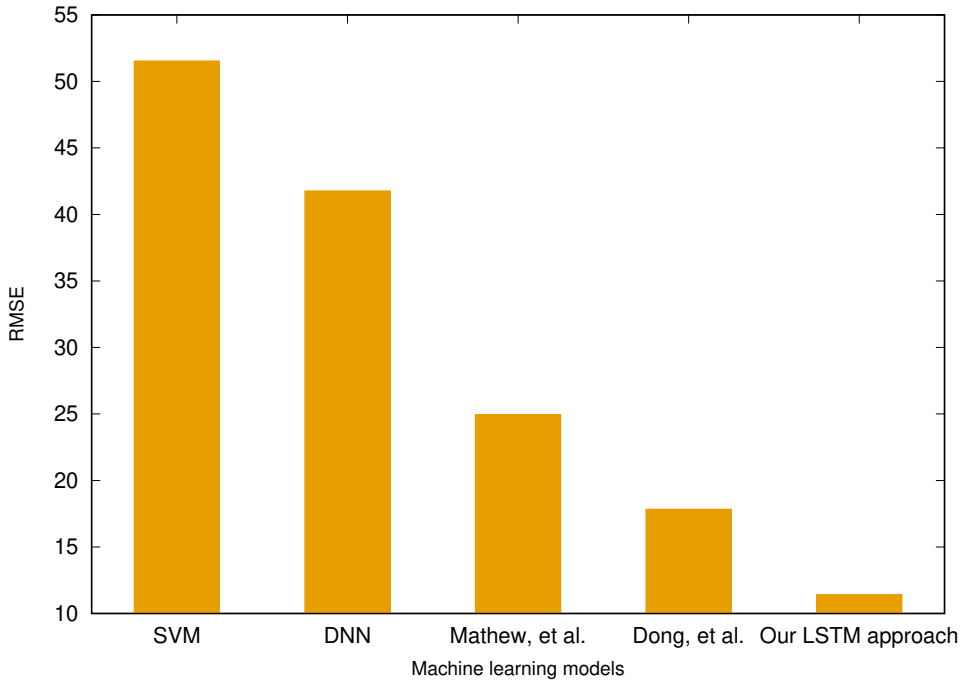


Figure 4.10: Plot which shows a comparison with other techniques.

The obtained results confirm that LSTM are a promising technique that can be applied to smart industries for predictive maintenance. However, unlike more "traditional" approaches (like feed forward DNN), they require a careful tuning in order to obtain good results.

4.2 Data collection framework for telemetry and anomaly detection of Industrial IoT Systems

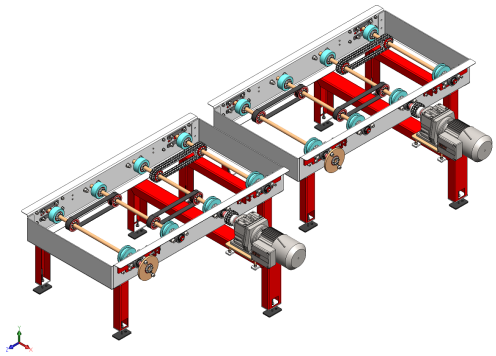
In the recent years, we observed a market growth of IoT applications in smart industry. The application of this technology in the industrial sector has significantly changed the way we interact with these system, leading to what we call Industrial Internet of Things (IIoT) and representing the core element of the new Industry 4.0 paradigm [65], [66]. In an industrial scenario consisting of large sets of sensors, Edge computing plays a key role providing fast processing capabilities that are able to meet the high frequency rate requirements of the industrial applications

[4]. However, the hardware constraints of these devices make them unsuitable for the execution of onerous tasks. In this sense, the Cloud acts as a central "brain" in the smart industry for the coordination of the edge devices, also exposing storage capabilities (useful to keep track of the system history) and computing services for the execution of complex operations (e.g., training of machine/deep learning algorithms, in depth data analysis, etc.) [67]. In such a context, a major challenge is represented by sensors heterogeneity due to the adoption of different communication protocols which require the implementation of frameworks for an efficient system monitoring. However, as we already said in paragraph 4.1, the simple monitoring is not sufficient to maintain an industrial system. To this aim, AI proposes different solutions to address this problem according to the application context [68]. In a scenario where a *run to failure* dataset can not be created (making impossible the use of predictive maintenance techniques), anomaly detection is a viable approach that allows to detect the occurrence of conditions or events that should not be considered normal (e.g., a fault) potentially dangerous for an industrial plant. Leveraging the above mentioned technologies (i.e., Edge, Cloud, and AI), this thesis proposes an IIoT data collection framework (based on S4T) that enables the telemetry and anomaly detection of industrial plants. In such a context, our major contribution consists in creating a synergy among all the components of the proposed architecture demonstrating its feasibility in a real scale replica industrial plant.

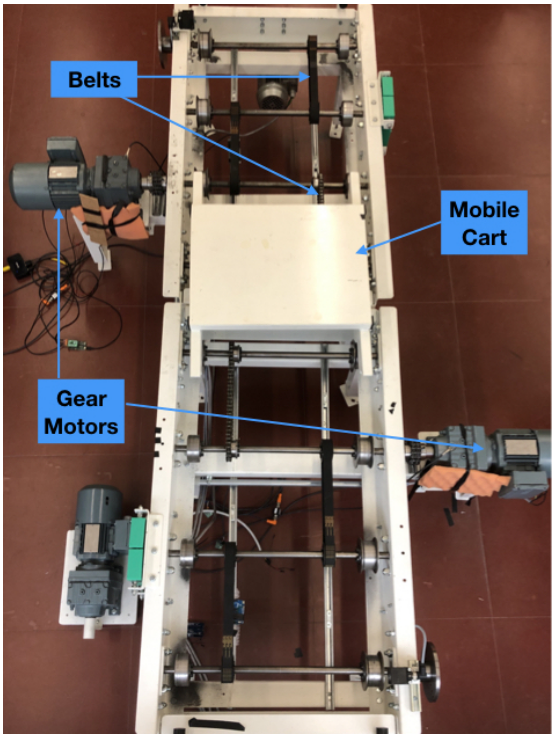
4.2.1 Industrial IoT testbed

In this paragraph we describe in detail, the industrial testbed we realized to test the proposed IIoT framework.

Figure 4.11 shows a part of the assembly plant used for the transportation of car pieces in automobile plants (see Figure 4.11a) equipped with two gear motors, and six belts (four made of rubber, and two made of steel) to transport the mobile cart as showed in Figure 4.11b. An important feature of the system is



(a) 3D CAD image of the plant



(b) Picture of the plant

Figure 4.11: Scale replica of an industrial plant used as testbed.

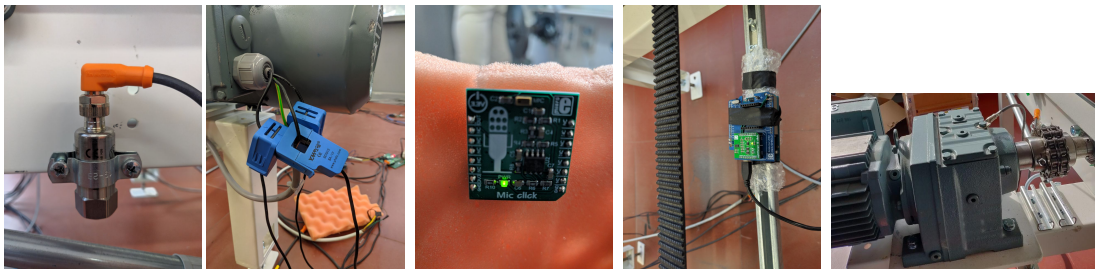
the possibility to inject different types of faults (mainly mechanical) such as: the introduction of external vibration, change the belt tension, increase the friction of the gears, and emulate the break of the cart proximity switch. Such a feature has been fundamental for our purposes, since it allowed us to have a complete understanding of the system dynamics even when subject to mechanical stress. The system was provided to us without sensors, so our first task has been to

decide how to instrument it by choosing the type of sensors to use to monitor it from the electrical and mechanical point of view.

With respect to the electrical part, we used a SCT-013 current sensor (see Figure 4.12b) which is able to measure the current in a non invasive way. This sensor works as a transformer: it generates a voltage which is proportional to the current flowing in the internal capacitor. This current (known as "primary current") generates a magnetic flux that produces in turn a "secondary current". The relationship that binds the output voltage of the sensor with the two currents is expressed in the following equation:

$$\frac{i_{secondary}}{i_{primary}} = \frac{V_{primary}}{V_{secondary}}, \tag{4.12}$$

where the voltage is an analog signal converted in a digital one via an Analog to Digital Converter (ADC).



(a) Vibration sensor (b) Current sensor (c) Noise sensor (d) Proximity sensor (e) Temperature sensor

Figure 4.12: IIoT testbed instrumentation.

For the mechanical part, we adopted the VTV-122 vibration sensor, a Micro Electro Mechanical System (MEMS) produced by IFM electronics. The sensor is powered by a direct voltage which can range between 9 and 32 V and generates in output an analog signal which respects the 4-20 mA standard.

For the audio analysis part, we used the SPQ0410HR5H-B sensor produced my Mikroe (see Figure 4.12c) to measure the noise generated by the plant by putting it nearby each one of the motors. Such a technique is broadly used in the industrial monitoring and allows to extract very useful information about the

state of the system. The sensor is an omni-directional microphone that generates as output an analog signal and communicates using Serial Peripheral Interface (SPI), UART and I2C.

To measure the belts loss of tension (a typical condition that happens when a mechanical fault occurs), we used the VL53L0X proximity sensor (or distance sensor) (showed in Figure 4.12d) produced by STMicroelectronics. At the time of writing, it still represents the state of the art since it is the smallest Time of Flight (ToF) sensor ever created. The sensor can measure distances up to two meters, to do that, it emits a ray of photons which is reflected when an obstacle is encountered, then it measures the distance (returned as an analog signal via I2C) according to the following equation:

$$distance = \frac{photon\ travel\ time}{2 \cdot c}, \tag{4.13}$$

where c is the speed of light (approximately $3 \cdot 10^8\ m/s$).

Finally to measure the motors temperature we used a sensor named TS2229 (see Figure 4.12e). In such a context, the main problem was given by the impossibility to access the internal parts of the motors. For this reason, we opted for the use of a non invasive sensor suitable to measure the temperature of solid surfaces that outputs an analog signal using the 4-20 mA protocol.

Even in such a small scenario made by a set a of few sensors, it is more than evident the communication protocols heterogeneity which is one of the major challenges in the industrial sector [69]. Evidently, such a condition becomes even more complex in a real industrial plant [70]. This is the principal motivation that pushed us to build an IIoT architecture capable to add an abstraction layer between the embedded boards interacting with sensors and the data acquisition system. In particular, to do that we exploited the S4T framework functionalities (already discussed in Chapter 3) that we adapted to make it suitable for an industrial scenario and building on top of it a telemetry application and an anomaly detection algorithm.

4.2.2 Industrial IoT architecture

The realization of an effective IIoT architecture is not easy and can be done in several ways. In [66], [71] authors describe the challenges and future directions in the design and implementation of IIoT frameworks. In [72] is presented a performance evaluation of two architectures for IIoT systems, namely: full-Cloud and Edge-Cloud on top of which is executed an anomaly detection task based on LSTM networks. In this work, the testbed consists in a simple Power Line Communication (PLC) which interacts with a computer using only the Message Queue Telemetry Transport (MQTT) protocol. However, as we saw in the previous paragraph, in a real industrial scenario is adopted a wide variety of communication protocols, in this sense the proposed architecture is designed to work with different protocols which make it suitable to be used in an industrial scenario. Authors in [73] implemented an industrial gateway acting as a bridge between the sensors and the Cloud; also in this case the system adopts a single communication protocol (i.e., I2C) to gather the data from sensors. Moreover, the system is not equipped with mechanisms to perform an anomaly detection and requires a constant monitoring by a human operator. The IIoT framework presented in [74] is used to monitor chemical emissions in an industrial plant. It consists of three layers: a ZigBee nodes layer, a middle layer of Long Range (LoRa) nodes, and another layer providing the connection to the Internet. Unfortunately, the use of only wireless communication protocols is discouraged in an industrial setting where the interference due to the other devices and the plant itself could result in significant data loss and distortion.

The proposed IIoT architecture for data collection and telemetry of an industrial plant is showed in Figure 4.13 and consists of two parts, namely: hardware a software.

With respect to the hardware, Figure 4.14 depicts the boards we adopted in our industrial framework that have been designed by our university spin-off

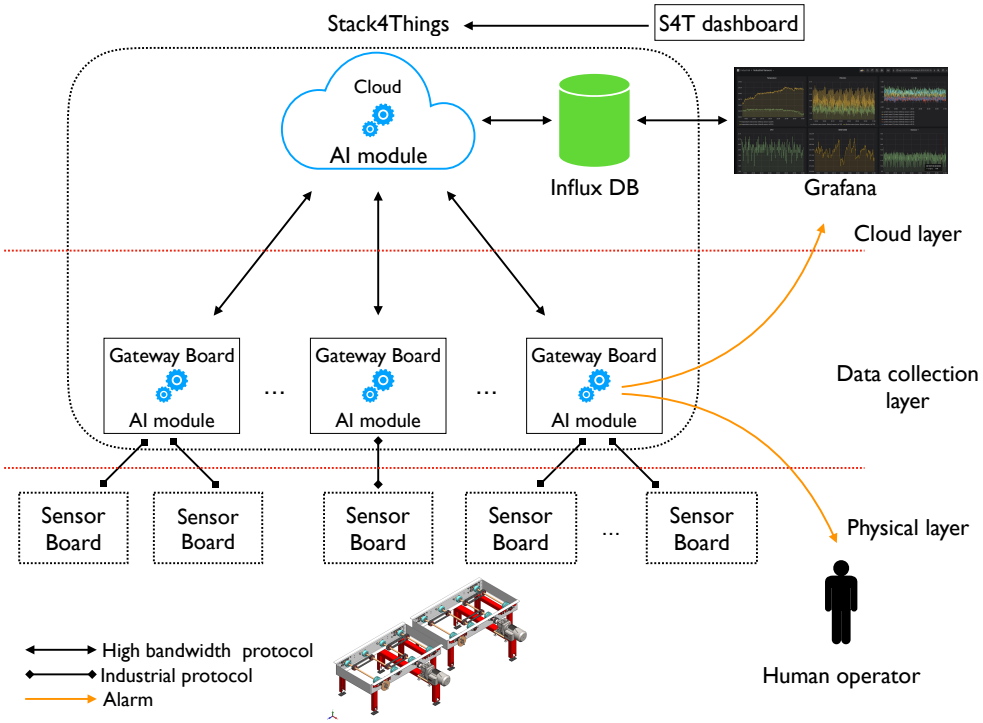
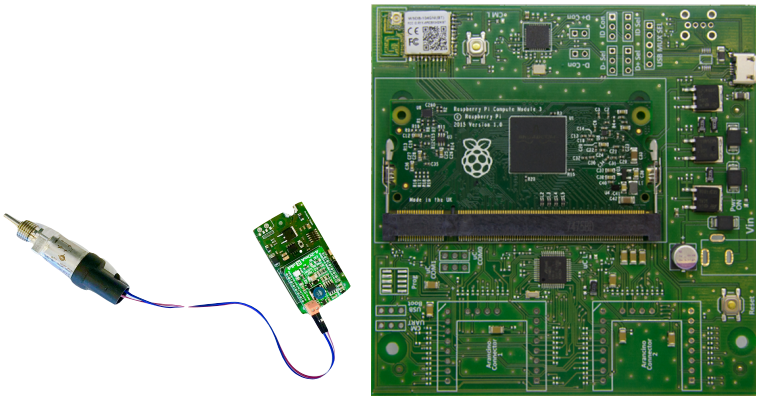


Figure 4.13: Proposed IIoT architecture.

smartme.IO[1]. Specifically, Figure 4.14a shows the sensor board named *Arancino mignon*, a board which mounts a SAMD21 MCU produced by Atmel and provided with several interfaces such as: I2C, SPI, 4-20mA, and many others. This type of boards have the faculty to be connected to multiple of the above described sensors, moreover thanks to their modularity, it is possible to equip them with expansion shields that allow the use of "traditional" communication protocols (e.g., BLE and Wi-Fi), or more specific such as: MODBUS and CAN which are typically adopted in industrial scenarios. Figure 4.14a shows the *Arancino* board (i.e., the gateway board), a more powerful version of the *mignon* in terms of computing power and number of interfaces that allow the interaction with the physical world. Unlike the Arancino mignon which is only equipped with a MCU, the Arancino board has a double environment consisting of the SAMD21 MCU and a MPU provided by the a Raspberry compute module that allows to perform

[1]http://smartme.io, accessed October 2020

more complex operations. A very interesting aspect of this board is its capability to make communicate the MPU and MCU using a serial communication which perfectly integrates with the S4T software architecture. As a design choice to make the board more compatible with the large number of *click modules*[2] (very common in the market), the MCU connector pins are the same of the Arduino Zero. Moreover, the board is equipped with a special connector that allows to attach a NVIDIA Jetson for those cases where a higher computing power is required.



(a) Sensor board (b) Gateway board

Figure 4.14: Hardware setup for data collection and management.

The software part consists of three layers where the physical one represents the "entry point" of our architecture and it is also the closest to the industrial plant itself. At this level, the micro-controllers of many sensor boards perform the computations over the electric signals generated by the sensors, creating the actual measurement that is passed to the upper layer. The data collection layer is responsible for collecting the data and acts as a bridge between the physical layer and the Cloud. In such a context, one of the biggest challenges during its implementation was due to the wide variety of protocols and data formats adopted by sensors, in this sense the layer has to manage and analyze the data gathered from the sensors boards and make it homogeneous before sending to the Cloud. Thanks to the AI module that we introduced in S4T (see paragraph

[2]https://www.mikroe.com/click, accessed October 2020

3.2), the gateway board can also perform a preliminary anomaly detection on the industrial plant and if an abnormal behavior is occurring, the board generates an alarm to the human operator that is monitoring the plant and to the upper layer.

The third layer is represented by the Cloud; here we run an instance of S4T coordinating the gateway boards (i.e., the Arancino boards) and storing the plant historical data in a database. Specifically, we adopted InfluxDB[3] a NoSQL database designed to work with embedded devices and particularly suitable for storing long time series. To visualize the data, we used a tool called Grafana[4] which perfectly integrates with InfluxDB and allows to display its values through a dashboard, thus making it suitable to perform the plant telemetry and show the occurrence of anomalies. Like the gateway board, also the Cloud has an AI module that in this case is used to perform not only the inference, but also the training process of onerous machine learning algorithms that cannot be executed on the Edge (i.e., the gateway boards). Finally, to access our industrial architecture we exploit the S4T dashboard exposing all the framework functionalities via a web browser.

4.2.3 Anomaly detection approach

Most of the existing approaches tackle the problem of preventing the faults of an industrial plant by fixing a constant maintenance scheme (even when this is not necessary). Unfortunately, such an approach is infeasible since it is a time and money consuming process. As we saw in the previous paragraph, the predictive maintenance is a technique that can address this problem, however it requires specific conditions in order to be performed, in this sense, anomaly detection is a viable alternative than can be used to recognize those conditions that can be dangerous for an industrial plant. To do that, an anomaly detection algorithm is based on the use of machine learning techniques to identify those samples in

[3]https://www.influxdb.com, accessed October 2020
[4]https://grafana.com, accessed October 2020

a dataset that do not follow the same trend (pattern) defined by the rest of the data they belong to [75].

Although anomaly detection results a valid solution to tackle the problem related to unnecessary maintenance, it also poses significant challenges which are mainly caused by the lack of anomalous data. In this sense, the possibility to inject faults in our system has been fundamental for the design and implementation of our algorithm other than giving us the opportunity to have a complete understanding of the plant when working in a normal or anomalous condition.

Before designing the anomaly detector, we analyzed the data collected through the proposed IIoT architecture in order to have a better understanding about its structure and spatial distribution. It is worth to mention that even if we instrumented our industrial testbed with the proximity sensor, in this work we used it only for telemetry purposes, for this reason in the following data analysis, we did not consider its data.

Figure 4.15 shows a 3D plot of the raw data coming from sensors using the PCA, an unsupervised machine learning technique that reduces the data dimensionality by projecting it in a new latent space that eases the visualization. By observing the image, it is evident that both the two data categories (i.e., anomalous and non-anomalous) are very mixed, implying that the data has to be preprocessed before using machine learning techniques for the anomaly detection. With respect to the data distribution, we can observe that even if data is mixed, it is also well clustered.

After this first examination, we continued the data analysis by applying a manual feature extraction process in order to select the most informative features in the system (i.e., the scale replica industrial plant). In particular, our goal was to identify the features exhibiting a clear pattern that could help a machine learning model during the anomaly detection process.

In such a context, we derived 259 features, namely: 1 feature each for the vibration, temperature, and current, and 256 features for the audio (i.e., the noise).

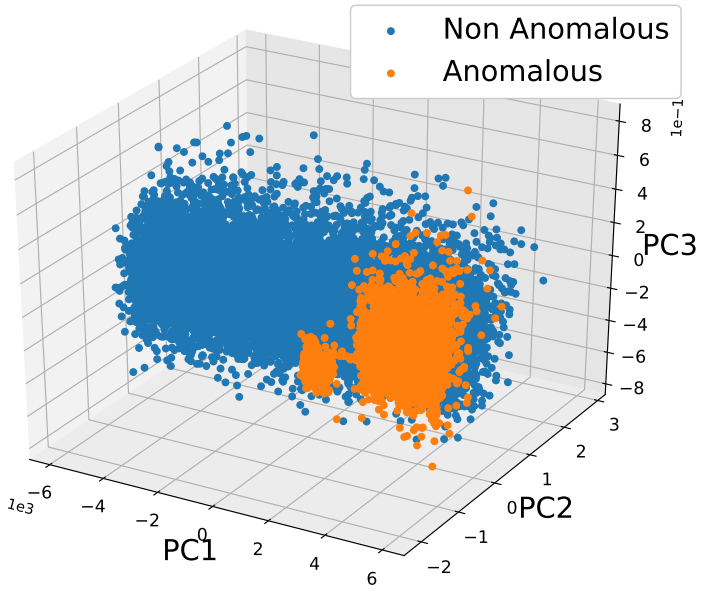
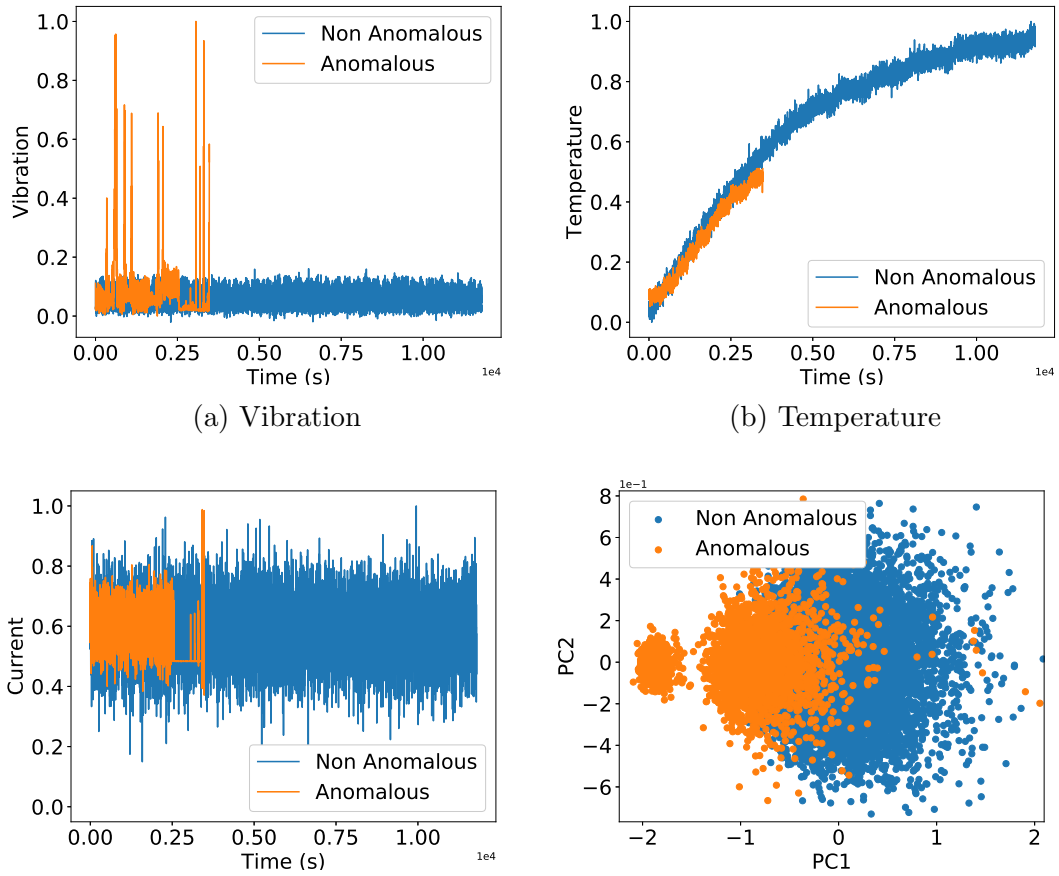


Figure 4.15: Three-dimensional PCA applied on raw sensor data.

With respect to the noise features, we obtained them computing the Fast Fourier Transform (FFT) of the audio signal captured by the microphone which returns 256 components in the frequency domain. In particular, we decided to use a frequency signal because the audio spectrum results to be more informative when compared with a time domain signal wave, and provides information about the system working conditions (i.e., anomalous or non-anomalous). Moreover, the use of frequency signals allows the application of filters to remove those components that can cause a signal distortion.

Figure 4.16 plots the above described features in both anomalous and non-anomalous working conditions where the 256 FFT features have been reduced using the PCA and plotted in a 2D space.

Focusing our attentions to Figures 4.16b and 4.16c, it is evident that they are not suitable to be used for the anomaly detection. This because they exhibit a trend that does not change when passing from an anomalous condition to a non-anomalous one. In Figures 4.16a and 4.16d, on the other hand, we can notice a change in trend between the two conditions, in this sense, we can exploit this behavior to use the vibration and the noise as "discriminant" features for the



(a) Vibration (b) Temperature

(c) Current (d) Noise (FFT)

Figure 4.16: Plots of the system features considering anomalous and non-anomalous working conditions.

anomaly detection. However, the vibration was affected by very high fluctuations which could cause a degrade of the training process. To avoid such a problem, we smoothed the vibration signal using a time sliding window where we computed the mean, maximum, and minimum of the vibration samples falling inside of it, thus extracting 3 features.

Considering the 259 features coming from the vibration and the noise, we were not able to solve the problem related to the "curse of dimensionality". The design of the proposed anomaly detection framework aims to reduce the number of features while keeping the information as much as possible. Figure 4.17 shows the building blocks of the anomaly detection algorithm which is the result of a

combination of four techniques, namely: Autoencoders (AE), deep Autoencoders (deep AE), PCA, and K-Means.

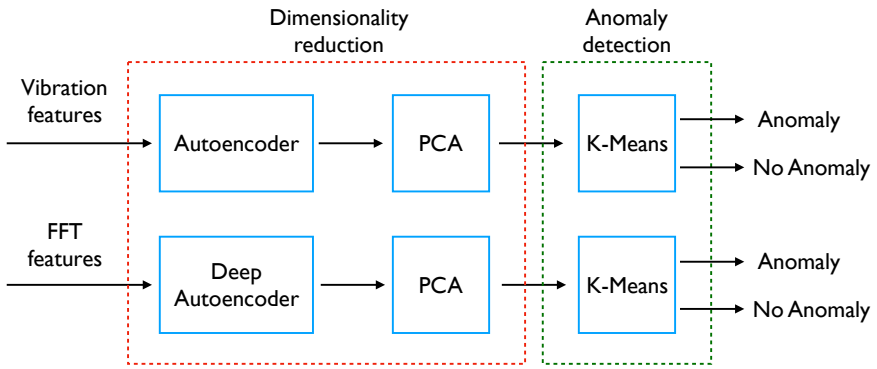


Figure 4.17: Building blocks of the proposed anomaly detection algorithm.

Starting from the left of Figure 4.17, the vibration and noise features are given as input to the AE and deep AE blocks to reduce their dimensions. Specifically, an AE is a feed forward neural network which tries to learn an efficient encoding of the data passed as input. From a topological point of view, in AEs and deep AEs, the input layer and the output one are exactly the same such that the network is able to learn and reproduce in output what has been passed as input. The peculiar aspect that differentiates these type of networks to the others is given by the fact that in this case the real output is not provided by the output layer, but it is provided by a specific hidden layer named *code layer* which contains the reduced representation of the data passed as input (as showed in Figure 4.18).

More formally, an AE (or a deep AE) can be described as the combination of two elements, namely: an encoder and a decoder. From a mathematical point of view they can be expressed as follows:

$$\begin{cases} \mathcal{E} : X \to Z \\ \mathcal{D} : Z \to X, \end{cases} \tag{4.14}$$

where the encoder $\mathcal{E}$ is a function that maps a n dimensional (where n is the number of features) input $X \in \mathbb{R}^n$ in a new m dimensional latent space (i.e,

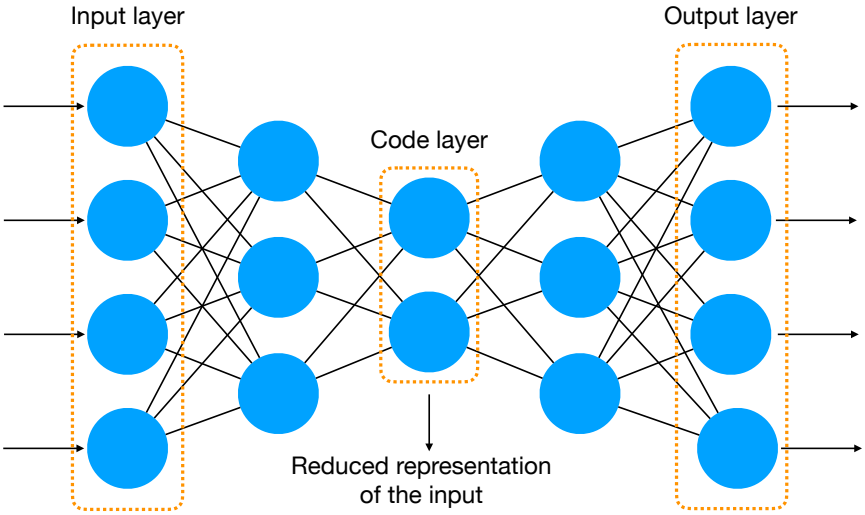


Figure 4.18: Autoencoder topology.

the code size) $Z \in \mathbb{R}^m$ with $m < n$. On the other hand, the decoder $\mathcal{D}$ is a function that reconstructs the input from the latent space bringing it back to the original space. Obviously, such a process implies a loss of information due to the compression of the dimensions. In this sense, given the following system of equations:

$$\begin{cases} z = \sigma(Wx + b) \\ \hat{x} = \sigma(W'z + b') \\ \mathcal{L} = \|x - \hat{x}\|^2, \end{cases} \tag{4.15}$$

the first equation describes the code computation z by the encoder where σ is the neural network activation function, and W and b are the encoder weight matrix and bias vector respectively.

With respect to the second equation, $\hat{x}$ is the input reconstructed by the decoder where, just like the previous case, σ is the neural network activation function, and W' and b' are the decoder weight matrix and bias vector.

Finally, the last equation represents the neural network loss function $\mathcal{L}$ computed as the norm of the original input x and the reconstructed one $\hat{x}$.

As mentioned above, the encoding and decoding cause a loss of information,

for this reason, the objective during the training process is to find the best set of neural network parameters that minimize the loss function $\mathcal{L}$:

$$\arg\min_{W,b,W',b'} \|x - \hat{x}\|^2 . \tag{4.16}$$

In particular, the satisfaction of this condition ensures as consequence the minimization of information loss.

After the dimensionality reduction of the AE and deep AE we were able to reduce the vibration features from 3 to 2 and the noise feature from 256 to 10 however, we observed that data was still to mixed, hence we decided to use PCA in order to separate the data through the application of another transformation. Specifically, the PCA finds a new reference system where the new axis contains the highest variance possible. As a result of this condition, if we consider a set of points belonging to two "classes" (e.g., anomalous and non-anomalous), these will be separated due to their different spatial distribution. Finally, the data transformed by the PCA is passed as input to the K-Means algorithm block that performs the actual anomaly detection. The methodology we adopted to train and validate our anomaly detection algorithm is the following. We first trained our model with the data collected from the plant under a non-anomalous condition. By doing so, the K-Means algorithm was able to learn and compute the centroids associated to a working condition of the system, thus acquiring the knowledge to detect the anomalies. In this sense, once the system is trained and a new set of data is passed as input, the K-Means evaluates the distance of this data with the pre computed centroids, and according to a threshold (empirically fixed) it outputs a binary value where 0 indicates that the data is not-anomalous while 1 indicates it is anomalous:

$$\begin{cases} if\ d > \tau\ 1 \\ if\ d \leq \tau\ 0, \end{cases} \tag{4.17}$$

Table 4.6: Mean transmission time and standard deviation from the sensor board to the gateway, and from the gateway to the cloud.

Data transmission times	
$SB - GB\ (ms)$	0.616 ± 0.002
$GB - CL\ (ms)$	12.1 ± 8.4

where d is the distance between the datapoint and the centroid, and τ is the threshold that establishes if the data is anomalous or not.

4.2.4 IIoT architecture experimental results

In this paragraph, we present the results obtained by testing our framework in terms of data transmission time (i.e., the time necessary to reach the Cloud) and anomaly detection inference time.

To measure the data transmission time, we divided the "path" between the sensor board and the Cloud into two parts where the first one is the end-to-end transmission time from the sensor board and the gateway board (SB-GB), and the second one is the end-to-end transmission time from the gateway board and the Cloud (GB-CL).

Table 4.6 depicts the data transmission time in SB-GB, and GB-CL parts where the results have been obtained by computing the mean and the standard deviation over 1000 data transmissions. In particular, the overall time that the data takes to reach the Cloud is in the order of milliseconds which makes our framework suitable for the application in an industrial scenario.

With respect to the inference time, we considered two different scenarios in which the model is deployed, namely: the Cloud and the Edge (i.e., the gateway board).

Tables 4.7, 4.8 show the mean inference time and standard deviation of the anomaly detection algorithm performed on the gateway board (powered with a RaspBerry compute module) and the Cloud respectively. Like the previous case, the results have been obtained by computing the mean and the standard

Table 4.7: Mean inference time and standard deviation of the anomaly detection algorithm on the gateway side.

Gateway side inference times	
$Noise\ (s)$	16.3 ± 0.3
$Vibration\ (s)$	5.62 ± 0.4

Table 4.8: Mean inference time and standard deviation of the anomaly detection algorithm on the cloud side.

Cloud side inference times	
$Noise(s)$	1.07 ± 0.01
$Vibration(s)$	0.44 ± 0.007

deviation on 1000 model inferences. As we expected, the inference time is lower on the Cloud because of its higher computing power, however, by using a more performing compute module (e.g., the NVIDIA Jetson) the inference time on the gateway side can be reduced in order to meet the strict response times of the industrial applications.

With respect to the anomaly detection algorithm, the dataset to train and test the proposed algorithm has been created using the IIoT architecture we described in Paragraph 4.2.2. At the end of this procedure, we obtained a dataset of $12,504$ faults where $9,210$ were caused by turning off the proximity switch of the cart with a consequent generation of a noise sound, and $3,249$ were generated by increasing the friction of the gears through the activation of the engine brakes, thus causing an increment of the vibrations. In particular, the number of anomalies associated to the gears friction is lower than the other category of anomalies because these type of faults can produce severe damages to the plant. In this sense, to preserve the industrial testbed, we collected a number of anomalies sufficient to validate the anomaly detection algorithm.

To better analyze the obtained results, we computed the confusion matrix, a "table" that allows to evaluate the performance of a machine learning algorithm by extracting three indices: the *precision*, *recall*, and F_1-score. These indices are defined in terms of *true positives* (TP), *true negatives* (TN), *false positives* (FP),

and *false negatives* (FN), where the first two refer to the number of samples correctly classified by an algorithm, while the last two represent the number of misclassified samples. Starting from these elements, the above indices are computed as follows:

$$precision = \frac{TP}{TP + FP}, \tag{4.18}$$

which represents the portion of samples predicted as positive actually correct. With regards to the recall, it is computed as follows:

$$recall = \frac{TP}{TP + FN}, \tag{4.19}$$

and measures the portion of actual positive samples correctly classified.

Due to the variability of these indices, sometime it could be difficult to determine the performance of an algorithm especially in those cases where there is the necessity to make comparisons with other techniques. In such a context, the F_1-score allows to have an immediate estimate of how good a model is. From a mathematical point of view, the F_1 score is an index bounded between 0 and 1, and defined as:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}, \tag{4.20}$$

and returns 1 when the algorithm correctly classifies every test sample, otherwise it returns 0.

Table 4.9: AE and deep AE networks configurations.

Training paramters	
Activation function	*ReLU*
Learning rate	0.001
Optimizer	*Adam*
Regularization rate	0.01
Training epochs limit	5000
Patience term	10

Table 4.9 represents the AE and deep AE training configurations. Specifically, the AE topology consists of three layers with 3 neurons for the input and the output, and 2 units for the code layer. We used ReLU as activation function and fixed the *learning rate* to 0.001 with the *Adam* optimizer. To prevent overfiting we used a L2 regularization technique with a *regularization rate* of 0.01, and an early stopping criterion to prevent network over training setting the *patience term* to 10 epochs. In this sense, even if we set the training epochs limit to 5000, the model stops the process as soon as it not able to reduce the loss function.

With respect to the deep AE, we used the same training configuration. The main difference in this case is given by the topology which consists of nine layers with 256 neurons units for both the input and output layers, 128, 64, and 32 for the encoder part, 10 units for the code layer, and 32, 64, and 128 units for the decoder.

Table 4.10: Algorithm confusion matrix on noise data.

		Actual	
		Anomaly (1)	No Anomaly (0)
Predicted	Anomaly (1)	12504	21
	No Anomaly (0)	0	7483

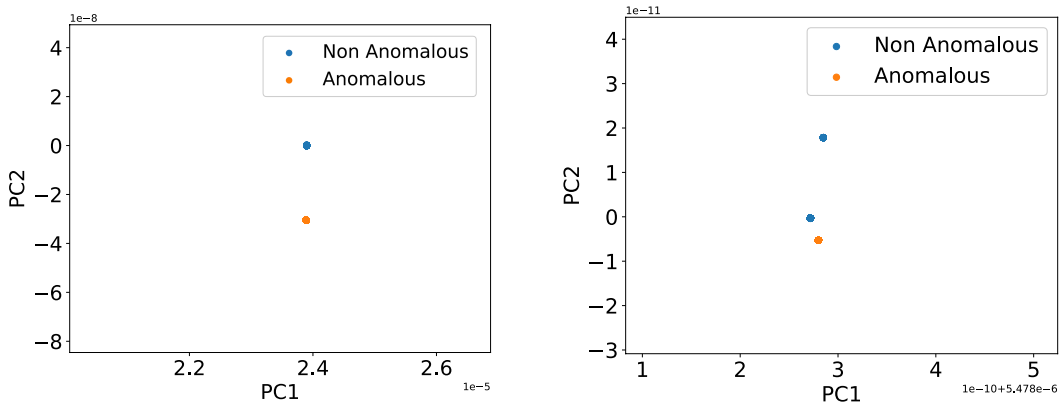
Table 4.11: Algorithm confusion matrix on vibration data.

		Actual	
		Anomaly (1)	No Anomaly (0)
Predicted	Anomaly (1)	2500	29
	No Anomaly (0)	0	1471

Figure 4.19a depicts the output of the proposed anomaly detection algorithm on the noise data. The two points here represented are two clusters dense of datapoints that demonstrate the effectiveness of our technique in totally separating the data. Such a result is also showed in Tables 4.10 and 4.12 where we report the confusion matrix and the scores of this experiment. In particular, the algorithm

Table 4.12: Performance indices of noise and vibration data.

Noise and Vibration performance indices		
Index	*Noise*	*Vibration*
Accuracy	0.998	0.992
Precision	0.998	0.99
Recall	1.0	1.0
F_1	0.998	0.994



(a) Noise results

(b) Vibration results

Figure 4.19: Anomaly detection output on noise and vibration data.

was able to correctly detect the majority of the anomalies with a precision of 0.998, a recall of 1.0, and an accuracy and F_1-score of 0.998

Figure 4.19b shows the results from the vibration data. Like the previous case, the three points here depicted are clusters of datapoints. Also here, the results demonstrate the effectiveness of the anomaly detection algorithm in separating the anomalous points to the non-anomalous ones. Moreover, from our experiments we noticed that the introduction of the mechanical faults not always generates an amount of vibrations to be considered anomalous. Nevertheless, the results returned from the confusion matrix (see Table 4.11) and the corresponding performance indices (see Table 4.12) are very good, with a precision of 0.99, a recall of 1.0, a F_1-score of 0.994, and an accuracy of 0.992.

Results showed that the data transmission from the sensors to the Cloud is in the order of milliseconds which makes our framework suitable for an industrial

scenario. With respect to the proposed algorithm, the obtained results are very good and demonstrate the effectiveness of anomaly detection techniques as a valid alternative to the predictive maintenance for the timely detection of anomalous behaviors, thus preventing total system breakdowns and supporting the human operators.

4.3 Fault prediction in Industry 4.0 using sensor data fusion

In Paragraphs 4.1 and 4.2, we proposed two different approaches to detect anomalous or dangerous behaviors in order to avoid a system breakdown which can have catastrophic consequences (in the worst case). Leveraging the IIoT architecture described in Paragraph 4.2, in this paragraph we present a third technique which exploits sensor data fusion to predict the type of fault occurring in an industrial plant. With the term fault prediction (or fault detection), we refer to a machine learning task that aims to classify the fault occurring in a system according to a set of measurements passed as input. On the other hand, the objective of sensor data fusion consists in merging several signals into a single one in order to create an indicator that correlates the information coming from a set of heterogeneous sources, thus improving the overall model predictive performance [76].

The problem of predicting the type of fault occurring in an industrial system can be performed in different ways. The surveys presented in [77], [78] put in evidence the trends and challenges related to the data fusion and fault detection in Industry 4.0. Authors in [79] propose an unsupervised algorithm based on Variational Autoencoders (VAE) for the anomaly detection on Linear Motion (LM) guides. The detection is performed by analyzing the spectrogram of signals generated in "healthy" and "anomalous" conditions which are passed as input to the VAE. Unlike the proposed approach, the one here presented is only able to detect the presence of anomalies. In this sense, our algorithm exploits sensor data fusion

to not only detect, but also classify the type of the fault occurring in the system. Moreover, if we consider a very large industrial scenario, in certain cases the only detection of the anomaly could be not sufficient because of the presence of a huge amount of components. In such a context, the use of a fault prediction technique is beneficial since it helps the human operator to better understand where the fault occurred, thus allowing the delivery of tailored maintenance schemes. The work presented in [80] propose the use on an ensemble of 5 different classifiers (i.e., DT, logistic regression, Naive bayes, multinomial Naive bayes, and KNN) for fault prediction that has been tested on 4 public available datasets. Even if the authors obtained good results with their technique, they did not validated it on a real industrial plant like in our case. In this sense, our goal is not only to demonstrate the effectiveness of the proposed algorithm, but also its feasibility in a real application scenario.

4.3.1 Sensor data fusion algorithm

As we mentioned in the previous paragraph, the gateway board (i.e., the Arancino board) and the Cloud are equipped with an AI module that allows to run machine learning applications on top of it. The possibility to inject faults in our scale replica plant allowed us also to label them during the data collection process. Indeed, such a feature has been fundamental for the implementation of this technique since it enabled the creation of a historical labeled database useful to train and test the proposed deep learning model. In particular, we tackled the problem of fault prediction as a sensor data fusion multi class classification problem, where the classes we wanted to classify were the following: *working*, *proximity*, and *brake*. With respect to the first class, it is referred to a condition in which the plant is working properly (i.e., the cart moves correctly back and forth), the second one represents a condition where the proximity switch is broken thus causing a mechanical fault that makes impossible the movement of the cart. Finally, the third class describes a condition where the friction of the gears cause

a mechanical stress and a sudden change of the engines belts tension.

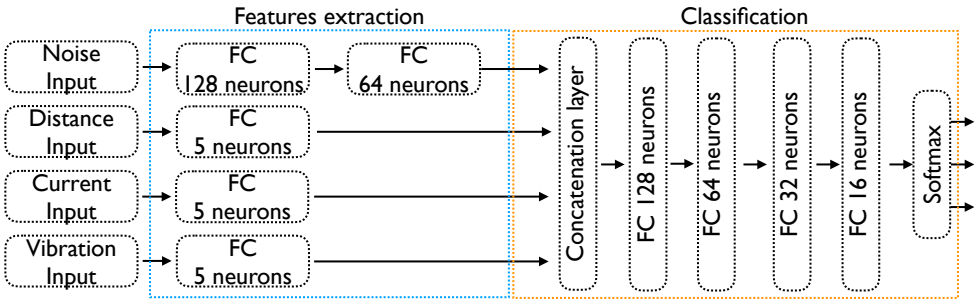


Figure 4.20: Sensor data fusion deep learning model.

Figure 4.20 shows the DNN model we designed exploiting sensor data fusion. Unlike the anomaly detection technique described in the previous paragraph, in this case the model takes a higher number of signal inputs, namely: noise, distance (i.e., the proximity sensor data), current (where all the three phases of the motors have been computed), and vibration. The temperature was the only signal that we did not consider since from our empirical tests it exhibited a low variability which made it non informative for the fault prediction task.

One of the biggest challenges when working with data fusion applications is the heterogeneity among the sensor measurements in terms of sampling rate. Specifically for this application, we sampled the current, vibration, and noise at 240 Hz, 100 kHz and 44 kHz respectively, while we used a sampling rate of 10 Hz for the distance input. In this sense, it has been necessary to perform a data resampling to guarantee the time coherence among the signals.

Before passing the data to DNN, we apply a preprocessing step to remove possible outliers which can affect the training phase. With respect to the noise, we used also in this work the FFT as it resulted effective in the detection of anomalies (see Figure 4.19a). Plotting the data related to the current and vibration, we noticed that these two signals had very high fluctuations that, as we said in the previous paragraph, can degrade the training process. To avoid these conditions we applied a time sliding window of 14 ms and 25 ms of widths respectively, performing a feature extraction process by extracting the maximum values of

current and vibration, thus obtaining two smooth signals. Finally, with respect to the distance we did not apply any preprocessing technique since the signal did not exhibit high fluctuations. Moreover, using a sampling rate of 10 Hz the signal was already smooth enough to be passed as input to the DNN. After the above described preprocessing step, we obtained $7,504$ samples for the working class, $9,210$ for the proximity class, and $3,294$ for the brake class with the following features scheme: 256 for the noise, 3 for the current (one for each phase), 1 for the vibration, and 1 for the distance.

A closer look to Figure 4.20 puts in evidence that its topology consists of two building blocks, namely: feature extraction and classification. With respect to the first part, it is responsible to perform an automatic feature extraction process of each the input separately, using one or more fully connected (FC in Figure 4.20) layers connected in cascade. Specifically, for the distance, vibration and current, consisting of 1 and 3 features respectively, we used one fully connected layer with 5 neurons. For the noise which has a much larger number of features, we used two fully connected layers of 128 and 64 neurons. The extracted features are merged together using a *concatenation* layer which implements the actual data fusion. Such a layer, represents the "input" of the second part of the DNN which performs the actual classification. Specifically, the classifier consists of 4 fully connected layers with 128, 64, 32 and 16 neurons respectively, while the last layer (i.e., the output) is composed by 3 units (one for each class we want to classify) on top of which we apply a softmax function.

4.3.2 Sensor data fusion results

After the data collection and labelling, we ended up obtaining a dataset of 20 K samples that we split using a K-fold cross-validation with K = 10. To train the above described DNN, we used Keras whose hyperparameters are reported in Table 4.13. In particular, we used ReLU activation function and *Adam* optimizer with a *learning rate* of 0.001. To prevent overfitting we used two techniques,

namely: a L2 regularization with a *penalty term* set to 0.01, and an early stopping technique where we fixed the *patience term* to 10 such that the training process is stopped in advance if for 10 consecutive epochs the loss function does not decrease. Finally, we set the training limit to 1500 epochs.

Table 4.13: Proposed DNN configuration.

DNN training parameters	
Activation function	*ReLU*
Learning rate	0.001
Training epochs limit	1500
Regularization term	0.01
Optimizer	*Adam*
Patience term	10

After the training process, the proposed data fusion model reached an accuracy of 95% on the test set. However, to better evaluate its performance, we computed the confusion matrix from which we extracted the precision, recall, and F_1-score for each of the 3 classes we wanted to detect. Table 4.14 shows the precision, recall, and F_1-score computed as the mean across the 10-fold cross-validation sets. The overall performance of the proposed model are good with respect to all the classes which means that our approach has been able to well generalize on the test data.

Table 4.14: Proposed DNN precision, recall, and F_1-score indexes.

Proposed DNN Confusion Matrix indexes			
Class	*Precision*	*Recall*	F_1 *score*
Working	0.95	0.88	0.91
Proximity	0.98	0.98	0.98
Brake	0.93	0.96	0.94

To demonstrate the effectiveness of our proposed approach, we compared it with a "monolithic" DNN which does not exploit sensor data fusion.
Figure 4.21 depicts the monolithic DNN topology. Specifically, the network consists of 6 layers (excluding the input) with 256, 128, 64, 32, 16, and 3 neurons

Table 4.15: Monolithic DNN precision, recall, and F_1-score indexes.

Monolithic DNN Confusion Matrix indexes			
Class	*Precision*	*Recall*	F_1 *score*
Working	0.94	0.86	0.90
Proximity	0.92	0.96	0.94
Brake	0.90	0.88	0.89

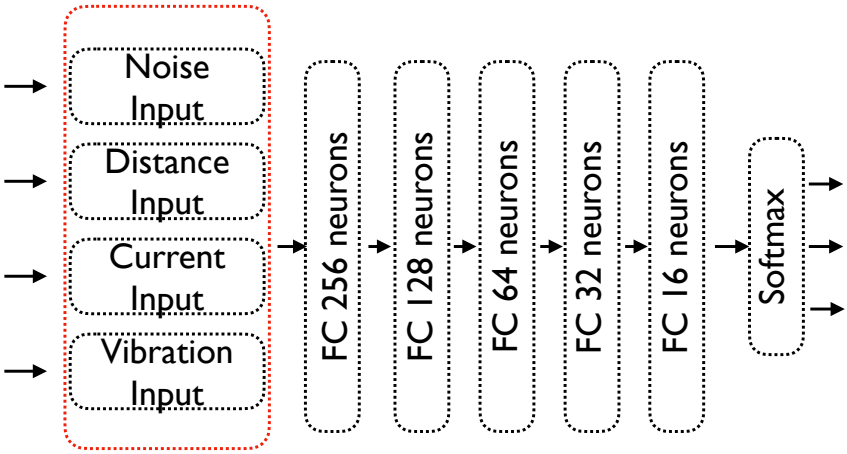


Figure 4.21: Deep learning model without sensor data fusion.

respectively, which are responsible not only for the feature extraction process, but also for the classification. The output layer is exactly the same we described for our proposed model. Unlike in data fusion, this model does not treat the inputs separately, but merges them together into a single one. In order to make a fair comparison between the two models, we used the same 10-fold cross-validation sets adopted for the training and test of the data fusion model; in this case, the monolithic model reached an overall accuracy of 91%. Even if the results obtained by the monolithic approach are comparable with ours, they are slightly lower. In this sense, the use of a data fusion approach can benefit from the use of a separated feature extraction process, thus reaching more accurate results. Figure 4.22 shows a comparison in terms of the loss between the two models. In particular, even if the proposed data fusion model has a larger number of layers, the loss trend is comparable, in fact, our model is able to reach the plateau after about 70 epochs which is a good result considering that the monolithic one reaches it

at 40 epochs.

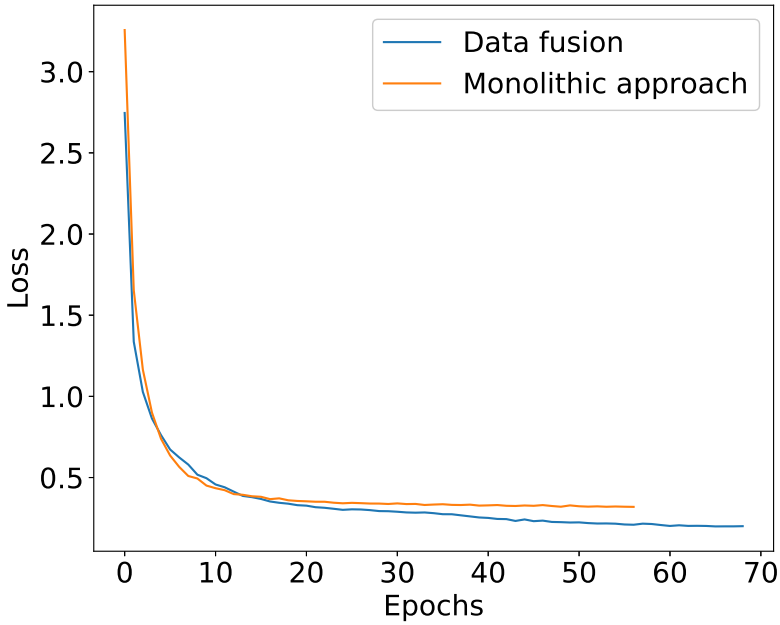


Figure 4.22: Loss comparison between our proposed model and the monolithic DNN model.

According to the results, both the DNNs reach very good performance, however the proposed data fusion approach is able to outperform the monolithic one in almost every index. Our approach results also to be more scalable than the monolithic that has to be re-designed in the case a new input would be added. Finally in terms of space occupation, the data fusion trained model occupied 839 KB on disk which is less than the 1.4 MB of the monolithic one, thus proving once again the benefits brought by sensor data fusion in terms of accuracy, scalability, and space.

To further prove the effectiveness of the proposed approach, we also compared it with SVMs which is a classical machine learning approach to perform classification tasks. Table 4.16 reports the hyperparameters setting we used for SVMs. Specifically, we used RBF as kernel, a penalty term C for the regularization to 1.0, and a gamma parameter fixed to "scale". Finally, we set the tolerance pa-

rameter to 0.001 as a stopping criterion, and the maximum number of training iteration to "auto", a special value used by Scikit-learn that allows the algorithm to have no train limit and stop only when the solver finds a solution.

Table 4.16: SVMs configuration.

SVMs training parameters	
Kernel	*RBF*
Training epochs limit	*auto*
Penalty term (C)	1.0
gamma	*scale*
Tolerance parameter	0.001

Table 4.17: SVMs precision, recall, and F_1-score indexes.

SVMs Confusion Matrix indexes			
Class	*Precision*	*Recall*	F_1 *score*
Working	0.95	0.83	0.88
Proximity	0.88	0.93	0.91
Brake	0.86	0.84	0.85

Considering the same cross-validation sets we used for the two DNNs, SVMs reached an overall accuracy of 88%, while Table 4.17 shows the confusion matrix results. Compared to SVMs the proposed DNN is able to outperform it in every index, moreover, SVMs do not allow a data fusion approach, in this sense we trained the algorithm using the same input structure adopted for the monolithic approach which results again in a bad scalability.

The obtained results demonstrate the effectiveness of the proposed data fusion approach. In a context where several sets of sensors generate a huge amount of heterogeneous data, our technique is a scalable solution which benefits from the separated analysis of each input signal improving the overall performance when compared with a "traditional" approach. Moreover, thanks to the distributed nature of the proposed IIoT architecture, we can perform a "knowledge sharing" among several industrial plants through the implementation of a federate learning approach, that allows to learn a global model capable to detect a larger number

of faults while also improving its generalization capabilities.

Chapter 5

Smart Health

According to statistics, the number of elder people is increasing. The aging is at the base of many diseases and for this reason it has a strong impact on the healthcare system. In such a context, one of the most common problems (especially in hospitalized patients) is the formation of pressure ulcers, a disease strictly related to the patient impossibility to move for long periods which can have bad physical and psychological consequences. Unfortunately, most of the proposed approaches require the use of invasive and expensive techniques. In this chapter, we describe a solution based on deep learning and wearable computing to deliver a smart health support system that can help the medical staff to easily prevent the formation of pressure ulcers in a cheap and non invasive way [81].

5.1 Pressure ulcers prevention using wearable computing and deep learning techniques

The aging of human population has a profound impact on several aspects of social life. From a biological point of view, this natural process produces different types of diseases which are mainly caused by the impairment of the organism. In such a context, smart health plays a key role providing useful tools that can prevent the occurrence of a disease or speed up the healing process.

Pressure Ulcers (PUs) also knows as *bedsores* are injuries caused by prolonged skin pressures which are very frequent in hospitalized patients. According to the National Pressure Ulcer Advisory Panel (NPUAP), PUs occur with a frequency between 10% and 18% in intensive care units, between 2.3% and 28% in long term care units and between 0% and 29% in home care units [82]. PUs appear as lesions on the skin which reduce the patients mobility, cause pain, and in some cases they can also have a bad psychological impact. Depending on the lesion, PUs are classified into different classes of severity [83]. In the first class the skin presents a color ranging from pink to red which disappears 30 seconds after the pressure removal. In the second class, the skin manifests the presence of vesicles and starts to lose continuity. Finally in third and fourth classes, there is a total loss of skin density (with possible muscle injuries in the worst cases).

The risk for a person to develop PUs can be estimated in a number of ways; Braden's scale is one of these which bases its analysis on factors such as movement activity, temperature, skin moisture, and nutrition [84]. Since PUs affect patients quality of life and prolong their hospitalization period (with a consequent increment of costs for the healthcare system), it is necessary to implement strategies to solve (but most importantly *prevent*) this problem that is becoming increasingly common.

Although the availability of different solutions, no one of them has been selected as standard; this becomes even more complicated for those patients exposing a medium risk factor for PUs formation. In a market characterized by fragmented techniques, emerges the necessity to implement a system acting as a bridge between the patients and the medical services thus allowing the deployment of tailored treatments [85]. Wearable technology is a promising solution that revolutionized they way we interact with devices and introduced a new way to generate and collect data. This aspect results to be a key factor when used to acquire useful information to deliver medical assistance [86].

Leveraging Cloud and wearable computing technologies, used in combination

with machine learning, in this thesis we present a system that collects inertial sensors data (worn by the patients), analyzes it, and performs real time motion activity predictions to help the medical staff to prevent and assess the risk of PUs formation on patients.

5.1.1 System architecture

When we started the design process, our main goal was to create a system that can be used by anyone in a simple way. By looking into the literature there are several techniques that can be used to address the problem of PUs prevention. For example, authors in [87] adopt wireless patches that are placed directly on the skin of the subjects to measure the contact pressure, temperature, movements, and humidity that are then sent to a base station. In [88], a system of multimodal sensor is proposed for the evaluation of PUs as support of hospitals and home environments. The solution presented in [89] is based on a ZigBee network of pressure sensors located on the bed to prevent the formation of PUs in patients with mobility problems. The approach described in [90] presents a technique to prevent PUs by adopting a system that provided electrical stimulation of the gluteal muscles. All these solution we presented however do not meet the requirements we wanted for our system, and have some disadvantages which include high costs, complex sensor positioning and bad comfort for the patient. In this sense, our intent was to build a non invasive system that should be easy to use, low cost, and most importantly accurate. To do that, we thought that a good solution to achieve the above mentioned requirements was the use of wearable sensors in combination with machine learning to predict the patient position over time and assess the risk of PUs formation.

Figure 5.1 depicts the clinical scenario we imagined for our system where a group of patients is located in a room wearing a smart version of a hospital gown with a set of wearable sensors attached in the abdominal area. Specifically, we considered to use a three-axis accelerometer and magnetometer to detect the

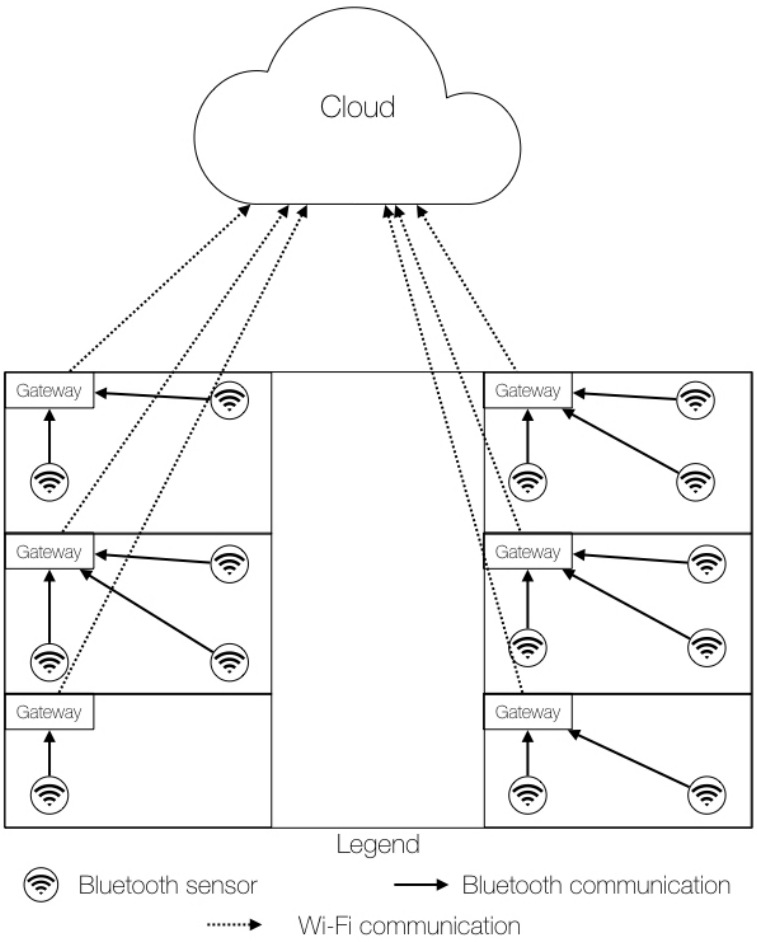


Figure 5.1: Clinical scenario.

patient motion and body orientation. To manage the signals coming from the sensors of each patient, we propose a hardware/software architecture such that each room has its own gateway connected to the Cloud acting as data accumulator.

The proposed architecture has been designed to be scalable; here the main function is performed by the gateway that acts as a local data sink and sends the information of the wearable devices to the Cloud. Then, after a data processing the information is stored in a database accessible by the medical staff of the hospital. In such a context, we used deep learning to build a model capable to take in input the raw data measurements and generate as output the most probable motion activity of the patient, in this sense, the patient "motion history" can be used to assess the risk of PUs formation and prevent their occurrence.

With respect to the hardware, for the gateway we used a Raspberry Pi 3 which provides a complete Linux system with good performance at low cost. For the wearable sensors, we used Flora (see Figure 5.2) a device produced by Adafruit which includes a LSM303 integrated circuit containing an accelerometer and a magnetometer. The board is also equipped with a BLE module through which it sends the data to the gateway (i.e., the Raspberry) as showed in Figure. 5.1.

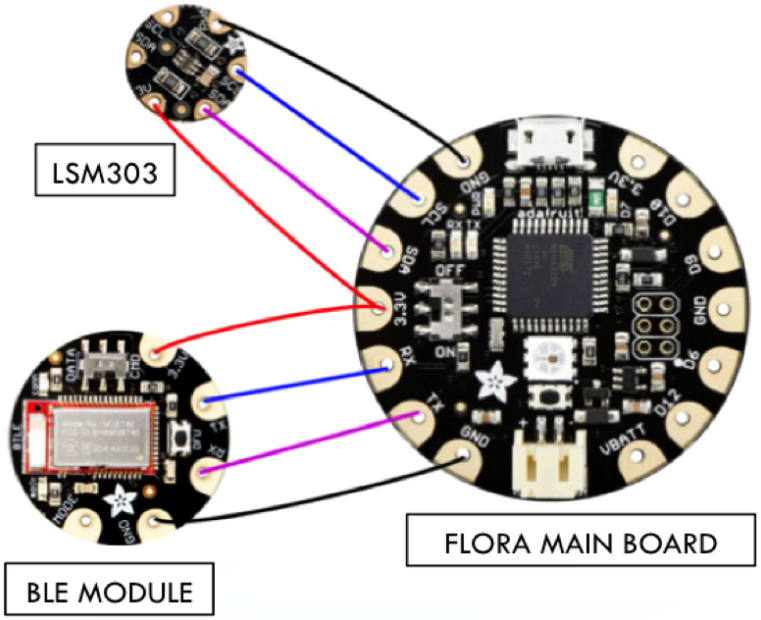


Figure 5.2: Wearables communication.

For the Cloud part, we adopted the S4T framework whose description has been provided in the previous chapters. Figure 5.3 depicts the communication flow between all the components of the proposed system architecture i.e., the wearable sensors, the Raspberry Pi 3, S4T, an Open Data CMS (Content Management System), and a web server. Considering a condition where S4T is installed from scratch, the first step required by the platform is to register the gateway board (i.e., the Raspberry Pi 3) which runs the lightning-rod engine. On the Cloud side, when a registration request arrives, it replies injecting the code containing the logic for the wearable sensing. After these two steps, the gateway is registered with the S4T platform and ready to perform BLE scans.

When the Raspberry detects the advertising data coming from a wearable device, it connects to it via BLE. On the wearable side, when the gateway asks for a connection, it immediately stops to send advertising data, and starts to send the sensors measurements to it. Since in a room there could be more than one patient, the Raspberry works in parallel processing, analyzing and pushing the data to an Open Data datastore by means of Comprehensive Knowledge Archive Network (CKAN) Representational State Transfer (REST) APIs. Through the above mentioned communication flow, the system collects labeled data to create a dataset for the deep model training, in this sense, S4T is also responsible to perform the offline training process and to deploy the trained models to the gateways via the lightning-rod plugin injection feature.

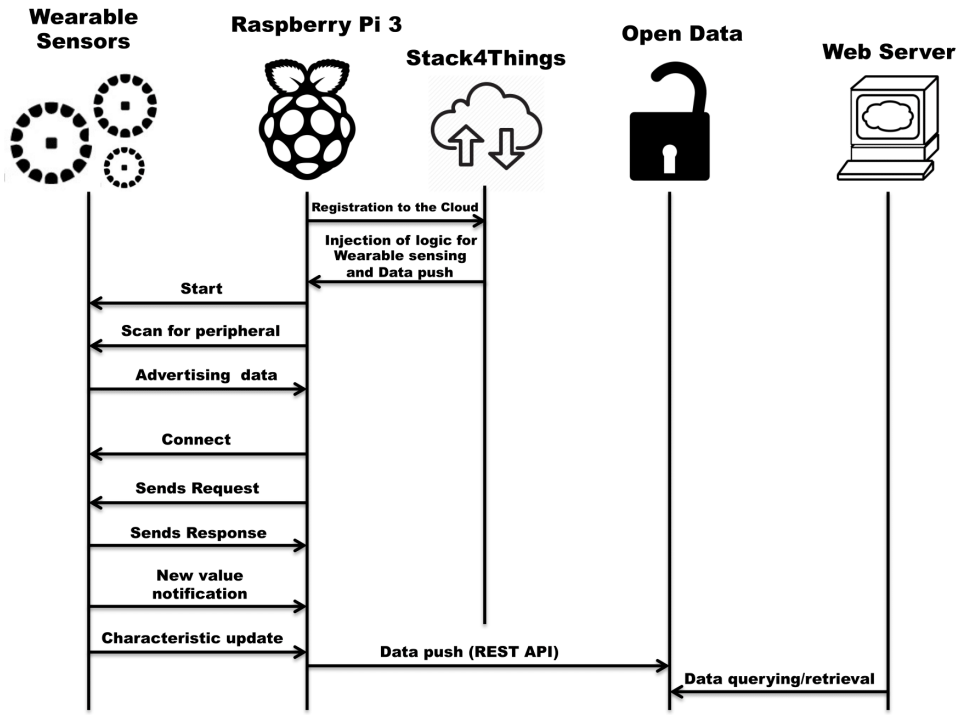


Figure 5.3: Communication flow.

5.1.2 Motion activity classification

The problem of classifying the patient motion activity is very challenging especially for the high variability in motion patterns. For example, the movements of an older adult and a younger one (even if they move in the same way) can be very different. Another problem derives from the sensors which are attached to the irregular surface of the hospital gown which can lead to the generation of wrong or noisy data. In particular, we decided to address this classification problem as a supervised deep learning approach in order to create a model capable to understand the correlation between the wearable sensors measurements and the patient position. The data collection has been done by involving six volunteers belonging to different profiles in terms of age, weight, height, and sex. It has been asked to wear the hospital gown in order to record the body position while performing five motion activities (i.e., staying seated, prone, supine, laying on the left, and on the right). Of course, for privacy reasons we did not insert the names and surnames of the volunteers, who signed a document that allows the processing of their data.

Figure 5.4 shows how the sensors were placed on the volunteers. Each one of them had no constraints on the movement that he/she could do and this has been fundamental so that the collected data derived from a natural motion scheme without any kind of bias. So far we discussed about how we collected data, let us now introduce the definition for a datapoint contained in the dataset. Specifically, a point can be seen as a tuple composed by the following elements: the patient id, the timestamp of the measurement, the sensor data collected from the accelerometer and the magnetometer, and the label representing the patient position associated to the sensor measurements; from a mathematical point of view, the datapoint can be defined as follows:

$$d_t =< (Id, Time, Ax, Ay, Az, Mx, My, Mz, Pos) >, \tag{5.1}$$

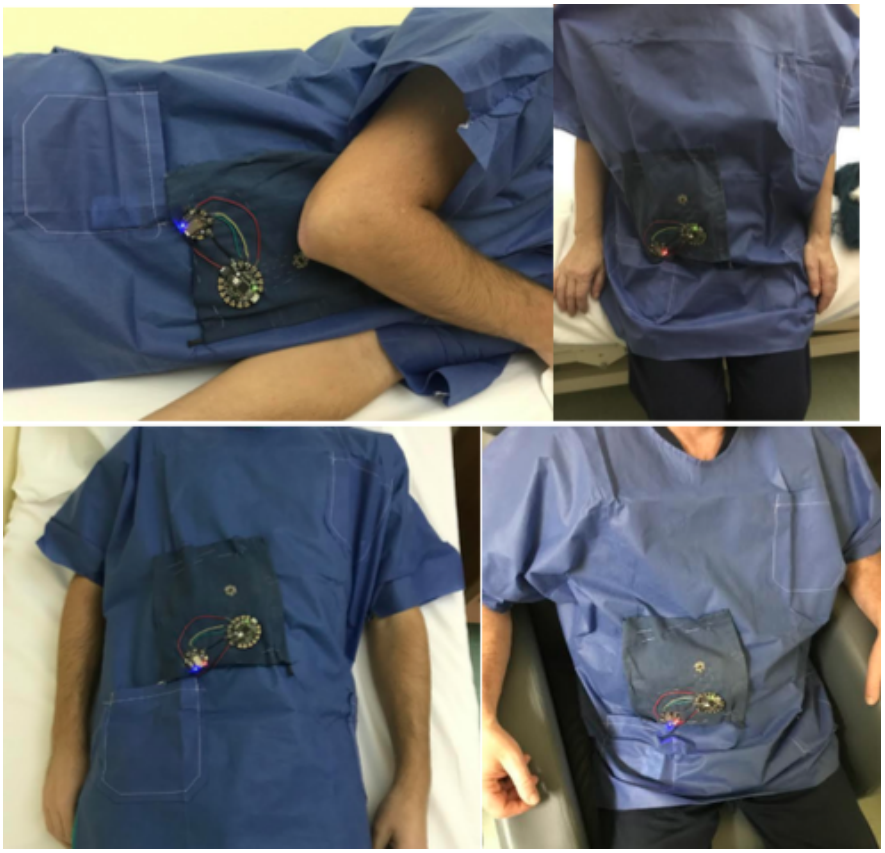


Figure 5.4: Monitoring on the involved volunteers using the hospital gown.

where d_t is the datapoint recorded at timestamp t. In such a context, our goal is to understand the patient motion activity over time in order to prevent the generation of PUs while he/she rests in the same position for a too long period. Analyzing the PUs state of the art, and thanks also to the help of a specialized personnel, we were able to define the main areas where the PUs usually occur and the corresponding hazardous positions. At the end of this analysis, we defined a set of position to monitor in order to prevent the PUs formation, and defined as follows:

$$\mathcal{P} =< Supine, Prone, Right, Left, Sitting, Movement > . \tag{5.2}$$

The final dataset $\mathcal{D}$ obtained by merging the datapoints d_t collected at different timestamps is showed in Table 5.1 where each row contains a set of labeled

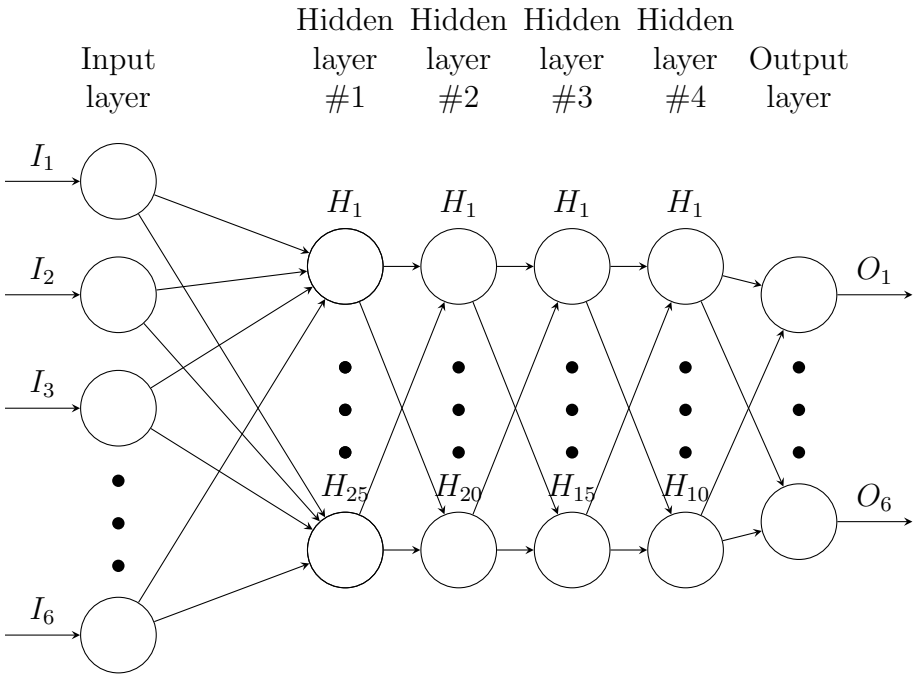


Figure 5.5: DNN architecture for the motion activity classification.

measurements associated to a patient Id.

Table 5.1: Labeled patient position dataset.

Id	$Time$	Ax	Ay	Az	Mx	My	Mz	Pos
0	2018/07/22 19 : 37 : 15	−0.04	0.24	9.77	26.82	8.18	−1.02	*Supine*
1	2018/07/24 19 : 42 : 15	7.53	−5.41	3.14	−14.73	40.27	8.98	*Left*
...	...	...	...	...	...	...	...	...
4	2018/07/27 22 : 37 : 15	−4.67	−8.87	1.80	4.73	21.91	−33.16	*Sitting*
5	2018/07/30 23 : 23 : 45	8.43	4.98	0.51	11.45	−35.36	3.67	*Right*

After the data collection phase we described, we got a total of 8, 708 samples which resulted to be a good amount to train and test our deep learning model. Finally, we divided the dataset into three sets, namely: the train set, the validation set and the test set representing the 70%, 15%, 15% of the original dataset respectively.

Figure 5.5 represents the DNN architecture that we designed. The input layer is composed by 6 neurons representative for the accelerometer and magnetometer

data i.e., (Ax,Ay,Az,Mx,My,Mz), while the output layer is representative for the 6 possible positions (i.e., Supine, Prone, Right, Left, Sitting, and Movement) we wanted to classify. After trying several models, we found out that four hidden layers were sufficient to represent the relationship between the input and the output. With respect to the units, we started with 25 neurons and decreased by five units as the network goes deep approaching the output layer. We used *Adam* as optimizer, and set the *learning rate* to 0.001 which results almost a standard choice when working with this kind of optimizer. We used ReLU as activation function which results also in this case one of the most used in the majority of the DNNs topologies. Finally, to avoid overfitting we used a L2 regularization technique. Table 5.2 shows the parameters used for DNN.

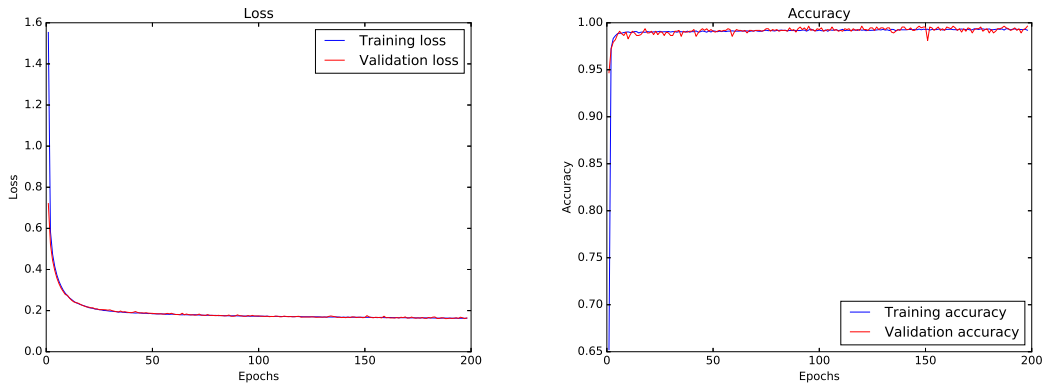
Table 5.2: DNN parameters for motion activity classification.

DNN parameters	
Number of hidden layers	4
Number of neurons	$[25, 20, 15, 10]$
Input dimension	6
Output dimension	6
Optimizer	*Adam*
Learning rate	0.001
Training epochs	500
Regularization parameter	0.01
Activation function	*ReLU*

5.1.3 Motion activity results

In this paragraph, we present the results obtained from testing the above described DNN. Specifically, we implemented the network using Keras and the results of the training process are showed in Figure 5.6a. In particular, the plot depicts the training and validation losses converging to values near to zero, thus demonstrating that the model learned correctly the relation between input and output. Such a condition is also showed by the plot represented in Figure 5.6b where both the curves reach very high level of accuracy (around 99%) which is

totally compatible with the results obtained in the test, where the proposed DNN reached a 99.56% of accuracy.



(a) Loss (b) Accuracy

Figure 5.6: Loss and accuracy curves for the training and validation sets.

We measured the performance of our DNN also computing the confusion matrix (showed in Figure 5.7) from which we extracted precision, recall, and F_1-score indices (showed in Table 5.3).

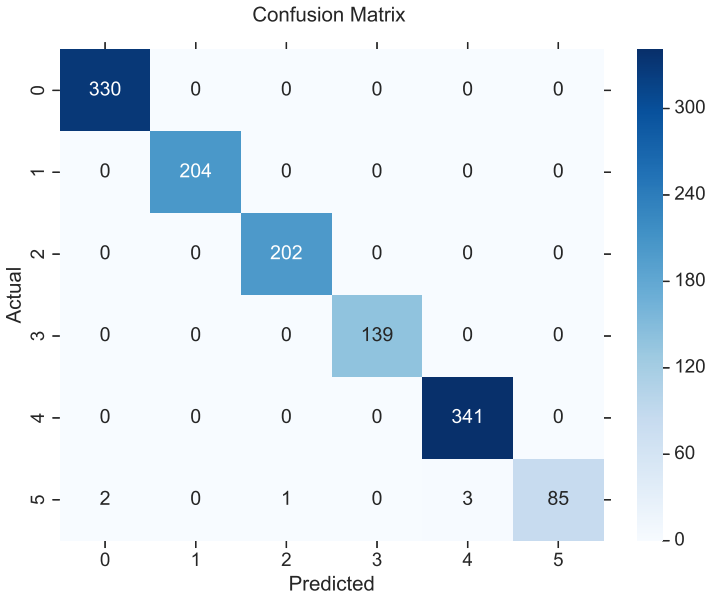


Figure 5.7: Confusion Matrix for our model.

In particular our model performed very well on the test set returning high values for all the performance indices, thus proving once again its effectiveness and

Table 5.3: Classification metrics report obtained from the confusion matrix computed on our model.

DNN Performance metrics			
Label	*Precision*	*Recall*	F_1
Supine	0.99	1.00	1.00
Prone	1.00	1.00	1.00
Right	1.00	1.00	1.00
Left	1.00	1.00	1.00
Sitting	0.99	1.00	1.00
Movement	1.00	0.93	0.97

generalization capabilities. We also compared our proposed approach with other two popular machine learning approaches, namely: SVMs and RF. With respect to the first one, this method is usually considered a valid alternative to the DNN since it is able to reach very good results both in regression and classification tasks. For the SVMs we used a RBF kernel function, a penalty term set to 1.0, a gamma factor set to *scale*, and a number of training epochs set to *auto* such that the algorithm terminates when a solution is found by the solver. The results obtained by the model are comparable with the ones reached by our approach, however, it is worth to mention that the way SVMs performs the classification is based on the 1-vs-1 method which requires to train $N \cdot (N-1)/2$ (where N is the number of classes) intermediate models in order to perform the actual classification. On the other hand, the DNN creates a single model since it is able to perform a multi-class classification, thus resulting faster in terms of inference time and reducing the model footprint. In addition, we also trained a RF classifier setting the number of trees generated by the algorithm to 100 which achieved an overall accuracy of 82.80%.

Table 5.4 summarizes the results obtained by each classifier and puts in evidence how our model outperforms in every index the other two techniques.

Table 5.4: Performance metrics comparison with other classification models.

Models comparison				
$Classifier$	$Acc.(\%)$	$Prec.(\%)$	$Rec.(\%)$	$F1(\%)$
$DNN(ours)$	0.9956	1.00	1.00	1.00
SVM	0.9932	0.99	0.98	0.99
RF	0.8280	0.70	0.82	0.75

5.1.4 PUs prevention system

The very last part of our architecture is the PUs prevention system through which the medical staff can monitor in real time a patient to see his/her "historical" movement activity. In particular, we implemented a webpage that allows the user to connect to a specific room of the hospital in order to have a visual risk assessment of PUs formation for each patient in that room.

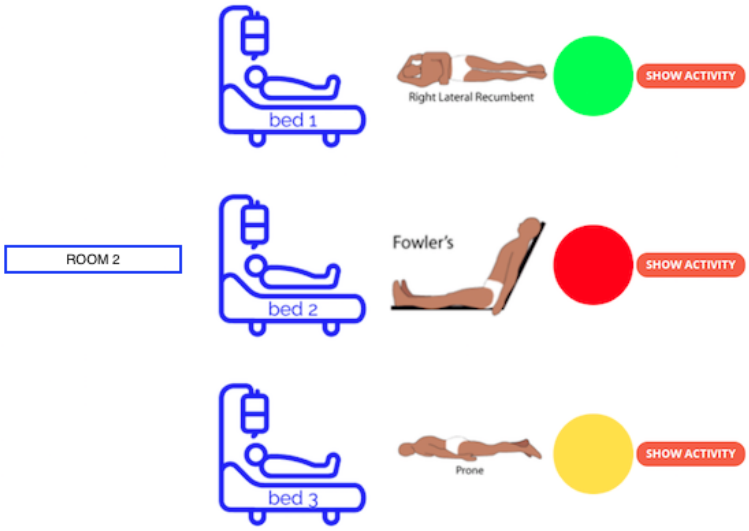


Figure 5.8: Room web page.

Figure 5.8 shows an example of a room webpage where for each patient is associated a circle than can be of three different colors (i.e., red, yellow and green) representing the level of risk. It is worth to mention that this risk assessment guidelines have been provided to us by a specialized medical staff. Specifically, if the DNN predicts the same position for more than two hours, this corresponds to a red color meaning a high level of risk. The yellow color is associated to a

condition in which the patient remained in the same position for half an hour, indicating a mid-level of risk. Finally, the green circle represents a condition where the patient frequently changed position, thus strongly reducing the possibility of PUs formation. The obtained results demonstrate the feasibility of the proposed PUs prevention system and suggest that the use of wearable devices can be considered a viable non invasive solution to prevent the problem of PUs formation, supporting the medical staff in this task. As a future work, we would like to start a collaboration with hospitals to perform a long term analysis of the system, with the goal to quantitatively measure the effectiveness of our algorithm in preventing the PUs formation, which however is beyond the scope of this work.

Chapter 6

Smart Agriculture

The use of modern technologies like IoT allowed an evolution of the traditional agriculture. Originally born as strictly manual, today this science can benefit from the use of tools that allow to improve the productivity and the overall effectiveness of the plantation works. A timely plant disease detection is one of the biggest challenges in this area because of the large number of plants in a plantation that require a constant supervision. In such a context, the use of the Cloud paradigm is not the best solution since it requires a constant Internet connection, thus making the plantations monitoring expensive and time consuming. In this chapter, we propose an ICPS solution for smart agriculture that leverages deep learning and quantization techniques to deliver plant health detection on the Edge [91].

6.1 Plant disease detection on the Smart Edge

The diagnosis of plant health conditions gained a lot of attention in smart agriculture. In such a context, the possibility to perform a timely detection of the early symptoms of a disease can avoid the spread of pandemics on the plantations. Today, most of the proposed solutions involve AI techniques that run on the smart Edge devices (e.g., ICPS) which are equipped with a hardware that enables them to work as sensors and actuators [1]. However, as we saw in the previous chap-

ters, the resource constraints of these devices in terms of power, memory, and computation make the execution of complex algorithms (e.g., machine learning and deep learning algorithms) quite challenging.

The shift of the computation from the Cloud to the Edge is not easy and requires a careful implementation of suitable algorithms. In particular, these devices are designed to be energy efficient because of their always-on capabilities that allow them to constantly "listen" to the surrounding environment, thus making possible the realization of context-aware applications and services [92]. In the recent period, *compression* and *quantization* techniques have been proposed to reduce the complexity of an algorithm while maintaining the performance [93].

In such a context, we developed an ICPS solution for smart agriculture to effectively detect plant diseases using smart Edge computing and machine learning techniques. This thesis proposes an example of our AI application called *Deep Leaf* running on a MCU of the STM32 family to detect coffee plant diseases, with the help of a Quantized Convolutional Neural Network (Q-CNN) model that we implemented to fit the hardware constraints. We also provide a quantitative analysis of the proposed solution making a comparison with other compression and quantization techniques by putting in evidence the differences in terms of accuracy, memory utilization, inference time, and energy consumption.

6.1.1 ICPS platform

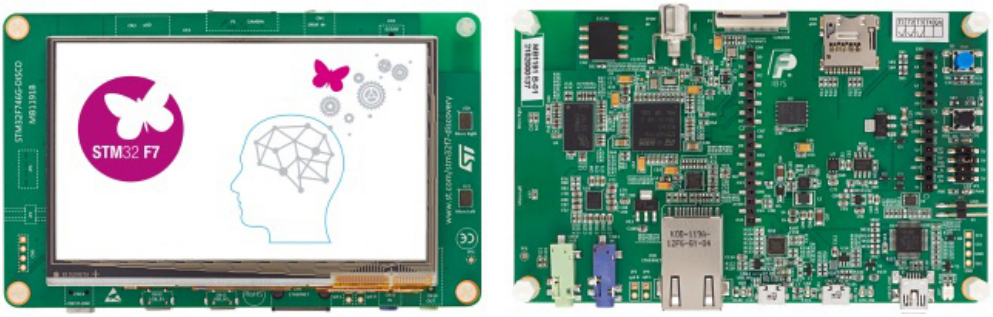
The proposed ICPS platform consists of two parts, namely: the X-CUBE-AI tool and the STM32F746GDISCOVERY (STM32-Discovery board) that we used for the realization of our system.

The STM32-Discovery board kit (showed in Figure 6.1) is a development platform for STMicroelectronics and ARM Cortex-M7 core-based STM32F746NG microcontroller. Table 6.1 shows the hardware specifics of the board. Specifically, the board includes a 4.3" RGB 480×272 color LCD-TFT with capacitive touch screen, two ST-MEMS digital microphones, 128 Mbit Quad-SPI Flash memory,

128 Mbit SDRAM (64 Mbits accessible), two user and reset push-buttons. The board also includes an ARM Cortex-M7 MCU, 1 MByte of flash memory and 340 Kbytes of RAM. Moreover, to expand board capabilities ST equipped it with audio in and out jacks, two USBs, one microSD slot, and an Arduino Uno expansion connector. Finally, the on-board ST-Link/V2-1 programmer enables virtual COM port, mass storage, and debug port functions. With respect to the power supply, the board can be powered from 3.6 to 12 V, thus facilitating its integration into the existing circuitry.

Table 6.1: STM32-Discovery board hardware specifics.

STM32-Discovery hardware	
Display	4.3" *RGB* 480*x*272 *color LCD TFT*
MCU	*ARM Cortex – M7*
RAM	340 KBytes
SDRAM	128 Mbits (64 *accessible*)
Flash	1 MByte



(a) Front. (b) Back.

Figure 6.1: STM32-Discovery board.

With respect to the software part, the board can be configured using the STM32CubeMX software that allows to manage the X-CUBE-AI[1] tool expansion package. This tool has the capability to automatically convert a pre-trained neural network model and inject it into a STM32 device creating an optimized library that is added to the users's project. X-CUBE-AI tool works with the most

[1]https://www.st.com/en/embedded-software/x-cube-ai.html, accessed October 2020

popular libraries today available such as: Keras, TensorFlow Lite, ONNX, Caffe, Lasagne, and ConvnetJS. However, the deployment of a neural network into an ICPS is challenging due to the limited memory and computations resources. In this sense, the X-CUBE-AI tool provides 8 bit quantization and compression techniques that allow to reduce the memory footprint of a model such that it can fit the hardware limitations of the STM32.

Compression technique

The X-CUBE-AI tool supports 3 different types of compression, namely: "none" which corresponds to a *target-factor* set to 1 (i.e., no compression is applied), x4, and x8. The compression technique is only applicable to fully connected (or dense) layers; such a choice derives from the fact that most part of the weights is concentrated in these layers. To compress the information, this technique analyzes each layer individually and applies a K-Means clustering where the number of centroids depends on the target-factor passed as input:

$$n_{centroids} = 2^{32/(target-factor)}. \tag{6.1}$$

Figure 6.2 shows an example of a compression where the blue dots are representative for the weights of a neural network dense layer, while the red ones are representative for the centroids computed by the K-Means according to equation 6.1.

Quantization technique

Other than TensorFlow Lite quantized models, X-CUBE-AI has also the capability to quantize models opportunely reshaped using the Command Line Interface (CLI) of the tool. Unlike the compression, quantization is applied to all network weights, biases, and activation functions, which are converted to a 8 bit precision. These are then mapped on an optimized C implementation of the supported kernels, thus improving the overall performance of the algorithm. In this sense, the

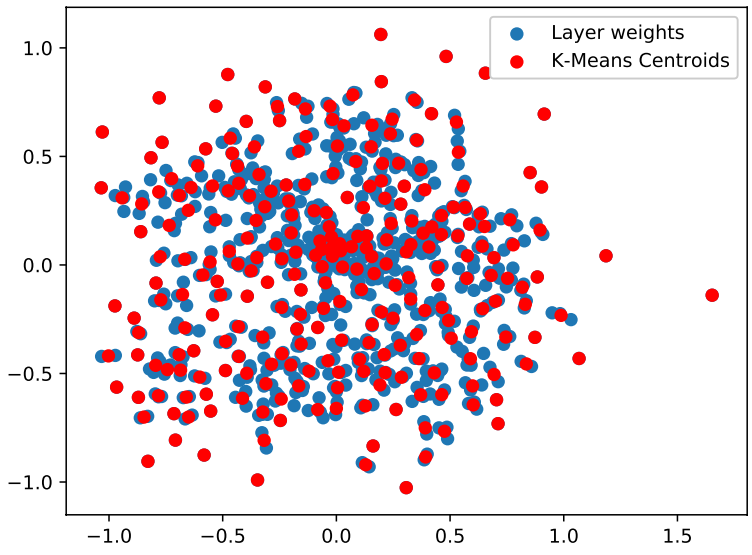


Figure 6.2: Compression technique using K-Means clustering.

goal of quantization is to reduce the model size while also improving CPU and hardware latencies with a little degradation in terms of the model accuracy.

Figure 6.3 depicts the quantization flow performed by the tool. In particular, the quantization flow consists of three phases: reshape model, quantize model, and predict with emulated quantized model. Since the Keras *.h5* files do not support quantized parameters, the first step consists in reshaping the model and creating a json file containing the tensor format representation. To do that, the *stm32ai* application provides a useful post-training quantization command that automatically generates the above mentioned files. After this phase, the tool allows to select one of the two possible quantization arithmetic, namely: fixed-point and integer. With respect to the first one, the fixed-point is obtained using the Qm,n format, a two's complement representation where the number of fraction bits n and integer bits m are specified with a fixed resolution. For the integer arithmetic, each real number r is represented as a function of the quantized value q, a scale factor s, and a zero-point parameter Z_p:

$$r = (q - Z_p) \cdot s. \tag{6.2}$$

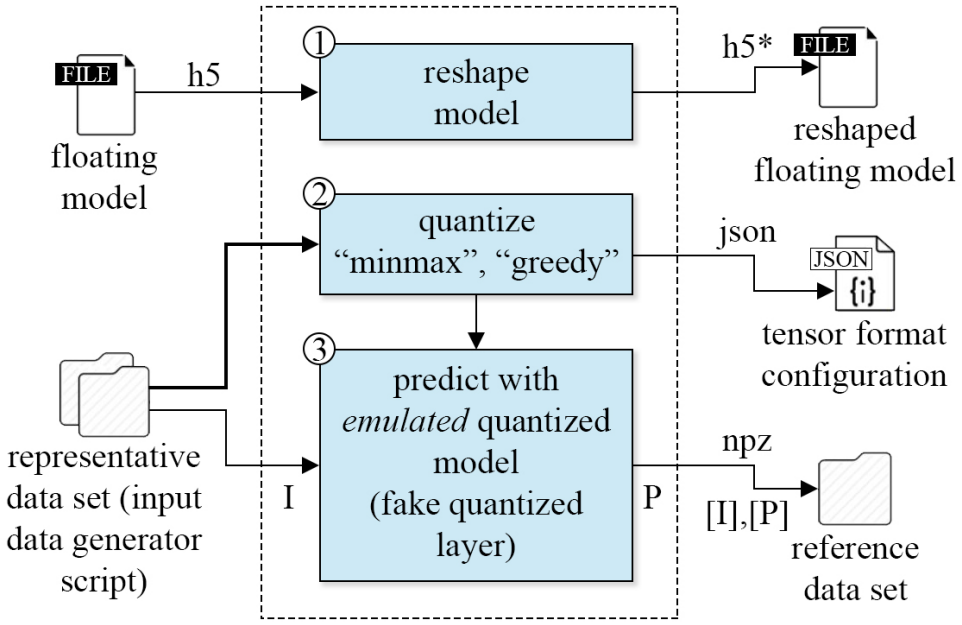


Figure 6.3: Quantization flow.

Once the quantization arithmetic is selected, it is possible to choose which type of algorithm should be used to find the model quantization parameters. Given a representative dataset (i.e., a portion of data which is representative for the training set of the model), the tool provides two options: *minmax* or *greedy*. In minmax, the algorithm performs a linear quantization, computing the maximum and the minimum values of the weights of the entire neural network in order to create an interval. Then, this interval is divided into 256 steps (as showed in Figure 6.4) where this number derives from the adopted 8 bit precision (i.e., $2^8 = 256$). According to the value of a weight, it is remapped into the "symbol" associated to the quantization step in which it falls. With respect to the greedy algorithm, it performs a layer by layer quantization using an iterative process through which it seeks for the best quantization parameters setup to maximize the model performance.

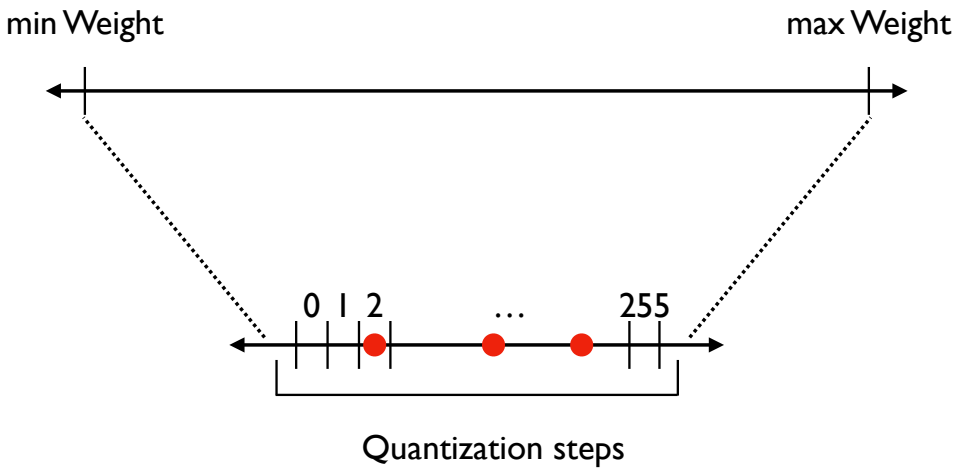


Figure 6.4: Minmax quantization.

Finally, the third and last phase is the prediction, during which the tool validates the quantized model using the representative dataset passed as input and generates in output its quantized version (i.e., the reference dataset) as a *numpy* npz file.

6.1.2 Deep Leaf detector

When working in plantations, the possibility of bringing a continuous internet connection may not be always guaranteed. To address this connectivity issue, the majority of the available solutions involve the use of technologies like LoRa to cover wide areas. However, if the one hand such techniques solve the problem related to the connection coverage, on the other they allow the transmission of very short message payloads, thus making very difficult the execution of complex tasks (e.g., sending images to the Cloud). Moreover, considering the large number of plants in a plantation, it makes ineffective the use of the Cloud computing paradigm which would require a huge amount of time and money to perform a constant monitoring and diagnosis.

In such a context, we propose an ICPS that exploits Edge computing to perform a real time early analysis of plants health condition, as a support to the

human operator during the diagnosis process.

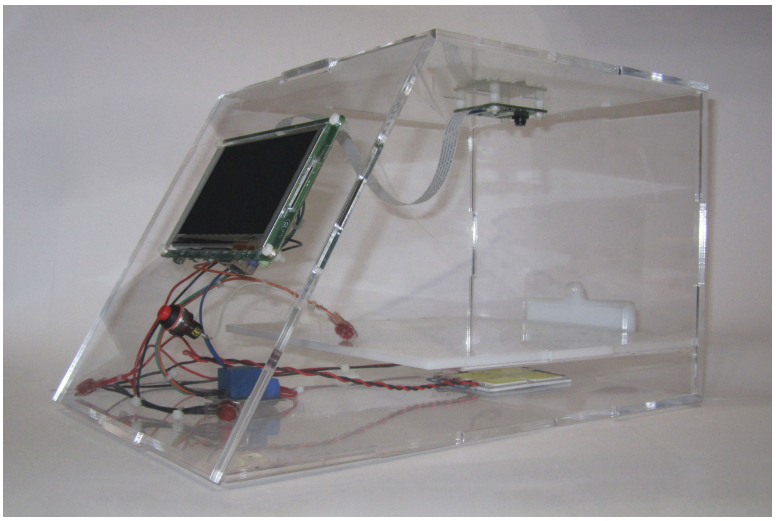
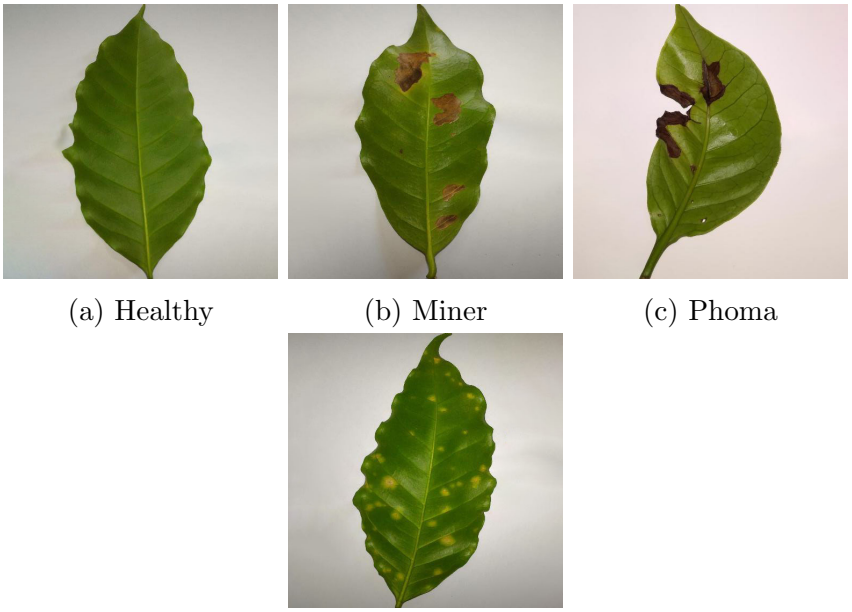


Figure 6.5: ICPS prototype based on STM32-Discovery board.

The main challenge we faced during the design and implementation of our solution was mainly related to the deployment of a software that can run effectively on the smart Edge, while maintaining the system performance in terms of accuracy. In the literature it is possible to find different techniques for plant disease detection, however, according to the hardware and the application context certain approaches could be preferred over the others [94]. For example, in [95] is proposed a plant disease detector based on RF. In this work, authors adopted a Histogram of Oriented Gradient (HOG) to perform the feature extraction process that are then passed to train the RF algorithm. However, the accuracy of this approach is very low (about 70%), in this sense our goal is to propose a technique with a much higher accuracy in inference process.

Authors in [96] present a plant disease detection system based on a CNN and a Learning Vector Quantization (LVQ) algorithm which reaches a very good accuracy (about 90%). Unfortunately, the algorithm implementation is unsuitable for the deployment on an Edge device which is one of the key aspects of our Deep Leaf detector.

The pruning technique described in [97] allows to compress and accelerate



(a) Healthy (b) Miner (c) Phoma

(d) Rust

Figure 6.6: Leaves belonging to all the different classes.

CNN models during the training and inference processes. Also in this case, the experimental results are very good both in terms of accuracy and inference time, however, depending on the hardware it may not be always possible to use a simple compression to fit the hardware constraints of a MCU.

Figure 6.5 shows the ICPS prototype we realized. The core element of our system is of course the STM32-Discovery board that we embedded in a poly-methylmethacrylate box together with a camera module and a LED matrix which are used to take pictures of the leaves and light them respectively. On top of this ICPS we run Deep Leaf an AI application able to detect the main biotic diseases affecting the plant coffee leaves through the analysis of their pictures.

The task of detecting plant health conditions has been tackled as a supervised machine learning approach. To this aim, the dataset for our application consists of pictures of healthy and diseased plants affected by one of the three typical biotic stresses of coffee plants (as showed in Figure 6.6 such as: *miner* representing a disease where larvae eat the leaf tissue, *phoma*, and *rust* representing two diseases

where a fungus deteriorates the leaf. From the original dataset[2], we extracted a total of 1,290 images of which 274 images belonging to the healthy class, 323 images to the miner class, 343 images to the phoma class, and 350 images to the rust class. The original images were 512x256 pixels with a ratio of 2:1, however the camera module attached to the Discovery board is only able to capture squared ratio images. For this reason, we performed a first preprocessing step to resize the images into a squared 224x224 1:1 ratio.

Then, we divided the images into training, validation, and test sets representing the 80%, 10%, and 10% of the original dataset, respectively. After this division, we noticed that the number of training and validation samples was too low, to overcome this problem we applied a data augmentation technique. Such a technique allows to generate new instances of the data passed as input by randomly applying different transformations such as movements, zooms, rotations, and skews [98].

Another problem was related to the low quality of the camera module which most of the times captured noisy pictures at a very poor resolution lower than the one available in the dataset. In this sense, to make the model more tolerant to noisy conditions, we performed another preprocessing technique on train and validation sets by randomly selecting a subset of images to which our algorithm applied one or more noise filters (i.e., Gaussian noise, Poisson noise, sparkle noise, vignette effect). As a result of this process, we obtained a larger number of training samples, while making the model more robust since it was trained also on images where "artificial noise" was added (as showed in Figures 6.7a and 6.7b).

At the end of the data augmentation process, we reached a total of 23,462 images for the training set and 3,057 images for the validation. For an easier visualization, we report in Table 6.2 the number of samples for the train, validation, and test sets.

[2]https://drive.google.com/file/d/15YHebAGrx1Vhv8-naave-R5o3Uo70jsm/view, accessed October 2020

(a) Vignette effect + Gaussian noise (b) Gaussian noise

Figure 6.7: Image examples with two different types of noise.

Table 6.2: Number of samples for training, validation, and test sets after the data augmentation process.

Class	Training	Validation	Test	Total images
Healthy	5,055	621	27	5,703
Miner	5,931	735	32	6,698
Phoma	6,297	781	34	7,112
Rust	6,179	920	40	7,139
Total	23,462	3,057	133	**26,652**

6.1.3 CNN architecture

For the classification of coffee plant pictures, we implemented a classifier based on a CNN whose structure is depicted in Figure 6.8.

From a topological point of view, a CNN consists of two parts. The first part, composed by convolution and pooling layers, is responsible to perform the feature extraction through which the network is able to learn the main internal structures of the images passed as input (e.g., lines, edges, shapes, etc.). The second part of the CNN makes the actual classification and consists of a sequence of fully connected layers terminated by a softmax, which outputs the most probable class associated to the input image. By taking a closer look to Figure 6.8, we can

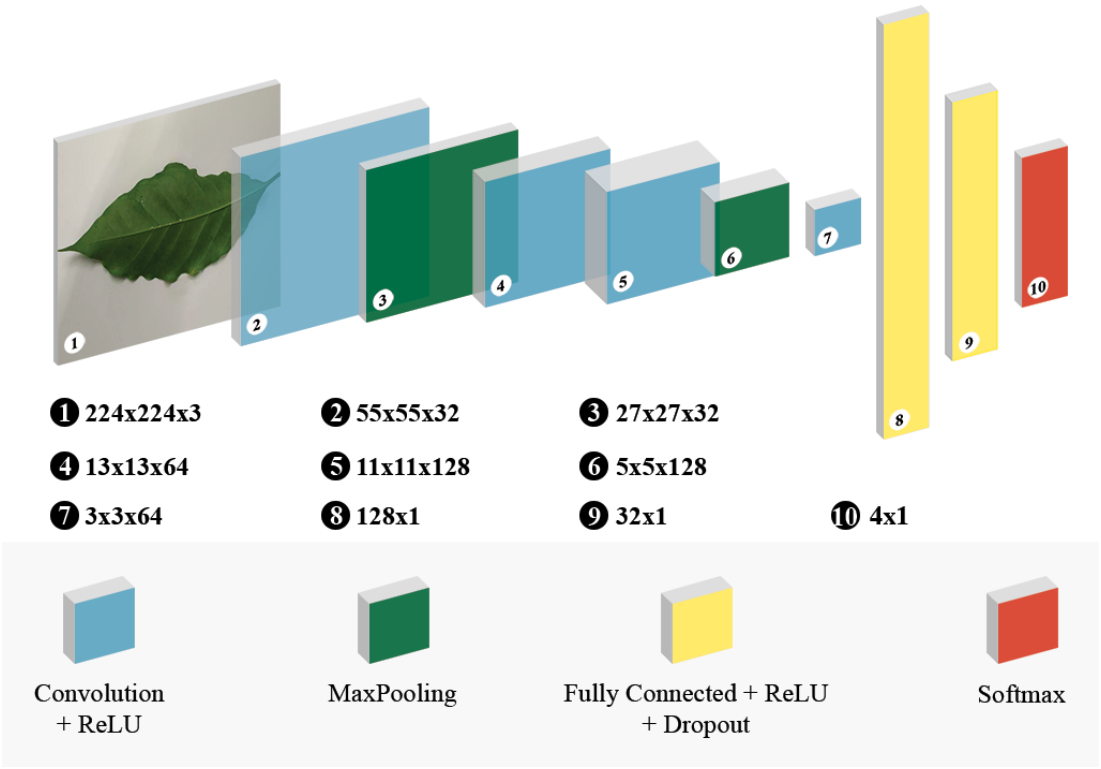


Figure 6.8: Complete structure of the CNN.

identify these two parts that we evidenced using different colors: blue and green for the feature extraction part which starts from the input layer (❶) and ends with convolution ❼; yellow and red for the classifier part starting from layer ❽ and ending with the softmax (❿).

The input layer ❶ is a 224x224 pixels RGB image, which passes through 4 convolution layers (i.e, ❷, ❹, ❺, and ❼) consisting of 32, 64, 128, and 64 filters, a kernel size of 8x8, 3x3, 3x3, and 3x3, respectively, with strides of size 4x4 and 2x2 applied only to the first two convolution layers. With respect to the pooling layers (i.e., ❸ and ❻), we adopted a max-pooling technique with pool size and strides of 2x2.

The classifier part is composed by two fully connected layers (i.e., ❽ and ❾) with 128 and 32 neurons, respectively. To avoid overfitting we used two dropout layers with 0.2 and 0.5 drop rates, together with an early stopping technique where we set a *patience term* to 30 epochs. Finally, the CNN terminates with a

softmax layer composed by 4 neurons (one for each class we want to classify).

Table 6.3 provides a summarized representation of the model parameters. We used ReLU as activation function, for each layer, *Stochastic Gradient Descent* (SGD) optimizer with a *learning rate* of 0.01 and we set the maximum number of training epochs to 150.

Table 6.3: Deep Leaf CNN parameters.

Deep Leaf CNN model architecture			
No. of layers	10	**Optimizer**	*SGD*
Activation function	*ReLU*	**Learning rate**	0.01
Pooling method	*Max-pooling*	**Patience**	30
Dropout rate	[0.5, 0.2]	**Max training epochs**	150

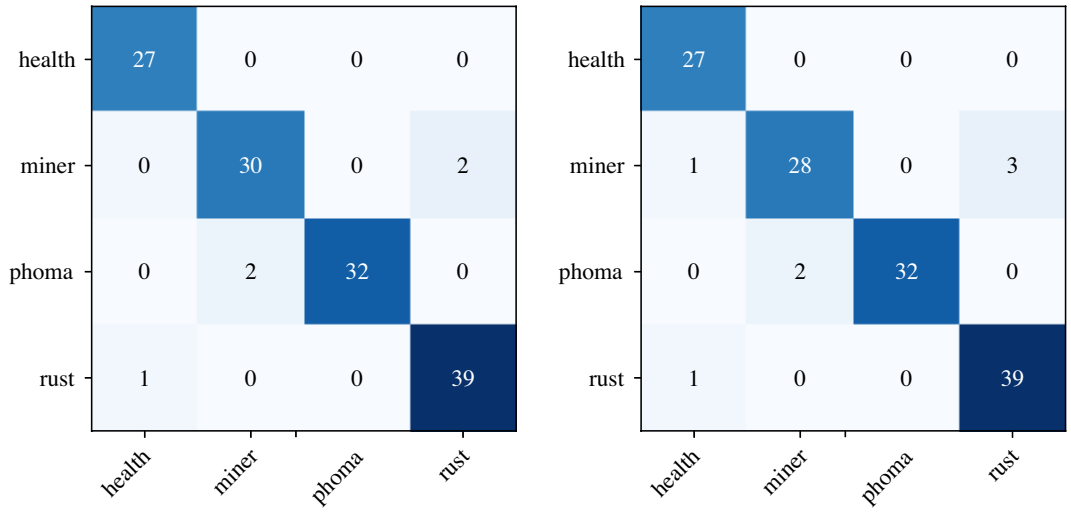
6.1.4 Quantitative analysis of Deep Leaf

In this paragraph we provide a quantitative analysis of five different models of Deep Leaf. Specifically, we considered the 32 bit original floating point model, its compressed version using a x8 target factor, a quantized one using the TensorFlow Lite converter, and the quantized models using the Qm,n and integer formats provided by the X-CUBE-AI tool. For each of the above mentioned models we computed the accuracy, precision, recall, and F_1-score, other than comparing the compressed and quantized models in terms of flash and RAM occupation, average inference time, and average energy consumption. The experimental results have been obtained deploying the models on the STM32-Discovery board using the train, validation, and test sets described in Table 6.2.

32 bit floating point model

The 32 bit floating point model achieved an overall accuracy of 96.24% on the test set. We also computed the confusion matrix (showed in Table 6.9a) from which we extracted the precision, recall, and F_1-score indices reported in Table 6.4. In general, the model performed very well reaching high values for all the

three indices, thus demonstrating its generalization capabilities. Using Keras we exported the model in a *.h5* file that occupied 2.1 MBytes on disk. However, as we explained in Paragraph 6.1.1 the STM32-Discovery board is equipped with only 1 MByte of flash memory, which makes impossible the deployment of the 32 bit model without applying a compression or quantization technique to reduce its memory footprint.



(a) Confusion matrix of 32-bit floating point, Compressed, TFlite quantized, and Integer quantized models.

(b) Confusion matrix of the classifier for Qm,n format quantized model.

Figure 6.9: Confusion matrices of the models.

Table 6.4: Precision, recall and F_1-score of the 32-bit floating point, Compressed, TFlite quantized, and Integer format quantized models.

Class	**Precision**	**Recall**	F_1**-score**
Health	0.96	1.00	0.98
Miner	0.94	0.94	0.94
Phoma	1.00	0.94	0.97
Rust	0.95	0.97	0.96

Compressed model

For the compressed model we tried both the available target-factors (i.e., x4 and x8). From our tests the performance in terms of accuracy of both the models were

comparable, for this reason we opted for a x8 compression model since it allows to have a higher reduction of the model memory occupation. After the compression, the model occupied 729.49 KBytes, however it could not still fit the MCU because the amount of required RAM (619.78 KBytes) was higher than the one available of 340 KBytes. To solve this problem, we configured the STM32-Discovery to provide the access to the SDRAM.

With respect to the performance, the confusion matrix computed on the test set resulted to be the same as the one showed in Figure 6.9a. Because of this condition, accuracy, precision, recall, and F_1-score (which are extracted from the matrix) resulted to be the equal to the ones obtained by the 32 bit floating point model. The *health* class reached values very close to 1 for precision, recall, and F_1-score, meaning that the model was able to correctly classify the leaves pictures belonging to this class. Results show an analogous condition also for the other classes i.e., *miner*, *phoma*, and *rust* where the proposed detector reached a precision of 0.94, 1.0, and 0.95, a recall of 0.94, 0.94, and 0.97, and a F_1-score of 0.94, 0.97, and 0.96 respectively.

As regards to the inference time, by running the model on each sample of the test set, it took on average $1,022.81$ ms with an average of $220,927,803$ CPU cycles and $31,593,500$ of *Multiply-and-ACCumulate* (MACC) operations considering a CPU clock speed of 216 MHz. In such a context, we introduce an useful metric represented as the ratio of the cycles/MACC which defines a model efficiency in terms of computing complexity (i.e., the smaller the ratio, the more efficient is the implementation). For the compressed model we obtained a cycles/MACC of 6.99.

The STM32CubeMX suite allows also to estimate the average energy consumption of a board according to its setup. In this sense, by specifying the board configuration (i.e., clock frequency and active peripherals), it is possible to estimate the average current consumption. For our application we enabled the following board peripherals: the Digital CaMera Interface (DCMI) to commu-

nicate with the camera module, the Direct Memory Access (DMA) and Direct Memory Access 2D (DMA2D) to manipulate the images, the LCD-TFT Display Controller (LTDC) to control the board display, and a timer (TM1). Moreover, because of the use of the external SDRAM, the Quad Serial Peripheral Interface (QSPI) is necessary for communication. According to this board configuration, the average absorbed current per millisecond (denoted as $\bar{i}$) was equal to 305.67 mA. In such a context, if we consider a supply voltage (V_{dd}) equal to 3.3 V and an average model inference time ($\bar{t}$) of 1022.81 ms, the average energy consumption ($\overline{E}$) for each inference is computed as:

$$\overline{E} = \bar{i} \cdot V_{dd} \cdot \bar{t}. \tag{6.3}$$

Plugging the above mentioned values into eq. (6.3) we obtained an average energy consumption of $1,031.72$ mJ for each inference. If we consider a scenario where the board is powered with a configuration of 3 Ni-MH(A2500) batteries (arranged in series), each with a capacity of $2,500$ mAh and a nominal voltage of 1.2 V, we obtain an expected lifetime of 2 days and 14 hours assuming that one inference is performed every minute.

TensorFlow lite quantized model

This model has been obtained through a quantization based on the TensorFLow Lite converter. This technique reduced the flash occupancy to 251.79 Kbytes and RAM to 33.25 KBytes, thus allowing the model to run without the help of the external SDRAM. In terms of performance, this model returned a confusion matrix that coincides with the ones of the above described models (i.e., floating point and compressed). This is a very interesting result because it means that we were able to reduce the memory footprint of the model (more than compression) while maintaining the same performance in terms of accuracy, precision, recall, and F_1-score.

Considering the inference time, this model took on average 364.44 ms which

is about a quarter of the time required by the compressed one with an average of 78,719,324 CPU cycles and 31,469,676 of MACC operations. In terms of cycles/MACC, also in this case the model obtained very good results with a value of 2.50 (about 3 times lower than the compressed model).

With respect to the power consumption, the model resulted very power efficient. In such a context, it was not necessary to enable the QSPI and SDRAM peripherals which have a huge energy impact. Assuming the same supply voltage V_{dd}, the average absorbed current per millisecond ($\bar{i}$) was 135.66 mA with an average energy consumption for each inference equal to 163.15 mJ (about 6.5 times less than the compressed model).

Qm,n format quantized model

The quantization using the Qm,n format representation is one of the techniques supported by the X-CUBE-AI tool. This method generated a model with 250.44 KBytes on flash and 32.35 KBytes on RAM occupation. In terms of performance, the model reached an accuracy of 95% which is slightly less that the one of the other models. Such a result is due to the Qm,n representation format which cause a higher loss of information during the DNN conversion with a consequent degradation of the performance. Figure 6.9b depicts the confusion matrix for the Qm,n model where we can notice a misclassification of 2 instances of the *miner* class which cause a reduction of precision, recall, and F_1-score (showed in Table 6.5). Nevertheless, the results are still very good with values ranging between 0.93 and 1.0 that are comparable with the ones showed in Table 6.4.

As far to the inference time, the model exhibited an average value of 299.60 ms with an average number of 64,712,955 CPU cycles, 31,593,508 MACC operations, and a cycles/MACC ratio of 2.05 which resulted to be the lowest values among the five models considered in this analysis. With respect to the hardware configuration, also in this case the SDRAM and QSPI peripherals were not necessary (like in the TensorFlow Lite quantized model). Hence, assuming the same

Table 6.5: Precision, recall and F_1-score of the classifier for Qm,n quantized model.

Class	Precision	Recall	F_1-score
Health	0.93	1.00	0.96
Miner	0.93	0.88	0.90
Phoma	1.00	0.94	0.97
Rust	0.93	0.95	0.97

Table 6.6: Models memory footprint

Model	Flash (KB)	RAM (KB)
Floating Point	-	-
Compressed	729.49	619.78
Quant. TFLite	251.79	33.25
Quant. Integer Format	251.79	32.60
Quant. Qm,n Format	250.44	32.35

average absorbed current (i.e., 135.66 mA) and V_{dd} (i.e., 3.3 V) and an average inference tme of 299.60 ms, we estimated an energy consumption for each inference of 134.12 mJ, which is again the lowest value.

Integer format quantized model

The final model we analyzed in this quantitative analysis is the integer format quantized model provided by the X-CUBE-AI tool. The model occupied 251.79 KBytes on flash and 32.60 KBytes on RAM and obtained the same accuracy and confusion matrix as the original floating point model. With respect to the inference time, the model required on average of 360.41 ms with an average of $77,848,890$ CPU cycles, $31,469,668$ MACC operations, and a cycles/MACC ratio equal to 2.47. Likewise the TensorFlow Lite and Qm,n quantized models, the integer one did not require the use the flash and QSPI peripherals, in this sense, assuming the same board setting configuration, the model returned an average energy consumption 161.34 mJ for each inference.

In Tables 6.6, 6.7, 6.8 we report the results we got from this analysis for each model, where the symbol "-" for the floating point model indicates an empty

Table 6.7: Models performance

Model	Inference Time (ms)	Energy Cons. (mJ)	Cycles/MACC
Floating Point	-	-	-
Compressed	1,022.81	1,031.72	6.99
Quant. TFLite	364.44	163.15	2.50
Quant. Integer Format	360.41	161.34	2.47
Quant. Qm,n Format	299.60	134.12	2.05

Table 6.8: Models accuracy

Model	Accuracy
Floating Point	0.96
Compressed	0.96
Quant. TFLite	0.96
Quant. Integer Format	0.96
Quant. Qm,n Format	0.95

value due to the impossibility to run this model on the board.

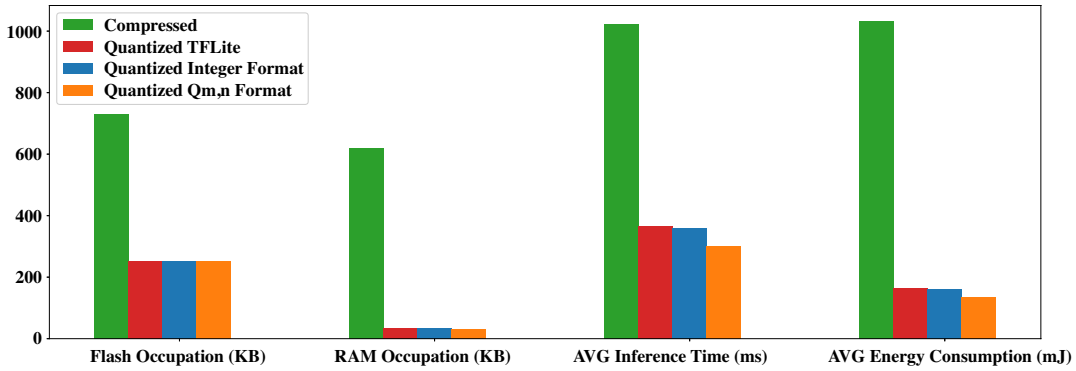


Figure 6.10: Histogram plot that compares the compressed and quantized models.

6.1.5 Lessons learned

By taking a closer look to Tables 6.6, 6.7, 6.8, we can derive interesting insights on the compression and quantization techniques explained above. In general the compression produced a model that reported the largest value of flash and RAM occupation, inference time and cycles/MACC ratio. In terms of prediction performance, the model maintained the same accuracy, precision, recall, and F_1-score

of the original 32 bit floating point one. Unfortunately the compressed model is not suitable to be used on the smart Edge due to its very high energy consumption which is a key aspect for a battery driven device. On the other hand, the quantized models using the TensorFlow lite converter and the X-CUBE-AI tool obtained very good results not only reducing the flash and RAM occupation, but also lowering the energy consumption (if compared to the compression technique) while maintaining (most of the times) the same performance. In particular the Qm,n format quantized model reached a slightly lower accuracy performance than the other models, however it reported the lowest values for flash and RAM, occupation, average inference time and energy consumption, and cycles/MACC ratio, thus resulting in the best trade-off among the five models. Figure 6.10 provides a visual representation of Tables 6.6, 6.7, 6.8 highlighting how the Qm,n model outperforms the others in almost every index.

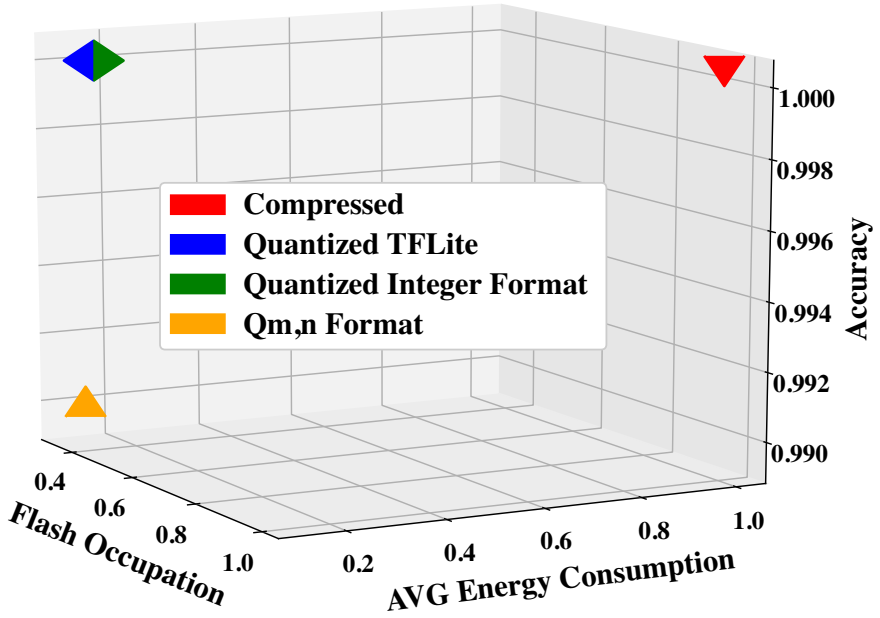


Figure 6.11: 3D scatter plot of the models putting in relationship flash occupation, accuracy and average energy consumption.

Figure 6.11 depicts a 3D scatter plot with respect to accuracy, flash occupation, and average energy consumption where we compare the compression and

quantization techniques. From the plot it is evident the effectiveness of the former over the latter however, it also puts in evidence that the performance of the quantization are heavily affected by several parameters (i.e., representation format, number of bits for the representation, quantization steps, etc,.). Moreover, such a technique requires the implementation of efficient algorithms to find the best set of quantization parameters which can take a lot of time. For example, the X-CUBE-AI tool took 5 hours to quantize our model, and in general this time tends to increase according to the DNN complexity.

The experimental results demonstrate the feasibility of the proposed ICPS and AI application that can be considered two valid support tools for the human operator to perform a timely recognition of plant diseases.

Chapter 7

Conclusions

The lessons learned from this thesis work are manifold. In general, the use of AI techniques for the five smart application contexts (i.e, home, city, industry, health, and agriculture) has been successful and allowed the realization of ICPS solutions that can help the human being during his daily life. In this sense, the core element of this work has been to demonstrate the potential of AI as an enabling technology, for the generation of advanced systems capable to make aware decisions thanks to their "reasoning" abilities.

On the other hand, from this experience emerged a very important aspect related to the implementation of this techniques. In particular, machine learning algorithms result to be very application specific, such that, the change of the context requires most of the times a complete model re-design. This is one of the main drawbacks of this technique where the strict relationship between the data and the model topologies makes it not suitable for a cross-domain usage.

Another consideration we can make on AI consists in the way it learns. Specifically, most of machine learning techniques today are implemented via a "black box" approach. Except for some cases, when working with these algorithms we do not have the total control of the learning process. For example, if we consider a DNN with a large number of layers, our awareness is restricted to the input we pass and to the output it generates. In this sense, the knowledge we have with

respect to the hidden layers about how and what they learn is very limited. If on the one hand, this property allows a fast model design, on the other it makes very difficult the debug phase which can require very long and time consuming processes.

The absence of rigorous design methods makes very hard the choice of the hyperparameters configuration. Today most part of the modelling is dictated by the experience and some "rules of thumb" which can be not sufficient. Sometimes the hyperparameters space is too large to be analyzed, such that the only way to procede is by using grid search techniques in order to find the most suitable setup. However, such a solution does not completely solve the problem for two orders of reason: *i.)* grid search methods can not seek for the entire hyperparameters space which must be limited to a subset. As a result of this condition, the solution found most of the times is sub-optimal. *ii.)* The hyperparameters validation can not be done a priori, but only during (or after) the training process, thus introducing a strong trial and error component. This is in part caused by the problem we mentioned above, which makes difficult to have a clear idea of the hyperparameters impact on each model component.

Another challenge is represented by the need of external feedbacks to enable the learning of complex tasks. Specifically, most of machine and deep learning applications require a supervised approach to perform the learning process. The absence of labelled data inevitably entails the use of an unsupervised approach which results much more limited than the supervised one. Such a condition poses a significant gap between the AI and human level intelligence which is able to learn complex concepts in an unsupervised way with a much lower number of samples [99]. This aspect derives from the supervised nature of the training algorithms (e.g., the backpropagation); in such a context emerges the need to migrate the learning towards a *deductive* reasoning via the development of new algorithms and training methods. Recent research works are starting to face these problems to enable the learning of disentangled representations of the reality [100] that are

at the base of the human intuition process.

The research challenges and problems here reported put in evidence that there is a lot of work to do in order to reach a human level AI. Despite huge steps have been made since the introduction of this technology, we are still living the eve of the AI. In this sense, if we want to reach a new level of intelligence we need to start building algorithms able to learn high level concepts abstractions (as the human does). These are the steps towards a future where "things" will start to have an advanced intelligence enabling an autonomous learning without the need of external teachers.

Bibliography

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions", *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[2] E. A. Lee, "Cyber physical systems: Design challenges", in *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, 2008, pp. 363–369.

[3] R. Baheti and H. Gill, "Cyber-physical systems", *The impact of control technology*, vol. 12, no. 1, pp. 161–166, 2011.

[4] P. Bellavista, J. Berrocal, A. Corradi, S. K. Das, L. Foschini, and A. Zanni, "A Survey on Fog Computing for the Internet of Things", *Pervasive and Mobile Computing*, vol. 52, pp. 71 –99, 2019, ISSN: 1574-1192.

[5] A. Imteaj and M. H. Amini, "Distributed sensing using smart end-user devices: Pathway to federated learning for autonomous iot", in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2019, pp. 1156–1161.

[6] D. Cogliati, M. Falchetto, D. Pau, M. Roveri, and G. Viscardi, "Intelligent cyber-physical systems for industry 4.0", in *2018 First International Conference on Artificial Intelligence for Industries (AI4I)*, 2018, pp. 19–22. DOI: 10.1109/AI4I.2018.8665681.

[7] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic, "Cyber-physical systems: The next computing revolution", in *Design Automation Conference*, 2010, pp. 731–736. DOI: 10.1145/1837274.1837461.

[8] S. Marsland, *Machine Learning: An Algorithmic Perspective, Second Edition*, 2nd. Chapman and Hall/CRC, 2014, ISBN: 1466583282.

[9] S. J. Russell and P. Norvig, *Artificial Intelligence - A Modern Approach (3. internat. ed.)* Pearson Education, 2010, ISBN: 978-0-13-207148-2. [Online]. Available: http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0136042597,00.html.

[10] B. Kröse, B. Krose, P. van der Smagt, and P. Smagt, *An introduction to neural networks*, 1993.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[12] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015, ISSN: 0028-0836.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[14] F. A. Gers, J. A. Schmidhuber, and F. A. Cummins, "Learning to forget: Continual prediction with lstm", *Neural Comput.*, vol. 12, no. 10, pp. 2451–2471, Oct. 2000, ISSN: 0899-7667. DOI: 10.1162/089976600300015015. [Online]. Available: http://dx.doi.org/10.1162/089976600300015015.

[15] A. Géron and a. O. M. C. Safari, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*. O'Reilly Media, Incorporated, 2019. [Online]. Available: https://books.google.it/books?id=O2VJzQEACAAJ.

[16] F. De Vita and D. Bruneo, "A deep learning approach for indoor user localization in smart environments", in *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2018, pp. 89–96. DOI: 10.1109/SMARTCOMP.2018.00078.

[17] Y. H. Wen, H. S. Chang, H. W. Kao, and G. H. Ju, "Location-aware services based on wi-fi network", in *The 16th Asia-Pacific Network Operations and Management Symposium*, 2014, pp. 1–4.

[18] A. K. Dey, G. D. Abowd, and D. Salber, "A context-based infrastructure for smart environments", in *Managing Interactions in Smart Environments*, London: Springer London, 2000, pp. 114–128, ISBN: 978-1-4471-0743-9.

[19] A. K. Bhattacharjee, D. Bruneo, S. Distefano, F. Longo, G. Merlino, and A. Puliafito, "Extending bluetooth low energy pans to smart city scenarios", in *IEEE SMARTCOMP*, 2017, pp. 1–6.

[20] R. N. Zahid Farid and M. Ismail, "Recent advances in wireless indoor localization techniques and system", *Journal of Computer Networks and Communications*, vol. vol. 2013,

[21] F. Zafari, A. Gkelias, and K. K. Leung, "A survey of indoor localization systems and technologies", *IEEE Communications Surveys Tutorials*, vol. 21, no. 3, pp. 2568–2599, 2019.

[22] Torres-Sospedra *et al.*, "Ujiindoorloc: A new multi-building and multi-floor database for wlan fingerprint-based indoor localization problems.", in *IPIN*, IEEE, 2014, pp. 261–270, ISBN: 978-1-4673-8054-6.

[23] D. Mascharka and E. D. Manley, "Machine learning for indoor localization using mobile phone-based sensors", *CoRR*, vol. abs/1505.06125, 2015. arXiv: 1505.06125.

[24] A. Salamah, M. Tamazin, M. Sharkas, and M. Khedr, "An enhanced wifi indoor localization system based on machine learning", Oct. 2016.

[25] M. Abadi *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015. [Online]. Available: `https://www.tensorflow.org/`.

[26] F. De Vita, G. Nardini, A. Virdis, D. Bruneo, A. Puliafito, and G. Stea, "Using deep reinforcement learning for application relocation in multi-access edge computing", *IEEE Communications Standards Magazine*, vol. 3, no. 3, pp. 71–78, 2019, ISSN: 2471-2833. DOI: `10.1109/MCOMSTD.2019.1900011`.

[27] F. De Vita and D. Bruneo, "Leveraging stack4things for federated learning in intelligent cyber physical systems", *Journal of Sensor and Actuator Networks*, under review.

[28] R. Dautov, S. Distefano, D. Bruneo, F. Longo, G. Merlino, and A. Puliafito, "Data processing in cyber-physical-social systems through edge computing", *IEEE Access*, vol. 6, pp. 29 822–29 835, 2018. DOI: `10.1109/ACCESS.2018.2839915`.

[29] V. Sciancalepore, F. Giust, K. Samdanis, and Z. Yousaf, "A double-tier mec-nfv architecture: Design and optimisation", in *2016 IEEE Conference on Standards for Communications and Networking (CSCN)*, 2016, pp. 1–6. DOI: `10.1109/CSCN.2016.7785157`.

[30] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration", *IEEE Communications Surveys Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017, ISSN: 1553-877X. DOI: `10.1109/COMST.2017.2705720`.

[31] E. G. M. V2.1.1, *Mobile edge computing (mec); framework and reference architecture*, 2019.

[32] M. V. *et al.*, "Human-level control through deep reinforcement learning", *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, ISSN: 00280836. [Online]. Available: `http://dx.doi.org/10.1038/nature14236`.

[33] P. Bellavista, A. Zanni, and M. Solimando, "A migration-enhanced edge computing support for mobile devices in hostile environments", in *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2017, pp. 957–962. DOI: 10.1109/IWCMC.2017.7986415.

[34] M. G. R. Alam, Y. K. Tun, and C. S. Hong, "Multi-agent and reinforcement learning based code offloading in mobile fog", in *2016 International Conference on Information Networking (ICOIN)*, 2016, pp. 285–290. DOI: 10.1109/ICOIN.2016.7427078.

[35] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Performance optimization in mobile-edge computing via deep reinforcement learning", in *2018 IEEE 88th Vehicular Technology Conference (VTC-Fall)*, 2018, pp. 1–6. DOI: 10.1109/VTCFall.2018.8690980.

[36] L. Huang, S. Bi, and Y.-J. A. Zhang, *Deep reinforcement learning for online offloading in wireless powered mobile-edge computing networks*, 2018. arXiv: 1808.01977 [cs.NI].

[37] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment", in *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, ser. Simutools '08, Marseille, France: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, 60:1–60:10, ISBN: 978-963-9799-20-2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1416222.1416290.

[38] A. Virdis, G. Stea, and G. Nardini, "Simulte - a modular system-level simulator for lte/lte-a networks based on omnet++", in *2014 International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, vol. 00, 2014, pp. 59–70. DOI: 10.5220/0005040000590070. [Online]. Available: doi.ieeecomputersociety.org/10.5220/0005040000590070.

[39] G. S. G. Nardini A. Virdis and A. Buono, "Simulte-mec: Extending simulte for multi-access edge computing", in *Proceedings of 5th International OMNeT++ Community Summit*, vol. 56, 2018, pp. 35–42. DOI: https://doi.org/10.29007/7g1p. [Online]. Available: https://easychair.org/publications/paper/VTGC.

[40] F. Chollet *et al.*, *Keras*, https://keras.io, 2015.

[41] Q. Li, Z. Wen, and B. He, "Federated learning systems: Vision, hype and reality for data privacy and protection", *ArXiv*, vol. abs/1907.09693, 2019.

[42] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications", *ACM Trans. Intell. Syst. Technol.*, vol. 10, no. 2, Jan. 2019, ISSN: 2157-6904. DOI: 10.1145/3298981. [Online]. Available: https://doi.org/10.1145/3298981.

[43] Y. Liu, J. J. Q. Yu, J. Kang, D. Niyato, and S. Zhang, "Privacy-preserving traffic flow prediction: A federated learning approach", *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7751–7763, 2020.

[44] X. Yao, T. Huang, C. Wu, R. Zhang, and L. Sun, "Towards faster and better federated learning: A feature fusion approach", in *2019 IEEE International Conference on Image Processing (ICIP)*, 2019, pp. 175–179. DOI: 10.1109/ICIP.2019.8803001.

[45] F. Longo, D. Bruneo, S. Distefano, G. Merlino, and A. Puliafito, "Stack4Things: A Sensing-and-Actuation-as-a-Service Framework for IoT and Cloud Integration", *Annals of Telecommunications*, vol. 72, no. 1, pp. 53–70, 2017, ISSN: 1958-9395.

[46] X. Zhu, J. Wang, Z. Hong, T. Xia, and J. Xiao, "Federated learning of unsegmented chinese text recognition model", in *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, 2019, pp. 1341–1345.

[47] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, T. Parcollet, and N. D. Lane, *Flower: A friendly federated learning research framework*, 2020. arXiv: 2007.14390 [cs.LG].

[48] G. Merlino, D. Bruneo, S. Distefano, F. Longo, and A. Puliafito, "Stack4things: Integrating iot with openstack in a smart city context", in *2014 International Conference on Smart Computing Workshops*, 2014, pp. 21–28. DOI: 10.1109/SMARTCOMP-W.2014.7046678.

[49] F. Sattler, S. Wiedemann, K. R. Müller, and W. Samek, "Robust and communication-efficient federated learning from non-i.i.d. data", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, no. 9, pp. 3400–3413, 2020.

[50] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtarik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency", in *NIPS Workshop on Private Multi-Party Machine Learning*, 2016. [Online]. Available: https://arxiv.org/abs/1610.05492.

[51] Z. Luo, F. Branchaud-Charron, C. Lemaire, J. Konrad, S. Li, A. Mishra, A. Achkar, J. Eichel, and P. Jodoin, "Mio-tcd: A new benchmark dataset for vehicle classification and localization", *IEEE Transactions on Image Processing*, vol. 27, no. 10, pp. 5129–5141, 2018.

[52] T. Nishio and R. Yonetani, "Client selection for federated learning with heterogeneous resources in mobile edge", in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019, pp. 1–7. DOI: 10.1109/ICC.2019.8761315.

[53] A. Reisizadeh, A. Mokhtari, H. Hassani, A. Jadbabaie, and R. Pedarsani, "Fedpaq: A communication-efficient federated learning method with periodic averaging and quantization", in *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, S. Chiappa and R. Calandra, Eds., ser. Proceedings of Machine Learning Re-

search, vol. 108, Online: PMLR, 2020, pp. 2021–2031. [Online]. Available: http://proceedings.mlr.press/v108/reisizadeh20a.html.

[54] D. Bruneo and F. De Vita, "On the use of lstm networks for predictive maintenance in smart industries", in *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2019, pp. 241–248. DOI: 10.1109/SMARTCOMP.2019.00059.

[55] F. De Vita, D. Bruneo, and S. K. Das, "A novel data collection framework for telemetry and anomaly detection in industrial iot systems", in *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 2020, pp. 245–251.

[56] F. De Vita, D. Bruneo, and S. K. Das, "On the use of a full stack hardware/software infrastructure for sensor data fusion and fault prediction in industry 4.0", *Pattern Recognition Letters*, vol. 138, pp. 30 –37, 2020, ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2020.06.028. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167865520302464.

[57] O. Aydin and S. Guldamlasioglu, "Using lstm networks to predict engine condition on large scale data processing framework", in *2017 4th International Conference on Electrical and Electronic Engineering (ICEEE)*, 2017, pp. 281–285. DOI: 10.1109/ICEEE2.2017.7935834.

[58] S. Bianchi, R. Paggi, G. L. Mariotti, and F. Leccese, "Why and when must the preventive maintenance be performed?", in *2014 IEEE Metrology for Aerospace (MetroAeroSpace)*, 2014, pp. 221–226. DOI: 10.1109/MetroAeroSpace.2014.6865924.

[59] X. Si, W. Wang, C.-H. Hu, and D. Zhou, "Remaining useful life estimation - a review on the statistical data driven approaches", *European Journal of Operational Research*, vol. 213, pp. 1–14, 2011.

[60] D. Dong, X. Li, and F. Sun, "Life prediction of jet engines based on lstm-recurrent neural networks", in *2017 Prognostics and System Health Management Conference (PHM-Harbin)*, 2017, pp. 1–6. DOI: 10.1109/PHM.2017.8079264.

[61] V. Mathew, T. Toby, V. Singh, B. M. Rao, and M. G. Kumar, "Prediction of remaining useful lifetime (rul) of turbofan engine using machine learning", in *2017 IEEE International Conference on Circuits and Systems (ICCS)*, 2017, pp. 306–311. DOI: 10.1109/ICCS1.2017.8326010.

[62] A. Kanawaday and A. Sane, "Machine learning for predictive maintenance of industrial machines using iot sensor data", in *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 2017, pp. 87–90.

[63] A. Saxena and K. Goebel, *Turbofan engine degradation simulation data set*, NASA Ames Research Center, Moffett Field, CA: NASA Ames Prognostics Data Repository, 2008. [Online]. Available: http://ti.arc.nasa.gov/project/prognostic-data-repository.

[64] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting", *J. Mach. Learn. Res.*, vol. 15, no. 1, 1929–1958, Jan. 2014, ISSN: 1532-4435.

[65] C. Vallati, S. Brienza, G. Anastasi, and S. K. Das, "Improving Network Formation in 6TiSCH Networks", *IEEE Transactions on Mobile Computing*, vol. 18, no. 1, pp. 98–110, 2019. DOI: 10.1109/TMC.2018.2828835.

[66] E. Sisinni, A. Saifullah, S. Han, U. Jennehag, and M. Gidlund, "Industrial Internet of Things: Challenges, Opportunities, and Directions", *IEEE Transactions on Industrial Informatics*, vol. 14, no. 11, pp. 4724–4734, 2018. DOI: 10.1109/TII.2018.2852491.

[67] M. A. O. Pessoa, M. A. Pisching, L. Yao, F. Junqueira, P. E. Miyagi, and B. Benatallah, "Industry 4.0, how to integrate legacy devices: A cloud iot approach", in *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, 2018, pp. 2902–2907. DOI: 10.1109/IECON.2018.8592774.

[68] R. Y. Zhong, X. Xu, E. Klotz, and S. T. Newman, "Intelligent Manufacturing in the Context of Industry 4.0: A Review", *Engineering*, vol. 3, no. 5, pp. 616 –630, 2017, ISSN: 2095-8099.

[69] V. Kulik and R. Kirichek, "The Heterogeneous Gateways in the Industrial Internet of Things", in *10th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, 2018, pp. 1–5. DOI: 10.1109/ICUMT.2018.8631232.

[70] S. Uludag, K. Lui, W. Ren, and K. Nahrstedt, "Secure and scalable data collection with time minimization in the smart grid", *IEEE Transactions on Smart Grid*, vol. 7, no. 1, pp. 43–54, 2016, ISSN: 1949-3061. DOI: 10.1109/TSG.2015.2404534.

[71] G. Campobello, M. Castano, A. Fucile, and A. Segreto, "Weva: A complete solution for industrial internet of things", in *Ad-hoc, Mobile, and Wireless Networks*, Cham: Springer International Publishing, 2017, pp. 231–238, ISBN: 978-3-319-67910-5.

[72] P. Ferrari, S. Rinaldi, E. Sisinni, F. Colombo, F. Ghelfi, D. Maffei, and M. Malara, "Performance Evaluation of Full-cloud and Edge-cloud Architectures for Industrial IoT Anomaly Detection based on Deep Learning", in *2019 II Workshop on Metrology for Industry 4.0 and IoT (MetroInd4.0 IoT)*, 2019, pp. 420–425. DOI: 10.1109/METROI4.2019.8792860.

[73] A. Khandelwal, I. Agrawal, M. I, S. S. Ganesh, and R. Karothia, "Design and Implementation of an Industrial Gateway: Bridging Sensor Networks into IoT", in *3rd International Conference on Electronics, Communication*

and Aerospace Technology, 2019, pp. 1–4. DOI: 10.1109/ICECA.2019.8821883.

[74] T. Addabbo, A. Fort, M. Mugnaini, L. Parri, S. Parrino, A. Pozzebon, and V. Vignoli, "An IoT Framework for the Pervasive Monitoring of Chemical Emissions in Industrial Plants", in *Workshop on Metrology for Industry 4.0 and IoT*, 2018, pp. 269–273. DOI: 10.1109/METROI4.2018.8428325.

[75] C. H. Park, "Anomaly Pattern Detection on Data Streams", in *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2018, pp. 689–692. DOI: 10.1109/BigComp.2018.00127.

[76] T. P. Banerjee and S. Das, "Multi-sensor data fusion using support vector machine for motor fault detection", *Information Sciences*, vol. 217, pp. 96 –107, 2012, ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2012.06.016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025512004185.

[77] A. Diez-Olivan, J. Del Ser, D. Galar, and B. Sierra, "Data fusion and machine learning for industrial prognosis: Trends and perspectives towards industry 4.0", *Information Fusion*, vol. 50, pp. 92 –111, 2019, ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2018.10.005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1566253518304706.

[78] S. R. Saufi *et al.*, "Challenges and opportunities of deep learning models for machinery fault detection and diagnosis: A review", *IEEE Access*, vol. 7, pp. 122 644–122 662, 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2938227.

[79] M. S. Kim *et al.*, "Unsupervised anomaly detection of lm guide using variational autoencoder", in *2019 11th International Symposium on Advanced Topics in Electrical Engineering (ATEE)*, 2019, pp. 1–5. DOI: 10.1109/ATEE.2019.8724998.

[80] E. Elahi, S. Kanwal, and A. N. Asif, "A new ensemble approach for software fault prediction", in *2020 17th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, 2020, pp. 407–412.

[81] G. Cicceri, F. De Vita, D. Bruneo, G. Merlino, and A. Puliafito, "A deep learning approach for pressure ulcer prevention using wearable computing", *Human-centric Computing and Information Sciences*, vol. 10, 2020. DOI: 10.1186/s13673-020-0211-8. [Online]. Available: https://doi.org/10.1186/s13673-020-0211-8.

[82] M. Kaşıkçı, M. Aksoy, and E. Ay, "Investigation of the prevalence of pressure ulcers and patient-related risk factors in hospitals in the province of erzurum: A cross-sectional study", *Journal of Tissue Viability*, vol. 27, no. 3, pp. 135 –140, 2018, ISSN: 0965-206X. DOI: https://doi.org/10.1016/j.jtv.2018.05.001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0965206X17301213.

[83] C. Díaz, B. Garcia-Zapirain, C. Castillo, D. Sierra-Sosa, A. Elmaghraby, and P. J. Kim, "Simulation and development of a system for the analysis of pressure ulcers", in *2017 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, 2017, pp. 453–458. DOI: 10.1109/ISSPIT.2017.8388686.

[84] D. Hayn, M. Falgenhauer, J. Morak, K. Wipfler, V. Willner, W. Liebhart, and G. Schreier, "An ehealth system for pressure ulcer risk assessment based on accelerometer and pressure data", *J. Sensors*, vol. 2015, 106537:1–106537:8, 2015.

[85] Z. Zhong and Y. Li, "A recommender system for healthcare based on human-centric modeling", in *2016 IEEE 13th International Conference on e-Business Engineering (ICEBE)*, 2016, pp. 282–286. DOI: 10.1109/ICEBE.2016.055.

[86] F. Hu, D. Xie, and S. Shen, "On the application of the internet of things in the field of medical and health care", in *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, 2013, pp. 2053–2058. DOI: 10.1109/GreenCom-iThings-CPSCom.2013.384.

[87] D. Sen, J. McNeill, Y. Mendelson, R. Dunn, and K. Hickle, "A new vision for preventing pressure ulcers: Wearable wireless devices could help solve a common-and serious-problem", *IEEE Pulse*, vol. 9, no. 6, pp. 28–31, 2018, ISSN: 2154-2317. DOI: 10.1109/MPUL.2018.2869339.

[88] M. Chang, T. Yu, J. Luo, K. Duan, P. Tu, Y. Zhao, N. Nagraj, V. Rajiv, M. Priebe, E. A. Wood, and M. Stachura, "Multimodal sensor system for pressure ulcer wound assessment and care", *IEEE Transactions on Industrial Informatics*, vol. 14, no. 3, pp. 1186–1196, 2018, ISSN: 1941-0050. DOI: 10.1109/TII.2017.2782213.

[89] T. Y. Wang, S. L. Chen, H. C. Huang, S. H. Kuo, and Y. J. Shiu, "The development of an intelligent monitoring and caution system for pressure ulcer prevention", in *2011 International Conference on Machine Learning and Cybernetics*, vol. 2, 2011, pp. 566–571. DOI: 10.1109/ICMLC.2011.6016779.

[90] H. Nisar, A. R. Malik, M. Asawal, and H. M. Cheema, "An electrical stimulation based therapeutic wearable for pressure ulcer prevention", in *2016 IEEE EMBS Conference on Biomedical Engineering and Sciences (IECBES)*, 2016, pp. 411–414. DOI: 10.1109/IECBES.2016.7843483.

[91] F. De Vita, G. Nocera, D. Bruneo, V. Tomaselli, D. Giacalone, and S. K. Das, "Quantitative analysis of deep leaf: A plant disease detector on the smart edge", in *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2020, pp. 49–56. DOI: 10.1109/SMARTCOMP50058.2020.00027.

[92] R. Varenne, J. M. Delorme, E. Plebani, D. Pau, and V. Tomaselli, "Intelligent recognition of tcp intrusions for embedded micro-controllers", in *New Trends in Image Analysis and Processing – ICIAP 2019*, M. Cristani, A. Prati, O. Lanz, S. Messelodi, and N. Sebe, Eds., Cham: Springer International Publishing, 2019, pp. 361–373, ISBN: 978-3-030-30754-7.

[93] L. Lai, N. Suda, and V. Chandra, *Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus*, *arXiv* preprint, 2018. arXiv: 1801.06601 [cs.NE].

[94] Y. Guo, "A survey on methods and theories of quantized neural networks", *ArXiv*, vol. abs/1808.04752, 2018.

[95] S. Ramesh *et al.*, "Plant disease detection using machine learning", in *2018 International Conference on Design Innovations for 3Cs Compute Communicate Control (ICDI3C)*, 2018, pp. 41–45. DOI: 10.1109/ICDI3C.2018.00017.

[96] M. Sardogan *et al.*, "Plant leaf disease detection and classification based on cnn with lvq algorithm", in *2018 3rd International Conference on Computer Science and Engineering (UBMK)*, 2018, pp. 382–385. DOI: 10.1109/UBMK.2018.8566635.

[97] J. Luo *et al.*, "Thinet: A filter level pruning method for deep neural network compression", in *IEEE International Conference on Computer Vision*, 2017, pp. 5068–5076. DOI: 10.1109/ICCV.2017.541.

[98] A. Mikołajczyk and M. Grochowski, "Data augmentation for improving deep learning in image classification problem", in *2018 international interdisciplinary PhD workshop (IIPhDW)*, IEEE, 2018, pp. 117–122.

[99] Y. Bengio, A. C. Courville, and P. Vincent, "Unsupervised feature learning and deep learning: A review and new perspectives", *CoRR, abs/1206.5538*, vol. 1, p. 2012, 2012.

[100] Y. Bengio, G. Mesnil, Y. Dauphin, and S. Rifai, "Better mixing via deep representations", in *International conference on machine learning*, 2013, pp. 552–560.