POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico
di Torino

Master's Degree Thesis

A Neural Network Application For Impedance-based Plant Monitoring: From A Development Framework Towards Edge Computing

Supervisors

Prof. Maurizio MARTINA

Prof. Danilo DEMARCHI

Ph.D. Umberto GARLANDO

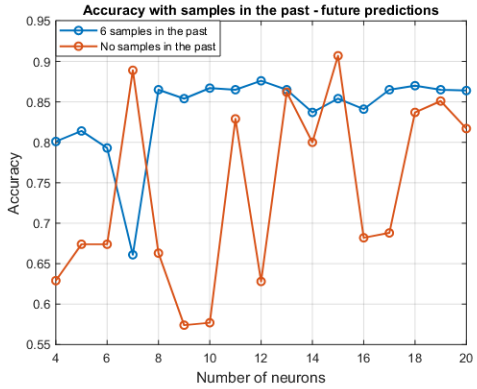
Candidate

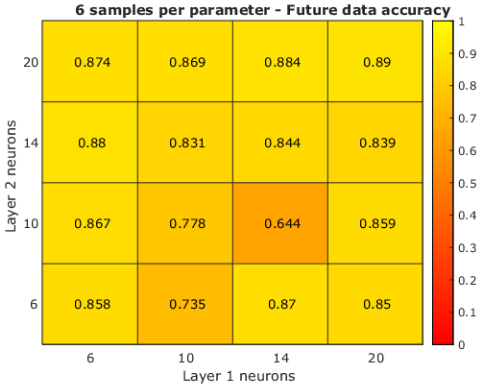
Federico CUM

October 2022

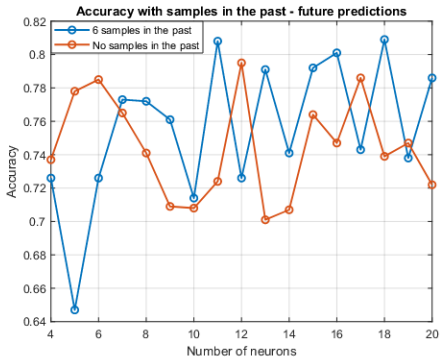
Summary

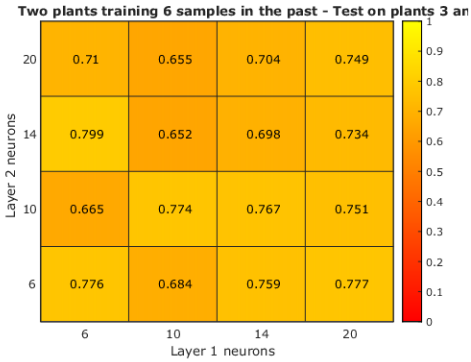
During the 21st century, the world is experiencing the most evident effects of climate change, such as desertification, rising temperatures, higher frequency of extreme meteorological events, rising sea levels, and lack of potable water. The world population is constantly growing and is expected to reach almost 10 billion people by 2050. In this scenario, agriculture is severely challenged due to the increasing demand for food and the worsening of environmental conditions: new techniques are needed to improve the yield of plantations by saving as many valuable resources, such as water and energy, as possible. The field of smart agriculture is working in this direction, and it is developing technologies that can make farming more efficient and autonomous. This thesis work aims to find suitable neural network topologies to predict a plant's status. Many techniques in the scientific literature employ machine learning on images to detect diseases or sufferance of the plants; in this work, the focus will be on using data from stem impedance sensors to accomplish this task. MINES researchers at Politecnico di Torino in collaboration with the University of Tel Aviv, have put great effort into the development of sensors that measure plants' stem impedance and also in understanding how this parameter is related to the life cycle and health of the plant [1] [2] [3] [4]. In this thesis, neural networks employing as features the environmental and stem impedance data are explored to predict the health status of a plant. When developing a machine learning model, the parameters to tune to make it work effectively are many, so many simulations must be performed. A Python framework for plant status classification was developed to ease this research process and compare many neural network structures. Since the final goal is to have an algorithm running on a microcontroller, the research was focused on networks with a limited complexity: one or two hidden layers with maximum 20 neurons per hidden layer. The dataset was composed of four plants, two considered healthy and two unhealthy. First, samples from all four plants were used to train the networks, and accuracy tests were performed on future data of the same plants for both one and two hidden layers networks.





Then, only two plants, one healthy and one unhealthy, were employed for training, and the test was performed on the other two. In both cases, the best results were obtained by employing six samples in the past for each feature considered.





In the last part of this work, some tests on a microcontroller were performed to understand the resources used to run these learning algorithms. The tool X-Cube-AI from STMicroelectronics was used to automatically create a ready-to-use C library to implement the inference process on an STM32 microcontroller. A model with one hidden layer with 10 neurons and another with two hidden layers with 20 and 14 neurons were loaded on the microcontroller to perform a simple benchmark.

	1 hidden layer model	2 hidden layers model
Network flash usage	448 B	1.97 KiB
Library flash usage	11.16 KiB	11.86 KiB
Total flash usage	11.59 KiB	13.86 KiB
Network RAM usage	72 B	136
Library RAM usage	1.38 KiB	1.88 KiB
Total RAM usage	1.45 KiB	2.01 KiB
Complexity	122 MACC	538 MACC
Inference time	0.016 ms	0.043 ms
CPU Cycles	2872	7798
Cycles/MACC	23.55	14.50
Accuracy	73.77 %	73.68%

The performance reached by the analyzed neural networks suggests that using impedance and environmental data to predict plants' status is possible. However, it is necessary to increase the prediction accuracy to obtain a reliable system to be, in the future, deployed on the field. One option for future improvements could be to extend the dataset to more than four plants; in this way, the networks can learn from a broader set of examples and, hopefully, improve the classification capability on completely unseen plants.

Table of Contents

List of Tables

List of Figures

Acronyms

MiNES
Micro & Nano Electronic Systems

ML
Machine Learning

NN
Neural Network

AI
Artificial Intelligence

SVM
Support Vector Machine

SGD
Stochastic Gradient Descent

ONNX
Open Neural Network eXchange

CSV
Comma Separated Values

RMSE
Root Mean Square Error

MCC
Matthews Correlation Coefficient

CEL

Cross Entropy Loss

Chapter 1

Introduction

1.1 Thesis structure

This thesis will follow the workflow adopted in this project and will be divided in three main parts:

- Development of a machine learning framework to easily train different neural network topologies
- Analyze and determine which topology performs better for the problem
- Find a proper toolchain for a microcontroller implementation of the algorithm

Since many simulations have to be done in order to understand if, and with which topologies, neural networks can be used to effectively predict health status of the plants there was the necessity to easily train a great number of neural networks in an automatic way. So, it was decided to develop a machine learning framework to ease this process. To do this Python was employed in particular the ML library PyTorch [5]. The Python framework just mentioned was initially developed by a previous student in his master thesis but was able to only perform regression-like predictions. Since in this study the goal is to classify plants based on their health there was the need of add this functionality and make the framework able to perform classification tasks. Considering that the developed piece of software was a valuable job it was decided to just modify it by adding the needed functionalities. Once this software had been developed, an instrument to easily train multiple neural networks was available. So an extensive analysis was performed to understand how these algorithms can be used in our case study. In the last part of the thesis, a package from STMicroelectronics, called X-Cube-AI, is used to load and benchmark on an STM32 microcontroller the previously trained machine learning models.

Chapter 2

Machine Learning Basics

Before diving into the details of the project it's necessary to give a brief introduction about machine learning, what is it and what are the concepts behind this innovative technology. Although the terms Artificial Intelligence (AI) and Machine Learning (ML) are often interchanged they are not the same concept: AI is the broader concept of having machines that learn in a way similar to the one of humans while ML is a wing of AI that groups the techniques that enable computers to learn new skills from data. This technology is becoming more and more popular in these years, however its theorization is quite old: in 1959 the AI pioneer Arthur Samuel wrote a program able to learn and improve at the game of checkers by playing thousands of games; he also gave an informal definition of ML: "Field of study that gives computers the ability to learn without being explicitly programmed". This definition sums up well a key characteristic of ML that differentiate it from the classical programming that is the explicit programming. In classical programming specific instructions are executed in an IF-THEN structure that is to say that they are executed only if certain conditions are met. On the other hand when dealing with ML we have automated process that allows computer to learn a new task based on past data related to the problem under analysis [6]. In 1998 the American computer scientist Tom Mitchell gave a more formal definition of machine learning: "A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on P, as measured by P, improves with experience E". Considered that the studies about ML are quite old, why did these techniques take so much time to become popular? Obviously researches have done big steps but back in the years the bottlenecks of ML were the availability of memory to store huge amounts of data and the computational power required by this algorithms to work efficiently. Nowadays memory and computational power have become more and more cheap and the quantity of data available is huge. In this evolved scenario machine learning have found the ideal conditions to be effectively used in modern applications [7].

2.1 Types of learning algorithms

Many ML algorithms exist and each one has its performance depending on the application. They can be grouped in some macro-categories:

- Supervised learning
- Unsupervised learning
- Semi-supervised learning
- Reinforcement learning

Supervised learning

In supervised learning a model is trained by giving it input data and expected outcomes, this way the model can learn over time by adjusting its predictions. In this case it is said that the model is fed with a *labeled dataset*. The performance of the algorithm is measured by the so called *loss function* which serves also to adjust the weights of the algorithm to get more accurate on future predictions. We can provide a simple example: suppose you want to predict housing prices given their size in feet square. The dataset will be composed by a single feature, the size of the house, and by the price of it (the expected prediction outcome). Referring to figure 2.1, one technique can be to find a function which approximate the price, this way a prediction can be done on new data which are not present in the original dataset (i.e. forecast new house prices). In the below example two possible approximations are displayed: linear and quadratic. The process of training a learning algorithm will consist in finding proper weights to get these approximating functions. In the case of a linear approach the training process will find the parameters θ_1 and θ_2 such that $price = \theta_1 \cdot size + \theta_2$ and this price must be as similar as possible to the actual one. To quantify how much the prediction is similar to the real price and adjust θ_1 and θ_2 a mathematical function, the *loss function* will be used. More information on how this process works will be given in next sections.

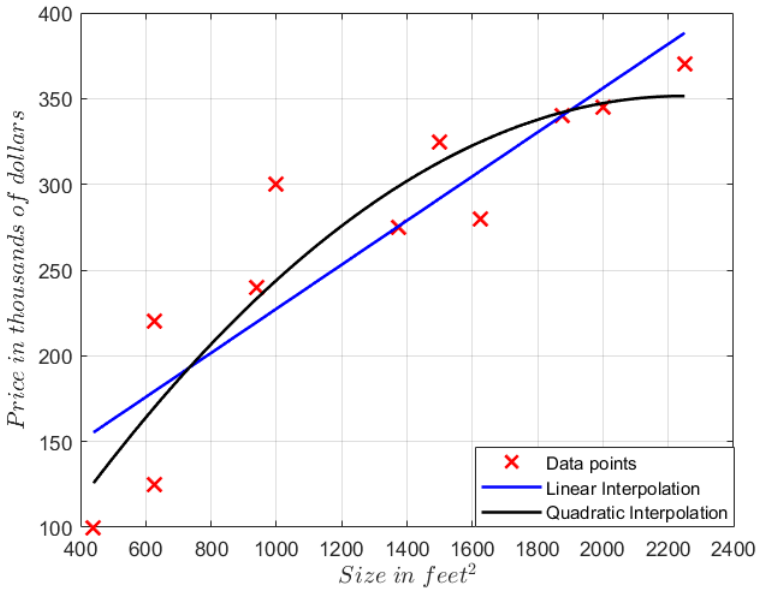


Figure 2.1: Approximations of housing prices

Unsupervised learning

In unsupervised learning data given to the algorithm during the training phase are not labeled and the machine is let by itself finding patterns, understanding how these data are correlated or grouping them. Rivedi sta frase perchè mi fa schifo. An example of unsupervised learning can be *cluster algorithms*: suppose we have two features x_1 and x_2 we would like to find a structure and group similar data together. In figure 2.2 is possible to observe how a learning algorithm could group data by their similarity in two clusters. One might think that this separation of the data is quite evident from the graph but in this simple example we had considered only two features, with more of them the graph would have been multi-dimensional and finding correlation would have been impossible by just plotting data and visually analyzing them.

Semi-supervised learning

In semi-supervised learning, only a small portion of the dataset is labeled, and a bigger portion is left unlabelled. This approach is employed when labeling the training examples is complicated or highly time-consuming.

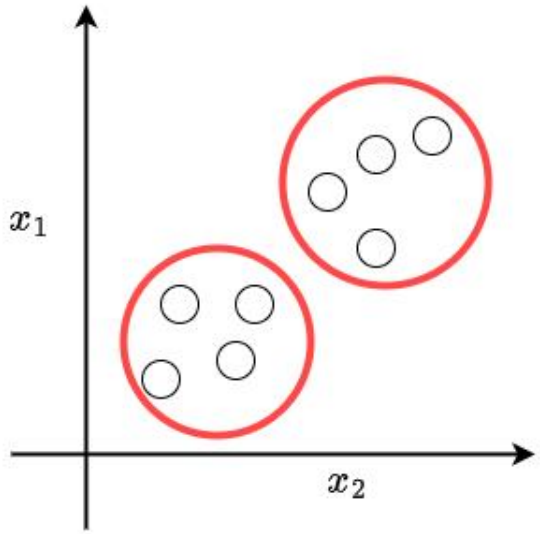


Figure 2.2: Unsupervised learning example: clustering

Reinforcement learning

Reinforcement learning refers to the training of models able to perform a sequence of decisions. For this task, the algorithm learns by trial and error, and a reward mechanism drives the learning process: when the algorithm gets closer to the desired behavior, it gets a reward; on the other hand, it is penalized. Examples of these models are self-driving cars or applications where a computer has to learn to play a game.

2.2 Classification vs Regression

In previous sections, we saw a possible predictive model for housing prices; these types of prediction are called *regression problems*. In this case, the variable to be guessed can vary on a specific range in a *continuous* way. Some examples of regression problems can be housing price predictions, stock market trends, predictions of a day temperature given other environmental parameters, and many others. On the other hand, we refer to *classification problems* whenever the variable to be predicted can assume only a discrete set of values. For example, the study in this thesis work aims to classify plants based on their health status, which can assume only two values: 1 = healthy, 0 = unhealthy. Another example can be handwritten digits recognition: there are ten classes which are the possible digits 0-9, and the ML model has to pick the correct one or also object recognition in which the algorithm may have to discriminate between different objects such as

cars, trucks, bicycle, and others so a number to each class is assigned and these numbers are the only possible outcomes of the learning algorithm. The following sections present some basic algorithms and optimization techniques used during this thesis work to understand better why some design choices were made.

2.3 Linear regression

Linear regression is the simplest example of a regression problem but can be very useful to analyze to understand many ML basic concepts. Here, some features x_i are given as input, and a prediction $\hat{y}$ of the actual value y is provided as output. The prediction $\hat{y}$ is called *hypothesis*, and it is obtained as follows:

$$\hat{y} = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + ... + \theta_j \cdot x_j \tag{2.1}$$

Where N is the number of the features considered in this example. $\hat{y}$ can be written more concisely:

$$\hat{y} = h_\theta(\vec{x}) = \sum_{j=0}^{N} \theta_j x_j = \theta \cdot X \tag{2.2}$$

Where

$$\theta = [\ \theta_0 \quad \theta_1 \quad .. \quad \theta_N\ ]$$

is a row vector if considering only one training example, and similarly

$$X = \begin{bmatrix} x_0 \\ x_1 \\ .. \\ x_N \end{bmatrix}$$

is a column vector. Considering that the training will be done on many training examples and not just one, the previous notation can be extended by having X an $(N\ x\ m)$ matrix:

$$X = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} & .. & x_N^{(1)} \\ x_0^{(2)} & x_1^{(2)} & .. & x_N^{(2)} \\ .. & .. & .. & .. \\ x_0^{(m)} & x_1^{(m)} & .. & x_N^{(m)} \end{bmatrix}$$

where m is the number of training examples employed and $x_j^{(i)}$ indicates the jth feature belonging to the ith sample. The training phase consists of finding the values of θ_j parameters so that the estimation $\hat{y}$ is as similar as possible to the

actual value y. Nevertheless, how is it evaluated how much the hypothesis is close to the actual value? It is used the so-called *loss function* (or similarly *cost function*). Many of them exist and are suitable for different ML algorithms; in the case of linear regression, the *Mean Squared Error* is used:

$$J(\theta_0, \theta_1, .., \theta_N) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y)^2 \tag{2.3}$$

Since this is a measure of the distance between the predicted value and the actual one, the value of $J(\vec{\theta})$ should be minimized by picking proper $\vec{\theta}$ values. To solve this minimization problem, a technique called *gradient descent* is used. An intuitive definition of gradient descent is given in [8]: "*Gradient descent is an iterative technique commonly used in machine learning and deep learning to find the best possible set of parameters/coefficients for a given model, data points, and loss function, starting from an initial, and usually random, guess*". Gradient descent can be easily visualized by picking just two input features. The loss function $J(\theta_1, \theta_2)$ is a 3D surface:

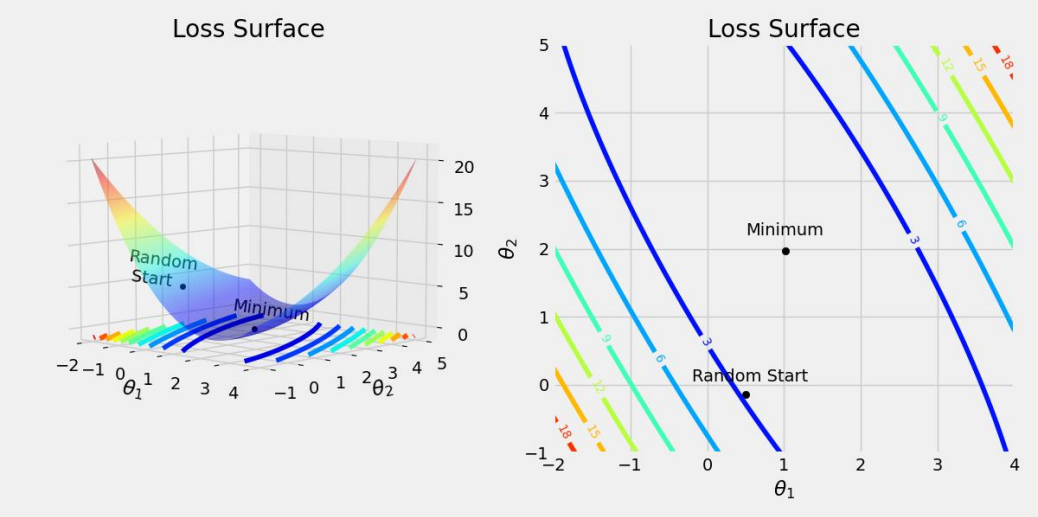


Figure 2.3: Right: Loss surface. Left: Contour Map. Courtesy [8]

Concerning figure 2.3 the process starts from a random choice of θ_1 and θ_2 . It gradually moves towards the minimum point of the loss surface, in which the θ parameters are optimal, and the loss (the distance between the prediction and the actual value) is minimized. In figure 2.3 the rightmost picture depicts the loss surface in a 3D way, while in the left one, a *contour map* is used, and colors are employed to highlight the different values for the loss function. Equations can express this process, and in gradient descent parameters are updated as follows:

$$\theta_j := \theta_j - \eta \frac{\partial J(\theta)}{\partial \theta_j} \tag{2.4}$$

It is important to notice that the ":=" sign in the equation indicates an update operation of the parameters and not the usual equal symbol. Every parameter θ_j is updated by a constant parameter η, the *learning rate*, weighted by the partial derivative of the loss function with respect to θ_j, i.e. the steepness of the J(θ) curve in the J-θ_j plane. In this process, the point on the surface is moved towards, hopefully, the absolute minimum of the curve. Equation (2.4) can be applied to a linear regression case:

$$\theta_j := \theta_j - \eta \frac{\partial}{\partial \theta_j} \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y)^2 = \theta_j - \eta \frac{\partial}{\partial \theta_j} \sum_{i=1}^{m} (h_\theta(\vec{x}^{(i)}) - y^{(i)})^2 \tag{2.5}$$

Reminding that $h_\theta(\vec{x}) = \sum_{j=0}^{N} \theta_j x_j$we can obtain the final equation for the update of the parameters for linear regression:

$$\theta_j := \theta_j - \eta \frac{1}{m} \sum_{i=1}^{m} (h_\theta(\vec{x}^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \tag{2.6}$$

From equation (2.6) it can be noticed that an update occurs every m samples, supposing to have in total M training examples, it is possible to select the value m and perform gradient descent in three different ways:

- if $m = M$ we are using all the examples in the training set, this technique is called *Batch Gradient Descent*
- if $m = 1$ only one training example is used, and this is called *Stochastic Gradient Descent*
- if $1 < m < M$ we have the so-called *Mini-Batch Gradient Descent*

But why this distinction? The natural process would be to compute the loss on all training samples available before performing an update, so to employ batch gradient descent. Although this process will find a minimum, it may take a long time to execute, especially in complex problems, so usually, the number of samples used before performing an update (usually called *batch*) is reduced to find a good trade-off between performance and speed. Another critical aspect to consider is the choice of the learning rate η, which heavily affects the effectiveness of the gradient descent. Visually, the learning rate represents the size of the 'steps' on the loss surface to move towards the minimum. A small η means small steps are taken, a minimum is reached, but the algorithm will be slow. In addition, more complex loss functions do not have a "bowl" shape like in figure 2.3, so many local minima

may be present. In that case, having too little learning rate can lead the algorithm to be stuck in one of the minima that are not optimal. On the other hand, if the value of η is increased, the execution speed will also increase; however, in this way, steps taken on the surface will be bigger, and it becomes possible to skip over a minimum missing it. Also, if the learning rate is set to a huge value, not only could the optimal point be skipped, but the algorithm might diverge and, instead of reducing the loss while training, may increase it, leading to poor results. Gradient descent is only one of the many methods to solve a minimization problem; however, it is the foundation of many other techniques which aim to solve some criticalities of gradient descent (local minima, saddle points, plateau). In [9] an overview of some gradient descent methods and an excellent visualization of their key concepts is presented.

2.4 Logistic Regression

Logistic regression is a different method for classification problems. In this case, the output of the machine learning algorithm should be a variable that changes in a discrete, instead of a continuous, way. Supposing we want to divide objects into two classes, performing the so-called "*binary classification*" we would like to have as output two possible states that can be, for simplicity, 0 or 1. To obtain such behavior, a function called "*logistic (or sigmoid) function*" is applied to the output of a simple linear regression case. In linear regression, the prediction, the hypothesis $h_\theta(\vec{X})$, was obtained as:

$$h_\theta(X) = \theta \cdot X \tag{2.7}$$

Differently, for logistic regression:

$$h_\theta(\vec{X}) = g(\theta \cdot X) \tag{2.8}$$

Where g, is the logistic function, defined as:

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2.9}$$

By applying g to our hypothesis, we get:

$$h_\theta(X) = \frac{1}{1 + e^{-\theta \cdot X}} \tag{2.10}$$

In figure 2.4 is plotted the graph of the logistic function. It can be noticed how this function assumes values in the range (0-1); however, to adapt the output to a binary classification problem, it is necessary to have as possible values only the limits 0 or 1. A threshold can be set; above it, the predicted class will be 1, and

below it will be 0. A natural choice can be 0.5, so the prediction y will have the following behavior:

$$h_\theta(X) \geq 0.5 \longrightarrow y = 1 \tag{2.11}$$

$$h_\theta(X) < 0.5 \longrightarrow y = 0 \tag{2.12}$$

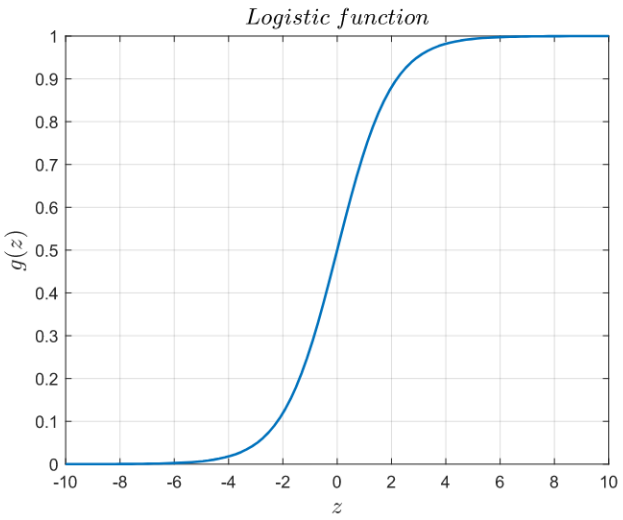


Figure 2.4: Logistic function

By selecting a threshold, the quantity $\theta \cdot X$ is the one that determines the value of a prediction; for this reason, it is called "decision boundary."

$$\theta \cdot X \geq 0 \longrightarrow h_\theta(X) \geq 0.5 \longrightarrow y = 1 \tag{2.13}$$

$$\theta \cdot X < 0 \longrightarrow h_\theta(X) < 0.5 \longrightarrow y = 0 \tag{2.14}$$

Since X is the matrix containing all input samples, the logistic regression training process will consist of modifying the parameters θ to have a proper decision boundary to get an accurate prediction.
Example: supposing to have a logistic regression problem with θ already defined as $\theta = \begin{bmatrix} \theta_0 & \theta_1 & \theta_2 \end{bmatrix} = \begin{bmatrix} -3 & 1 & 1 \end{bmatrix}$ we can easily find the decision boundary. Considering just one training example, $\mathrm{X} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}$ given that $x_0 = 1$, it is possible to evaluate the hypothesis and the corresponding decision boundary.

$$h_\theta(X) = g(\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2) \tag{2.15}$$

Prediction is equal to 1 if the argument of the logistic function is greater than zero, 0 otherwise.
Predict y=1 if $-3 + x_1 + x_2 \geq 0 \longrightarrow x_1 + x2 \geq 3$
Predict y=0 if $-3 + x_1 + x_2 < 0 \longrightarrow x_1 + x2 < 3$
In figure 2.5 is represented the decision boundary just calculated in the more complex case of multiple training examples. Black circles indicate a prediction equal to 0, and the red crosses instead equal to 1. The green dashed line represents the decision boundary identified by the equation $x_1 + x_2 = 3$. It can be noticed that training examples are divided, and above the line, a prediction will be equal to 1; on the contrary, it will be 0 below the same line. In this simple example, only two input features were considered, so, it was possible to plot the decision boundary; in real problems where many features are considered, it is not possible to perform this analysis due to the difficulty of representing higher dimension functions. Another essential aspect is that the decision boundary is a property not of the training set but of the hypothesis and parameters. The learning algorithm will use the training set to tune its decision boundary to get accurate predictions.

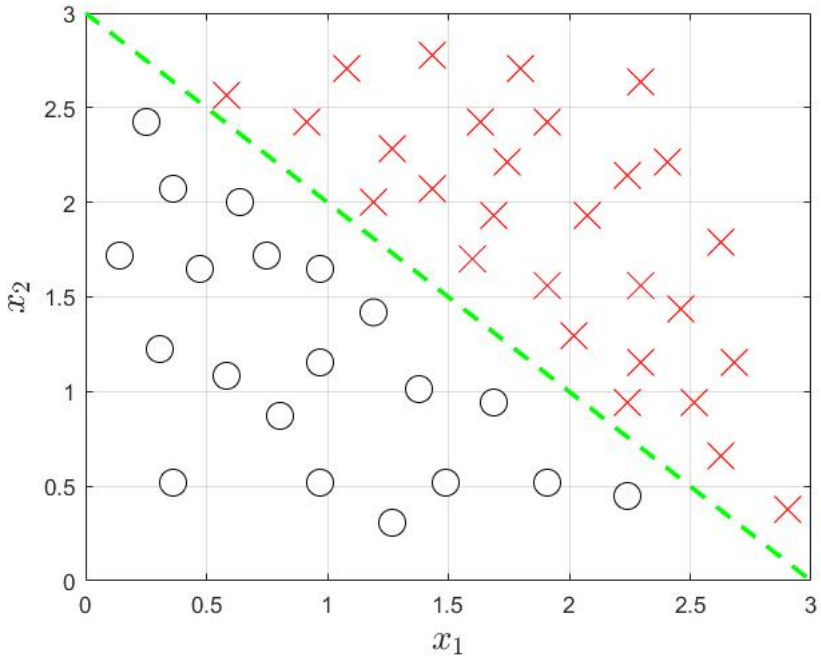


Figure 2.5: Decision boundary

At this point, gradient descent can be again applied to optimize parameters θ;

however, it is needed to define a new cost function.

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} log(h_\theta(x^{(i)})) + (1 - y^{(i)}) log(1 - h_\theta(x^{(i)}))] \tag{2.16}$$

where $y^{(i)}$ are the correct labels given by the dataset. To understand better how this cost function works, is possible to re-write it for one training example only:

$$J(\theta) = -y \cdot log(h_\theta(x)) - (1 - y) log(1 - h_\theta(x)) \tag{2.17}$$

This equation is a compact form of the following system, which is more clear to understand:

$$J(\theta) = \begin{cases} -log(h_\theta(x)), & \text{if } y = 1 \\ -log(1 - h_\theta(x)), & \text{if } y = 0 \end{cases} \tag{2.18}$$

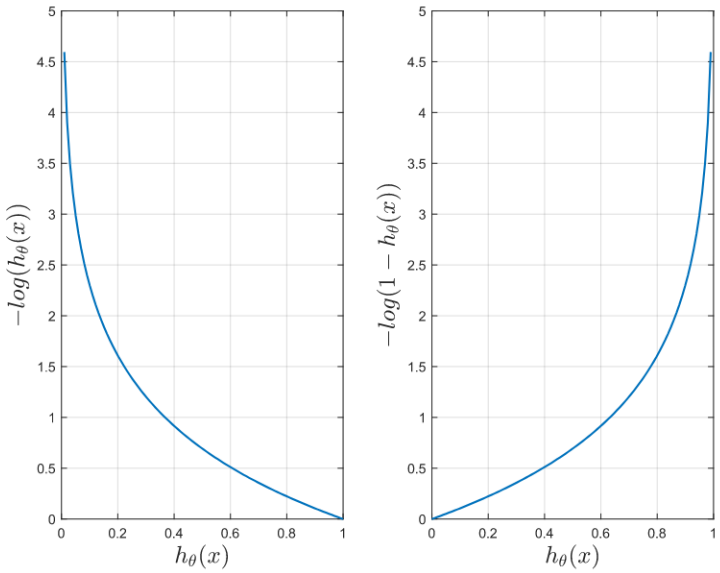


Figure 2.6: Logistic regression cost function: on the left y=1 case, on the right y=0 case

By plotting these two functions is possible to understand how the cost is calculated. The above graphs are plotted considering $h_\theta(x)$ varying in the (0,1) range continuously, but this is only for visualization purposes since we already discussed that the hypothesis should assume only two possible values. Supposing that the actual value of the prediction y is 1, we would consider the leftmost function: here, if the hypothesis is equal to 1, we can notice that the cost approaches 0. On the other

hand, if $y = 1$ and the hypothesis is equal to 0, we can notice that the cost drastically increases. For the y=0 case, the concept is similar, with obvious adjustments. With this mechanism, it is possible to run gradient descent to tune parameters θ, so the decision boundary, to get accurate predictions: correct prediction will have a zero cost while wrong ones will increase the cost. At this point, gradient descent will optimize θ to get the smaller possible cost. Lastly, we notice that gradient descent for linear and logistic regression has the same behavior; the only change is the loss function employed.

2.5 Neural Networks

Neural networks are a set of algorithms that try to mimic the structure and functioning of neurons in a human brain. The functioning of the neuron is very complex but can be simplified as follows: the cell receives an electrical input through the dendrites, which can be thought of as the *input wires*, and this signal is elaborated in a certain way inside the cell, then it is re-transmitted to other neurons through the axon. In figure 2.7 a simplified neuron structure is represented.

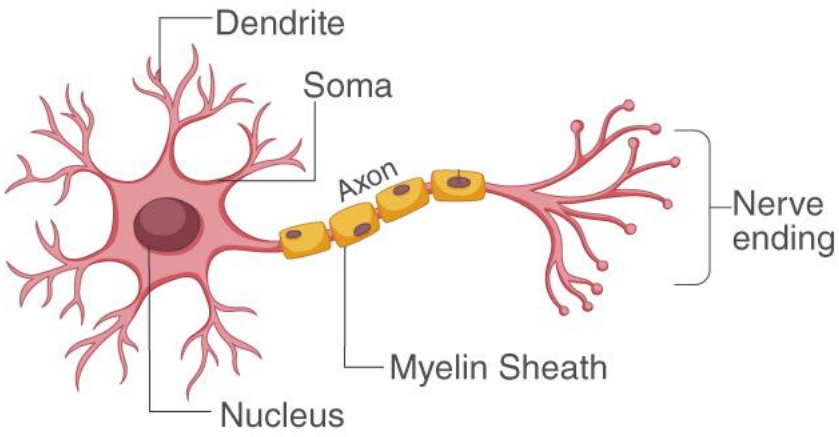


Figure 2.7: Human neuron simplified structure

Neural networks are made up of the so-called artificial neurons that mathematically mimic the above process. Like a human neuron, the artificial one receives some input signals, which, in this case, are real numbers. The output to be transmitted

will be the sum of all the inputs with the application of a non-linear function called *activation function*. For example, it is possible to employ the logistic one seen in the previous chapter as an activation function, obtaining a logistic neuron. Figure 2.8 represents the model for the artificial neuron, which will be the basic building block for the neural network.

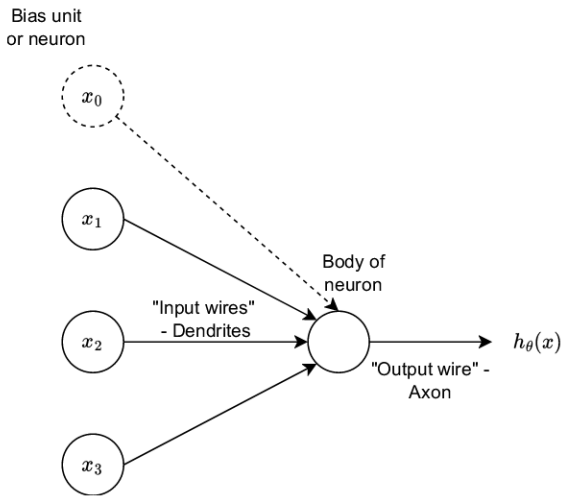


Figure 2.8: Artificial neuron model

x_1,x_2, and x_3 are the input values; each dendrite has its weight, a multiplicative factor applied to each input that tells how much the corresponding input is "relevant" for the neuron, and the "cell body" performs a sum operation. Then the activation function is applied to the weighted sum of the inputs and given as output. Note that the x_0 input is called a "bias" unit and is set to 1 by definition. Supposing $g(\cdot)$ is a generic non-linear activation function, it is possible to write the output for a single neuron.

$$h_\theta(x) = g(\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3) \tag{2.19}$$

At this point, it is possible to combine many neurons to get a neural network, as shown in figure 2.9. In this figure, the following notation is used:

$$a_i^{(j)} = \text{"Activation of unit i in layer j"}$$

$$\theta^{(j)} = \text{Matrix of weights mapping from layer j to layer j+1}$$

Applying the same procedure seen in (2.19) it is possible to obtain the output equations for the neural network in figure 2.8.

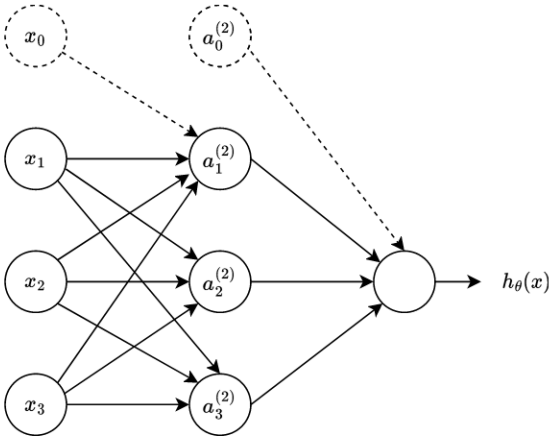


Figure 2.9: Example of neural network

$$\begin{aligned} a_1^{(2)} &= g(\theta_{10}^{(1)}x_0 + \theta_{11}^{(1)}x_1 + \theta_{12}^{(1)}x_2 + \theta_{13}^{(1)}x_3) \\ a_2^{(2)} &= g(\theta_{20}^{(1)}x_0 + \theta_{21}^{(1)}x_1 + \theta_{22}^{(1)}x_2 + \theta_{23}^{(1)}x_3) \\ a_3^{(2)} &= g(\theta_{20}^{(1)}x_0 + \theta_{21}^{(1)}x_1 + \theta_{22}^{(1)}x_2 + \theta_{23}^{(1)}x_3) \\ h_\theta(x) = a_1^{(3)} &= g(\theta_{10}^{(2)}a_0^{(2)} + \theta_{11}^{(2)}a_1^{(2)} + \theta_{12}^{(2)}a_2^{(2)} + \theta_{13}^{(2)}a_3^{(2)}) \end{aligned} \tag{2.20}$$

These equations can be written more compactly by working with vectors and matrices.

$$\begin{aligned} a_1^{(2)} = g(\theta_{10}^{(1)}x_0 + \theta_{11}^{(1)}x_1 + \theta_{12}^{(1)}x_2 + \theta_{13}^{(1)}x_3) &\longrightarrow a_1^{(2)} = g(z_1^{(2)}) \\ a_2^{(2)} = g(\theta_{20}^{(1)}x_0 + \theta_{21}^{(1)}x_1 + \theta_{22}^{(1)}x_2 + \theta_{23}^{(1)}x_3) &\longrightarrow a_2^{(2)} = g(z_2^{(2)}) \\ a_3^{(2)} = g(\theta_{20}^{(1)}x_0 + \theta_{21}^{(1)}x_1 + \theta_{22}^{(1)}x_2 + \theta_{23}^{(1)}x_3) &\longrightarrow a_3^{(2)} = g(z_3^{(2)}) \\ z^{(2)} &= \theta^{(1)}x \\ a^{(2)} &= g(z^{(2)}) \\ h_\theta(x) = z^{(3)} &= \theta^{(2)}a^{(2)} \end{aligned} \tag{2.21}$$

Where $\theta^{(1)}$ is 3 by 4 matrix in this case and $z^{(2)}$,$a^{(2)}$ have dimension 3 by 1, and $h_\theta(x)$ is a scalar so dimension 1 by 1. To understand the dimension of the weight matrix is possible to apply a simple rule: if network has s_j units in layer j, s_{j+1} units in layer j+1, then $\theta^{(j)}$ will be of dimension s_{j+1} by $(s_j + 1)$. Note that the bias unit must not be considered. In this case $\theta^{(1)}$ is the matrix mapping from layer 1 to 2; $s_1 = 3$ and $s_3 =$ so the dimension will be s_2 by $(s_1 + 1) = 3$ by 4. This process of evaluating the output given the inputs and weight matrix is called *forward propagation*.

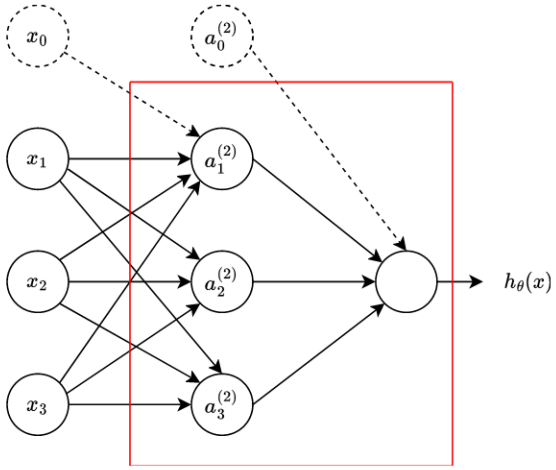


Figure 2.10: Example of neural network

Considering the part of neural network inside the red frame in figure 2.10 and evaluating the output we get $h_\theta(x) = g(\theta_{10}a_0^{(2)} + \theta_{11}a_1^{(2)} + \theta_{12}a_2^{(2)} + \theta_{13}a_3^2)$. This part is similar to logistic regression, where $g(\cdot)$ was the logistic function. The difference is that the features of this logistic regression are the values in the hidden layer. Instead of using inputs, the terms $a_1^{(2)}$, $a_2^{(2)}$ and $a_3^{(2)}$ are employed and they themselves learnt from inputs x_0,x_1,x_2,x_3. The neural network can learn its features for the subsequent layers, so it is possible to get more complex hypotheses.

2.5.1 Examples

AND function

$$x_1, x_2 \in \{0,1\}$$

The desired output y will be:

$$y = x_1 \; AND \; x_2$$

Considering the neural network in figure 2.11 it is possible to write the equation for the hypothesis $h_\theta(x)$. Applying the rule for the matrix of weights, θ is a row vector with three elements, $\theta = [\theta_{10}^{(1)} \;\; \theta_{11}^{(1)} \;\; \theta_{12}^{(1)}] = [-30 \;\; 20 \;\; 20]$. The hypothesis will be:

$$h_\theta(x) = g(-30 + 20x_1 + 20x_2) \tag{2.22}$$

At this point, using the logistic function as activation is possible to write a truth table for this neural network to see if it performs an AND operation.

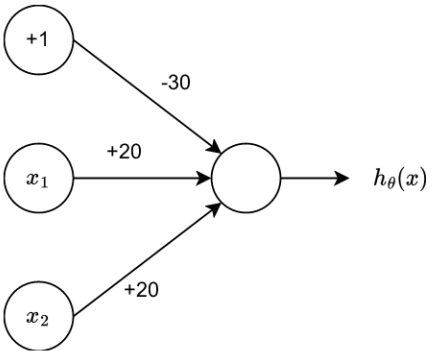


Figure 2.11: Neural network for AND logic function

x_1	x_2	$h_\theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

OR function

The same procedure of the previous example can be used to build the OR logic function.

$$x_1, x_2 \in \{0,1\}$$

The desired output y will be:

$$y = x_1 \, OR \, x_2$$

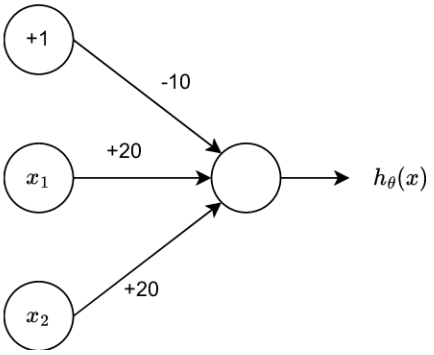


Figure 2.12: Neural network for OR logic function

The hypothesis will be evaluated as: $h_\theta(x) = g(-10 + 20x_1 + 20x_2)$. As before, writing the truth table for this network is now possible.

x_1	x_2	$h_\theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	$g(10) \approx 1$
1	1	$g(30) \approx 1$

NOT function

$$x_1, x_2 \in \{0,1\}$$

The desired output y will be:

$$y = NOT(x_1)$$

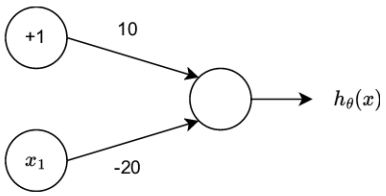


Figure 2.13: Neural network for NOT logic function

As before the hypothesis is evaluated: $h_\theta(x) = g(10 - 20x_1)$ and the truth table is calculated.

x_1	$h_\theta(x)$
0	$g(10) \approx 1$
1	$g(-10) \approx 0$

A more complex function: XNOR

Combining simple functions such as those seen in the previous example makes it possible to build even more complex functions. This concept is crucial to understand because it determines whether a model is suitable to perform specific predictions or not. Figure 2.14 shows how a neural network is created to have an XNOR logic function. Since the XNOR can be obtained by combining basic logic functions, the structure is divided into more layers, and each layer's output will be the input feature for the subsequent one.

x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_\theta(x)$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

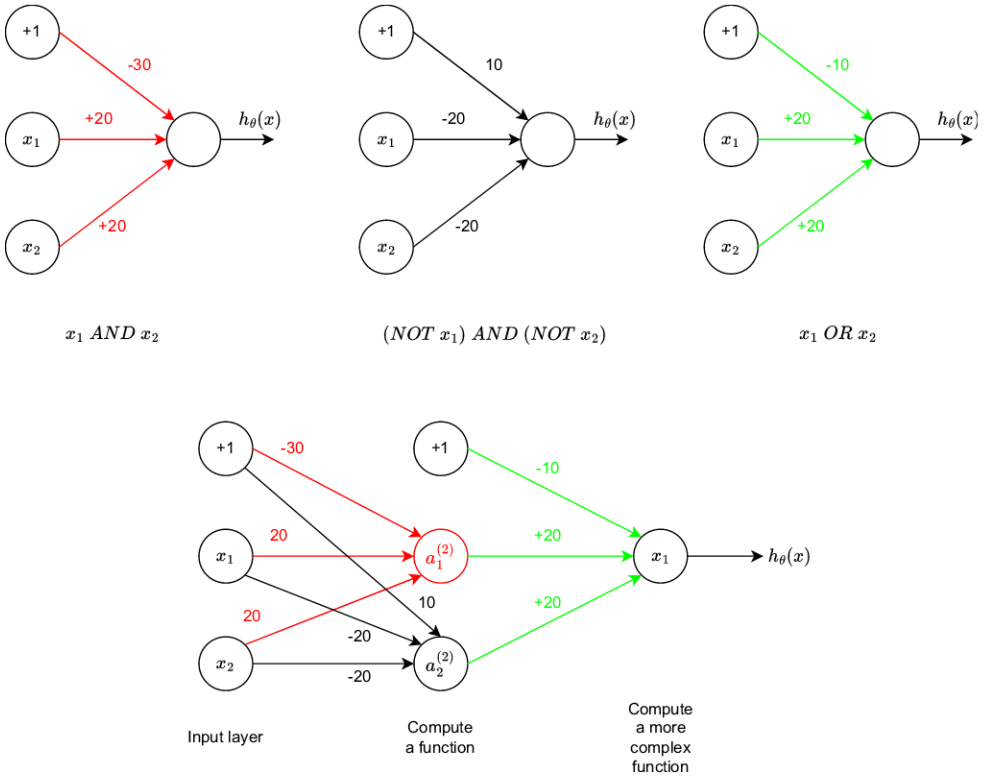


Figure 2.14: XNOR function neural network

2.5.2 Backpropagation algorithm

Suppose now to have the neural network in figure 2.15. The structure of the network can be divided into three parts:

- Input layer gives the input values to the network, and the number of neurons in this layer represents the number of features used in the problem.
- Output layer: neurons that produce the output of the algorithm
- Hidden layers: all layers that are not input or output are referred to as hidden layers.

In the examples seen so far, the weights θ of the connections were already given; however, in real problems, they have to be found during the training phase. Also, in neural networks, the concepts of cost function and gradient descent are used but, in this case, with some modifications due to the different structures of the algorithm. The so-called *backpropagation algorithm* is used for the neural networks. A complete mathematical analysis of this algorithm is quite complex, so it will be given only a basic outline of what the algorithm is doing. More mathematical

details can be found at [10], and a very informative video visually explaining the backpropagation process is available at [11].

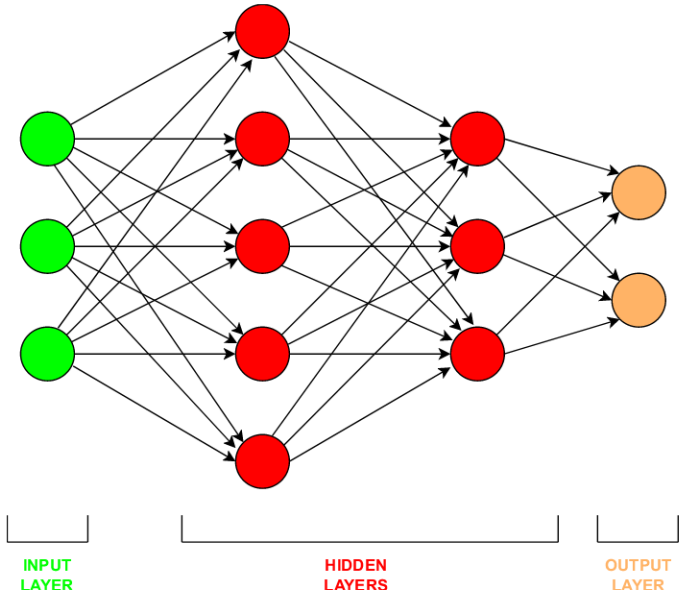


Figure 2.15: A simple feedforward neural network

The starting point is a neural network with random weights, so there are no capabilities for prediction better than random guessing. The first step is to perform the so-called forward propagation, i.e., evaluating the neural network output given as input the training examples. At this point, the cost is computed: many loss functions exist and have different characteristics based on the type of prediction implemented with a neural network (regression, classification) but what they all do is return a number that indicates how much the predicted output is similar to the expected one, which is given in the training dataset. The following notation will be used:

- L = total number of layers in the network
- s_l = number of units (not counting bias units) in layer l
- $\theta^{(l)}$ = matrix mapping layer l to $l+1$
- $\theta_{ij}^{(l)}$: element of matrix $\theta^{(l)}$ which maps element j of layer l to element i of layer $l+1$
- $a_j^{(l)}$ = "activation" of neuron j in layer l
- $a^{(i)}$ = vector of activations of layer l

- x = input vector, the size is the number of features
- y = output vector, y_j = output of neuron j in last layer L
- $g(\cdot)$ is a generic activation function

It is possible to express forward propagation in a vectorized form:

$$\begin{aligned} a^{(1)} &= x \\ z^{(2)} &= \theta^{(1)} a^{(1)} \\ a^{(2)} &= g(z^{(2)}) \\ z^{(3)} &= \theta^{(2)} a^{(2)} \\ a^{(3)} &= g(z^{(3)}) \\ z^{(4)} &= h_\theta(x) = \theta^{(3)} a^{(3)} \end{aligned}$$

At this point, a new quantity is defined: $\theta^{(l)}$ is a vector containing the error associated with neurons of layer l. It is now possible to go backward and evaluate the error vectors for each layer in the neural network, starting from the output layer. The following equations are not demonstrated, so to see the mathematical details it is possible to check [10].

$$\begin{aligned} \delta^{(4)} &= a^4 - y \\ \delta^{(3)} &= (\theta^{(3)})^T \delta^{(4)} \cdot g^{'}(z^{(3)}) \\ \delta^{(2)} &= (\theta^{(2)})^T \delta^{(3)} \cdot g^{'}(z^{(2}) \end{aligned} \tag{2.23}$$

Note that the dot symbol "$\cdot$" distinguishes the element-wise product from the matrix product. $\delta^{(1)}$ is not present because the first layer coincides with the training dataset. It is also possible to obtain a relationship between the gradients of the cost function with respect to the network weights and the quantities just calculated:

$$\frac{\partial J(\theta)}{\partial \theta_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)} \tag{2.24}$$

At this point, the gradients of the cost function for each parameter $\theta_{ij}^{(}l)$ are known, and it is possible to perform gradient descent to minimize the cost function. Also for neural networks, the ideal situation would be to plug into the neural network all the training set values before performing an update, i.e., to employ *batch gradient descent*. However, this may be too computationally expensive, so *mini batch* and *stochastic gradient descent* are more viable choices in most cases.

2.5.3 Activation functions overview

In previous examples, the concept of cost function was introduced, but just the logistic one was analyzed. Many of these functions exist, and they heavily influence the performance that a neural network can reach; for this reason, an overview of the most common ones is now presented [12].

Linear activation

Linear activation is the simplest type of activation, and basically, it consists in performing no more than the weighted sum of the inputs. Results can be limited because linearly combining the input will result in an output that is still linear, so a simple linear regression is performed.

$$f(x) = x$$

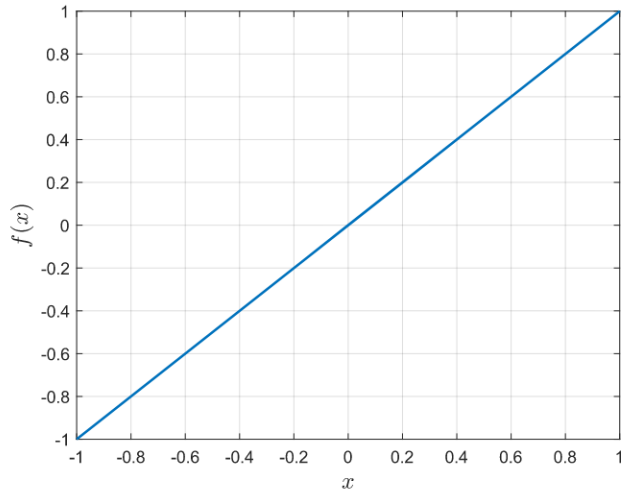


Figure 2.16: Linear activation function

To predict more complex functions or, in the classification case, have a more complex decision boundary, it is necessary that the activation function introduces some non-linearities.

Hyperbolic tangent activation

The hyperbolic tangent activation function (or *tanh*) is commonly used in machine learning, it varies in the range [-1,+1] and has a similar behavior to the sigmoid activation function (see figure 2.4).

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.25}$$

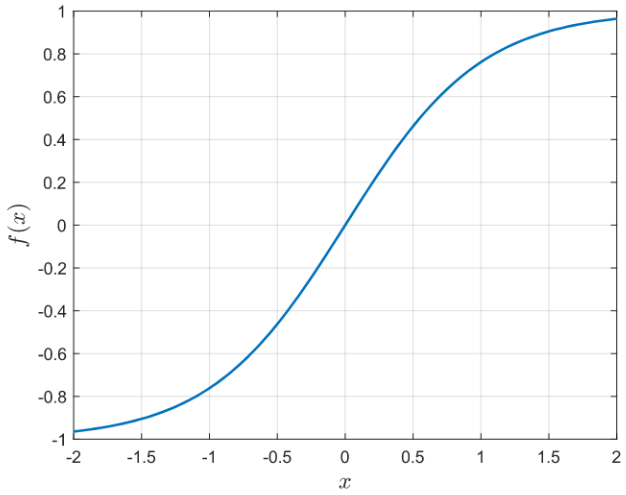


Figure 2.17: Hyperbolic tangent activation function

It is convenient to look at the gradients of these functions to better understand the differences between logistic and tanh function. In figure 2.18, these derivatives are plotted: since data employed in machine learning algorithms are usually centered around zero, it is useful to look at gradients around zero. For the tanh activation function, this value is bigger than the logistic function. Having bigger values for the gradients during the training phase will result in bigger steps during gradient descent procedures. So the tanh function can be employed when a faster convergence is needed, always keeping in mind that larger learning steps can cause the skipping of optimal minima of the loss function.

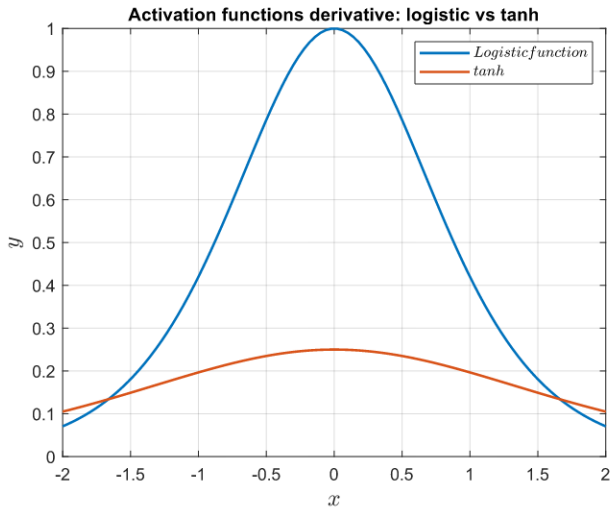


Figure 2.18: Derivatives of logistic and tanh functions

ReLU

Another widely used activation function is the *Rectified Linear Activation Unit*, or *ReLU* for short. Sigmoid and tanh activation present the problem of saturation. For both of them, when the input value increases, the output is saturated to the value 1, so moving towards greater inputs results in a smaller sensitivity concerning input variations. In the same way, when the input becomes smaller, sigmoid and tanh functions saturate at the values 0 and -1, respectively. This aspect can also be seen in figure 2.18, where the derivatives that are linked to the sensitivity are larger around zero but tend to decay when inputs are smaller or bigger. Another aspect that has to be considered is the *vanishing gradients problem*. When a neural network is trained, the error is backpropagated through each layer and used to update the weights. This error dramatically decreases for each additional layer, so large networks may have some problems to be trained effectively. Moreover, non-linear activation functions such as logistic or tanh are computationally expensive due to the necessity of performing exponential calculations. The *ReLu* can be useful to solve the problems presented above. The graph of this function is presented in figure 2.19. Given the input x the ReLU is equal to zero if x<0 while equal to x when x>0. This function can be calculated as:

$$f(x) = max(0, x)$$

It can be noticed that no exponentials are needed, the function never saturates, and it also keeps a degree of non-linearity, allowing the neural network to learn

more complex decision boundaries. Looking at the derivative of the ReLU, it is possible to notice that it assumes value 0 when x<0 and equal to 1 when x>0, in this way, the problem of low sensitivity for larger input values is avoided. For the particular case x=0, the derivative is not defined because left and right derivatives are not equal; however, the derivative in this point can be set to 0 for simplicity.

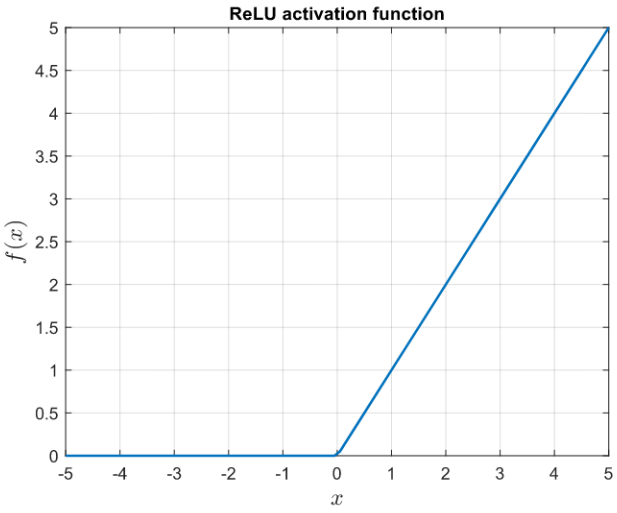


Figure 2.19: ReLU activation function

In [13] it is possible to find more information about the ReLU activation function.

Leaky ReLU

When employing the ReLU activation function, another problem, called *Dying ReLU problem*, may arise: some neurons might remain inactive with no regard to the input provided, and if this happens, the backpropagation algorithm is no more able to update effectively the neural network weights leading to a performance drop. To solve this problem is possible to modify the ReLU function by providing a negative slope in the region where x<0 obtaining the so-called Leaky ReLU activation function, which can be mathematically described in as:

$$f(x) = \begin{cases} a \cdot x, & \text{if } x < 0 \\ x, & \text{if } x >= 0 \end{cases} \tag{2.26}$$

where a is the slope in the region of negative x values. The corresponding plot is shown in figure 2.20.

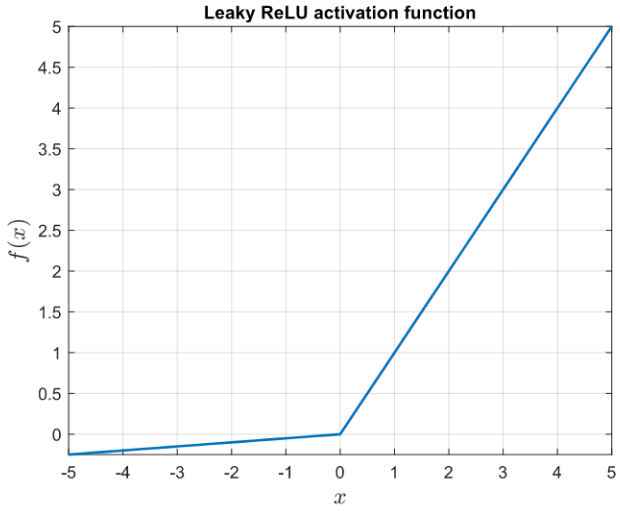


Figure 2.20: Leaky ReLU activation function with a=0.01

2.6 Machine Learning Additional Information

2.6.1 Normalization

Normalization is a technique widely used in machine learning that allows the algorithms to obtain a better result and to converge faster to a good minimum of the loss function. Different features can have ranges that vary a lot; if no data manipulation is performed, the features with greater value will be dominant compared to the others. The normalization process avoids this and scales all the features to the same range, usually set to [0,1] or [-1,1]. More details about normalization are given in [14].

2.6.2 Overfitting and Underfitting

The concepts of underfitting and overfitting are strictly related to the model's fitness for the learning task it has to face. The best way to understand this is to see how the model performs both on data used during the training and on new data. Underfitting is when a model performs poorly both on training and testing data. Overfitting occurs when the model scores exceptional results on training data but fails on new data from the test dataset.

2.6.3 Train-Test split

The train-test split technique consists in dividing the dataset in two portions, one for the training and the other for the test. A typical split is to reserve 80% of the dataset for the training and the remaining 20% to the test. This technique helps detect overfitting and underfitting: if the model performs poorly on the train and test portions, it probably suffers from underfitting because it can model neither the training data nor unseen test data; if, on the other hand, it performs exceptionally well on the train split but poorly on the test portion, the model may be affected by overfitting. The ideal situation is when a model performs well on the training dataset and the unseen data from the test split, highlighting a good generalization capability.

Chapter 3

Neural Network Training Framework

3.1 Introduction

One of the aims of this thesis work is to find out if neural networks can be efficiently used to predict the plant's health status and, if so, which topologies (i.e., how many layers, how many neurons per layer) are more suitable to accomplish this task. Moreover, since the objective is to implement the learning algorithm on a microcontroller which will also be responsible for getting data from plants and the environment, the focus will be on finding networks able to perform health status classification without employing too many resources on the platform such as flash memory and RAM. Besides memory aspects, also the computation time is a crucial problem because the more time the network spends on the computation, the more the microcontroller has to run instead of going into standby mode; in a system that aims to be autonomous and powered by solar panels the time microcontroller spends in running mode can determine a significant impact on the energy consumption and on the effectiveness of the system. For these reasons, accurate neural network research must be done. Since there are no strict rules in machine learning to be applied to have an effective system but only guidelines and, considering that there are scarce resources in the literature addressing the problem of plant classification employing impedance and environmental data, a trial and error approach is a good choice. Many aspects determine how a neural network performs, features used, number of hidden layers, number of neurons in each layer, learning rate, activation functions, cost functions, and many others. Since designing each neural network and then changing the above parameters can be time-consuming and good results are not guaranteed, it is helpful to have a tool to perform this search automatically. For the above reasons, it was chosen to

implement a framework to train neural networks efficiently. The first version of this framework was implemented by a previous student of Politecnico di Torino who did a great job defining the program's data structures and general workflow. However, this first version could train neural in a parametric way but was able to perform only regression tasks. The major add-on implemented during this thesis was the capability of train neural networks to perform classification, in particular, binary classification to discriminate between healthy and unhealthy plants. The software was implemented using the *Python*[15] programming language, in particular version 3.9. To easily install all required packages and avoid compatibility problems of different packages already installed on the computer, the software *Anaconda*[16] was employed. With its intuitive GUI, *Anaconda navigator*, this software allows to easily install the required packages inside a Python virtual environment[17]. The section "Appendix A" of this document presents a small guide on how to setup correctly the developed framework using the Anaconda virtual environments.

3.2 Framework starting point

This section presents an overview of the functions already implemented inside the framework before the beginning of this thesis work. Since many functionalities are implemented and going into deep details about the code is beyond the scope of this work, a general overview of the framework is presented. For more details, it is possible to refer to the thesis of Alessandro Lovesio present in the Politecnico di Torino online library [18]. The framework is developed using the open-source Python machine learning library called Pytorch[5]. This library was released in 2016 and gained rapid popularity in many ML applications; moreover, many important companies, such as Tesla, Uber, and Airbnb, employ this library for their products. Pytorch offers a set of tools that drastically improve the efficiency of training machine learning models, such as:

- automatic differentiation: helpful to perform automatically the process of backpropagation seen in the previous chapter.
- support for GPU computing in order to speed up the training process
- template classes to create a custom dataset
- template classes to create neural networks
- compatibility with popular Python libraries like Numpy and Pandas
- compatibility with ONNX format to convert models between different machine learning libraries

3.2.1 Directory tree

Figure 3.1 represents the files organization of the framework:

- "Data": this folder stores the files containing data on the tobacco plants used in this research. The data file location can be any folder since the path of the desired files can be passed through a setting file; however, it is convenient to have an ordered organization with each plant having its directory containing its CSV files.

- "results": it stores results of different training processes. This folder will be filled with other subfolders corresponding to the performed simulation, each of them will eventually store the model (or models) trained saved as a .pth file, plots of the behavior of the network (only for regression problems), and a .txt file that summarizes the performance of the network (or networks).

- "src": the src folder stores all the Python source files that compose the framework:

  - *status_now_classes.py*: implements all the classes useful for the training of neural networks, such as a custom neural network class inherited by the "nn" module of PyTorch, a Dataset class, the core functions that allow for the training process and many other important functionalities
  - *status_now.py*: script that performs the training of a single neural network
  - *status_now_finder.py*: script that performs a sweep over some parameters of a neural network (number of hidden layers, neurons per hidden layer) and runs one training process for every specified combination of parameters.
  - *status_now_dispatcher.py*: file used to call other scripts to have an easier execution of the software
  - *status_now_test.py*: script that performs a test of an existing neural network on a dataset specified by the user, different from the one employed in the training
  - *utilities.py*: stores all the generic functions employed in other parts of the code framework

Inside the "src" folder, there is a directory called "settings" that contains all the settings files that the user can create or modify to run a simulation. The next sections will give more information about the structure of settings files and how to use them.

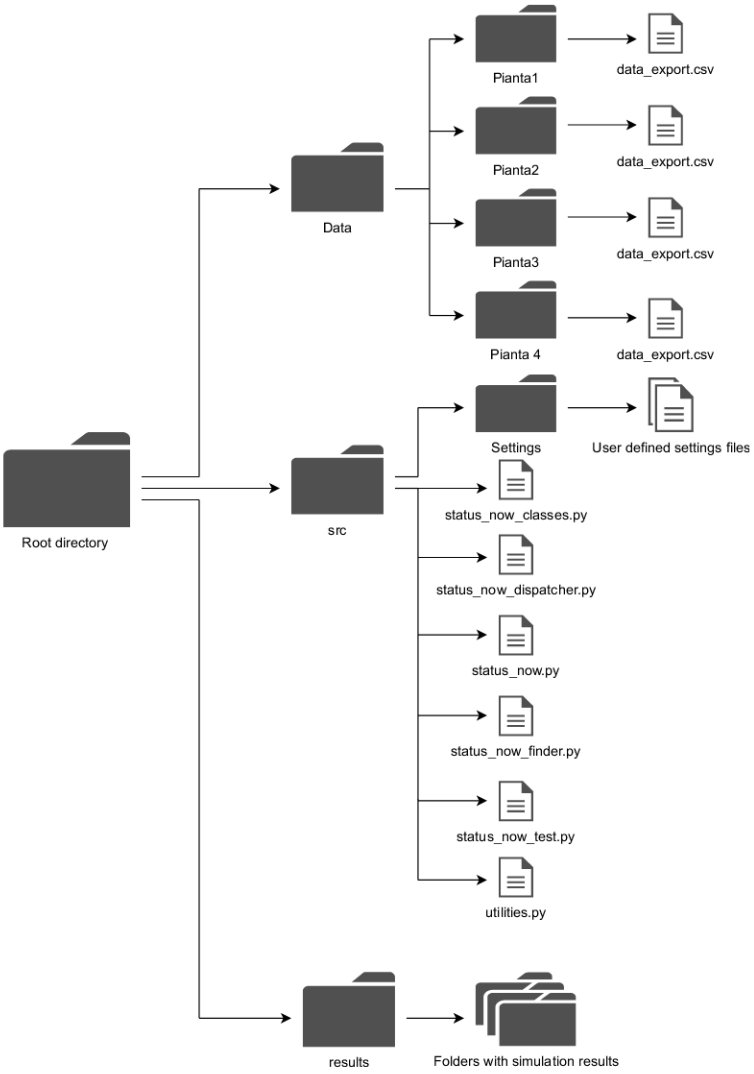


Figure 3.1: Directory tree of the framework

3.2.2 Dataset structure and management

As mentioned above, each plant analyzed in this thesis has its measurements stored in a CSV file. The data used are from tobacco plants whose parameters have been monitored from 23-03-2021 to 28-07-2021, measuring once an hour. Parameters measured from the plants include classical environmental ones (temperature, air humidity, impedance, and others), but what is more promising and innovative is the usage of impedance data extracted from the stem of the plants. Sensors to measure this kind of data were developed by MINES researchers in collaboration with Tel Aviv University and are described in [1][3]. Using neural networks that

employ this kind of data will open many possibilities because, up to now, image processing has been the primary technique used together with machine learning in smart agriculture. Relieving on sensors such as impedance ones can lead to simpler and cheaper detection systems. The four plants are kept in different conditions to have a more effective dataset that can better represent the different health conditions. In particular, two plants are regularly watered while the remaining are not. Going into deeper details, the measured parameters are:

- *Temperature* measured in Celsius degrees;
- *Air relative humidity* measured in percentage;
- *Ambient light* measured in lux
- *Soil moisture* measured in kilopascals;
- The *date* of the measurements
- Stem *impedance modulus* expressed in ohms;
- Stem *impedance phase* measured in degrees;
- The *health status* of the plant: 0 if the plant is labeled unhealthy, 1 if healthy.

In listing 3.1, a sample of the CSV files is presented:

Listing 3.1: Exampe of CSV file

```
1 Unnamed: 0,Status,Temperature [C],Air Humidity [RH],Ambient Light [
    lux],Moisture [KPa],Date,impedance_modlus,impedance_phase
2 0,1,29.42558288,41.17279052,11955.2,-5.652781040,2021-03-23
    14:37:53,973.782052,-90.776159
3 1,1,30.81535339,40.67840576,15953.92,-5.624867456,2021-03-23
    15:37:53,2729.400701,-87.17937
4 2,1,29.25941467,40.46173095,8442.880000,-3.683619830,2021-03-23
    16:37:53,1892.752663,-88.013926
```

The first operation the framework performs when launched is always the creation of the dataset using data extracted from CSV files. To accomplish this task, the Pandas [19] library is employed. The Pandas *read_csv()* function is used to read all CSV files and create a data frame which contains each plants' data. At this point the just created data frame is put into a list that will be then passed to another function: *buildDataset()*, whose definition is shown in listing 3.2

Listing 3.2: buildDataset() function definition

```
1
2 def buildDataset(plant_df, plant_list, dataset_settings,
    plants_to_use_list=None, remove_n=0, start_date=None,
```

```
3                   end_date=None, date_param_name="Date", testMode=
    False):
```

This function accepts the following arguments:

- *plant_df*: list of data frames created with the read_csv() function;
- plant_list: list of strings with names chosen for each plant;
- *dataset_settings*: list of dictionaries that specify how to build the columns in the dataset. It gives information such as how to filter, constrain and normalize data;
- *plants_to_use_list*: it specifies which plants are used to train the neural networks;
- *remove_n*: number of elements to be removed at the end of the data frame;
- *start_date*: specifies the starting date from which data will be put into the dataset;
- *end_date*: it specifies the last date for which data are put into the dataset;
- *date_param_name*: it specifies the name of the column in the data frame at which the dates of the measurements are written;
- *testMode*: flag helpful for debug purposes.

Data of the plants are limited in the range of dates specified by *start_date* and *end_date*, constrained within a specified values range, filtered (example: exponential average), transformed (for example, by applying the logarithm function), and then normalized in the range [-1,+1]. The *buildDataset()* function returns a single Pandas data frame in which all plants data are merged. To fully exploit the capabilities of PyTorch, however, it is better not to work with Pandas data frames but to convert them to some PyTorch data types. In particular, one of the most powerful objects of this library is the *Dataloader* class: it allows to pass data in batches to a network automatically; in this way, it is possible to avoid manipulating the previous data frame directly and to ease the training process. PyTorch implements a template class called *Dataset*[20] that is accepted by the Dataloader. At this point a custom class,*PlantsDataset*, which inherits from *Dataset* is created: due to the inheritance characteristics only some methods of this class had to be implemented, the *___init___()*, the *len()* and the *get_item()* methods. The behaviour of *len()* and *get_item()* is intuitive from their names: the first one returns how many elements are present in the dataset while the second returns a single item from it. Note that as the number of elements, the number of batches

the dataset is composed is intended. For example: having 100 training examples and a batch size of 10 elements, the *len()* method will return 10; in the same case the *get_item()* method will return a single element that is a batch containing 10 training examples.

Listing 3.3: ___init___() method

```
1    def __init__(self, dataframe, plants, parameters, outputs,
     n_samples=24, n_overlap=0, amount=1, amount_start=0):
```

Listing 3.3 represents the function definition of the ___init___() method. It is helpful to be analyzed because it offers a good overview of how the dataset is manipulated to be used together with PyTorch. The first parameter accepted, *dataframe*, is the Pandas data frame returned by the *buildDataset()* function and so it contains all the measurements of the plants. The parameter *plants* is a list of strings containing the names of the plants to be used (for example *pianta1, pianta2*). *Output* specifies which columns are used as the expected outputs for the neural network. The remaining parameters tell how to organize samples in the dataset.

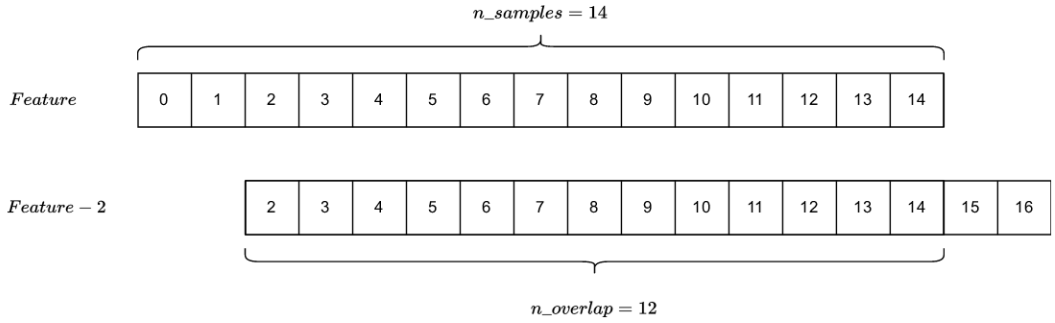


Figure 3.2: Features windows

The parameter *n_samples* tells how many successive time samples are used for each feature. For example: considering the impedance modulus, setting $n\_samples = 14$, the input data for the neural network regarding impedance modulus will be the current measurement of impedance and 13 previous measurements. Considering that many features are present in the dataset (Temperature, Air humidity, etc.), this parameter will affect each of them. Supposing to use six features, with *n_samples* equal to 24 (so one day of measurements, since one sampling per hour is performed), we will have $24 \cdot 6 = 144$ input data for the network, so 144 neurons in the input layer will be needed. Taking different samples of each feature can help to identify the correlation of the state of health when some events happen: for example when a plant is watered, the effects on its parameters, like the impedance, take place not instantaneously but after some time; having more samples can help to consider also

these kinds of events when making a prediction. Moreover, collecting input samples in this way will ensure that the time order of measurement remains unchanged when the dataset is shuffled, and this is fundamental when dealing with time series, like in this case. The *n_overlap* parameter tells how many samples are common to two subsequent inputs given to the neural network. Regarding figure 3.2, it can be seen that two successive input data of the same feature have 12 samples in common, that is to say that the time window is shifted by two time samples. To clarify this concept example tables 3.1 and 3.2 are provided.

	t	**t-1**	**t-2**
Impedance modulus	sample1	sample2	sample3
Impedance modulus -1	-	sample1	sample2
Impedance modulus -2	-	-	sample1

Table 3.1: Time window example: n_samples = 5, n_overlap=2

	t	**t-1**	**t-2**	**t-3**	**t-4**
Impedance modulus	sample1	sample2	sample3	sample4	sample5
Impedance modulus -2	-	-	sample1	sample2	sample3
Impedance modulus -4	-	-	-	-	sample1

Table 3.2: Time window example: n_samples = 5, n_overlap=3

The last two parameters specify how many samples are taken from the input data frame. *Amount* indicates the percentage of the total number of samples that will be inserted in the final dataset, so it is a value that can range from 0 to 1. *Amount_start* specifies the point, in percentage, from which starting to collect data to put into the dataset. For example: setting *amount_start=0.1* and *amount=0.8* will imply that 80% of total samples will be taken from the Pandas data frame, starting from the position corresponding to the 10% of the data. With this mechanism, it is possible to build different datasets starting from the same data frame, and this is helpful, for example, in creating a training and test dataset. The operations described in this chapter are common to every framework's functionality. They are a key part of the framework since an optimal dataset is a key aspect of obtaining a good predictor. After the dataset is correctly manipulated, it can be used by the program to train neural networks.

3.2.3 Single neural network training

The first helpful functionality to perform training allows for training a single neural network. The core of it is the function *train_loop*, which is implemented in the

file *status_now_classes.py*, and on which the whole training procedure in the framework is based.

Listing 3.4: *train_loop definition*

```
1 def train_loop(dataloader, model, loss_fn, optimizer):
```

This function performs a single training step, which means a complete run on all the batches of the dataset, and its definition is shown in listing 3.4. It accepts four input parameters:

- *dataloader*: this parameter belongs to the Pytorch Dataloader class, and it is useful to create iterable objects that can easily be passed to the network for training. The objects returned by the dataloader consist of batches of training examples whose dimension is specified by the batch size parameter
- *model*: this object belongs to a class that inherits from the Module PyTorch class [21]. It allows the definition of a neural network specifying the hidden layers, number of neurons, and activation functions. Also in this case, only some methods of the model object have to be implemented: the *___init___()* method helpful to define the shape of the neural network and the *forward* method that given in input vector returns the prediction of the algorithm. In Appendix B is possible to find a list of the possible layers and activation for this class
- *loss_fn*: this object is useful to calculate the loss, so the "goodness" of a prediction; moreover, it allows to use the *backward* method, which automatically performs the process of backpropagation, allowing to update of neural network weights. In appendix C, an overview of the loss functions available in Pytorch is given [22]
- *optimizer*: it implements an optimization algorithm (such as gradient descent); what is needed is to have evaluated the gradients of the loss function for each parameter, that is what the backward method of the previous argument does. Appendix D presents a list of available optimizers in Pytorch [23]

Since before this thesis, the framework could perform only classification tasks, the functions described here will be analyzed again in later sections to see what modifications have been made. In listing 3.5 is shown the code for *train_loop*:

Listing 3.5: *train_loop* implementation

```
1 def train_loop(dataloader, model, loss_fn, optimizer):
2     device = 'cuda' if cuda.is_available() else 'cpu'
3     size = len(dataloader.dataset)
4     train_loss, train_RMSE, n = 0, 0, 0
```

```
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X.to(device))
        y = y.to(device)
        pred.squeeze_()
        y.squeeze_().squeeze_()
        loss = loss_fn(pred, y)
        train_loss += loss.item()

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if isinstance(pred.tolist(), list) and isinstance(y.tolist(), list):
            train_RMSE += sum(([(xi - yi) ** 2 for xi, yi in zip(pred.tolist(), y.tolist())]))
            n += len(pred.tolist())
        elif isinstance(pred.tolist(), (int, float)) and isinstance(y.tolist(), (int, float)):
            train_RMSE += (pred.tolist() - y.tolist()) ** 2
            n += 1
        else:
            print("Warning: RMSE calculation may be inaccurate due to wrong type conversion for calculation")
    train_loss /= size
    train_RMSE = math.sqrt(train_RMSE / n)
    print(f"Training Error: \n RMSE: {train_RMSE:>8f}, Avg loss: {train_loss:>8f} \n")
    return train_loss, train_RMSE
```

First, the computation device is selected among CPU or GPU depending on if the *cuda* utility is available. Then, a loop over all the batches inside the dataset is performed: measurement and labels of the plants are passed to the model, and a prediction is made. At this point the loss is evaluated using the *loss_fn* function and the backward propagation is performed using the *backward* method. Finally, the optimizer is employed to update the network weights. At the end of the execution, the loss and the Root Mean Square Error (RMSE) are returned. For the classification case that will be analyzed later, other evaluation metrics instead of the RMSE will be returned. The function just described helps perform a training step, however, it is also necessary to test the network on a different dataset to evaluate its performance. For this reason a similar function called *test_loop* is introduced.

Listing 3.6: *test_loop* implementation

```
def test_loop(dataloader, model, loss_fn):
```

test_loop has almost the same code as *train_loop*, but in this case, only the network's predictions are computed using the model, but the parameters are not updated. It is possible to see, looking at listing 3.6, that the optimizer is no longer needed as a parameter to be passed to the function.

Status_now functionality

The entire process of training a single neural network is implemented in the file *status_now.py*. In particular, three functions are implemented:

- *main()*: used when *status_now.py* is called as a standalone script. It imports all the data from CSV files and selects a *setting* object, which is fundamental for the program's execution. The next section will give more details about the *setting* object. If the script is used as a module the *main()* is not executed

- *statusTrain()*: this function is in charge of managing the process of training by calling the appropriate functions, such as the one described above, *train_loop*. When executed, it returns data about the trained model, in particular, evaluation metrics, plots of samples, and errors committed by the model.

- *statusNowPrint()*: function used to save results and actual model file produced by the *statusTrain()* or *statusTrainKFold()* function.

Listing 3.7 shows the definition of *statusTrain() function*.

Listing 3.7: *statusTrain()* definition

```
1 def statusTrain(plant_df, plant_list, chosen_setting, batchMode: bool
      = False):
```

Parameters:

- *plant_df*: list of Pandas data frames obtained using the read_csv() function

- *plant_list*: list of the name attributed to the plants (i.e. pianta1, pianta2, etc.)

- *chosen_setting*: a setting object that contains an indication of the model to be created

- *batchMode*: flag used to enable or disable the plots

Setting object

Before continuing with the overview about *statusTrain()* it is convenient to describe the *setting* object since it will be used inside that function. This class is a container for all the information used to build the network, and it is implemented in the file *status_now_classes.py*. In particular, it provides a way to specify all the characteristics the target neural network should have in a setting file. All the parameters stored in this class will be used in the *statusTrain()* function and a list with a brief description is presented in table 3.3.

Variable	Description
start_date	Dataset start date
end_date	Dataset end date
plants_to_use_list	List of plants used for training
params_to_use	List or features used for training
outputs	Name of network's output
n_samples_per_parameter	Number of samples for each prediction
n_overlap_of_samples	Overlap of two consecutive groups of samples
learning_rate	Learning rate for optimization algorithm
momentum	Momentum for optimization algorithm [9]
batch_size	Batch size
model	Neural network model
loss_fn	Loss function
epochs	Number of epochs
folds	Number of folds for K-Fold cross validation
testMode	Flag used for debug purposes
remove_n	Flag to remove last n samples
trainWithAllData	Specifies if
dataset_setting	List of dictionaries with information about the manipulation of the dataset

Table 3.3: Setting class variables

Particular attention has to be given to the *dataset_setting* parameter: it is a list of dictionaries that contains information that indicates to the *statusTrain()* function how the dataset will be manipulated before it is used. To do this, the indications collected from *dataset_setting* are then passed to the *buildDataset()* function described in section 3.2.2. In table 3.4, it is possible to read all the keys in this class and what they represent.
At this point it is possible to continue the analysis of the *statusTrain()* function, in figure 3.3 it is represented a flowchart of the operations performed inside it. First, all the parameters passed through the *chosen_setting* dictionary are collected. Then, the dataset is created by employing *buildDataset()* function, and then a *PlantsDataset* object is instantiated; during this process, data are shuffled to allow a proper learning process; now, the dataset is ready to be used. Inside *chosen_setting* is specified the desired number of epochs for the training; this number is used to loop over all the batches of the dataset. Inside this loop the *train_loop()* function is called to perform an optimization step on the specified model (the one

Key	Description
param_name	Name assigned to the feature
input_name	Name of the CSV column
norm_data_range	Specified range for the feature
norm_data_median	Mid-range of the feature
transform_function	Applied data transformation
transform_function_kwargs	Arguments for transform function
filter	Filter function
filter_kwargs	Arguments for the filter function
constrain_min	Feature lower limit
constrain_max	Feature upper limit

Table 3.4: Dataset setting variables

passed through the *chosen_setting* dictionary). Then the model is tested after the optimization using the *test_loop* function. The model that exhibits the lower error will be saved at the end of the loop. When the iterations over the whole dataset are concluded, the function *statusTrain()* returns the best model found, the last model trained, and, for each of them, a summary of the performances and all the figures associated.

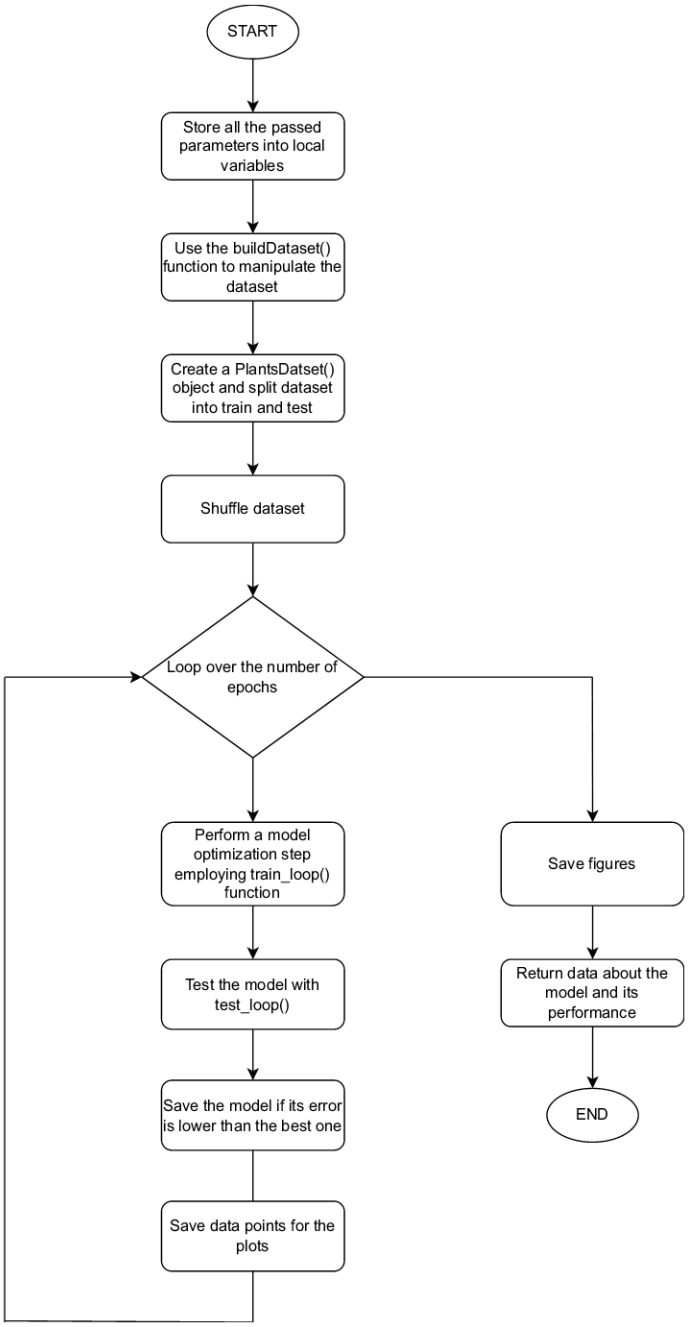


Figure 3.3: *statusTrain() flowchart*

3.2.4 Multiple neural networks training

The other key feature of this framework is the possibility to train multiple neural networks automatically, and it is implemented in the file *status_now_finder.py*.

Like *status_now* functionality, it is also possible to use the file as a standalone script or as a module. The key function of this framework capability is *statusFinder()* whose definition is shown in listing 3.8.

Listing 3.8: *statusFinder() definition*

```
def statusFinder(plant_df: list, plant_list: list, search_type: str,
    save_folder_root: Path = None, useKFold: bool = True,
    decideWithFullData: bool = True, finder_setting: FinderSetting =
    None, batchMode: bool = False):
```

This function accepts the following parameters:

- *plant_df*: list of Pandas data frame obtained by CSV files
- *plant_list*: list of names assigned to the plants
- *search_type*: type of sweep of the neural network parameters
- *save_folder_root*: directory in which all the results and models obtained will be saved during
- *useKFold*: a flag that indicates if the K-Fold cross-validation technique is used
- *decideWithFullData*: a flag that is used to decide if all data are used for training
- *finder_setting*: new setting object specific for the sweep functionality
- *batchMode*: a flag that enables or disables the plot of images

Finder setting

For this framework feature, a new setting type is needed to tell the software how to perform the simulations. For this reason, a new class, called *FinderSetting* is implemented. It stores a set of variables that will be used by *statusFinder()* to batch-train multiple neural networks, and its implementation is presented in listing 3.9.

Listing 3.9: FinderSetting class

```
class FinderSetting:
    def __init__(self, search_type):
        self.common_settings = None
        self.RMSE_threshold_to_save = float("inf")
        self.loss_threshold_to_save = float("inf")
        self.Accuracy_to_save = 0
        self.F1_score_to_save = 0
        self.MCC_to_save = 0
```

```
9           if search_type == "time1":
10              self.start_date = ""  # YYYY-MM-DD hh:mm:ss format
11              self.end_date = ""  # YYYY-MM-DD hh:mm:ss format
12              self.window_min = 0  # days
13              self.window_max = 0  # days
14              self.window_change_step = 0  # days
15              self.window_move_step = 0  # days
16              self.min_samples_for_training = 1  # Amount of samples in
     window to start training
17              self.plants_to_cycle = []  # List of plant names to cycle
     one. Training can happen only on a single plant for this type of
    search
18          elif search_type == "nnShape1":
19              self.layer1_neurons_list = []
20              self.layer2_neurons_list = []
21          elif search_type == "nnShape2":
22              self.n_layers_values = []
23              self.n_neurons_values = []
24          elif search_type == "nSamplesPerParameter1":
25              self.n_samples_per_parameter_min = 0
26              self.n_samples_per_parameter_max = 0
27              self.n_samples_per_parameter_step = 1
28          else:
29              raise Exception("Search type not implemented.")
```

The class only implements the *___init___()* method, which is called as soon as the object is instantiated. The variable *common_settings* is an instance of the *dataset_setting* class and contains the basic information about the models to be created. The other parameters are specific to the search type performed, so a "switch" structure is used to define only the needed quantities in every case. Four types of search are implemented, but more can be easily added by adding new if statements with the proper variables inside, as well as the search implementation inside the *statusFinder* function:

- *time1*
- *nnShape1*
- *nnShape2*
- *nSamplesPerParameter1*

time1 search aims to find the best temporal window in which to perform the training. This window is searched between the values *start_date* and *end_date* and for one plant at the time. The process begins by varying the time frame size between the values *window_min* and *window_max* and each variation is made by an amount equal to *window_change_step*. Then, the position in time

of the window is moved by a quantity *window_move_step* at the time. It is important to notice that each step of the window corresponds to a new model trained. *nnShape1* allows to try different configurations for a neural network. In particular, many networks with two layers are trained, and the amount of neurons belonging to each layer is changed. The parameters that must be provided are two lists containing the number of neurons the user wants to try in each layer; then, every combination of the values present in these two lists is used. For example, supposing the list for layer 1, *layer1_neurons_list* equal to [2,4] and the one for layer 2, *layer2_neurons_list* equal to [3,6], then the following combinations of layers will be trained: 2-3,2-6,4-3,4-6.

nnShape2 allows for varying the number of neurons per layer and the number of hidden layers used. In this case, it is only possible to have the same amount of neurons for each layer inside a single simulation. For example having the parameter *n_layers_values* equal to [2,4] and *n_neurons_values* equal to [3,6] will mean training a total of four neural networks, in particular:

- a neural network with 2 layers and 3 neurons per each layer
- a neural network with 2 layers and 6 neurons per each layer
- a neural network with 4 layers and 3 neurons per each layer
- a neural network with 4 layers and 6 neurons per each layer

The last type of search concerns how many samples to look at in the past to make a prediction. This functionality modifies the parameter *n_samples_per_parameter* of dataset_setting object. This type of sweep is performed in the range *n_samples_per_parameter_min* - *n_samples_per_parameter_max* with variations equal to *n_samples_per_parameter_step*. The simulations described above require, inside each iteration, defining a different structure for the neural network or the dataset. Once these aspects are defined, the training process consists of just training a single neural network at each cycle. This can be accomplished by calling the *statusTrain()* function. For this reason the *statusNowFinder()* function loops over the different setups to be analyzed, for each of them defines the appropriate network and dataset structure, and then calls the *statusTrain()* function. After the training process is ended, the RMSE and loss of the best and last model found are stored, and if they are higher than the thresholds *RMSE_threshold_to_save* and *loss_threshold_to_save*, the current model and the corresponding plots are saved inside the *results* folder. At this point, the process is repeated for each model considered in the selected sweep. Figure 3.4 shows a flowchart of the behavior of the *statusFinder()*

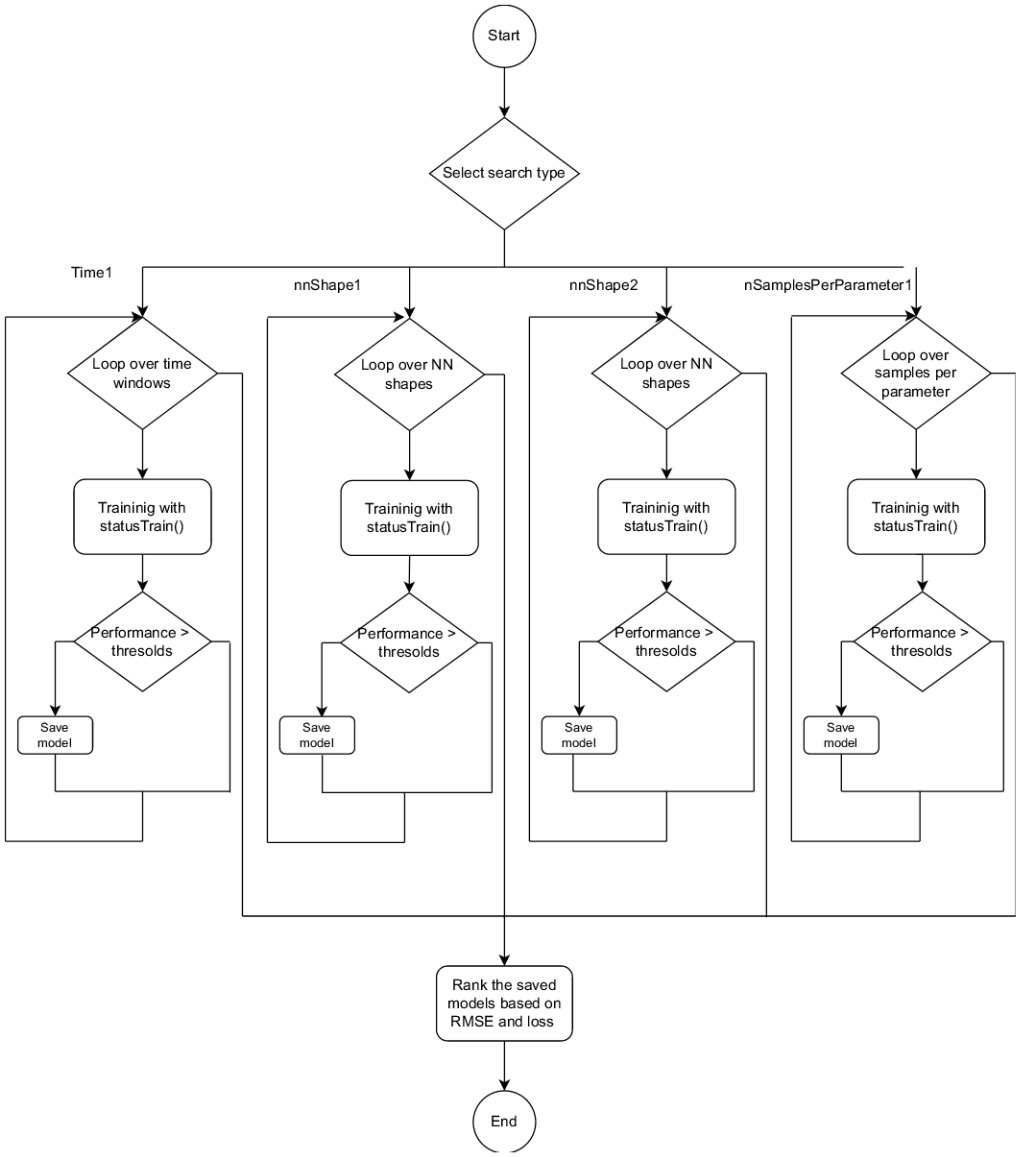


Figure 3.4: *statusFinder() workflow*

3.2.5 Dispatcher and setting files

The *Dispatcher* is a functionality developed in the file *status_now_dispatcher.py* that allows the user to run single or multiple neural networks simulations without launching directly the *status_now.py* or *status_now_finder.py* files. It also allows the complete usage of the setting files in which a *Setting* or *FinderSetting* object can be defined to have an easy way to launch different simulations by just providing different setting files. A setting file is created by the user each time a simulation has

to be performed and it creates a *Setting* or *FinderSetting* object depending on the type of mode, single or multiple training, the user wishes to use. The *Dispatcher* accepts as an argument a setting file, and it then launches the simulation by calling the *status_now.py* or *status_now_finder.py* with the parameters specified by the user inside the setting file. All the setting files the user wants to use must be stored in the *settings* folder.

3.2.6 Dispatcher usage

The scripts described up to now can be launched as standalone elements, however, it is more convenient to employ them by calling the dispatcher since it allows to use custom setting files without modifying the code of *status_now.py* or *status_now_finder.py*. The syntax to be used from the command line is the following:

```
1    python3 status_now_dispatcher.py script -f <Setting File> -b <
   Batch>
```

The *script* argument can be set to *status_now* or *status_now_finder* and it is used to select the framework functionality to use. The *-f <Setting File>* argument specifies the name of the setting file used for the simulation. The specified setting file is searched inside the *settings* folder, and an error is returned if it is not present. The *-b <Batch>* argument is used to select between batch mode or not batch mode. By setting *<Batch>* to "yes," batch mode is selected, and no plots are displayed on the screen by the framework; on the contrary, if <Batch> is set to "no," the batch mode is disabled, and the framework will print on screen the plot containing features values, loss values, and RMSE values. Note that for the classification case, no plots regarding the evaluation metrics are produced, so only images about the values of input features will be printed on screen if classification is performed and batch mode is not selected. To allow the user to easily perform simulations, template files for *status_now* and *status_now_finder* search types are provided in the *settings* folder. In this way, the user can easily compile them with the desired values for the simulation.

Example: single neural network training

Suppose to have a setting file called *single_training.py* inside the *settings* folder, and the user wants to launch a single training with batch mode enabled. It is possible to launch the following command.

```
$ python3 status_now_dispatcher.py status_now -f single_training
-b yes
```

Multiple neural networks training

Suppose to have a setting file in the *settings* folder called *sweep_search.py* and the user wants to launch a sweep over the neural network parameters with batch mode enabled. It is possible to type the following command to perform this simulation.

```
$ python3 status_now_dispatcher.py status_now_finder -f
sweep_search -b yes
```

To display a help page for the *Dispatcher* it is also possible to type the following instruction.

```
$ python3 status_now_dispatcher -h
```

3.3 Framework modifications

So far, the general structure of the framework and its functionalities have been analyzed. The first version of the software was able to train neural networks effectively but was only able to perform regression problems. This section presents the modifications to make the framework able to train neural networks suitable for classification problems. In particular, binary classification is the objective of the training since the plants have to be classified as healthy or unhealthy. The code was modified by adding the required functions but respecting the software's original workflow as much as possible.

3.3.1 Dataset manipulation

A new feature needs to be added to the dataset to perform binary classification inside the framework: the *Status* of the plant. It is a variable that can assume two values: 0 if the plant is considered unhealthy and 1 if it is considered healthy. This new feature does not belong to the available CSV files. To obtain an updated dataset, a new file, called *modify_csv.py*, is created, and it contains two useful functions to manipulate the CSV files and get them ready to be used to train a classifier: *add_columns()* and *remove_columns()*. The definitions of these functions are presented in listings 3.10 and 3.11.

Listing 3.10: *add_columns()* definition

```
1 def add_columns(filename,to_be_added_dict=None, by_dict=False,by_date
    =False,start_date=None,stop_date=None,value=None,col_name=None):
```

add_columns, as the name suggests, allows for the addition of one or more columns in the CSV file, whose path is specified by the *filename* parameter. This function is helpful to label the data, in this case, by adding the observed status of the plants. The addition of a column can be made by setting the flag *by_dict* to True and providing a list of dictionaries, *to_be_added_dict*, to the function. Each dictionary inside the provided list must have a key that will be the name of the column added and the corresponding value, which is a list containing all the numbers to be added. Another way to add one or more columns is by specifying a range of dates in which a certain value has to be inserted, and outside of it, another value is put. To modify the CSV file in this way is necessary to set the flag *by_date* to True, provide a date range by setting *start_date* and *end_date*; provide a list of column names to be added (parameter *col_name*); provide a list,*value*, in which each element is a list itself that specifies, for each column to be added, the value that has to be set inside the date range and the one outside of it. For example, suppose providing the *col_name*= [var1,var2] and *value*= [[2,3] , [4,5]]; it is created a column var1 in which values inside the specified date range are set

to 2 and to 3 outside; then another column called var2 is created with the value 4 inside the date range, and 5 outside.

Listing 3.11: *remove_columns()* definition

```
def delete_columns(filename, col_to_delete):
```

The second function, called *delete_columns*, helps delete one or more columns from a CSV file, and it is shown in listing 3.11. The parameters to be passed are, respectively, the name of the target file to be modified and the name of the column that must be deleted. These two functions were used to label the plants, in particular, by visual inspections of the plants performed by previous students it was decided to perform the labeling as follows:

- *Plant 1*: healthy from 23-03-2021 to 21-04-2021 and unhealthy later
- *Plant 2*: unhealthy from 23-03-2021 to 21-04-2021 and healthy later
- *Plant 3*: unhealthy from 23-03-2021 to 21-04-2021 and healthy later
- *Plant 4*: healthy from 23-03-2021 to 21-04-2021 and unhealthy later

For what concerns the dataset a little modification is made to the *buildDataset()* function. The labels given to the plants must not be transformed, normalized, or constrained. It was added to the framework the possibility to exclude some columns from this process since, by default, *buildDataset()* applies this operation to all the columns in the data frame.

Listing 3.12: Modified *buildDataset()* definition

```
def buildDataset(plant_df, plant_list, dataset_settings,
    plants_to_use_list=None, remove_n=0, start_date=None,
        end_date=None, date_param_name="Date", testMode=False,
    exceptions=None):
```

It can be noticed that the function accepts a new parameter, called *exceptions*, and it is a list of strings containing the name of the columns that have to be left unmodified. In this case, it is possible to exclude the column containing the output labels from the manipulation process.

3.3.2 Single neural networks training - Modifications

To perform classification problems the *statusTrain()* function inside *status_now.py* file is modified. First, a flag called *binaryClassificationFlag* is added to the *Setting* class, which is set by the user when binary classification has to be performed. Concerning the simple flowchart of figure 3.5, the original framework's single

training function is kept as it was and can be used when it is needed to perform regression problems. At the same time, the new parts of the code are added to the other branch of the behavior. From now on, the instructions described belong to

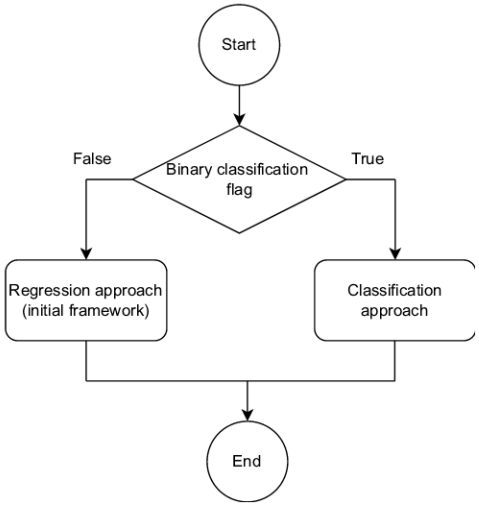


Figure 3.5: Modified *statusTrain()* workflow

the branch in charge of performing classification. First, the dataset is created using the *buildDataset()* function. For the case of plants status classification as *exceptions* parameter the "*Status*" string is passed to indicate that the *Status* column in the CSV files has to be left untouched. At this point, a new parameter is introduced: the evaluation metric. In the setting class, a new variable is added, and it is called *binaryClassificationMetric*; it is a string and can assume three values:

- Accuracy
- F1 Score
- MCC

Many techniques exist to evaluate a learning algorithm, and accuracy is the simplest and more intuitive. However, accuracy can be misleading, so it was added inside the framework the possibility to use other evaluation metrics. For the framework case, the *F1 Score* and the *Matthews Correlation Coefficient*. Accuracy can be misleading when evaluating a learning algorithm's results. The following example is proposed to understand why: suppose to perform binary classification and have a dataset that is not well balanced between positive and negative cases. Suppose that 95% of the labels are 0 and the remaining 5% are 1. It is possible to predict always the result 0 without considering the features, and the resulting accuracy would be 95%, which seems an incredible result, but in reality, it is not how a

learning algorithm should work, and accuracy is not showing its real performances. To avoid this problem the metrics *F1 Score* and *Matthews Correlation Coefficient* are inserted in the framework.

F1 Score

The F1 Score is evaluated starting from two other quantities, *precision* and *recall*. The precision is defined as the number of correctly classified positive results divided by all the actual positive results. The recall is the number of true positives divided by all items that should be identified as positive.

$$Precision = \frac{true\ positives}{true\ positives + false\ positives} \tag{3.1}$$

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives} \tag{3.2}$$

The F1 Score is defined starting from these two quantities as the harmonic mean between precision and recall and can assume values between 0 and 1.

$$F1\ Score = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{3.3}$$

To understand this metric better, it is possible to calculate the F1 Score for the previous simple example. The values for true positives, true negatives, false positives, and false negatives can be written inside the so-called *confusion matrix* as follows:

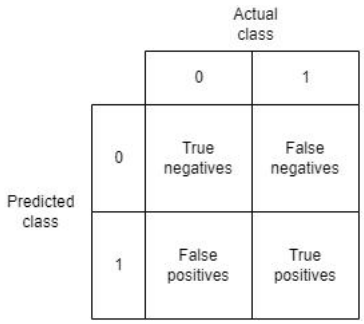


Figure 3.6: Confusion matrix

For this specific case, the confusion matrix results as follows:

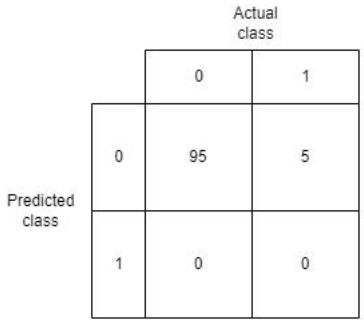


Figure 3.7: Example confusion matrix

At this point, it is possible to evaluate the F1 Score. By looking at the F1 Score definition, it is possible to notice that if precision or recall is equal to 0, the F1 Score will be equal to 0. In this case, recall is equal to 0, so it is also the F1 Score indicating that the algorithm employed is not adequate.

Matthews Correlation Coefficient

The other metric introduced inside the framework is the *Matthews Correlation Coefficient* (MCC). Accuracy and F1 Score are both sensitive to class imbalance, which can be present inside the dataset and can be misleading. The MCC can consider all four cases present in the confusion matrix to give a score that considers the algorithm's performance concerning the eventual class imbalance in the dataset, an aspect not considered by accuracy and the F1 Score. The MCC can assume values in the range [-1,1], where 1 indicates the perfect classifier, 0 is a classifier not better than random guessing, and -1 is a classifier perfectly opposite to the perfect one. Since the formula which defines the MCC is quite long, a more concise notation is employed:

- true positives $\longrightarrow$ tp
- true negatives $\longrightarrow$ tn
- false positives $\longrightarrow$ fp
- false negatives $\longrightarrow$ fn

The MCC is defined as:

$$MCC = \frac{(tp \cdot tn) - (fp \cdot fn)}{\sqrt{(tp + fp) \cdot (tp + fn) \cdot (tn + fp) \cdot (tn + fn)}} \tag{3.4}$$

It is possible to understand the properties of this metric by looking at this formula. Considering a perfect classifier, we have that fp=fn=0. In this case, evaluating

will lead to MCC=1, so a perfect prediction. Supposing an algorithm that always misclassifies, we have that tp=tn=0 and so MCC=-1, indicating that predictions are always opposite to the correct ones. Suppose instead to have a dataset. It is also possible to consider the situation in which the dataset is composed of only one label. In this case, the MCC will always be equal to 0; this situation will be encountered in the next section when predicting future data of a single plant. This situation is useful to know if a classifier can predict the status of a "stable" plant. However, MCC is unsuitable for this case so simple accuracy will be considered. A more exhaustive explanation of the concepts and advantages of the *Matthews Correlation Coefficient* are provided in [24] [25] [26].
Going back to the single training functionality, it is possible to select which metric to use among the three just described. The metric selected will be the one employed to find the best model to be trained; for example: if *binaryClassificationMetric* is set to "Accuracy," the model with the best accuracy will be saved, and the same is done for the cases "F1 Score" and "MCC." The functions useful for the calculation of the MCC and F1 Score are implemented in the file *utilities.py* and are called *MatthewsCorrCoeff* and *F1_Score*, respectively.
At this point, the dataset is shuffled and split into train and test portions using the *PlantsDataset* class. Test and train dataset labels are then counted, and the result is printed on the screen to inform the user how many samples present a label equal to 0 and equal to 1, in this way, the user can understand if he is working with a balanced case study or not. Now it is all ready to perform the training process. In figure 3.8 it is present a flowchart that describes the operations performed to train a single neural network, it is similar to the one of the original *statusTrain()* function presented in figure 3.3 with some differences. The core of the function is the same, a loop over the selected number of epochs, but in this case, the single training step is performed by a different function called *train_class_loop()*. This function is slightly different from *train_loop* and is in charge of performing a training step for the binary classification case, which will be described in the next section. After the training step, the model is again tested using the function *test_class_loop()* that is similar to *train_class_loop()* with the difference that, in this case, the model is only tested, but no backpropagation is applied to the network. At every epoch, after the training step and the test are performed, the best model is updated if the results are better for the selected evaluation metric than those obtained in the previous cycles. At the end of the epochs, the last and the best model (according to the evaluation metric selected) are saved in the *results* folder as ".pth" files. In the same folder, a file called *results.txt* is created, and it reports the performance of the trained models.

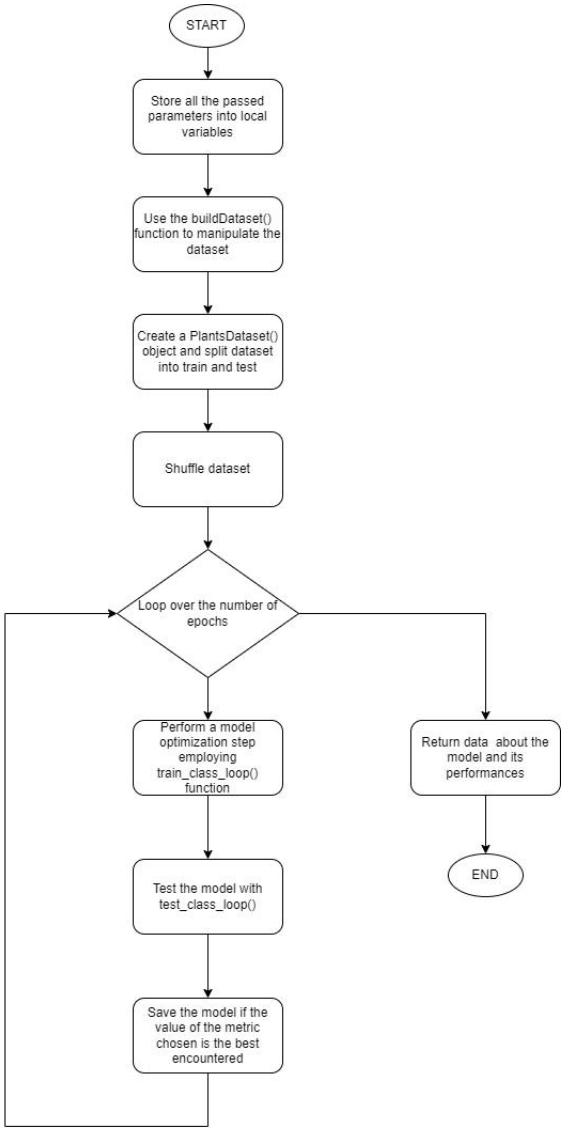


Figure 3.8: *statusTrain()* workflow modified

train_class_loop() and test_class_loop()

Listing 3.13: *train_class_loop()* definition

```
1 def train_class_loop(dataloader, model, loss_fn,optimizer):
```

In listing 3.13 is presented the definition of the *train_class_loop()* function. It can be noticed that the function accepts the same four parameters of the original *train_loop()* function employed for regression problems. What is changed is that, in this case, retrieving the outputs of the neural networks is different from the

regression case. When dealing with regression problems, the output is obtained by looking at the value of the output neuron, and the prediction quality is evaluated by simply computing the difference between the calculated value and the expected one, which is included in the dataset. When performing binary classification, the situation is different: by looking at the output neurons, a label, 0 or 1, has to be extracted and compared to the expected one. To do so, two cases are implemented in the framework, employing two different loss functions listed below. Other methods to accomplish this task exist and can be added to the framework by making some modifications to the code; however, only these two, which are the most commonly used, are implemented in this work.

1. *nn.CrossEntropyLoss()* loss function
2. *nn.BCEWithLogitsLoss()* loss function

By passing one of these two functions to *train_class_loop()* using the *loss_fn* the first or the second case is selected by the code. If the *nn.CrossEntropyLoss()* is used; the neural network's output layer should be composed of two neurons, the first associated with the label 0 and the second with the label 1. The values of the output neurons are compared to make a prediction, and the index of the neuron with the greater value will be the predicted value. For example, if the first output neuron (index 0) has a greater value than the second one (index 1), the prediction will be class 0. Regarding the neural network training, the code is similar to the one shown in listing 3.5 for *train_loop*. The significant differences take place in the outputs extraction from the network and are shown in listing 3.14. The code loops over the samples provided by the passed *dataloader*, the values of the output neurons are computed, and the actual value of the predictions is evaluated by the *max()* Pytorch function, which returns the index of the neuron with the greater value. At this point, the number of true positives, true negatives, false positives, and false negatives is evaluated in order to be able to compute the confusion matrix. Then the accuracy, F1 Score, and MCC are computed, and the calculated confusion matrix is printed on the screen.

Listing 3.14: *CrossEntropyLoss* definition

```
1         with torch.no_grad():
2             count1=0
3             count0=0
4             precision = 0
5             recall = 0
6             n_samples = 0
7             n_true_positives = 0
8             n_true_negatives = 0
9             n_false_negatives = 0
10            n_false_positives = 0
```

```
11              n_positives = 0
12              n_negatives = 0
13              for X,y in dataloader:
14                  if X.size()[0] != 1:
15                      outputs = model(X.to(device))
16                      y = y.to(device)
17                      # Squeeze removes the dimensions equal to 1 in
    the tensor. E.G if I have a tensor which has size [3000,1,1] it is
18                      for i in y:
19                          if i.item()==1:
20                              count1+=1
21                          if i.item()==0:
22                              count0+=1
23                      # Due to the default normalization of the
    framework the status is a variable which is either +1 if  the
    plant is fine
24                      #print(f'Labels: {y}. Size: {y.size()}')
25                      _,predictions = torch.max(outputs,1)
26                      for i in range(len(predictions)):
27                          if y[i] == predictions[i]:
28                              if y[i] == 1:
29                                  # True positives
30                                  n_true_positives += 1
31                              else:
32                                  n_true_negatives += 1
33                          else:
34                              if y[i] == 1:
35                                  n_false_negatives += 1
36                              else:
37                                  n_false_positives += 1
38                      # Update the number of cases analyzed
39                      n_samples+=len(predictions)
40          precision = util.bin_precision( n_true_positives,
    n_false_positives)
41          recall = util.bin_recall( n_true_positives, n_false_negatives
    )
42          F1_score = util.F1_score(precision=precision, recall=recall )
43          MCC = util.MatthewsCorrCoeff(TruePos= n_true_positives,
    TrueNeg= n_true_negatives, FalsePos= n_false_positives, FalseNeg=
    n_false_negatives)
44          accuracy = (n_true_positives+n_true_negatives)/n_samples
45          print(f'Count of TRAIN dataset values: {count1} positive
    cases, {count0} negative cases')
46          print(f'Wrong predictions: {n_false_positives+
    n_false_negatives}')
47          print(f'Number of correct predictions for training dataset:
    {(n_true_positives+n_true_negatives)}/{n_samples}. Training
    accuracy: {100*accuracy:.3f} %')
```

```
48          print(f'True positives: {n_true_positives}. True negatives: {
        n_true_negatives}. False positives: {n_false_positives}. False
        negatives: {n_false_negatives}')
49
50          # Long print statement to draw the confusion matrix
51          print(f'TRAIN CONFUSION MATRIX\n\n\n\t\t\t\t      ACTUAL\n\t\t
        \t\t    0\t    1\t\n\t\t\t      __|________|________|\n\t\t\t\t\t|\t|\t
        |\n\t\t\t0\t|   {n_true_negatives}\t|   {n_false_negatives}\t|\n\t\t
        \t       __|________|________|\n\tPREDICTED\t\t|\t|\t|\n\t\t\t1\t|   {
        n_false_positives}\t|   {n_true_positives}\t|\n\t\t\t       __|
        ________|________|\n\n')
52
53
54          print(f'Training precision: {precision:.3f}. Training recall:
         {recall:.3f}. Training F1 score: {F1_score:.3f}. MCC: {MCC:.3f}')
55          return train_loss/size, accuracy, F1_score, MCC
```

If the *BCEWithLogitsLoss* is employed, the output layer has to contain just one neuron, and its output represents the probability that the class predicted is 1. So it is possible to predict by setting a threshold for the output neuron above which the predicted class is 1 and below it is 0. This threshold is set to the value 0.5. Since *BCEWithLogitsLoss* applies the sigmoid function to the output neuron to evaluate the probability of a specific class. It is sufficient to look at the value of the output neuron to understand if the probability is greater or smaller than 0.5. If it is greater than zero, the prediction will be equal to 1 and 0 otherwise (refer to the graph of the sigmoid function in figure 2.4). Also in this case, the part of the code which implements the training is similar to the one of *train_loop* (refer to listing 3.5). Listing 3.15 shows the code created to evaluate a prediction using the *BCEWithLogitsLoss()* function. The training examples are iterated through the dataloader, and the value of the output neuron is computed for each of them. At this point, following the procedure described above, the value of the output neuron is compared to 0, and the value of the prediction is determined. Again, the true positives, true negatives, false positives, and false negatives are evaluated to compute the confusion matrix. Finally, all the possible evaluation metrics are computed, and the confusion matrix is printed on the screen.

Listing 3.15: *BCEWithLogitsLoss*

```
1           for X,y in dataloader:
2               predictions = torch.empty(y.size())
3               outputs = model(X.to(device))
4               y = y.to(device)
5               y.squeeze_()
6               outputs.squeeze_()
7               #print(outputs)
8               for i in y:
9                   if i.item()==1:
```

```
                        count1+=1
                    if i.item()==0:
                        count0+=1
                for index in range(len(outputs)):
                    if outputs[index]>=0:
                        predictions[index]=1
                    else:
                        predictions[index]=0

                for i in range(len(predictions)):
                    if y[i] == predictions[i]:
                        if y[i] == 1:
                            # True positives
                            n_true_positives += 1
                        else:
                            n_true_negatives += 1
                    else:
                        if y[i] == 1:
                            n_false_negatives += 1
                        else:
                            n_false_positives += 1
                # Update the number of cases analyzed
                n_samples+=len(predictions)
        precision = util.bin_precision( n_true_positives, n_false_positives)
        recall = util.bin_recall( n_true_positives, n_false_negatives)
        F1_score = util.F1_score(precision=precision, recall=recall )
        MCC = util.MatthewsCorrCoeff(TruePos= n_true_positives, TrueNeg= n_true_negatives, FalsePos= n_false_positives, FalseNeg= n_false_negatives)
        accuracy = (n_true_positives+n_true_negatives)/n_samples
        print(f'Count of TRAIN dataset values: {count1} positive cases, {count0} negative cases')
        print(f'Wrong predictions: {n_false_positives+n_false_negatives}')
        print(f'Number of correct predictions for training dataset: {(n_true_positives+n_true_negatives)}/{n_samples}. Training accuracy: {100*accuracy:.3f} %')
        print(f'True positives: {n_true_positives}. True negatives: {n_true_negatives}. False positives: {n_false_positives}. False negatives: {n_false_negatives}')
        # Long print statement to draw the confusion matrix
        print(f'TRAIN CONFUSION MATRIX\n\n\n\t\t\t\t     ACTUAL\n\t\t\t\t\t   0\t   1\t\n\t\t\t\t      __|_______|_______|\n\t\t\t\t\t|\t|\t|\n\t\t\t0\t|  {n_true_negatives}\t|  {n_false_negatives}\t|\n\t\t\t\t      __|_______|_______|\n\tPREDICTED\t\t|\t|\t|\n\t\t\t1\t|  {n_false_positives}\t|  {n_true_positives}\t|\n\t\t\t\t      __|_______|_______|\n\n')
```

```
44
45
46          print(f'Training precision: {precision:.3f}. Training recall:
        {recall:.3f}. Training F1 score: {F1_score:.3f}. MCC: {MCC:.3f}')
47          return train_loss/size, accuracy, F1_score, MCC
```

3.3.3 Multiple neural network training - Modifications

The possibility of training a single neural network to perform classification tasks is also extended to the case of multiple neural network training by making some modifications on the *status_now_finder.py* file. The modified workflow is similar to the one represented in figure 3.4. The difference now is that the classification function and metrics can also be used for each of the available search types previously presented. Also in this case, the *BinaryClassificationFlag* is used to discriminate between classification and regression problems. Figure 3.9 shows a flowchart explaining the operation performed by the functionality. First, the general parameters and settings required by the *statusFinder()* function are collected. Note that the definition of *statusFinder()* is not changed so its definition and accepted arguments are the same described in the previous sections (refer to listings 3.8 and 3.9). From the *finder_setting* argument, the search type and all the flags necessary for the execution are retrieved. Then, the function checks the *BinaryClassificationFlag* (which is passed inside the *finder_setting* argument) and, if False, the normal operations for regression problems seen in previous sections are performed; if True, the algorithm loops over the structures of the neural network specific to the search type, it performs the training using *statusTrain()* function and saves all the models that meet a certain threshold for the specified evaluation metric. Note that the evaluation metric and the thresholds for the model are specified inside the *finder_setting* object. When all the specified models are trained, the function creates a file called *rankings.txt*, in which all the models saved are ranked based on their performance. Once the execution is concluded, all the models and results are saved inside the *resutls* folder. Regarding the usage of the software employing the dispatcher, no changes are made with respect to the original code, so the same commands can be typed to perform the simulation; what is changed is just how it is performed. For information about the framework's usage, refer to 3.2.6.

Figure 3.9: Modified *statusFinder()* workflow

3.4 Converting PyTorch model to ONNX

The Python models trained with the framework are saved in files with the .pth extension. In the last chapter of this thesis a tool, called X-Cube-AI, is used to automatically create an optimized C library that allows the implementation of the algorithm on a microcontroller. This software can convert files from the main Python deep learning frameworks (such as Keras or Tensorflow) but the .pth format

of Pytorch is not supported. What is supported is the ONNX, an open format built to represent machine learning models [27], so a Python script to convert .pth files to ONNX is written and shown in listing 3.16.

Listing 3.16: *PyTorch to ONNX conversion*

```
import torch
import torch.onnx
import status_now_classes
import pandas as pd
import utilities as util


model = torch.load('../results/1234TrainingImage1/0004
    _best_model_hiddenLayers_1_neurons_10.pth')
model.eval()
batch_size = 1
num_features = 8
x = torch.randn(batch_size,num_features,requires_grad = True)
output = model(x)
print(model)
torch.onnx.export(model,x,'Four_plants_10.onnx',export_params=True,
    opset_version=10,do_constant_folding=True,input_names=[],
    output_names=[],dynamic_axes={ 'impedance_phase':{0:'batch_size'},
     'impedance_modulus':{0:'batch_size'}, 'moisture':{0:'batch_size'
    },'temperature':{0:'batch_size'}, 'Ambient Light': {0:'batch_size'
    },'day_of_year':{0: 'batch_size'},'hour':{0:'batch_size'},'
    AirHumidity':{0:'batch_size'}, 'Status':{0:'batch_size'}     })
```

The model is loaded by using the *torch.load()* command specifying the path of the model to be converted. Then the *torch.onnx.export()* command is used to convert and save the model as an ONNX file.

Chapter 4

Framework Simulations

In this chapter, the developed framework is employed to perform many simulations useful to find a suitable neural network structure. First, the training is performed on all four plants available, and the test on future data belonging to the same plants. Then, only two plants are used for training, and the remaining are used to test the neural networks. Finally, the trained models are converted from the Pytorch .pth file to the ONNX [27] format.

4.1 Training on four plants

In this section, all four plants available are used for the training phase. The samples used are from 24-03-2021 to 28-05-2021, and 80% of them are used as the training dataset while the remaining 20% as a test. Many combinations of possible structures are tried to compare different results; in particular, many parameters are changed to see how they affect the quality of the prediction. Different setups analyzed:

- Different number of layers
- Different number of neurons per layer
- Different combinations of features
- Different number of samples employed for each prediction

Moreover, a testing phase on future data of the same plants used for training is performed.

4.1.1 One hidden layer networks

First, simpler networks with only one hidden layer are trained, and the number of neurons is varied from 4 neurons to 20 neurons. All the hyperparameters employed for this simulations are shown in table 4.1.

Hyperparameter	Value
Activation function	ReLU
Loss function	Cross entropy loss
Features	Impedance modulus,Impedance phase,Soil moisture Temperature,Ambient light,Air humidity Day of the year,Hour
Samples per parameter	1
Overlap of samples	0
Batch size	10
Epochs	50
Learning rate	0.001

Table 4.1: Simulation parameters

In figures 4.1, 4.2 and 4.3 are shown the results of the simulation for the three evaluation metrics available for classification problems, that are accuracy, Matthews correlation coefficient, and F1 Score, versus the number of neurons, computed over the 20% of data used as test dataset. Red curves represent the calculated metrics values, while the black-dashed lines represent the trend for these plots. All the figures show that when more neurons are employed, predictions tend to be more precise. Another thing to notice is how these three graphs are similar; since F1 Score and Matthews correlation coefficient are quantities that take into account also the balancing of the dataset, the fact that their plots are similar to the one of the accuracy indicates that the used dataset is correctly balanced between positive and negative cases, i.e., healthy and unhealthy plant samples. Considering accuracy, the performances of the networks range from a minimum of 82% of the network employing only four neurons up to 93.5% reached by the ones with 13,16, and 19 neurons. To understand better the quality of the models obtained, they are then tested on samples of the same four plants starting from 29-05-2021 to 20-07-2021; in this way, it is possible to appreciate how the networks perform on future data belonging to the same plants.

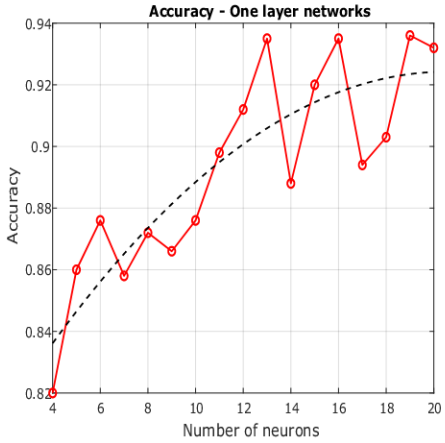


Figure 4.1: Accuracy vs number of neurons

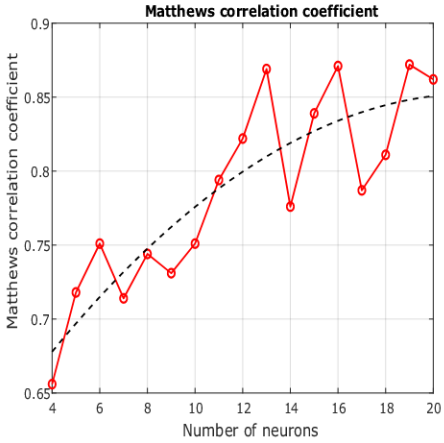


Figure 4.2: Matthews correlation coefficient vs number of neurons

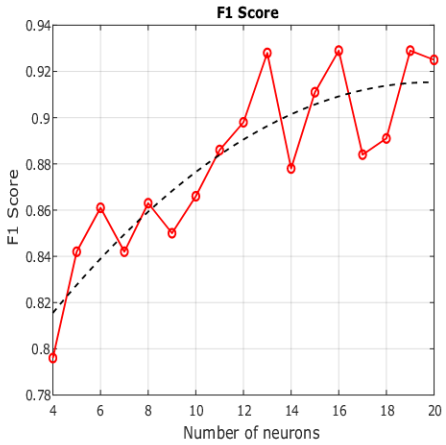


Figure 4.3: F1 Score vs number of neurons

The results of this simulation are shown in figure 4.4. In this case, results vary a lot, also for close numbers of neurons. The black dashed line indicates the trend, a slight rise in performance can be noticed as the number of neurons approaches the upper limit; however, differently from the previous case, this behavior is less evident, and it is difficult to judge which architecture is performing better.

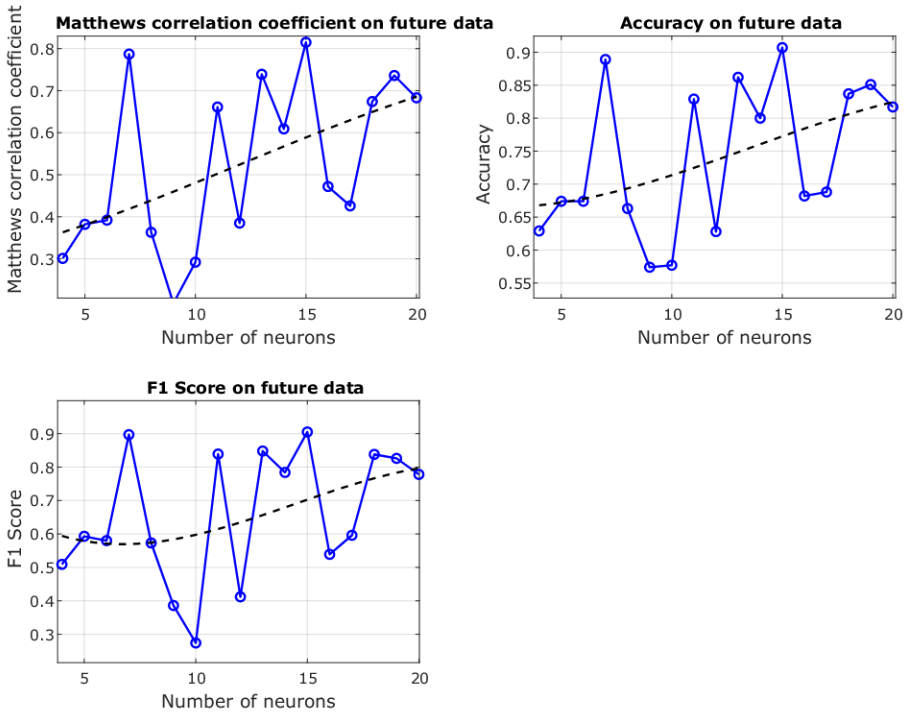


Figure 4.4: Test results on future data

Predictions with only impedance

The same structures employed in the previous section are used in this paragraph. However, this time the only features used are the *impedance modulus* and *impedance phase* to understand if it is possible to predict health status only employing this characteristic of the plants. The simulation results are depicted in figure 4.5. Table 4.2 describes the settings employed for this analysis.
As before, the three evaluation metrics are computed for each model and these values are plotted against the number of neurons inside the only hidden layer inside the networks. The red lines represent the actual values, while the black dashed one represents the trend for the prediction accuracy. In this case, a performance drop can be noticed with respect to the previous case in which all the available features are implemented. In particular, the best model, according to the plots, is the one with 18 neurons that reaches a prediction accuracy of 80 %. This result can be compared to the best model obtained in the previous case, which showed an accuracy of 92.9 %. Also, since the dataset is well-balanced between positive and negative cases, all the evaluation metrics show the same behavior, so an analysis based on simple accuracy can be performed without worrying about misleading

Hyperparameter	Value
Activation function	ReLU
Loss function	Cross entropy loss
Features	Impedance modulus Impedance phase
Samples per parameter	1
Overlap of samples	0
Batch size	10
Epochs	50
Learning rate	0.001

Table 4.2: Simulation parameters - Only impedance

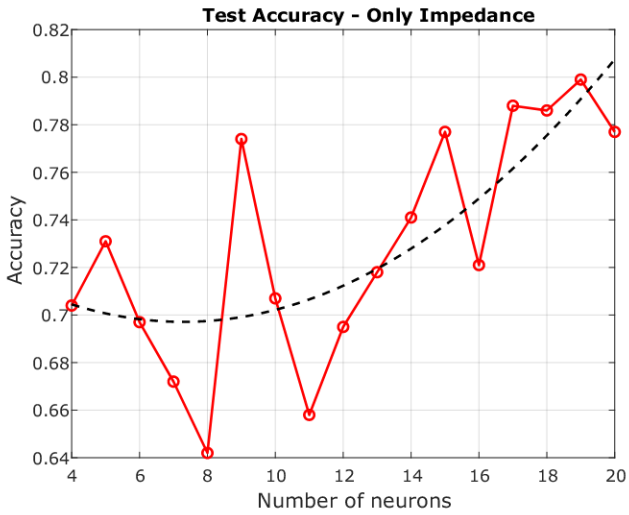


Figure 4.5: Accuracy vs number of neurons - Impedance only

results. Regarding the accuracy trend versus the number of neurons, on average, it tends to grow when approaching a larger number of neurons; however, these results also present much variability for close number of neurons considered. In particular, the model with 9 neurons presents an accuracy close to 64 %, while the successive one, with 10 neurons, performs a lot better, with an accuracy close to 80 %. Also in this situation, the models are tested on future data belonging to the same plants. It can be noticed, looking at figure 4.6, that the performance of future data is awful. Looking at accuracy, It is possible also to notice, looking at accuracy, that this value is always below 50 %. Considering that the classification performed is the binary one, having an accuracy below 50 % means that the model's predictions would perform better if reversed, and this situation is not acceptable.

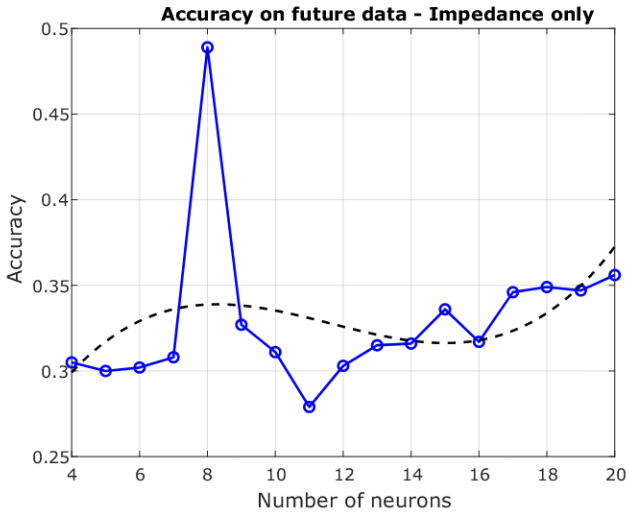


Figure 4.6: Future data results - Impedance only

Figure 4.7 shows the model's performance on each plant taken individually. All the networks perform perfectly on plant 1 and almost all of them, excluding the network with 8 neurons inside the hidden layer, fail to predict the status of plant 3. Performance on plants 2 and 4 are better than the ones for plant 3, but still not acceptable since the accuracy does not reach, in any case, 50 %. With respect to the case in which all the features are employed, the predictor performs optimally on plant 1 and then presents worse performance on the rest of the plants; however, in this new scenario, the situation is even more evident, and overfitting seems to be stronger. This fact is probably due to the reduced number of features used that are not enough to predict the plant's status accurately.

Impedance and moisture

Since implementing a prediction with only impedance data led to poor results due to strong overfitting, in this section, another feature, the soil moisture, is added when performing the training. Table 4.3 shows the settings used in the following simulations.
Also in this section, the performance analysis will be the same. First, the prediction quality is evaluated on the 20% of the plant samples left as a test dataset; then, the same analysis is performed on future data. In this case, since the employed dataset is the same as other cases and so are the labels belonging to it, the plots of F1 Score and MCC are excluded since they present the same accuracy behavior due to the balance in the dataset. The first results are shown in figure 4.8 and are quite better than the previous case in which only impedance was considered: here,

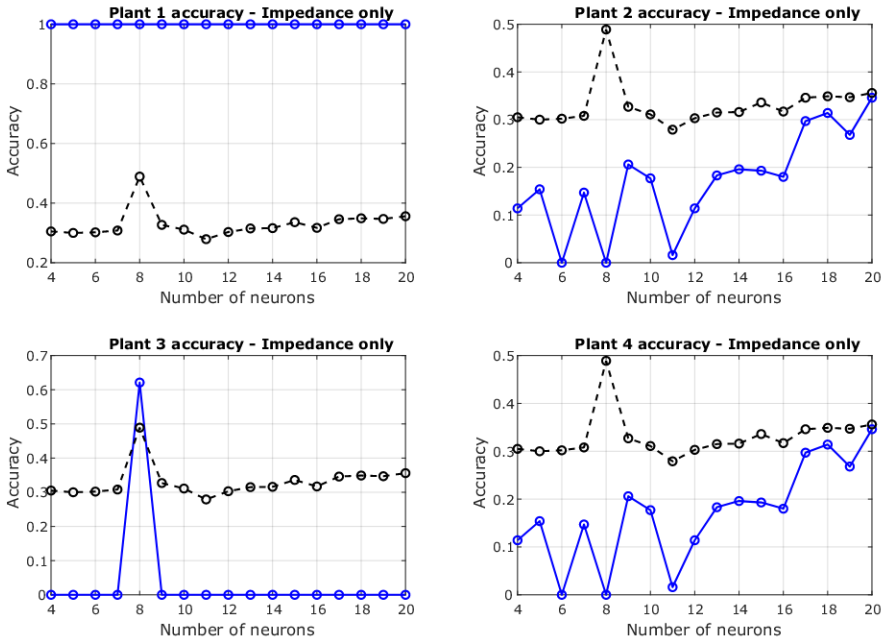


Figure 4.7: Future data results - Impedance only plant by plant

Hyperparameter	Value
Activation function	ReLU
Loss function	Cross entropy loss
Features	Impedance modulus Impedance phase Soil moisture
Samples per parameter	1
Overlap of samples	0
Batch size	10
Epochs	50
Learning rate	0.001

Table 4.3: Simulation parameters - Impedance and soil moisture

the accuracy range from 79% to 84 % while previously was from 64% to 80%. Like before, the performance is quite variable and shows a slightly increasing trend with the number of neurons.

Regarding future predictions, whose results are shown in figure 4.9, adding the moisture boosts the performance, but still, they are not optimal. In this case, the

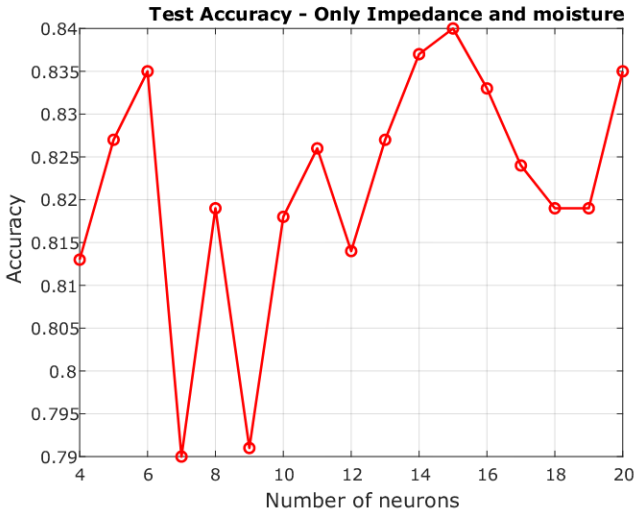


Figure 4.8: Accuracy vs number of neurons
Impedance and soil moisture

accuracy ranges from 45% to 61%, which is better than the impedance alone case, in which the same parameter varied from 25% to less than 50%. For this reason, we can notice how adding a new significant feature to the network helped to reduce the problem of overfitting even if performance is still not satisfying.

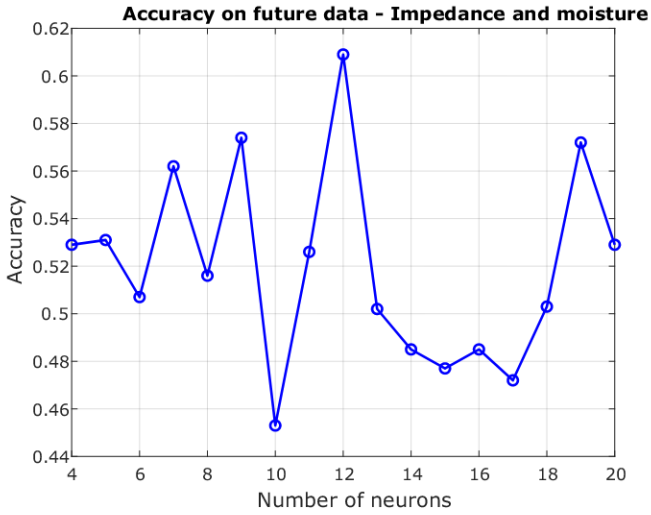


Figure 4.9: Accuracy vs number of neurons - Future data

Samples per parameter

In this section, the number of samples in the past for each prediction is varied to see if it positively affects the algorithm's performance. In particular, four cases are analyzed:

- 6 samples in the past
- 12 samples in the past
- 18 samples in the past
- 24 samples in the past

Since each sample is acquired every hour, this technique implies using time windows of 6,12,18, and 24 hours, respectively. The time windows move by one sample, meaning that a new prediction is provided every hour. Figure 4.10 shows the obtained results on the 20% of the dataset used for testing. All the curves representing the simulations where some samples in the past are employed show better performance than the case in which no past samples are considered. All the cases that consider past samples have similar performance that is in the range of 90-95%; however, the case where 6 past samples are considered is promising: it performs well for almost all the number of neurons, moreover, considering that only 6 samples in the past are considered it results in fewer input neurons, resulting in a simpler network. This aspect can be favorable, looking forward to a low-power implementation on a microcontroller.

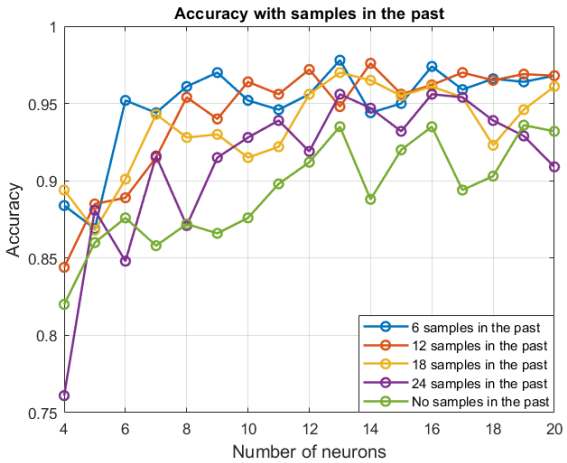


Figure 4.10: Accuracy vs number of neurons - Samples in the past

Figure 4.11 shows the simulation results on future data of the same plants used for training. In this situation, an overall drop in the performance is present since

almost any network reaches the score of 90% accuracy, while in the previous case, many topologies reached the 95%. Moreover, greater oscillations in the performance are present with respect to the previous case in particular for the cases with 18,24 and no samples in the past. The network which employs 6 and 12 neurons shows good and quite stable performance over the number of neurons, in particular, the networks with 6 samples in the past perform well also in this situation.

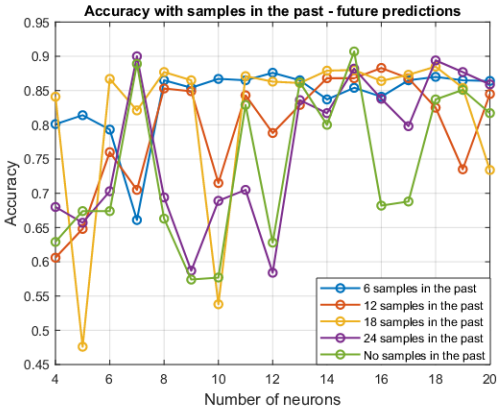


Figure 4.11: Accuracy vs number of neurons - Samples in the past - Test on future data

4.1.2 Two hidden layers networks

After analyzing simpler networks composed of just one hidden layer, a new analysis is performed on more complex networks containing two hidden layers. To perform this search the functionality *nnShape1* is used. This analysis is done by selecting only some possible values for each hidden layer: 6,10,14, and 20 neurons. The same case studies of the previous section are proposed here to see the differences with networks with only one hidden layer. The first simulations are performed considering all the available features, and the main settings are shown in table 4.4.

Hyperparameter	**Value**
Activation function	ReLU
Loss function	Cross entropy loss
Features	Impedance modulus,Impedance phase,Soil moisture Temperature,Ambient light,Air humidity Day of the year,Hour
Samples per parameter	1
Overlap of samples	0
Batch size	10
Epochs	50
Learning rate	0.001

Table 4.4: Simulation parameters

Also in this case, the dataset employed is well balanced between negative and positive cases, so only the plots regarding the accuracy are reported since it is more intuitive than MCC or F1 Score. Since in this new simulation the accuracy will be plotted against the numbers of neurons of layer 1 and layer 2, a visualization that employs a color map was chosen where colors tending to the yellow represent better performance, and more red cells indicate a worse behavior. As usual, the model is first trained on 80% of the dataset and tested on the remaining 20%, and the corresponding results are shown in figure 4.12. It can be noticed a good overall accuracy that ranges from 86.7% to 98%. In particular, more complex networks seems to behave better, especially the ones with more neurons in the second hidden layer; in fact from the figure, it can be observed that networks composed of 20 neurons in the second hidden layer are the ones that exhibit the best behavior.
Again, the trained models are tested on future samples belonging to the same four plants, and another colormap is produced and shown in figure 4.13. It can be observed that there is a performance drop when analyzing future samples, and this is quite visible, noticing how, overall, the colormap tends to have more red colors. In this case, simpler networks, i.e., the ones located on the bottom side of the colormap, present worse performance compared to the more complex ones

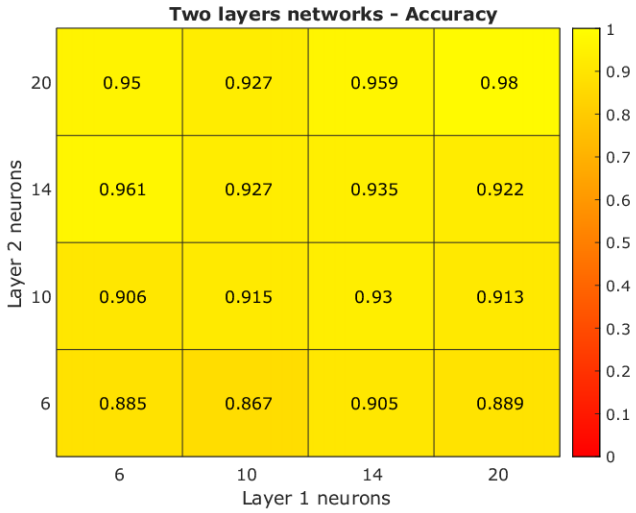


Figure 4.12: Two layers networks - Accuracy

located closer to the top side.

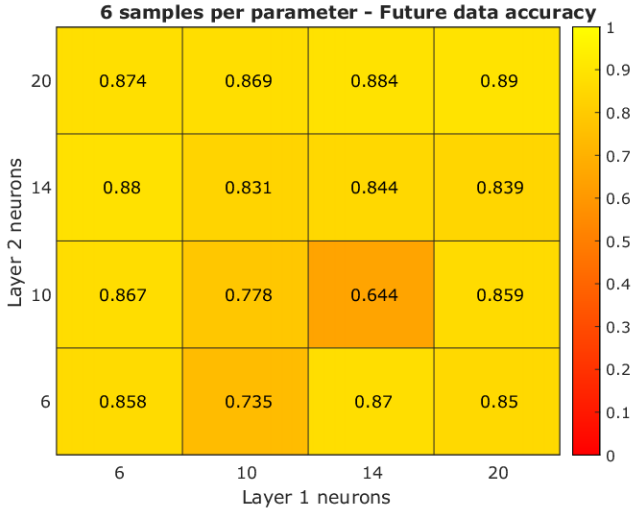


Figure 4.13: Two layers networks - Accuracy on future data

Figure 4.14 presents an insight on how the trained neural networks perform on each plant.
On plant 1, results are excellent, with almost all the networks reaching an accuracy of 100%. The situation drastically changes looking at plant 2, where most of the structures perform below the 50% of accuracy with just two networks able to score almost 100%. These exceptions correspond to the two networks that did not

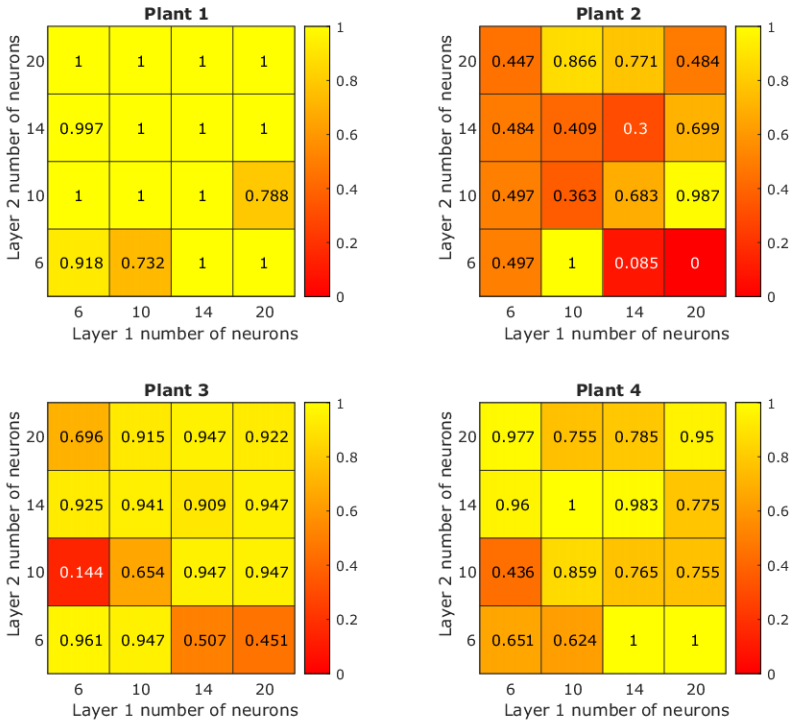


Figure 4.14: Two layers networks - Accuracy on future data plant by plant

perform perfectly on plant 1. Looking at plant 3, many structures perform well, but much variability is present since some simple networks (bottom of the map) score good results while others do not. In general, moving to the top of the colormap, so toward more complex networks, the accuracy tends to increase with the only exception of the structure with 6 and 20 neurons in the two hidden layers. Plant 4 presents intermediate performance between plants 2 and 3; almost all the networks created score accuracy higher than 75%, but also in this case, great variability of the results is noticed.

Predictions with only impedance

In this section, neural networks are trained using only impedance modulus and phase. Results on the test dataset are shown in figure 4.15.
From the colormap, it is quite visible that performance has dropped with respect to the previous case. The behavior across the different networks is almost constant, and the accuracy ranges from 74.4% to 83%, with better performance concentrated toward more complex structures. At this point, it is possible to analyze the behavior

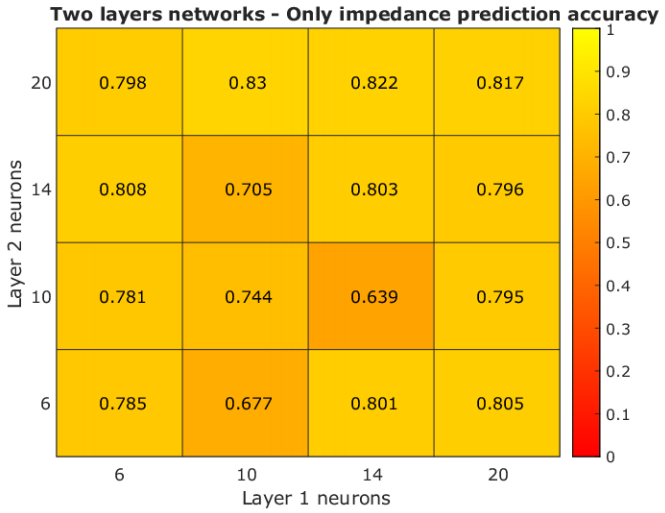


Figure 4.15: Two layers networks - Only impedance implementation

on future samples by looking at figure 4.16. The dominant color in the picture is red, indicating abysmal performance; in fact, the best score is 61%, but most networks don't reach 50 % accuracy. It is possible to look at figure 4.17 to understand better how the networks are behaving. The prediction accuracy on plant 1 is perfect, with all the networks scoring 100% accuracy; however, the fact that the networks perfectly predict plant 1 penalizes the prediction capability on the other plants, obtaining awful results for plants 2,3, and 4.

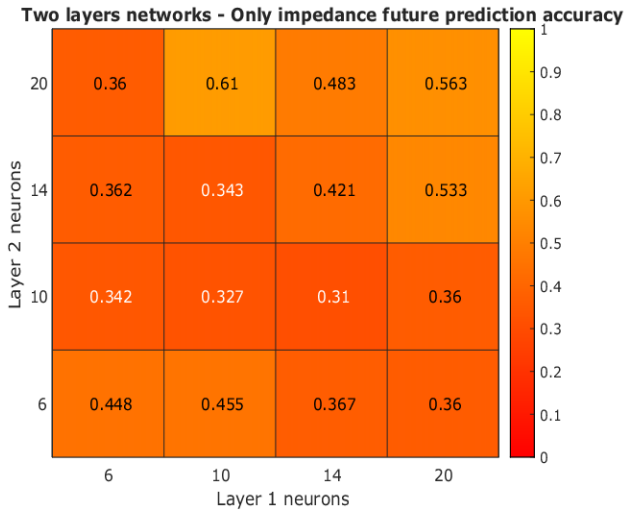


Figure 4.16: Two layers networks - Only impedance implementation - Future predictions

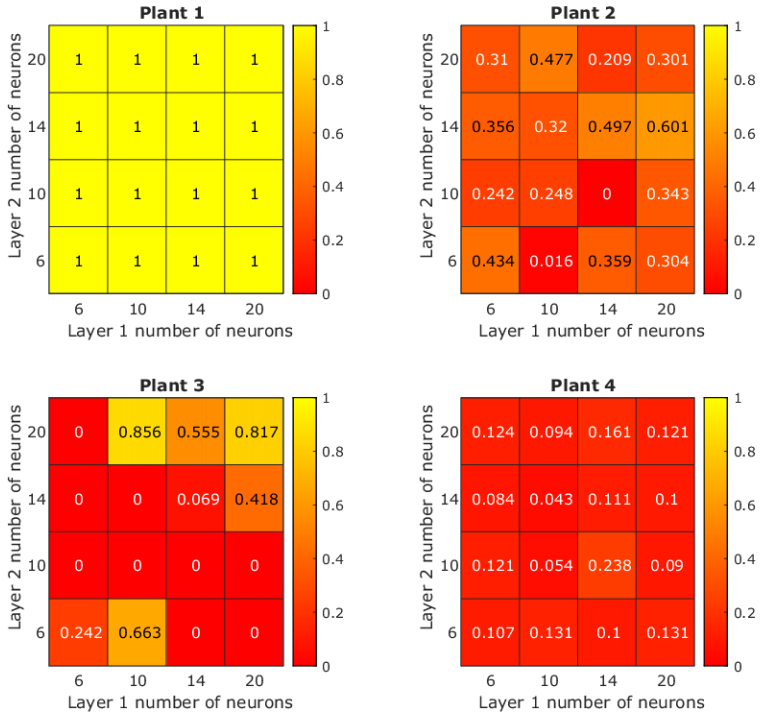


Figure 4.17: Two layers networks - Only impedance implementation - Future predictions plant by plant

Predictions with impedance and moisture

In this section, networks with two hidden layers are trained using impedance and moisture as features. Looking at figure 4.18, a slight increase in performance is observed except for the network with 6 neurons in both hidden layers, which fails by reaching almost null accuracy. This model is discarded from this analysis since it can be due to a particularly bad minimum found by the gradient descent algorithm during training, and it is in strong contrast with all the other models trained. The accuracy on the present data of the plants rose compared to the case with only impedance employed and reached above the 85% for many networks. In figure 4.19, the models are tested on future data, and the accuracy drops drastically, suggesting strong overfitting and that impedance and moisture are not enough to get a reliable prediction. This fact is confirmed by figure 4.20 where performance is perfect on plant 1 and decent on plant 2 but drops drastically for plants 3 and 4.

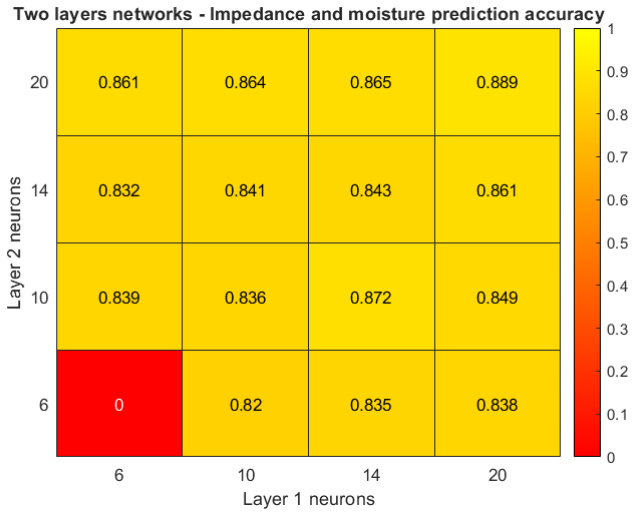


Figure 4.18: Two layers networks - Impedance and moisture

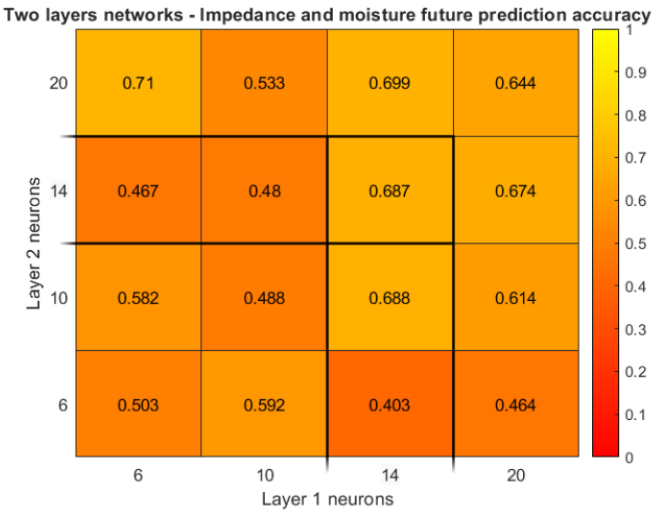


Figure 4.19: Two layers networks - Impedance and moisture - Future predictions

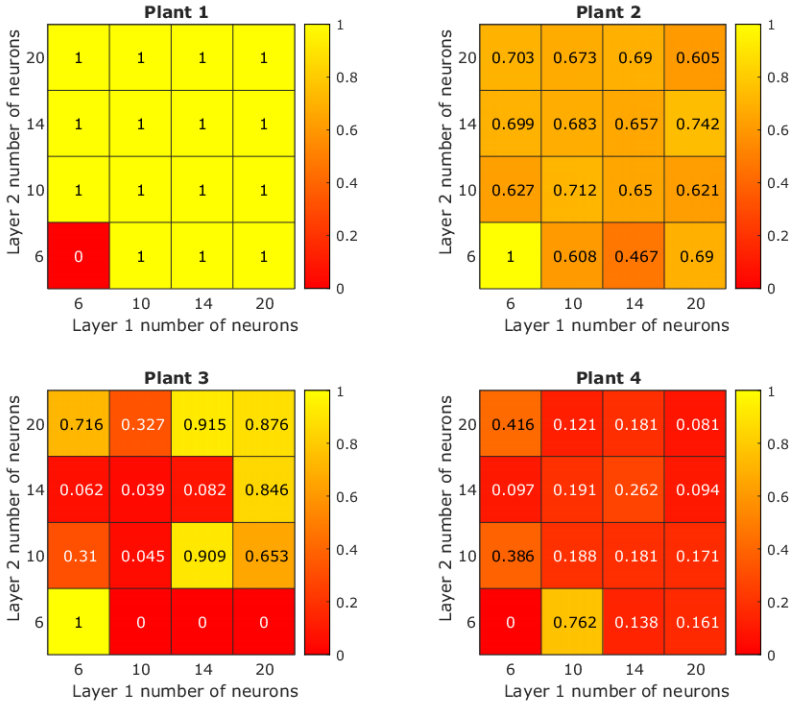


Figure 4.20: Two layers networks - Impedance and moisture - Future predictions plant by plant

Samples per parameter

In this section, networks with two hidden layers employing all the features available are trained using a different number of past samples for each feature to perform a single prediction. For each number of past samples, the networks are tested first on the 20% of the dataset excluded from training, then predictions on future samples belonging to the same four plants are evaluated. Figure 4.21 shows the results on the 20% test dataset; it can be noticed that performance is excellent, and almost any network performs over 95% of accuracy. The same networks are evaluated on

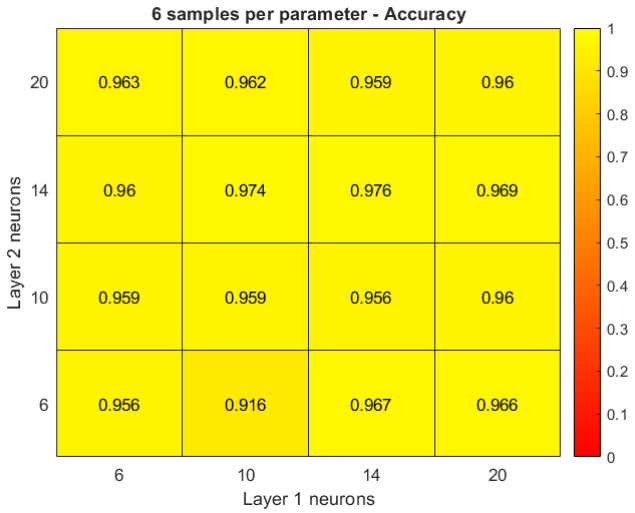


Figure 4.21: Two layers networks - 6 samples in the past

the same four plants but considering subsequent samples; the results are shown in figure 4.22. The general performance has reduced with respect to the previous case, but it is still acceptable since most of the networks perform above 80% with the best results obtained by a network with 6 and 10 neurons in the hidden layers. It can also be noticed a sort of "noisiness" among the results since some networks present accuracy that is a lot lower than best cases; an example of this effect is the network with 14 and 10 neurons which scores only 64.4% of accuracy. Looking at these results, it is possible to effectively predict the future status of the plants used for training.
The same analysis is performed for the other cases of 12,18, and 24 samples in the past. The results follow the same trend as the case of 6 samples: the accuracy prediction is excellent on the test dataset, and a performance drop is noticed when predicting future data. Another important aspect is that for the 12,18 and 24 samples cases, the accuracy on future data is worse than the 6 samples case. Since employing fewer past samples leads to better performance, it does not make sense

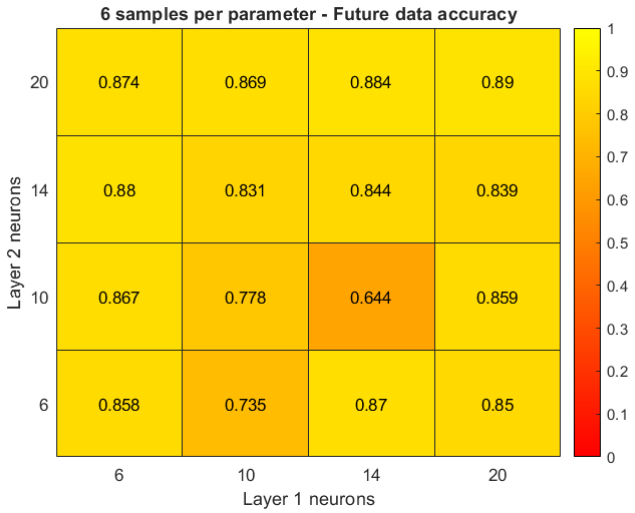


Figure 4.22: Two layers networks - 6 samples in the past - Future predictions

to pick more than 6 samples in the past because the number of features to be used in those networks will be larger, leading to greater network complexity not justified by an increase in performance. The following figures present results for the networks with 12,18, and 24 samples in the past.

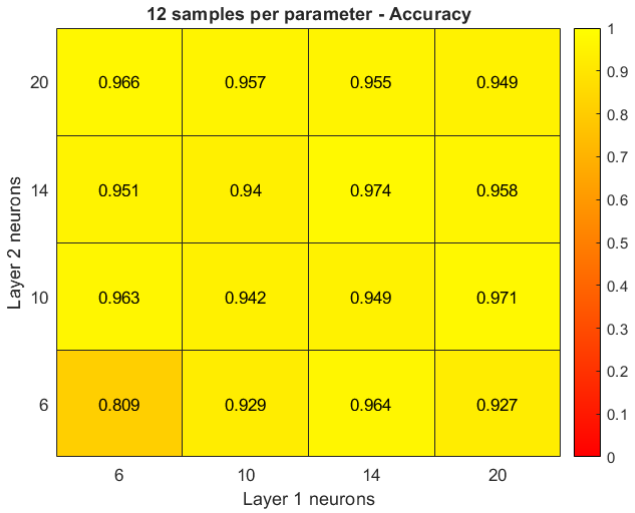


Figure 4.23: Two layers networks - 12 samples in the past

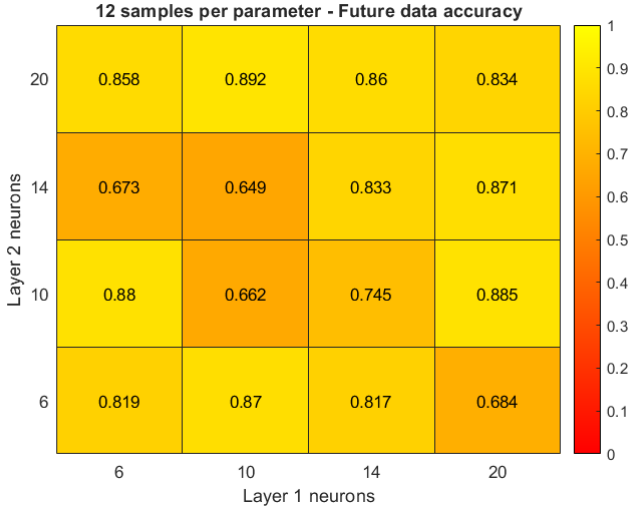


Figure 4.24: Two layers networks - 12 samples in the past - Future data

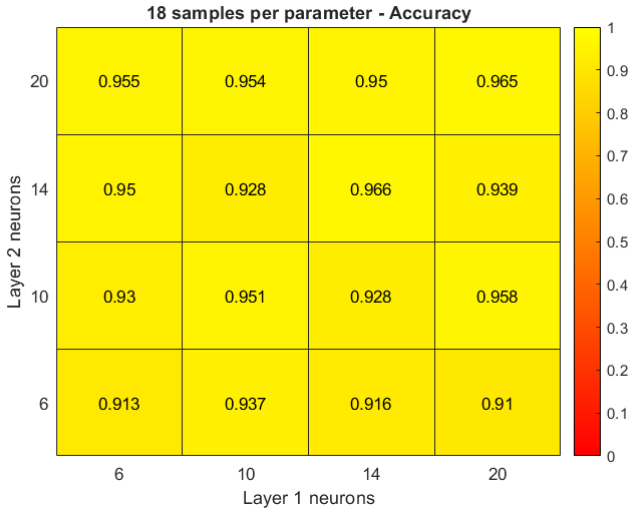


Figure 4.25: Two layers networks - 18 samples in the past

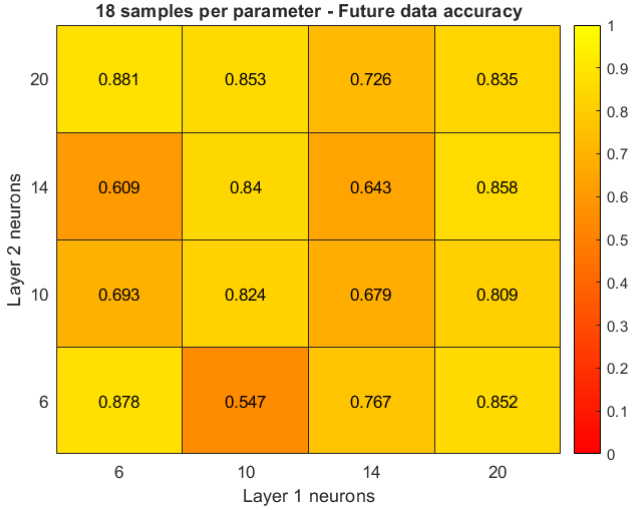


Figure 4.26: Two layers networks - 18 samples in the past - Future data

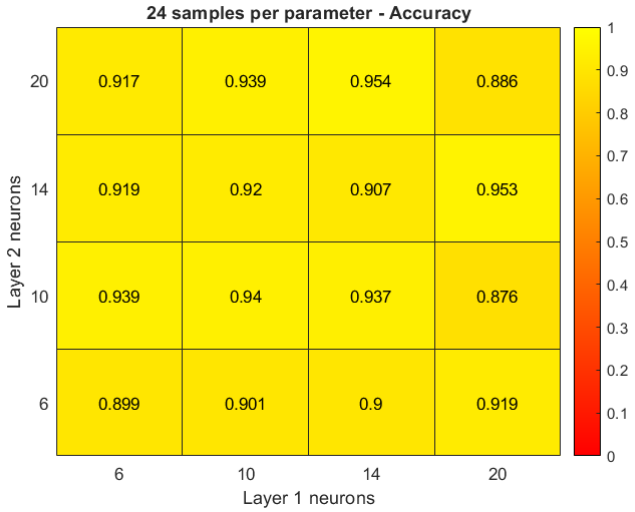


Figure 4.27: Two layers networks - 24 samples in the past

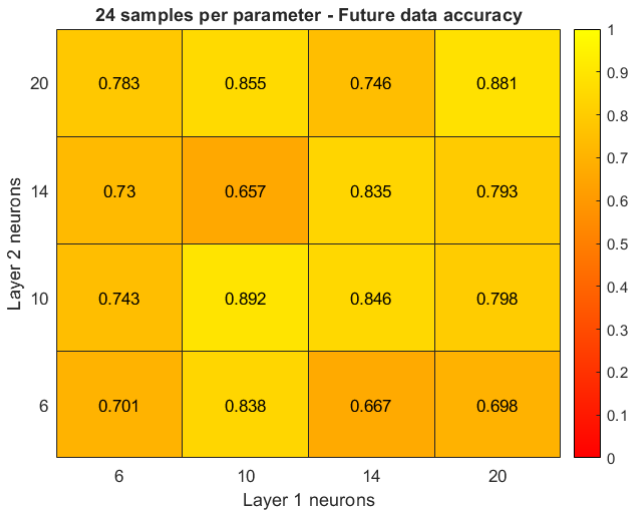


Figure 4.28: Two layers networks - 24 samples in the past - Future data

4.2 Training on two plants, test on the others

Up to now, all the neural networks were trained on all four available plants and tested data of the same plants, whether from the same time frame or a future one. This section uses a different methodology which is more similar to how an implementation of a classifier's actual application should work: the networks are trained on a set of plants and used to predict the state of health of different plants. Since the dataset available is composed of four plants, 2 of them are used for the training phase, while the others are for the testing. One healthy and one unhealthy plant are taken for training, and the same is done for the testing dataset using the remaining plants.

4.2.1 One hidden layer networks

The same procedure used in the previous training process is also followed in this situation. The first case analyzed is considering networks with only *one hidden layer*, the number of neurons is varied, and the results are reported considering future data belonging to the same plants employed for training, but now also for the future data belonging to the remaining two plants. The results of the training processes are plotted in figure 4.29. Two cases are presented: the blue

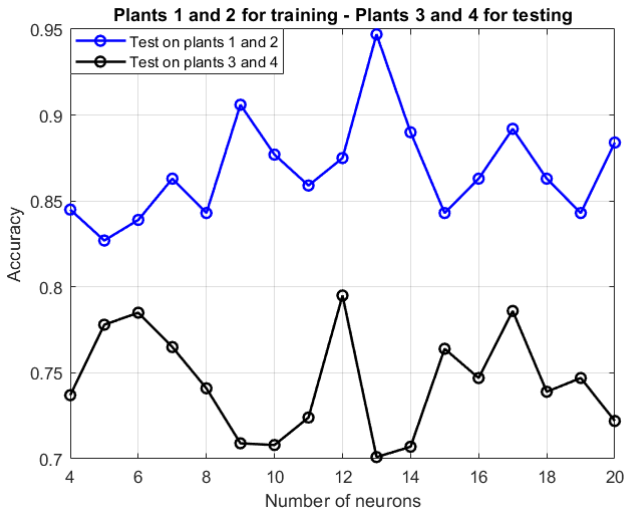


Figure 4.29: Two plants training: one layer networks

line represents the results of the networks when tested on future data of plants 1 and 2 (the ones used for training), while the black line shows the performance of plants 3 and 4. The first case shows good performance, reaching a peak accuracy of 95 %. In the second case, a significant performance drop is observed: prediction

accuracy is always greater than 75%; however, it dropped by more than 10%. The best results for what regards the test on unseen plants 3 and 4 are reached with networks having 10,12 and 16 neurons, which present an accuracy of almost 78 %. This performance drop between seen and unseen plants suggests the presence of overfitting, probably due to the simplicity of the network combined with a great quantity of data belonging only to two different plants.

Predictions with only impedance

In previous sections, some networks were trained by employing only impedance modulus and phase as features; this situation is also analyzed here to see if the results found when working on four plants are also valid in this new case.
With respect to the previous case, where all the features were employed, this situation shows less accurate predictions also on the plants 1 and 2 used for training, with an accuracy in the range of 50-60%. Moving to unseen plants 3 and 4, the situation also worsens with most of the networks performing correct predictions in less than 10% of the cases and a few of them reaching still not acceptable performance around 50%. These results confirm that using only impedance features does not lead to great results both on future data belonging to the same plants used for training and on future data of unseen plants.

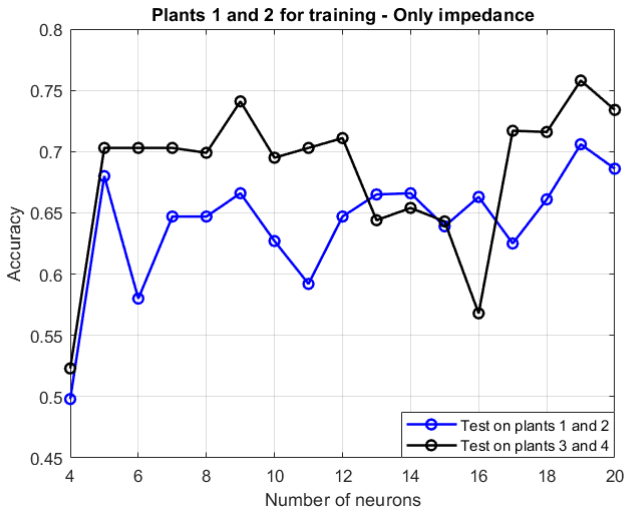


Figure 4.30: Two plants training - Impedance only

Predictions with impedance and moisture

A similar analysis to previous sections is now performed. The soil moisture is added to the impedance and is used as a feature of the trained neural networks.

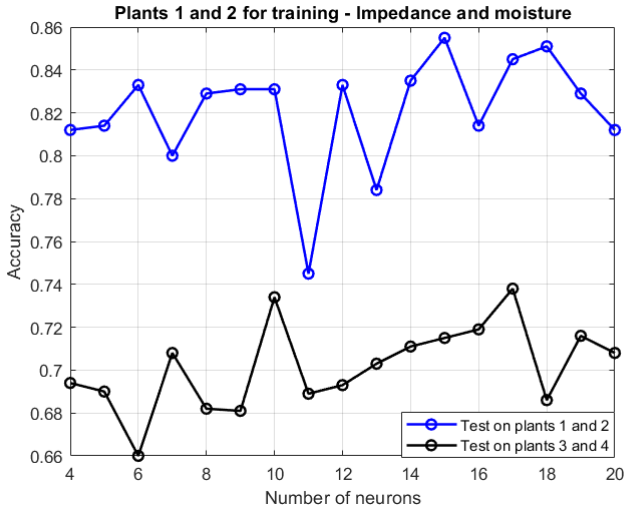


Figure 4.31: Two plants training - One layer networks - Impedance and moisture

Adding moisture as a feature increased the overall performance on test data belonging to plants 1 and 2, reaching almost 80% for all the number of neurons employed.

On the other hand, the results on plants 3 and 4 are still not acceptable. Some networks perform exceptionally well on the test plants reaching almost 90 % of accuracy. However, most models perform poorly, scoring less than 40%.

Prediction with samples in the past

Now the networks with only one hidden layer are trained using samples from the past to make a prediction. Figure 4.32 shows the results on plants 1 and 2. all the networks perform well, but it can be noticed that the case which shows better results is the one in which 6 samples in the past are considered. A general performance drop is noticed when moving to figure 4.33, suggesting the presence of overfitting. In this case, the networks with 6,12, and no samples in the past show accuracy on average around 75% while networks with 18 and 24 samples perform worse.

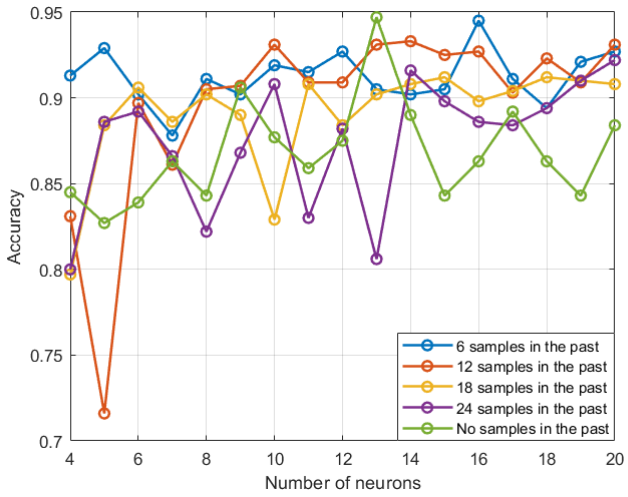


Figure 4.32: One layer networks - Samples in the past - Test on 1-2

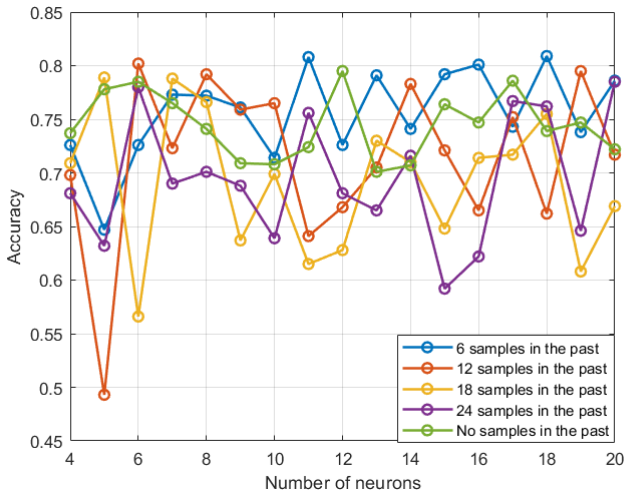


Figure 4.33: One layer networks - Samples in the past - Test on 3-4

4.2.2 Two hidden layers networks

This section will analyze networks composed of *two hidden layers*. As done previously for this type of analysis, only some values for each hidden layer will be considered: 6,10,14, and 20 neurons. The search *nnShape1* of the statusNowFinder functionality is employed to perform the simulations.The model will be first evaluated on the plants used for training (plants 1 and 2) and then tested on data belonging to plants 3 and 4. Figure 4.34 represents the performance of the trained

networks on plants 1 and 2, using the 20% of these plants' samples which were excluded from the training process. The image shows excellent performance, always higher than 90%. However, to better understand the neural networks' quality, it is possible to observe figure 4.35, which shows the accuracy on plants 3 and 4. In this case, it is visible how the performance dropped with respect to plants 1 and 2. This behavior suggests that the networks are overfitting, so they try strongly to predict the status of plants 1 and 2 correctly, but, in this way, they lose generalization capability. It is worth noticing the model composed of 20 neurons in the first hidden layer and 10 in the second one, which can achieve 78% of accuracy. This result is not excellent but shows that it is possible to train a neural network on some plants and use it on others that are entirely new from the network's perspective.

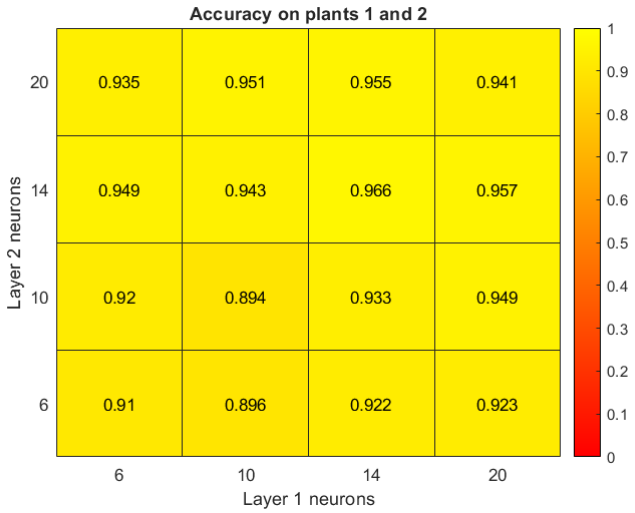


Figure 4.34: All features employed - Test on plants 1 and 2

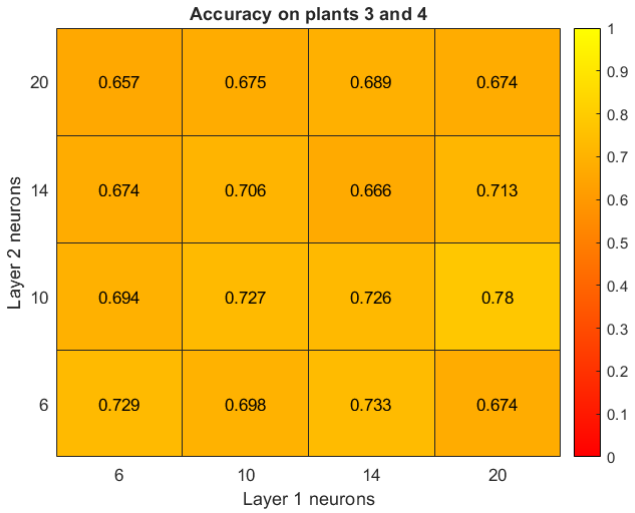


Figure 4.35: All features employed - Test on plants 3 and 4

Predictions with only impedance

Like in previous sections, also for the training with 2 plants, networks employing only impedance data are trained. The results of the simulations for plants 1 and 2 are shown in figure 4.36. The performance reached is around 75%, with the best network reaching 79.4% and the worse 68.4%.

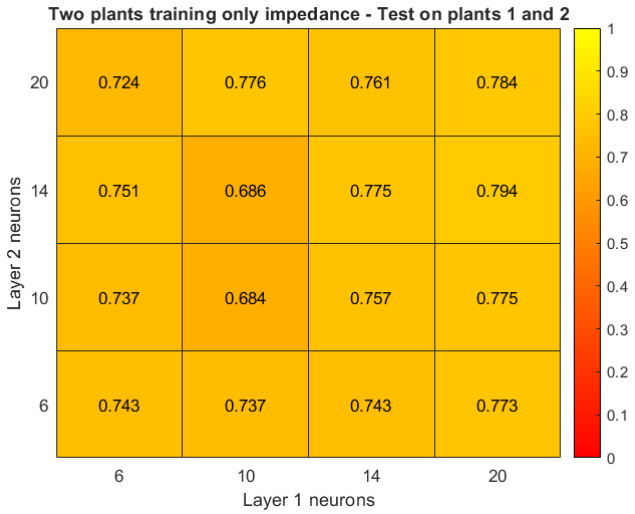


Figure 4.36: Only impedance - Test on plants 1 and 2

The networks are tested on unseen plants 3 and 4; the results are represented in

figure 4.37. Also in this case, the neural networks show a performance drop that can

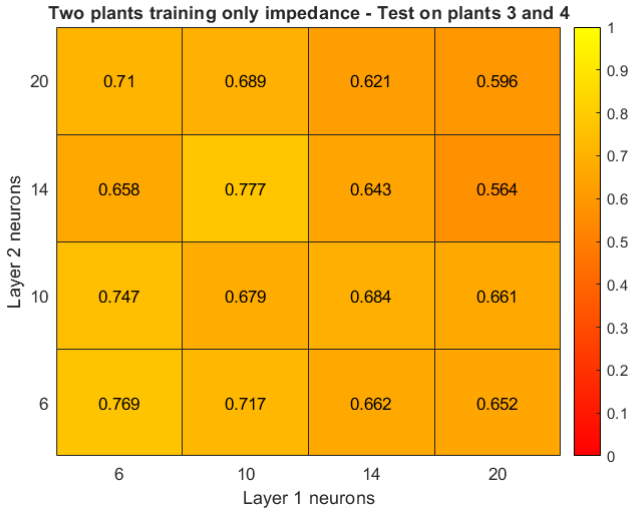


Figure 4.37: Only impedance - Test on plants 3 and 4

be due to overfitting, however, with respect to the previous case, this phenomenon is less evident since the performance difference between seen and unseen plants is reduced. Looking at figure 4.37, it is possible to see slightly better performance of simpler networks, so the ones located in the bottom left corner of the table, while moving towards the top right corner result in an accuracy reduction indicating that more complex networks tend to be more subject to overfitting.

Predictions with impedance and moisture

Following the usual scheme, figures 4.38 and 4.39 show the performance on plants 1-2 and 3-4, respectively, employing as features only impedance data and soil moisture. Looking at figure 4.38, we can observe that the performance on known plants is better than the previous case where only impedance was considered as a feature. Here, many networks perform above the 85% of accuracy, and the best one achieves 88.6% (10-20 neurons). The situation is different in figure 4.39, where the classifiers are evaluated on plants 3 and 4. In this case, the results tend to decrease. The performance ranges from 53.9% up to 74.5% showing a great variability between different networks. With respect to the case where only impedance was considered, here, overfitting seems to be stronger; in fact, the performance difference between the two tests is overall larger, and results tend to be more "noisy," highlighting the fact that the networks tend to classify known plants accurately and to misclassify unseen plants.

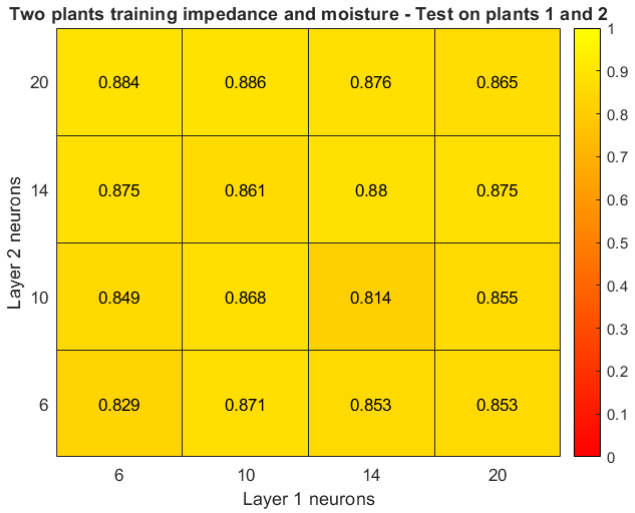


Figure 4.38: Impedance and moisture - Plants 1 and 2

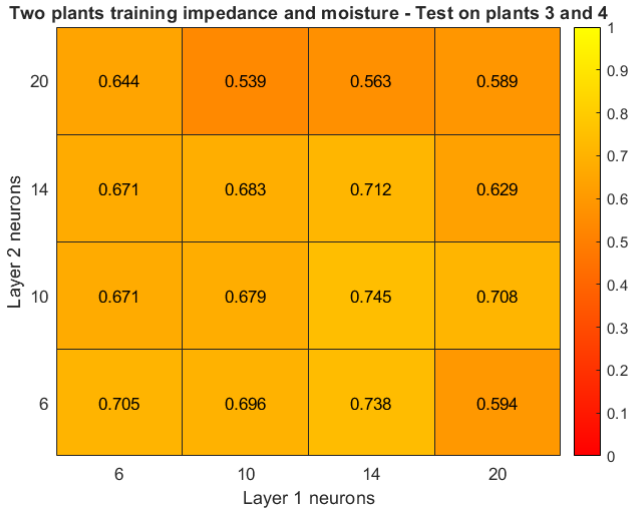


Figure 4.39: Impedance and moisture - Plants 3 and 4

Samples in the past

In this section, the networks are trained using different numbers of past samples to perform a prediction. The case analyzed are the following:

- 6 samples in the past
- 12 samples in the past

- 18 samples in the past
- 24 samples in the past

Starting from the first case, networks employing 6 samples to make a prediction are trained. The test results on plants 1 and 2 are reported in figure 4.40. As happened many times in previous simulations, the performance on known plants is auspicious, showing scores constantly above 95% of accuracy. Figure 4.41 shows how the same network performs on unseen plants 3 and 4. Regarding the results

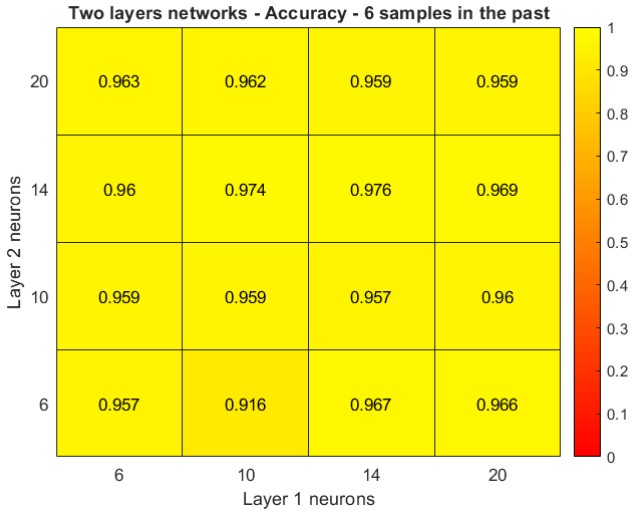


Figure 4.40: 6 samples in the past - Test on plants 1 and 2

on plants 1 and 2, in this case, the accuracy has decreased; however, employing 6 samples for a prediction leads to some good models even if overfitting is still present. In particular it is interesting to compare figure 4.41 with 4.35 in which the same networks are trained without considering samples in the past. It is possible to notice that adding 6 past samples to perform a prediction leads to a better and more stable performance through the various structures considered. In particular, the model composed of 20 neurons in both hidden layers is worth to be noticed since it shows great performance on both pairs of plants with 95.9% and 89%.
The same analysis is performed employing 12 samples for a single prediction and results are presented in figure 4.42 and 4.43. The tests on plants 1 and 2 give good results, as it is common on the plants used for training.

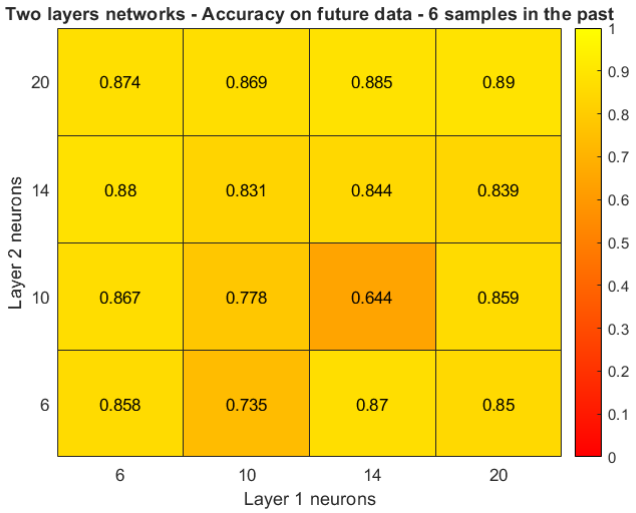


Figure 4.41: 6 samples in the past - Test on plants 3 and 4

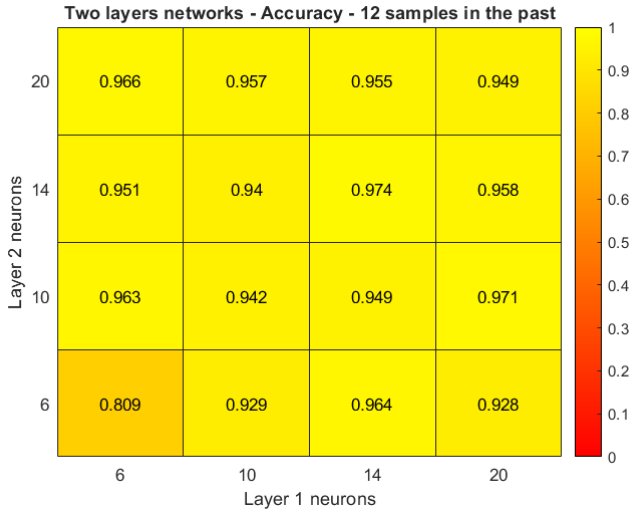


Figure 4.42: 12 samples in the past - Test on plants 1 and 2

Considering the accuracy on plants 3 and 4 instead, it is possible to see how, concerning the case of 6 past samples, the performance is less good overall and that many networks are strongly overfitting since the difference between the accuracy in the two cases has increased.

Figures 4.44 and 4.45 represent the results considering 18 past samples. As for the previous case, the accuracy on plants 1 and 2 is around 95% for almost all the trained networks. Moving to results on plants 3 and 4, it is possible to notice

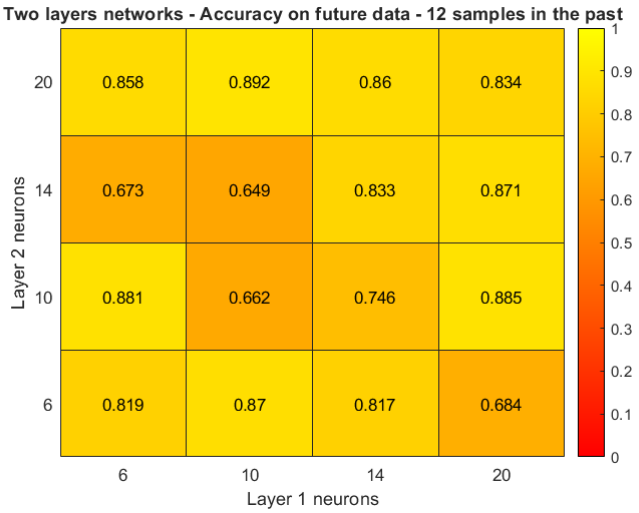


Figure 4.43: 12 samples in the past - Test on plants 3 and 4

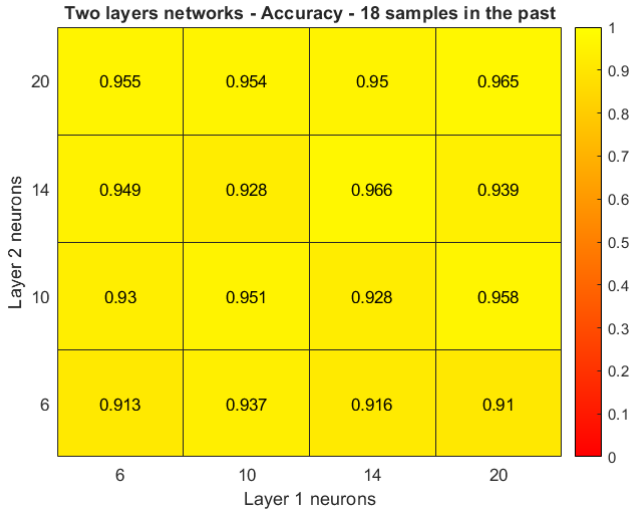


Figure 4.44: 18 samples in the past - Test on plants 1 and 2

that the overall performance is decreasing while picking more samples in the past. Still, some model scores good performance; however, the variability of the results across the structures analyzed has grown, and the overall difference between results on plants 1-2 and 3-4 suggests that increasing the past samples considered is also boosting the overfitting from which networks are affected.

Finally, the case of 24 samples in the past is considered, and the trend seen in the previous case continues here: for what refers to plants 1 and 2, performance

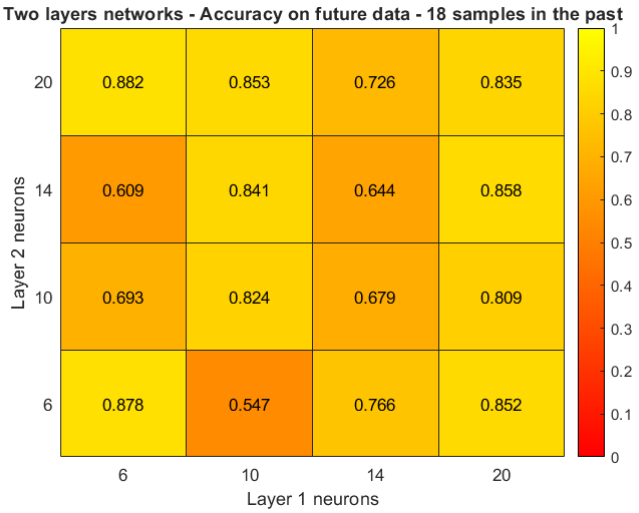


Figure 4.45: 18 samples in the past - Test on plants 3 and 4

is still good and above 90% for almost all the networks. Considering the test on plants 3 and 4 and adding even more samples for a prediction led to a worse performance than the previous cases where a smaller number of samples in the past was considered.

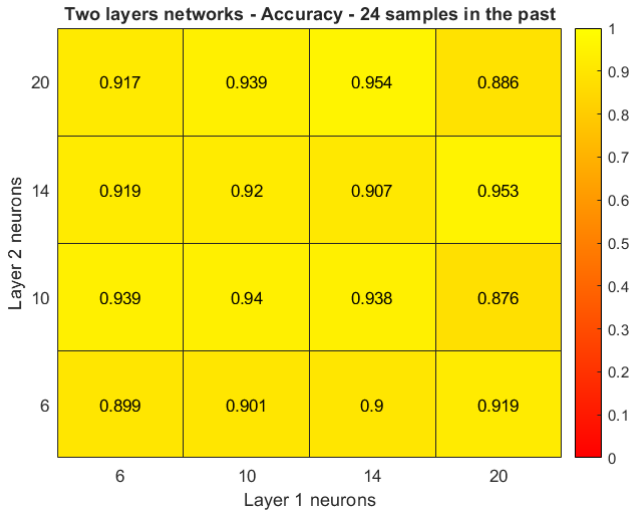


Figure 4.46: 24 samples in the past - Test on plants 1 and 2

Overall the networks that employ 6 past samples for a prediction perform better and are less complex, so it seems more convenient to use these networks rather than the ones with 12,18, or 24 samples in the past.

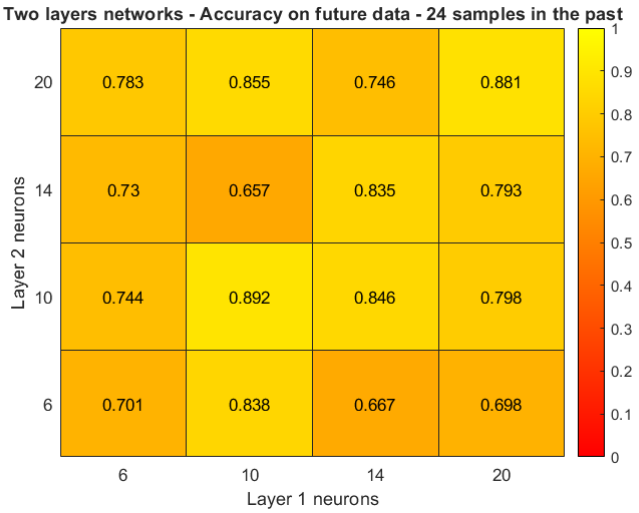


Figure 4.47: 24 samples in the past - Test on plants 3 and 4

4.3 Conclusions from the analysis

In the previous sections, many simulations were performed to understand the possible performance of a neural network to predict plants' status. First, all four plants were used as training, and the test was performed on future data from the same plants. Regarding networks with one hidden layer, it was possible to reach an accuracy on future data of 80% when employing all the features available. This result was further boosted to around 85% when employing 6 samples in the past to perform a prediction. Some tests using only impedance data or impedance and moisture combined led to not satisfying results. Moving to two hidden layers networks, the performance accuracy rose above 84% and using 6 samples in the past, in this case, did not boost the performance like in the previous case. For two hidden layers networks employing only impedance or impedance and moisture did not show good results. Second, the networks were trained only on two plants, leaving the other two as test datasets to emulate a situation closer to the final goal of the entire project: having a classifier to detect the status of completely unseen plants. In this case, one hidden layer networks showed worse performance than the case with all four plants used as training and this result was expected since the test plants are completely new and networks showed signs of overfitting. However, accuracy was around 75% and reached almost 80% when employing 6 samples in the past. These results are not excellent but suggest the possibility of predicting the plants' status by employing relatively simple networks. Also, in this case, one hidden layer networks that use only impedance or moisture did not show good results. Regarding networks with two hidden layers, they showed strong

overfitting when all the features were employed, reaching a maximum of 73% on the test plants. This situation was mitigated by employing 6 samples in the past, leading to performance above 85%. Also in this case, employing only impedance or impedance and moisture did not show promising results.

Chapter 5

Toward Microcontroller Implementation

In this chapter, the focus will be moved toward a possible microcontroller implementation of the networks studied in previous chapters. Even if the neural networks analyzed are still not optimal and much work is needed to completely understand how to solve the problems of overfitting highlighted in the previous chapter, it is worth analyzing the costs of a firmware implementation of such neural networks. In particular, since the final aim of this project is to integrate a plant status classifier to perform edge computing, it is interesting to understand how many resources these algorithms require to run on a microcontroller and the possible performance. In particular, a Nucleo board from STMicroelectronics was employed to perform some microcontroller test implementations. In particular the *NUCLEO-F446RE* equipped with an *STM32F446RET6U* was used. STMicroelectronics provides a package for artificial intelligence called X-CUBE-AI [28], which provides an automatic neural network library generator that is already optimized for what regards computation and memory and can convert trained models from the main machine learning frameworks such as TensorFlow, Keras, and ONNX to a library that the user on the microcontroller can employ. This package, together with the *STM32CubeMX* code generator and the *STM32CubeIDE* are employed to load one of the models trained in the previous chapter and perform some analysis on it for what regards memory occupation, computation time, and power consumption.

Figure 5.1 from X-Cube-AI documentation shows the general structure of the software. Up to this point, the trained model is available in the ONNX format [27]. The model is imported by specifying simple information such as the file format used to save the neural networks model (ONNX in our case), the name of the networks and features, a compression factor used to reduce the memory impact of the algorithm and the target STM32 microcontroller. From these simple

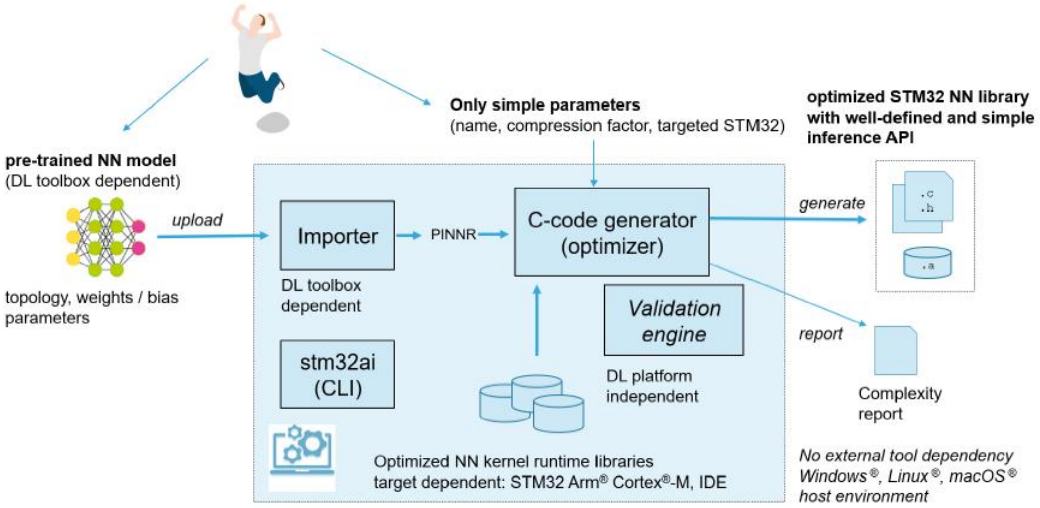


Figure 5.1: X-Cube-AI core from user manual [28]

parameters, the software can generate a ready-to-use C-code library that allows the user to employ the previously trained neural network model. An important part of the X-Cube-AI is the validation engine that allows the user to test the C-model library created, compare it to the saved model to see if, during the transition between the ML framework to the C-code, some numerical degradations occurred (in particular when the compression factor is used). Moreover, a complexity report will be produced and analyzed to evaluate the overall cost of the selected model. This section presents the step-by-step procedure to load a model on the STM32 microcontroller. All the procedure is performed using the STM32CubeIDE, which embeds the STM32CubeMX code generator and easily allows the installation of the X-Cube-AI add-on package. First, a project is created by selecting "*New > STM32 project*". The window that pops up is the board selector, which allows the selection of the desired Nucleo board; in this case, the *NUCLEOF446RE*. Then the program asks to name the project, and after that, a new window appears asking if initialize all the peripherals in default mode; in this case, where just a simple analysis is required, the answer will be yes, but when dealing with a real project the user might consider a different choice. At this point, the STM32CubeIDE should be a screen like the one in figure 5.2.

From here, it is necessary to activate the X-Cube-AI package by clicking on *Software Packs > Select Components*, and the screen on figure 5.3 will appear. In this new window, looking inside the red-dashed frame, check the box, and the X-Cube-AI package is now activated, and it is possible to click *Ok*. Back to the screen in figure 5.2

Back to the screen in figure 5.2, on the left is appeared a new entry called *Software*

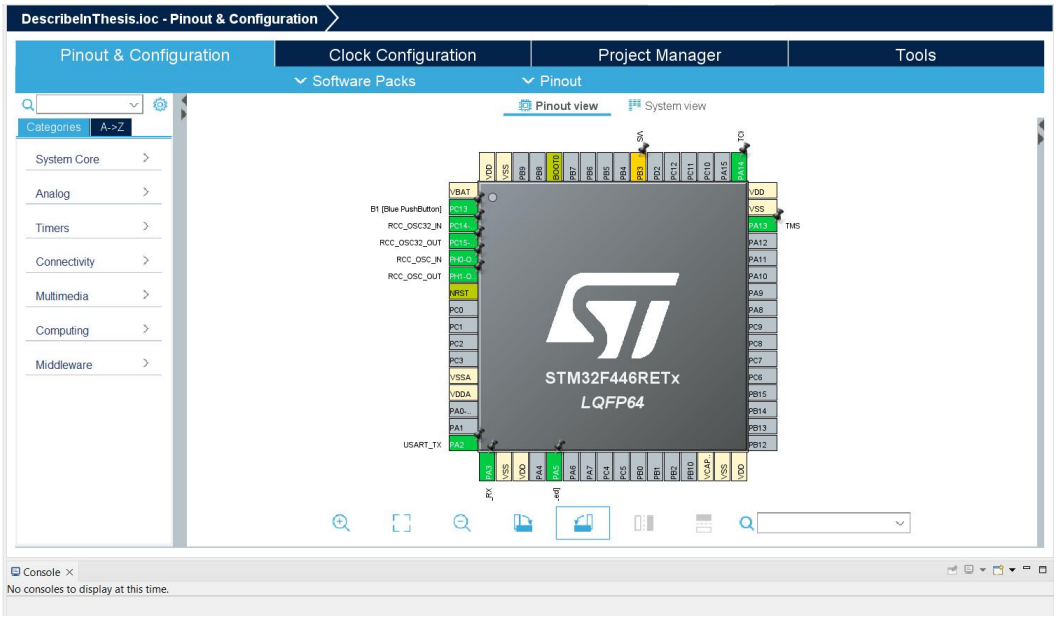


Figure 5.2: Initial screen of the STM32 project

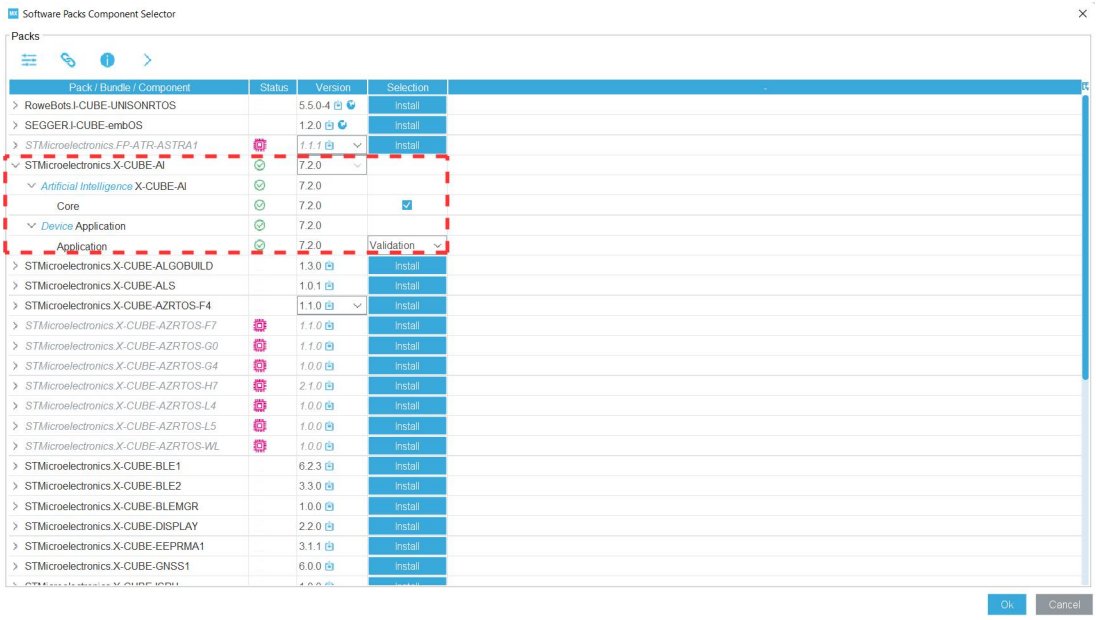


Figure 5.3: Activation of X-Cube-AI

Packs, select it and then click on *STMicroelectronics.X-CUBE-AI<version>*, where *<version>* corresponds to the downloaded version of X-Cube-AI. After doing so, the window in figure 5.4 will appear. From here, it is possible to add a model to

be loaded on STM32 by clicking the "+" icon near *Platform settings*.

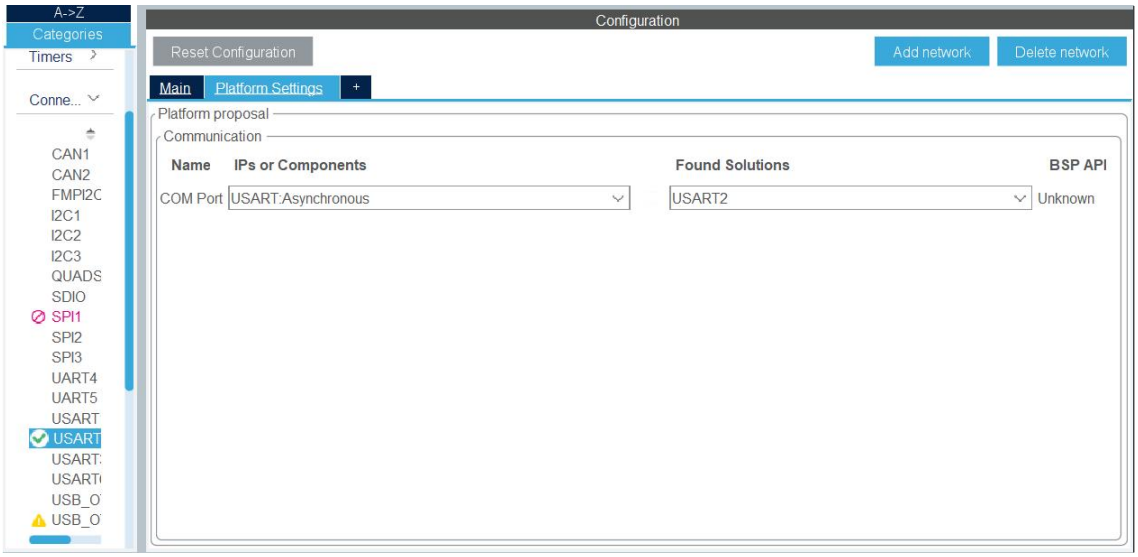


Figure 5.4: Loading model screen

After this, the window in figure 5.5 will pop up. From here, it is necessary to insert the name of the desired model (which will also be the name of the created C-library for the model), select the format of the saved model (in our case, it is the ONNX format), and then load the model file by clicking on *Browse*. Once the model is added to the project, it is possible to click on *Analyze* to print on screen some useful preliminary information on the network, such as RAM and Flash, necessary to run the model.
It is also possible to set a compression factor that reduces the dimension of the weights of the network, saving some memory space. When the model is correctly loaded, it is possible to use the *Validation engine* presented in figure 5.1 to perform some tests on the automatically generated C-model. Two kind of validation techniques are available: *Validate on desktop* and *Validate on target*. For both of them, it is possible to test the network using random or custom input data provided through CSV files.

Validation on desktop

This execution mode allows comparing the generated C model with the original model created using the ML framework. This operation runs on the host and not on the microcontroller. The validation on desktop is not mandatory, however, it is recommended by STMicroelectronics, in particular for the cases in which a compression factor is applied to the model. Figure 5.6 shows the workflow for the validation on desktop.

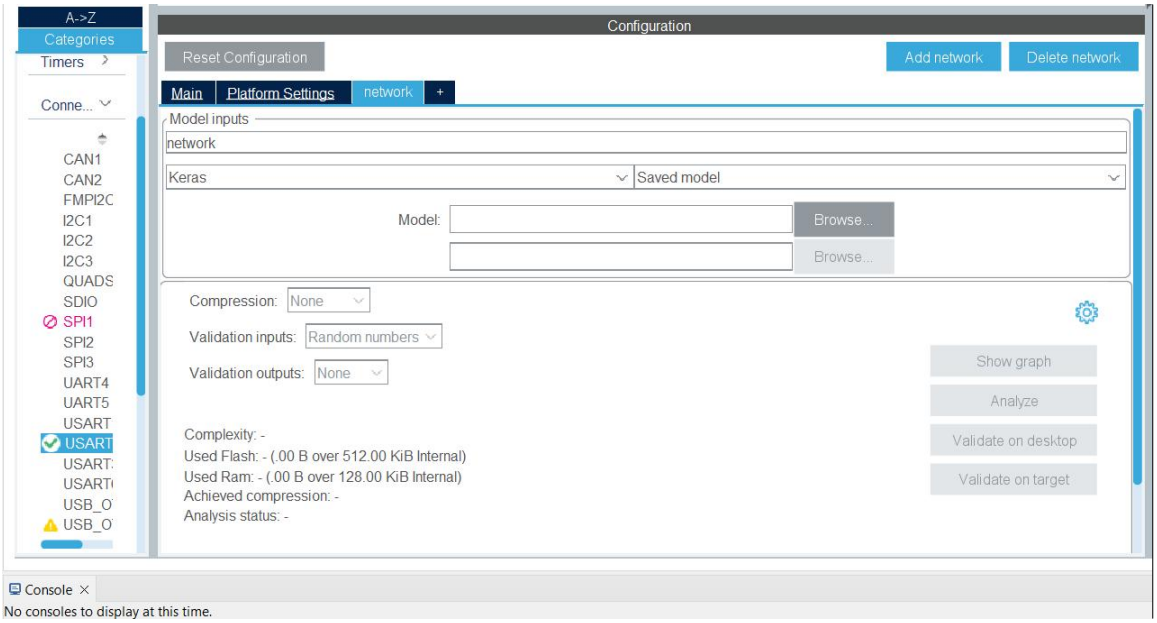


Figure 5.5: Loading model files

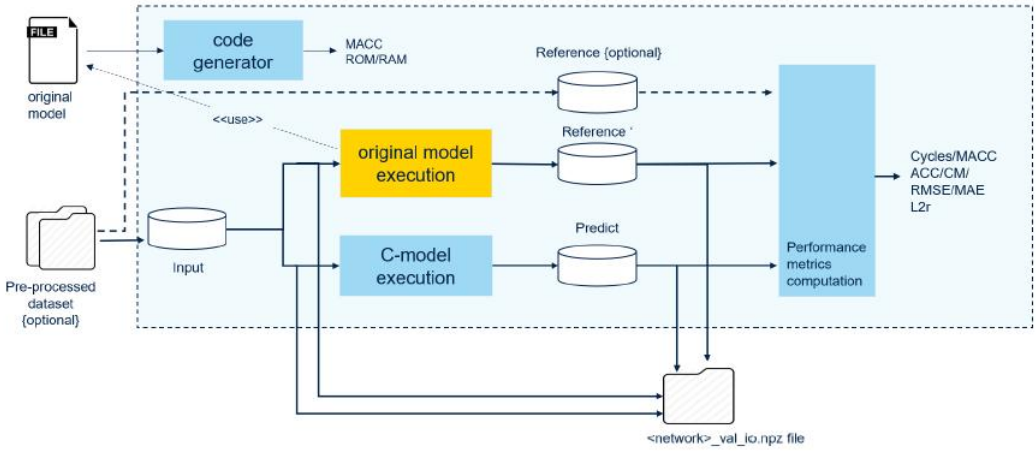


Figure 5.6: Validation on desktop workflow [28]

The evaluation on desktop process uses the input data (random or provided by the user) on both the original and the created C model. The networks' outputs are written on file so that it is possible to perform some post-processing if needed. Then some performance metrics such as accuracy, RMSE, and others are evaluated. It is important two notice that, in this validation case, eventual manipulations of the output of neurons should be done with some processing. For example: with the model trained in previous sections, the procedure to evaluate the prediction

starting from the output neurons' values was to compare their values and, based on the result, pick a particular prediction. In this case, the comparison operation is not performed by the validation engine, so the results returned are just the values of the output neurons. To get the predictions, some post-processing on the results file is necessary.

Validation on target

The second validation mode compares the original framework model with the generated C model running on the target device, so on the microcontroller. The workflow of this operation is shown in figure 5.7.

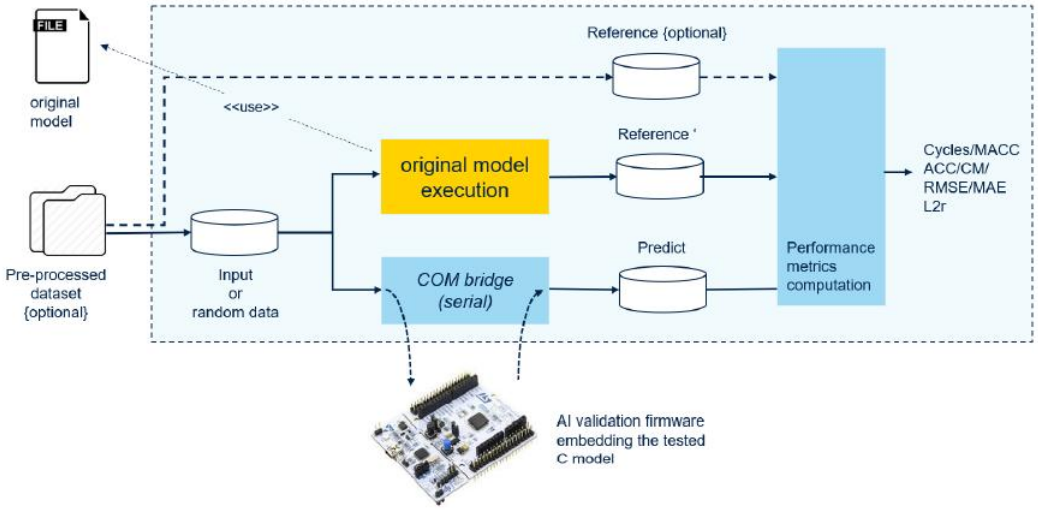


Figure 5.7: Validation on target workflow [28]

The process is similar to the validation on desktop case, with the difference that now the C-model is not running on the host device but the target. Also in this case, random or custom data are injected in both the C and the original models. The networks' outputs are written on file, and a report containing many performance metrics is reported. Also in this case, eventual manipulation to get the actual predictions of the model has to be done with some post-processing.

Evaluation metrics

Whenever a validation process is run X-Cube-AI returns a report containing valuable information about the ML model and many evaluation metrics useful to evaluate different aspects of the algorithm implementation. A complete list of the available metrics is in the user manual [28] (see section 15.2). For this simple analysis, only some of them are considered:

- MACC and cycles/MACC
- Memory-related metrics
- Accuracy
- Confusion matrix

MACC and *cycles/MACC* are measures of the computational complexity of the neural network. In particular *MACC* refers to the number of *Multiply and Accumulate* operations needed to perform inference. This metric is independent of the C-model implementation generated by X-Cube-AI, so it can be evaluated immediately when the model is analyzed. Instead, *cycles/MACC* is a metric that allows for on-device profiling of the performance by measuring the number of clock cycles the target device uses to perform an inference divided by the *MACC*. Memory-related metric refers to information about the usage of Flash and RAM. In particular, weights and activations of the neural networks are stored inside the Flash of the target device, so it is helpful to estimate how much memory is used to store these values. Moreover, when performing an inference, intermediate values of the neural network need to be stored on RAM to compute the output; for this reason, it is also useful to estimate the volatile memory usage required. Regarding the accuracy and confusion matrix, no further explanations are needed since they are the classical, well-known parameters useful to evaluate the quality of a prediction.

Example

A model with 2 hidden layers with 20 and 14 neurons saved in the ONNX format is loaded on the microcontroller. The validation process on target is performed, providing the model with two CSV files containing the input features and expected outcomes. When the process is completed, a report is produced, and it is now analyzed. First, some information about the network is reported:

Validation report

```
1
2  Generated C-graph summary
3  ------------------------------------------------------------
4  model name            : four_plants_20_14
5  c-name                : network
6  c-node #              : 5
7  c-array #             : 12
8  activations size      : 136 (1 segments)
9  weights size          : 2016 (1 segments)
10 macc                  : 538
11 inputs                : ['node_0_output']
```

```
outputs                 : ['node_12_output']

C-Arrays (12)
```

In the listing, some information about the loaded model extracted from the report is reported; in particular, the size in Bytes of the network activations and weight are shown as the number of MACC operations. Note that the memory usage reported here does not correspond to the total memory necessary to run the model on the target device. Since X-Cube-AI also generates a C-Library that allows using the model, the library's size has to be considered and is bigger than the memory used by the model itself. For this case, the Flash used by the library is 11.86 KiB which, added to the 1.97 KiB (2016 Byte / 1024 = 1.97 KiB)occupied by the weights, results in a total Flash size of 13.82 KiB. Note that KiB refers not to the Kilobyte (kB) but the Kibibyte. To convert the number of bytes to Kibibytes, it is necessary to divide by 1024. For what concerns the RAM used, the neural network itself needs the 136 Bytes of the activations ($136 \cdot 1{,}024 = 0.133$ KiB) and 1.88 KiB for the execution of the library, for a total of 2.01 KiB. Some information about the inference process is reported in the following part of the validation report. In this case, the number of CPU cycles used to perform an inference is 7798, which, considering that the microcontroller employed is running at 180 MHz, corresponds to $7798 \cdot \frac{1}{180\ MHz} = 0.043\ ms$ which is the value written in the report. Moreover, it is also reported the time spent by the network for the computation of the values of each layer and the corresponding percentage concerning the total inference time.

content/STM32/network_validate_report.txt

```

Results for 1216 inference(s) - average per inference
 device                 : 0x421 - UNKNOW @180/180MHz fpu,art_lat=5,
   art_prefetch,art_icache,art_dcache
 duration               : 0.043ms
 CPU cycles             : 7798
 cycles/MACC            : 14.49
 c_nodes                : 5

c_id  m_id  desc              output                    ms          %
-------------------------------------------------------------------
0     1     Dense (0x104)     (1,1,1,20)/float32/80B    0.014
   33.1%
1     2     NL (0x107)        (1,1,1,20)/float32/80B    0.003
   7.7%
2     3     Dense (0x104)     (1,1,1,14)/float32/56B    0.018
   41.5%
3     4     NL (0x107)        (1,1,1,14)/float32/56B    0.003
   6.6%
4     5     Dense (0x104)     (1,1,1,2)/float32/8B      0.005
   11.0%
```

```
16 ____________________________________
17                                                    0.043 ms
```

In the following part of the report, some performance metrics are reported. In particular, the accuracy and confusion matrix are evaluated for both the C-generated and original models. In the last part of the below listing, the Cross accuracy and confusion matrix resulting from comparing the C and the original is reported. It can be noticed that a score of 100% of cross accuracy is obtained; this means that the C model and the original one behave in the same way, and no degradation was introduced in the transition to the C model.

content/STM32/network_validate_report.txt

```
1
2  Accuracy report #1 for the generated stm32 C-model
3  ------------------------------
4  notes: - computed against the provided ground truth values
5         - 1216 samples (2 items per sample)
6
7   acc=73.68%, rmse=4.405611515, mae=3.851439714, l2r=0.980058432, nse
     =-7660.57%
8
9   2 classes (1216 samples)
10  ---------------------
11  C0        306    .
12  C1        320   590
13
14 Accuracy report #1 for the reference model
15 ------------------------------
16 notes: - data type is different: r/float32 instead p/float64
17        - computed against the provided ground truth values
18        - 1216 samples (2 items per sample)
19
20  acc=73.68%, rmse=4.405611515, mae=3.851439714, l2r=0.980058432, nse
     =-7660.57%
21
22  2 classes (1216 samples)
23  ---------------------
24  C0        306    .
25  C1        320   590
26
27 Cross accuracy report #1 (reference vs C-model)
28 ----------------------
29 notes: - data type is different: r/float64 instead p/float32
30        - the output of the reference model is used as ground truth/
     reference value
31        - 1216 samples (2 items per sample)
32
33  acc=100.00%, rmse=0.000000000, mae=0.000000000, l2r=0.000000000,
     nse=100.00%
34
35  2 classes (1216 samples)
```

```
36  ________________________
37  C0          626     .
38  C1            .     590
```

The same validation procedure is applied to a model composed of one hidden layer with 10 neurons to compare it to the model with 2 hidden layers with 20 and 14 neurons.

	1 hidden layer model	**2 hidden layers model**
Neurons in 1st-2nd hidden layer	10 - /	20 - 14
Network flash usage	448 B	1.97 KiB
Library flash usage	11.16 KiB	11.86 KiB
Total flash usage	11.59 KiB	13.86 KiB
Network RAM usage	72 B	136B
Library RAM usage	1.38 KiB	1.88 KiB
Total RAM usage	1.45 KiB	2.01 KiB
Complexity	122 MACC	538 MACC
Inference time	0.016 ms	0.043 ms
CPU Cycles	2872	7798
Cycles/MACC	23.55	14.50
Accuracy	73.77 %	73.68%

Table 5.1: 1 hidden layer model vs 2 hidden layers model

Chapter 6

Conclusion and Future Perspective

In this thesis, a neural network framework was employed and then used to train a great number of neural networks. An extensive analysis of neural networks showed the feasibility of plants' status prediction with relatively simple neural networks. Networks with one hidden layer showed a prediction accuracy on future data belonging to the plants used for training of 80 %, which can be boosted to 85 % if 6 samples in the past for each feature are used. When these networks are used on the unseen plants, the results drop to around 75% when no samples in the past are used and to 80% when 6 past samples are employed. Networks with two hidden layers reached a prediction accuracy of 84% on the future data from plants used in the training phase, but using samples in the past did not boost the performance in this situation. Regarding the test on unseen plants, the networks with two layers showed a performance drop to 73%, which is mitigated by using 6 samples in the past, leading to 85% accuracy on completely unseen plants. Finally, the X-Cube-AI tool was employed to load and benchmark two networks on an STM32 microcontroller. Two neural networks were loaded on the microcontroller:

- a network with one hidden layer with 10 neurons
- a network with two hidden layer with 20 and 14 neurons

The results obtained for these two network when validated on the target device were around 73% of prediction accuracy; however the simpler network with just one hidden layer showed a smaller Flash and RAM usage as well as a faster execution time. In the future, the first objective is to boost the neural network's performance, and a possible option is to augment the dataset with more different plants to be used in the training phase. Once better networks are obtained, the automatic

C-library generator provided by X-Cube-AI can be used to ease the process of writing the firmware of a classifier operating on the edge.

Chapter 7

Appendix

Appendix A: How to setup the framework using Anaconda

In this section it is explained how to install easily all the Python modules and their dependencies to be able to run the framework. To do so, the program used is Anaconda that allows to easily create virtual environments in which it is possible to safely use all the Python packages needed by the user in a project. Anaconda can be downloaded from the official site [29]. Once the installation is complete it is possible, to use the file *ml_framework_env.yml* to automatically setup a virtual environment with all the packages needed for the framework execution by typing the following command.

```
1    $ conda env create -f ml_framework_env.yml
```

This command creates a virtual environment whose name is specified inside the yml file. It is possible to change the environment name inside the yml to create an environment with a different name. To see if the environment is correctly created type:

```
1    $ conda list
```

and the name of all the environment created will be printed. 1 To correctly execute the framework the created virtual environment should be acctivated, to do so the following command is used.

```
1    $ conda activate <environment_name>
```

Once the virtual environment is correctly activated it is possible to launch the framework.

Appendix B: Available Pytorch activation functions

Pytorch activation functions
ELU
Hardshrink
Hardsigmoid
Hardtanh
Hardswish
LeakyReLU
LogSigmoid
MultiheadAttention
PReLU
ReLU
ReLU6
RReLU
SELU
CELU
GELU
Sigmoid
SiLU
Mish
Softplus
Softshrink
Softsign
Tanh
Tanhshrink
Threshold
GLU
Softmin
Softmax
Softmax2d
LogSoftmax
AdaptiveLogSoftmaxWithLoss

Table 7.1: Available Pytorch activation functions

Appendix C: Available Pytorch loss functions

Pytorch loss functions available in the *torch.nn* package.

Pytorch loss functions
nn.L1Loss
nn.MSELoss
nn.CrossEntropyLoss
nn.CTCLoss
nn.NLLLoss
nn.PoissonNLLLoss
nn.GaussianNLLLoss
nn.KLDivLoss
nn.BCELoss
nn.BCEWithLogitsLoss
nn.MarginRankingLoss
nn.HingeEmbeddingLoss
nn.MultiLabelMarginLoss
nn.HuberLoss
nn.SmoothL1Loss
nn.SoftMarginLoss
nn.MultiLabelSoftMarginLoss
nn.CosineEmbeddingLoss
nn.MultiMarginLoss
nn.TripletMarginLoss
nn.TripletMarginWithDistanceLoss

Table 7.2: Available Pytorch loss functions

Appendix D: Available Pytorch optimizers

Pytorch optimization algorithms available in *torch.optim* pacakge.

Pytorch optimizers
Adadelta
Adagrad
Adam
AdamW
SparseAdam
Adamax
ASGD
LBFGS
NAdam
RAdam
RMSprop
Rprop
SGD

Table 7.3: Available Pytorch optimizers

Bibliography

[1] Lee Bar-on, Aakash Jog, and Yosi Shacham-Diamand. «Four Point Probe Electrical Spectroscopy Based System for Plant Monitoring». In: *2019 IEEE International Symposium on Circuits and Systems (ISCAS)* (2019), pp. 1–5. DOI: 10.1109/ISCAS.2019.8702623 (cit. on pp. ii, 31).

[2] Lee Bar-On, Sebastian Peradotto, Alessandro Sanginario, Paolo Motto Ros, Yosi Shacham-Diamand, and Danilo Demarchi. «In-Vivo Monitoring for Electrical Expression of Plant Living Parameters by an Impedance Lab System». In: *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)* (2020), pp. 178–180. DOI: 10.1109/ICECS46596.2019.8964804 (cit. on p. ii).

[3] Umberto Garlando, Lee Bar-On, Paolo Motto Ros, Alessandro Sanginario, Sebastian Peradotto, Yosi Shacham-Diamand, Adi Avni, Maurizio Martina, and Danilo Demarchi. «Towards Optimal Green Plant Irrigation: Watering and Body Electrical Impedance». In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)* (2020), pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9181290 (cit. on pp. ii, 31).

[4] Umberto Garlando, Lee Bar-On, Paolo Motto Ros, Alessandro Sanginario, Stefano Calvo, Maurizio Martina, Adi Avni, Yosi Shacham-Diamand, and Danilo Demarchi. «Analysis of In Vivo Plant Stem Impedance Variations in Relation with External Conditions Daily Cycle». In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)* (2021), pp. 1–5. DOI: 10.1109/ISCAS51556.2021.9401242 (cit. on p. ii).

[5] *Pytorch documentation*. Available here (cit. on pp. 1, 29).

[6] *«An Introduction to Machine Learning»*. Available here (cit. on p. 2).

[7] *«Machine learning is popular right now»*. Available here (cit. on p. 2).

[8] Daniel Voigt Godoy. *Deep Learning with PyTorch Step-by-Step: A Beginner's Guide. Volume I - Fundamentals*. 2021 (cit. on p. 7).

[9] *A visual explanation of gradient descent methods*. Available here (cit. on pp. 9, 39).

[10] *Neural networks and deep learning*. Available here (cit. on pp. 20, 21).

[11] *3Blue1Brown's video on backpropagation*. Available here (cit. on p. 20).

[12] *Neural networks cost functions overview*. Available here (cit. on p. 22).

[13] *ReLU activation function*. Available here (cit. on p. 25).

[14] *Normalization*. Available here (cit. on p. 26).

[15] *Python documentation*. Available here (cit. on p. 29).

[16] *Anaconda documentation*. Available here (cit. on p. 29).

[17] *Python virtual environments*. Available here (cit. on p. 29).

[18] *DESIGN OF A NEURAL NETWORK DEVELOPMENT FRAMEWORK FOR PLANT MONITORING APPLICATIONS*. Available here (cit. on p. 29).

[19] *Pandas documentation*. Available here (cit. on p. 32).

[20] *Pytorch Dataset*. Available here (cit. on p. 33).

[21] *Pytorch Module*. Available here (cit. on p. 36).

[22] *Pytorch: torch.nn*. Available here (cit. on p. 36).

[23] *Pytorch: torch.optim*. Available here (cit. on p. 36).

[24] *Matthews Correlation Coefficient Introduction*. Available here (cit. on p. 53).

[25] *Matthews Correlation Coefficient advantages*. Available here (cit. on p. 53).

[26] *Matthews Correlation Coefficient advantages pt.2*. Available here (cit. on p. 53).

[27] *Open Neural Network Exchange format*. Available here (cit. on pp. 61, 62, 98).

[28] *X-Cube-AI documentation*. Available here (cit. on pp. 98, 99, 102, 103).

[29] *Download Anaconda*. Available here (cit. on p. 112).