POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico di Torino

Master's Degree Thesis

Design of a Neural Network development framework for plant monitoring applications

Supervisors

Prof. Maurizio MARTINA

Prof. Danilo DEMARCHI

Ph.D. Umberto GARLANDO

Candidate

Alessandro LOVESIO

December 2021

Summary

This thesis is part of the promising and fast-growing technology aspect of the "agri-food" sector. With the world's population growing rapidly and climate change posing always increasing risks to farming, food production must be made more efficient and secure. Much work has already been performed in this research field, especially by the MINES Research Group of the Politecnico di Torino under "the Plant Project". The project goal is to develop a self-sustaining monitoring system to be deployed on the field so that the plants that are being cultivated can be supervised autonomously, and the farmers can be remotely provided with the health status of their crops. This approach allows for a more efficient and timely response to negative factors affecting the plants. The scope of this thesis, specifically, is to study and develop machine-learning-based algorithms to understand plant status. As part of the thesis' efforts, a framework was developed in the Python language to test different architectures of machine learning and data processing algorithms to discover the best solutions to link environmental and impedance data about plants to their health status. In particular, data about tobacco plants grown and monitored at the Politecnico di Torino laboratories has been employed.
The developed software takes advantage of the data gathered by the MINES Research Group of the Politecnico di Torino, with a large and continuously growing dataset of environmental and impedance data of the tobacco plants. Software developed by the MINES Research Group can generate easy to handle CSV files with all the data needed to explore the various solutions and train and validate the neural networks. This data has been dispensed to the tool developed under this thesis work and, along with different configuration files describing how the data should be handled and processed, the topology of the Neural Network as well as how it should be trained, the program can train a Neural Network as specified and save the statistics associated with it. The included statistics consist of Mean Square Loss and Root Mean Square Error over the test subset or the whole dataset, as well as visual representations of the training process such as the evolution of the errors over the iterations and graphs of the predictions versus the actual data that can be used to check how the model behaves over all the available datasets.
To facilitate the exploration of the various architectures so that the code does not

have to be changed for every training run, the software can be executed from a Command Line Interface by providing the various settings files as parameters. It is also possible to specify if the training should be a one-off effort or if multiple training attempts should be performed by sweeping a specific setting to find their optimal values without running multiple training series one at a time. This paradigm allows the user to search for the best parameters, leading to better predictions.
The framework just described can be represented with the block diagram of figure 1.

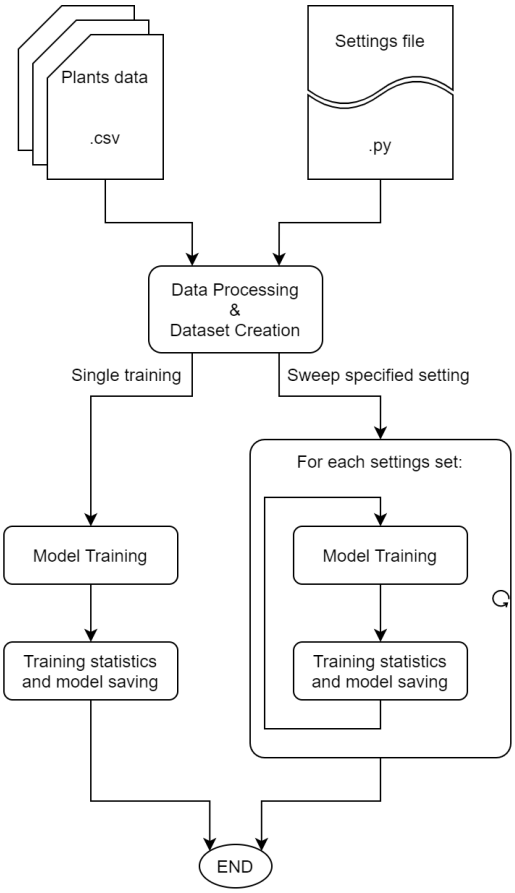


Figure 1: Block Diragram of the framework.

Different results have been obtained while developing and using this framework. While the training efforts could benefit from having many more weeks and months of data and data about many different plants, many interesting conclusions can be observed.
It has been noticed that individual plants respond differently even when subjected to similar conditions. Similar to vital human signs, the general trends are similar

for every individual, but the specifics vary from subject to subject.
It has also been observed that the best predictions can be made in periods that are close to watering events. There might be multiple and yet to be understood reasons for this, but one of the most likely is that far from water events, the vital signs of the plants are more sensitive to other environmental variables.
By performing sweeps over specific settings, it was found that it is optimal to pick samples in the past when making predictions, particularly samples up to 20-50 hours old. This notion suggests that the plants have a low pass filter behavior.
Better prediction results can also be obtained employing Neural Networks with two or more hidden layers and at least ~100 neurons per hidden layer. Increasing amounts of layers and neurons, while providing slightly better results, need a lot more resources that are not justified by the minor improvements.

Table of Contents

List of Tables VIII

List of Figures IX

Acronyms XIII

1 Introduction 1
- 1.1 Thesis Overview . . . 1
  - 1.1.1 Thesis Structure . . . 5
- 1.2 State of the art on plant status . . . 5

2 Machine Learning notions 9
- 2.1 Machine Learning fundamentals . . . 9
  - 2.1.1 Classification vs Regression . . . 10
  - 2.1.2 Supervised learning vs. Unsupervised learning . . . 10
  - 2.1.3 Training process . . . 11
  - 2.1.4 Feature Scaling . . . 13
  - 2.1.5 Bias – variance tradeoff . . . 14
  - 2.1.6 Neural Networks . . . 16
- 2.2 Machine learning applied to plants . . . 21

3 Software Implementation 23
- 3.1 Introduction . . . 23
  - 3.1.1 Framework structure . . . 24
- 3.2 Data preparation and processing . . . 27
- 3.3 Machine Learning "foundation" . . . 35
  - 3.3.1 Dataloader . . . 36
  - 3.3.2 Neural Network model . . . 38
  - 3.3.3 Loss Function . . . 42
  - 3.3.4 Optimizer . . . 44
  - 3.3.5 Train Loop and Test Loop . . . 47

3.4 Status Now: prediction of the current plant status software implementation . . . 50
3.4.1 Setting class . . . 51
3.4.2 Standard implementation . . . 54
3.4.3 K-Fold validation implementation . . . 58
3.5 Status Now Finder: finder of the best predictor software implementation . . . 64
3.5.1 Time search: search for the data that gives the best predictions 70
3.5.2 Neural Net Shape search: search for the best Neural Net topology . . . 72
3.5.3 Past samples search: search for how long in the past the samples are useful . . . 73
3.6 Utilities . . . 74
3.7 Dispatcher: running the scripts from a shell . . . 79
3.8 On the Field . . . 80

4 Capabilities and Results 82
4.1 Framework capabilities . . . 82
4.2 Status Now results . . . 83
4.3 Status Now Finder results . . . 94

5 Conclusions 100

A Software usage 102
A.1 Using the Status Now functionality . . . 104
A.2 Using the Status Now Finder functionality . . . 114
A.3 Adding a new Status Now Finder search type . . . 120

Bibliography 125

List of Tables

2.1 Neural Networks with a single neuron emulating logic gates. 19

3.1 Dataframe obtained after the CSV parsing. 29
3.2 Dataframe obtained from the *buildDataset()* function. 31
3.3 Example of the internal Inputs Dataframe of the *PlantsDataset* class. 33
3.4 Example of the internal Outputs Dataframe of the *PlantsDataset* class. 33
3.5 Available Non-linear Activations from the PyTorch library. 41
3.6 Available Linear Layers, Recurrent Layers, Dropout Layers, Transformer Layers, and Sparse Layers from the PyTorch library. 41
3.7 Available Normalization Layers, Padding Layers, Pooling layers, and Convolution Layers from the PyTorch library. 42
3.8 Available Loss functions from the PyTorch library. 44
3.9 Available NN Optimizer algorithms from the PyTorch library. . . . 46
3.10 All the variables of the *Setting* class. 52
3.11 All the keys of the dictionaries in the *dataset_settings* list. 53

List of Figures

1 Block Diragram of the framework. iii

1.1 The second sustainable development goal: Zero Hunger. Extracted from [1] 1
1.2 Example of a training and validation error visualization. 3
1.3 Block Diragram of the framework. 4

2.1 K-fold data assignment. 13
2.2 Gradient descent towards minimum. 14
2.3 Different outcomes of training on the same training set. 15
2.4 A generic Neural Network. 16
2.5 Structure of an artificial Neuron. 17
2.6 Structure of a biological Neuron. Extracted from [17] 17
2.7 Neural Networks with a single neuron emulating logic gates. 19
2.8 Comparison of different functions used for activations. 20

3.1 Organization of the framework. 25
3.2 Flow of the data during initial preparation. 28
3.3 Block diagram of the *buildDataset()* function. 30
3.4 Block diagram of the *PlantsDataset* ___*init*___*()* method. 34
3.5 Concept behind the *SubsetRandomSampler*. 38
3.6 Forward propagation and backpropagation. 43
3.7 Different implementations of the Gradient Descent algorithm, the Stochastic Gradient Descent algorithm is necessary with multiple batches. 45
3.8 Block diagram of the *train_loop()* function. 48
3.9 Block diagram of the *test_loop()* function. 49
3.10 Block diagram of the *statusTrain()* function. 55
3.11 "*visual_data_figure*" example figure. 57
3.12 "*figure_validation*" example figure. 58
3.13 K-fold data assignment. 59
3.14 K-fold set division example. 60

3.15 Block diagram of the *statusTrainKFold()* function. 63
3.16 Block diagram of the *statusFinder()* function. 68
3.17 Block diagram of the sweeping portion of the *statusFinder()* function. 69
3.18 Example of timeframes generated for training. 71

4.1 Model visualization over the whole dataset. Trained with 80% of the data, early end date. 85
4.2 RMSE and loss variations. Model trained with 80% of the data, early end date. 85
4.3 Model visualization over the whole dataset. Trained with 100% of the data, early end date. 86
4.4 RMSE and loss variations. Model trained with 100% of the data, early end date. 86
4.5 Model visualization over the whole dataset. Trained with 80% of the data, full date range. 87
4.6 RMSE and loss variations. Model trained with 80% of the data, full date range. 87
4.7 Model visualization over the whole dataset. Trained with 100% of the data, full date range. 88
4.8 RMSE and loss variations. Model trained with 100% of the data, full date range. 88
4.9 Comparison of the ReLU and HardTanh functions used for activations. 90
4.10 Advanced model visualization over the whole dataset. Trained with 80% of the data, full date range. 92
4.11 RMSE and loss variations. Advanced model trained with 80% of the data, full date range. 92
4.12 Advanced model visualization over the whole dataset. Trained with 100% of the data, full date range. 93
4.13 RMSE and loss variations. Advanced model trained with 100% of the data, full date range. 93
4.14 RMSE of the best trained model over the whole dataset depending on the number of layers and neurons. 95
4.15 Complexity of the model depending on the number of layers and neurons. 95
4.16 Complexity of the model depending on the number of layers and neurons. Log scale. 96
4.17 Complexity x RMSE of the best trained model over the whole dataset product depending on the number of layers and neurons. 96
4.18 RMSE of the best trained model over the whole dataset depending on the number of past sampled. 98
4.19 Complexity of the model depending on the number of past sampled. . . 98

4.20 Complexity x RMSE of the best trained model over the whole dataset product depending on the number of past sampled. 99

Acronyms

MINES
Micro&Nano Electronic Systems

ML
Machine Learning

AI
Artificial Intelligence

NN
Neural Network

ReLU
Rectified linear unit

CLI
Command Line Interface

DMA
Direct Memory Access

ARIMA
Autoregressive integrated moving average

MLP
MultiLayer Perceptron

NNGP
Neural Network Gaussian process

BNN

Bayesian Neural Network

CNN

Convolutional Neural Network

SVR

Support Vector Regression

GRNN

General Regression Neural Network

KNN

K-Nearest Neighbors

RBF

Radial Basis Function network

MKL

Multiple Kernel Learning

SVM

Support Vector Machines

RNN

Recurrent Neural Network

LSTM

Long-Short Temporal Memory

GBC

Gradient Boosting Classification

WPE

Weighted Permutation Entropy

OS

Operating System

CUDA

Compute Unified Device Architecture

CSV

Comma Separated Values

SGD

Stochastic Gradient Descent

RMSE

Root Mean Square Error

Chapter 1

Introduction

1.1 Thesis Overview

This thesis is part of a larger research field that aims to improve plants growth and yield, intending to make food production more reliable, efficient, and secure. Food production is an essential factor of the second ONU Sustainable Development Goal: Zero Hunger [2]. With rapid population growth in areas of the world where food production is not trivial, mainly due to climate, understanding plants' health has never been more critical.

Figure 1.1: The second sustainable development goal: Zero Hunger.
Extracted from [1]

Much work has already been performed in the industry, especially at Politecnico di Torino under the MINES Research Group's "The Plant Project". The project's primary goal is to develop a self-sustaining monitoring system that can be deployed on the field to observe the plants being cultivated and inform the farmer on the health of their plants so that it can be improved. Specifically, extensive work on stem impedance measuring has been performed with tobacco plants grown on-site with different watering cadences to simulate different plant health scenarios. Work has also been performed on demonstrating that correlation exists between the impedance measures and soil moisture. Granger causality tests have been utilized to show causation between

the watering events and the impedance changes.
Plenty of work has also been conducted on the measuring devices used to capture impedance data, as well as on the in situ low power devices that would eventually be employed on the field to make real-time measurements and predictions.
This thesis, in particular, was proposed with the objective of closing the link between variables such as the impedance measured on the plants' stem as well as environmental conditions and the plants' health status, creating an actual model that could show how these are coupled. Machine Learning was chosen as the medium for this problem because it allows for great flexibility and has enormous potential over more traditional methods.
The work under this thesis focused especially on developing in the Python Language a software tool that allows for researching the best machine learning algorithms for predicting the plants' status. Fully parametric setting files and the capability to run the software with a Command Line Interface allow the user to:

1. Choose to perform a single training or a sweep;
2. Compile the appropriate setting file with the data processing settings, the training settings and, optionally, the sweep settings;
3. Run the software with the compiled setting;
4. Retrieve the single training or the sweep statistics and models.

With this fully parametric structure, the "core" of the code does not have to be changed between different runs. This architecture allows the user to perform any of the tasks that have been just mentioned in parallel and more than once at the same time. For example, it is possible to create a new settings file while multiple training tasks are being performed by looking at data obtained in the past.
By having these setting files, the users are not only allowed to choose between performing a single training or, instead, a training sweep over a specific setting, but they can also fully customize the training run in every detail. Regarding the data processing, the user can choose how the variables should be created from the input data files. The timeframe can be chosen by specifying a start and an end date. The data source, a range constraint, a filter, a transformation, and the normalization can be set for each variable. It is possible to create as many variables as necessary, and it is also possible to create multiple variables from a single source. For example, from the "date" parameter, it is possible to extract both the day's hour and year's day by applying different transformations. However, it is also possible to extract from environmental variables a regular version of the variable with just a simple smoothing filter but also a squared version of the same variable or again another version where the logarithm is applied. This feature can be helpful as the dependence might not always be strictly linear. The user can then

choose which kind of model should be employed, its topology, and how the training should be performed. A critical setting is the "Number of samples per parameter", which sets how far in the past the model should look to make a prediction. The simplest version can make predictions only on present data, but more complex versions can take data even days in the past. The model creation allows the user to specify any type of model topology with any kind of neuron and activation function available in the PyTorch library. The training settings allow the researcher to tune the learning rate, momentum, batch size, number of epochs, number of folds (if K-Fold validation is employed), and the type of loss function that has to be used. The training can be executed in two modes, an Interactive mode, and a Batch mode. Both modes produce the same results, but the interactive mode allows the user to check the training process and its progress in real-time. At the end of the process, the plots that are updated during every training cycle are saved among the other statistics in a text format. The plots show the validation and training errors for every training iteration and a visual representation of how the model performs over the entire training, validation, and test datasets. A log of the training process (with every chosen setting and the final errors) is also produced, and if a sweep was executed, a ranking with the best models is also provided.

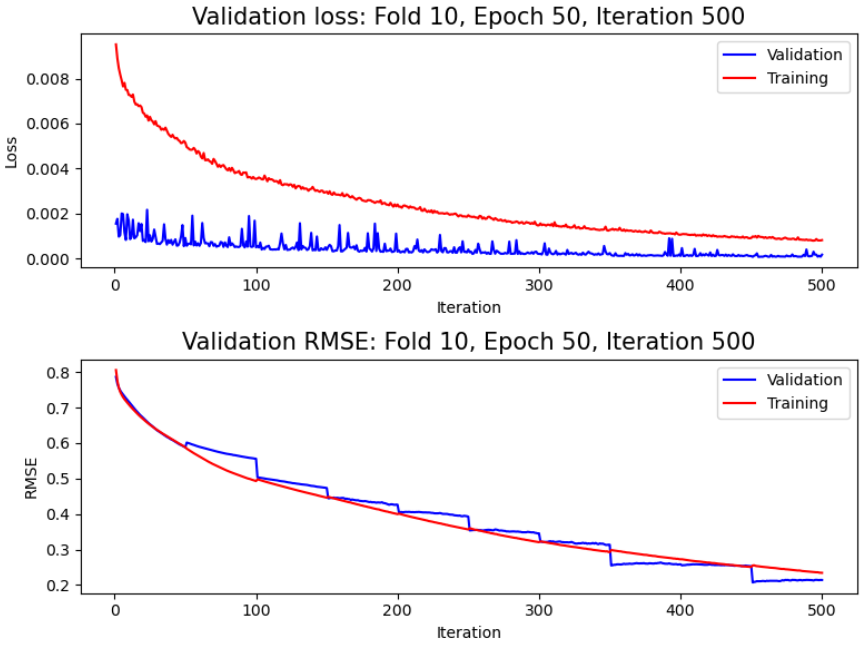


Figure 1.2: Example of a training and validation error visualization.

The framework just described can be represented with the block diagram of figure 1.3.

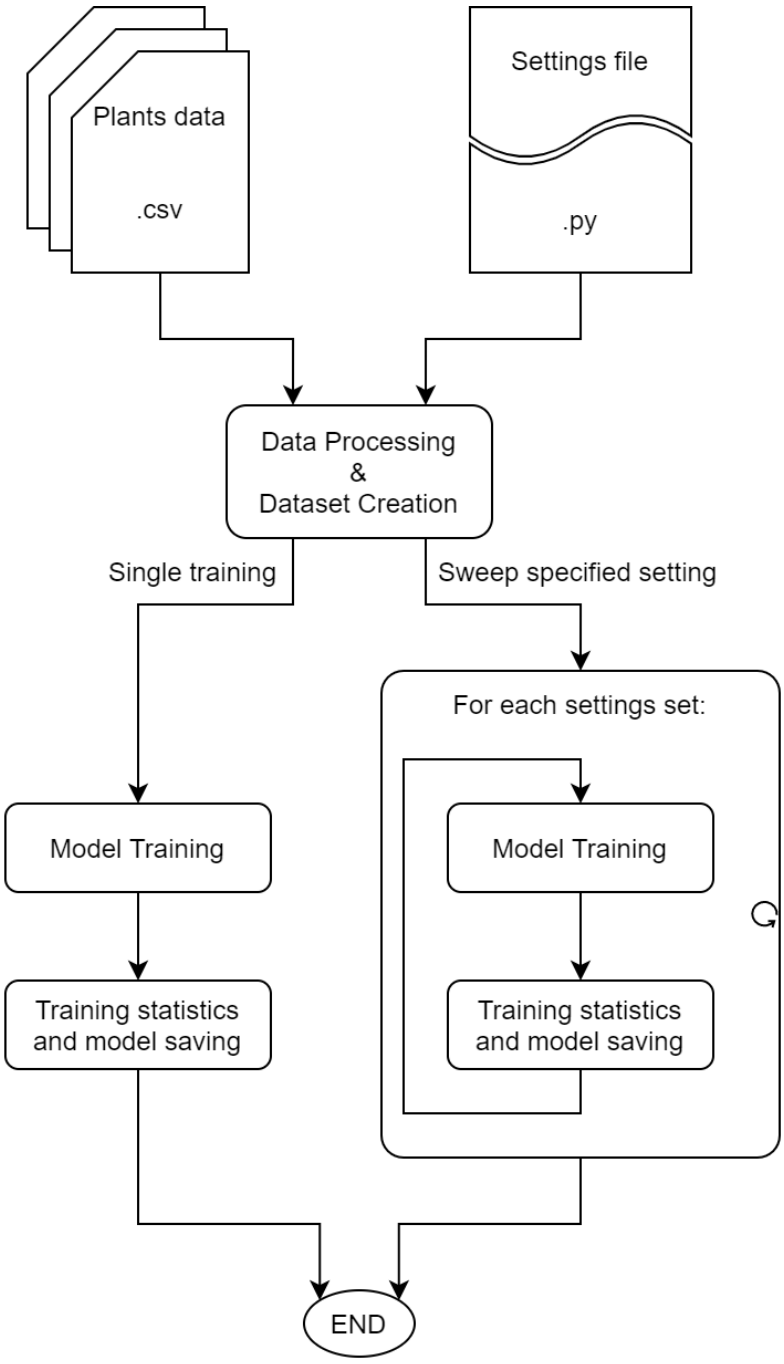


Figure 1.3: Block Diragram of the framework.

1.1.1 Thesis Structure

This thesis is structured on three main parts besides this introduction and the conclusion. The following section presents the state of the art on Plants Status in more detail and what the current scientific literature offers in terms of Machine Learning applied to this kind of problem. The second chapter focuses on the Machine Learning theoretical aspects that have been employed and how the ML concepts can be used for predicting the plants' status. The third chapter, the main one, is focused on the developed software. Every aspect of the software is detailed to show the reasoning behind it and how it was implemented. The description includes the data processing, the Neural Nets architecture, the training process, the searching process for finding the best settings, and details on the most important utilities developed. A brief section about a dispatcher that allows for much more manageable and parametric execution of the programs is also included. The fourth chapter focuses on the obtained results. Two main types of results are presented: what were found as the best settings for data processing and for the ML architecture, as well as final results on the prediction of the plants' status. After the conclusion, Appendix A presents how the developed software can be used in the intended manner.

1.2 State of the art on plant status

As already mentioned in the introduction, much work has already been done on extracting impedance data from plants and understanding its relationship with such plants' status and health. This work started by researching methods to measure the plants' vital parameters. One of the most promising methods, also utilized for this thesis work, focuses on measuring the impedance of the plant's stem. An In-Vivo four-point probe measurement setup has been developed, and the obtained data has been extensively analyzed[3][4]. Various setups with different probe distances and measuring equipment have been tested, paving the way for future on-the-field measurement stations. Impedance modulus and phase data at various frequencies can be recorded with this method, and the way they change has been studied, and electrical models of the plant stem have also been proposed. It has been demonstrated that impedance changes of both the modulus and the phase angle are linked to watering events, while other environmental factors also help correlate the changes[5]. These correlation checks were also supplemented by Granger causality[6] tests to verify that not only correlation exists between the soil moisture and the impedance data, but there is also causation among them. It has also been demonstrated that there is a relation between the electrical impedance of the plant stem and the daily cycle[7]. These effects are superposed to those mentioned before, caused by changes in soil moisture due to watering.

Most of the published research work is limited to what has been just mentioned, with additional information being very scarce. In order to compensate for this lack of resources on the matter of Machine Learning applied to data measured directly from live plants, research about studies and publications on "similar" topics has been conducted. The material that has been found can be partitioned into two different sets: studies on Machine Learning applied to agriculture in general or to other "live" entities, and studies on Machine Learning applied to fields that are not related to plants but that show similar behaviors (time variation, need to be monitored...).
As the final objective is to deploy the monitoring system on the field, and the system also has to be self-sustaining, the power consumption and so the complexity of the ML-based algorithms have also been taken into account. In this regard, it was found that it is possible to run ML-based algorithms efficiently on microcontrollers if the hardware allows for specific optimizations. It is also important to remember that, keeping in mind our objective, it is crucial to minimize the energy consumption more than the power as batteries will be needed. It is also significant to consider the hardware of the microprocessors carefully. For example, if the final algorithm heavily employs multiplications, it is fundamental to choose a device that can perform in hardware this operation[8][9]. The same concept can be applied to more complex operations. For example, logarithms or square roots might be needed for the data processing, in which case it is wise to select a processing unit that includes the hardware support for these operations. Some processors also allow computations on "external" units and allow the transfer of data to be elaborated via DMA[8]. Others also have additional units which can be used explicitly for Machine Learning tasks[8][9]. Such features can help reduce energy consumption significantly.
Moving on to the processing of the data, many possibilities are available to improve the performance of the ML algorithm by applying different operations to the data before it is provided to the Neural Network. This is especially important for time series that present seasonality due to the day-night cycle. Some of these solutions can be[10]:

- Transformations (i.e., applying the logarithm, the square root, a power, or other mathematical functions to every element in the series)
- Differentiation (subtraction of the previous "cycle" from the current one, like subtracting the data of the previous day from the data of the current one to only consider variations)
- Moving average
- Exponential smoothing[11]

- Autoregressive models
- Autoregressive integrated moving average (ARIMA) models

Besides applying these methods, it is also always recommended to scale each time series so that each one varies within the same range. The reasons for doing this are explained in the Chapter dedicated to Machine Learning in the Feature Scaling section.
When making predictions on variables that change over time, it was found that some types of algorithms perform better than others. Neural Networks perform quite inconsistently depending on the specific structure and the specific problem. Research on this problem has shown measured the performance of different models when applied to time series prediction problems[10]:

- Best models:
  - MLP (MultiLayer Perceptron): Best with moving average
  - NNGP (Neural Network Gaussian Process): Best with differentiation
- Decent models:
  - BNN (Bayesian Neural Network)
  - SVR (Support Vector Regression – SVM derived)
- Mediocre models:
  - GRNN (General regression neural network)
  - KNN (k-nearest neighbors)
- Models to avoid:
  - RBF (Radial basis function network)

A special mention must be given to a specific type of algorithm that has been successfully applied in the agriculture field, MKL (Multiple Kernel Learning)[12]. MKL consists in employing multiple types of kernels or neurons within the same algorithm. As some kernels perform better in certain situations and vice-versa, the better-suited ones in a specific situation should prevail by having multiple types of them. For example, it could be possible to have a branch of the network with normal neurons typical of an MLP and another branch based on Gaussian processes. This method dramatically increases flexibility, but complexity increases a lot.
Until now, all the algorithms that have been presented are especially useful when the objective is to make a prediction on a variable that changes over time that represents the health status of a plant. Another possible approach to tackle the

problem is to view cases where the plant's health is not good as anomalies. Anomaly detection algorithms may be particularly useful in this instance. Anomaly detection algorithms, unlike the previous algorithms, predict whether an anomaly could be occurring in any particular instant or not. Many different and also novel algorithms have been developed, such as[13]:

- Support Vector Machines (SVMs): the typical algorithm used for Anomaly detection, creates a "bound" where cases that fall outside are considered anomalies
- Recurrent Neural Networks (RNNs): contains neurons connected in loops, allows for memory saving thanks to the feedback. Abrupt changes can be identified well, especially with Long-Short Temporal Memory (LSTM) models as they keep track well of past data.
- Gradient Boosting Classification (GBC): Combines various weak predictors, typically Decision Trees
- Weighted Permutation Entropy (WPE): Takes advantage of the fact that sudden changes should entail strang changes in entropy

On a final note, it must be recognized that the medical and physiological fields could inspire approaches when working with plants, as in both cases the subject is a living being. In the medical field, CNNs (Convolutional Neural Networks), SVMs (RBF and linear), and Random Forest Networks have been employed with great results[14][15], but on very specif tasks. These tasks usually involve cardio-circulatory and neurological activity measures but always separately and usually involve a precise issue and not a broad "health" prediction. An interesting approach that has been used in the medical field is the so-called "Transfer Learning". The approach takes advantage of already well-trained neural networks to begin the training over new data without starting completely from zero. If there are commonalities in the systems that are being studied, better training results should be observed on the new model as some of the "learnings" would be shared between the models. This method has been applied to Bayesian Networks, Convolutional Neural Networks, and also on Multilayer Perceptrons[14].

Chapter 2

Machine Learning notions

2.1 Machine Learning fundamentals

Before diving into the developed software, an introduction about the Machine Learning fundamentals that have been employed is necessary to create a background sufficient to understand the choices that have been made. As such, this is not intended as a full and exhaustive description of Machine Learning.
First of all, what is Machine Learning, and why was it chosen to predict the status of plants? Machine Learning is a branch of Artificial Intelligence. It is utilized in cases where it is necessary to build a model based on known data (the "Training Set") without explicitly programming it to make predictions on different data. It is especially advantageous when the relation defining the model is not known a priori. Nowadays, Machine Learning has applications in a considerable amount of fields. From predicting shopping preferences to self-driving cars, from email spam filters to medical diagnosis, its usage is so widespread that ML may seem like a *panacea*. Thanks to its many different facets, very different solutions can be found for very different and complex problems. Obviously, as anything that seems too good to be true, it has its drawbacks and issues. ML algorithms are not explicitly programmed, but only the general structure is developer-defined. It may very well be possible that a solution to the problem that needs to be solved may not be found (even if it might exist), or the results may fall far from expectations. This has been the case for many ML projects in the past decade.[16]
That said, ML is one of the best tools at our disposal for tackling challenging problems, the problem at the roots of this thesis being an excellent example.
What now follows are a few sections dedicated to some machine learning fundamentals that have been key to this thesis work.

2.1.1 Classification vs Regression

As already anticipated before, ML can be used to make predictions based on new data after the model has been tuned with a training set. However, what kind of predictions can be made? We can subdivide this problem into Classification problems and Regression problems.
Classification problems are when the model is tasked with detecting different categories. To provide an example, we can think of an Optical Character Recognition (OCR) system: An image of a character (input) is provided to the ML-based algorithm, and the model outputs the identified character. We have a finite set of characters (letters, numbers, signs), and the model is expected to pick one of them.
On the other hand, regression problems require the model to predict a specific value in a defined range provided the inputs. The model usually acts as a multivariate, typically non-linear function where, given a set of input values, we are provided by the model with an output value that should better fit the data employed during the training process. The simplest version of a regression problem is Linear Regression, where the model can be represented in this form:

$$y = \theta_0 + \theta_1 x \tag{2.1}$$

According to this notation, y is the prediction or the hypothesis (the "output"), x represents the features (the "inputs"), while θ_i are the coefficients of our model. y may be a scalar if there is a single output or a vector in case there are multiple outputs. x may be a scalar if there is only one input or a vector if the inputs are multiple. The dimensions of θ_i depend on the dimensions of x and y. In this particular case θ_0 has the same dimensions of y while θ_1 has dimensions dependant on both x and y, where if x has dimensions m and y has dimensions n then θ_1 has dimensions $m \times n$.
It is possible to build a Classification algorithm starting with a Regression algorithm by introducing activation functions such as the Logistic function or the Rectified linear unit (ReLU) and introducing decision thresholds. For example, Logistic regression is one of the simplest models that can be used for classification, where a logistic function is applied to a regression model.
The regression and classification models can be made a lot more complex and in many ways, some of which will be described in the following sections.

2.1.2 Supervised learning vs. Unsupervised learning

A fundamental distinction to be made about Machine Learning is between Supervised learning and Unsupervised learning. While both fall under the broad ML umbrella, they represent very different concepts and usages of Machine Learning, and different algorithms are utilized to employ them.

Supervised learning includes the Classification and the Regression problems that have been mentioned in the previous section. Supervised learning means that for training the dataset includes an expected result as in the "right output" that we expect the fully trained model to predict.
However, this is not always what is needed. In some cases, we may want to take advantage of Machine Learning to actively search for clusters (Clustering) or for Anomaly Detection. In the first case, the objective usually is to group objects in different clusters by finding unique traits that differentiate them. For example, this is used widely in Market research utilizing data from surveys to identify different market segments to better target potential customers. The customers, however, do not identify themselves as belonging to a specific market segment, so Unsupervised Learning is needed. For the Anomaly Detection case, the use case is quite different. In this situation, the objective is not to identify different clusters as there is only a large "normal objects" cluster, but to identify objects that fall outside "normal", the so-called outliers. This is very useful for cases where most of the data in the dataset represent what should be expected, and only a few elements represent anomalies. Usage examples can be fraud detection and system health monitoring, where new types of fraud in the first case or new types of faults for the second may eventually appear. ML-based Anomaly detection algorithms may identify such outliers and respond accordingly without being explicitly programmed to expect them.

2.1.3 Training process

The process that defines Machine Learning, which sets it apart from other algorithms, is the training process. The training procedure allows the model with the defined topology to become "intelligent" and not just take random guesses. What the training process does is tuning the internal parameters of the model so that, once it is given new inputs, not part of the training set, it can make a reasonable prediction.
The model is first initialized with random parameters in order to prepare it for the training. A great random initialization is essential for good training as the parameters will slowly change starting from those initial values. The training process works by cyclically tuning each parameter so that the cost function is gradually reduced for every iteration and is minimized at the end of the training process. The cost function, or loss function, is a function that indicates the quality of the model and how good the predictions it makes are.
Multiple training algorithms have been invented and developed, and all have different advantages and disadvantages. One of the most common algorithms is Gradient Descent, and it works, as the name suggests, by computing the gradient of the cost function. In particular, each component of the gradient relative to each model parameter is used to increase or decrease its value by the partial derivative amount

multiplied by a factor α called the learning rate. The higher the learning rate, the more the parameters are changed in each iteration.
So, the model parameters are updated according to this procedure, that is iterated multiple times, conforming to the following equation (θ_j are the model paramters, j is the unique number of each paramter of the model, t is the iteration number, J is the cost function):

$$\theta_{j,t} = \theta_{j,t-1} - \alpha \frac{\partial}{\partial \theta_{j,t-1}} J(\boldsymbol{\theta}_{t-1}) \tag{2.2}$$

Every step in this iterative process is called an epoch, and the total amount of iterations is the number of epochs.
During the training process, it is possible to monitor the training by employing a validation set, with data that is not part of the training set, to check if the model is improving. The loss function should, in fact, decrease as the training progresses not only when applied to the training set but also to the validation set. However, the validation set that was just mentioned includes data that is not actively used for training, which might be an issue if not a lot of data is available. So other solutions have been developed, such as the K-fold cross-validation. It is essential to monitor the training process to fine-tune the learning rate and check if the training improves the model. In particular, if the learning rate is too high, the cost function might increase as divergence occurs. On the other hand, if the learning rate is too low, it might take too many iterations to train the model.

K-fold cross-validation

The k-fold cross-validation method allows for training to be computed on all the available data by swapping the holdout data used for the validation with data that is part of the training set. This swap accurs cyclically, as an outer loop of the training process. How the data assigments change for each cycle is shown in figure 2.1. As it can be seen, for each fold the Validation data changes.

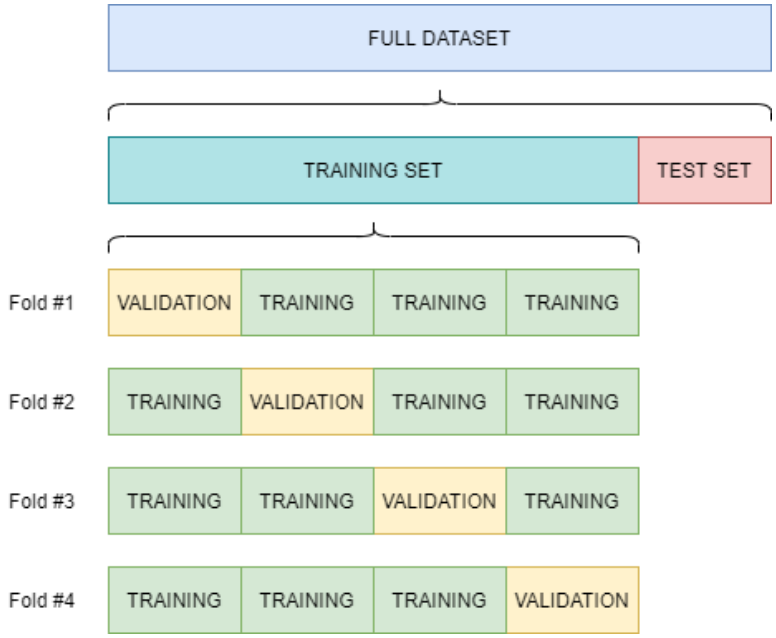


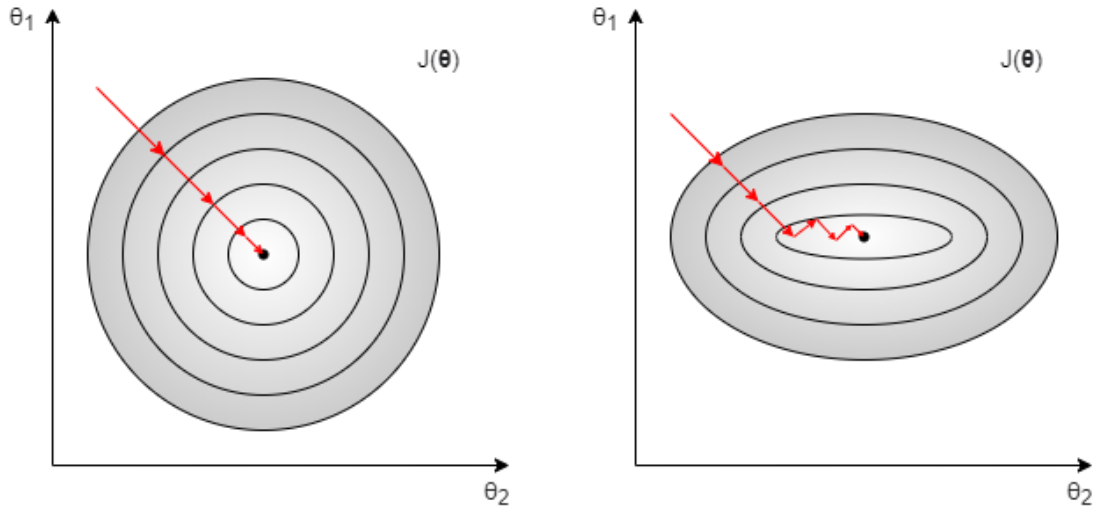
Figure 2.1: K-fold data assignment.

In contrast to the regular holdout validation, where a certain amount of data is kept for validation only, this type of cross-validation is advantageous when the available dataset is small or not very large. However, it becomes harder to understand how the model will react to new data as it will have been trained on all the available data.

2.1.4 Feature Scaling

As mentioned previously, our ML-based software may take multiple inputs, called features, provide them to the model and obtain a prediction as output. These inputs may represent very different types of variables, each one with potentially very different ranges. For example, a measure of an angle measured in degrees may have values that range between 0° and 360°, while a variable about the hour of the day may range from 0 to 24 and, on the opposite end, ambient lighting measures may reach values in the tens of thousands of lux.
This may not seem like a problem as long as these numbers stay within the sizes of the variables employed in the software. However, during training, this is a big issue as all the model parameters get initialized with random values. These random values have the same initial variance for any input type, which may cause issues when applying algorithms like Gradient Descent as the descent may happen faster over some variables and slower over others. Instead, it is preferable if the descent happens more evenly over all variables so that convergence can occur much faster.

This is especially noticeable in the example on figure 2.2a, where both features have the same scale, when compared to the example of figure 2.2b, where the feature on the vertical axis has a smaller scale compared to the one on the horizontal axis. The red arrows represent the descent over the cost function, which occurs much faster when the features have the same scaling.



(a) With Feature Scaling. **(b)** Without Feature Scaling.

Figure 2.2: Gradient descent towards minimum.

To avoid this issue, it is better to restrict the range of the features by scaling them or performing a normalization. The ideal case is to perform a normalization in the following range, where x_i is the i^{th} feature:

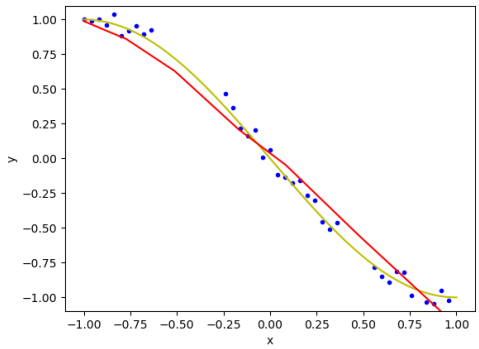
$$-1 \leq x_i \leq +1 \tag{2.3}$$

This ensures that the issues presented before are avoided.

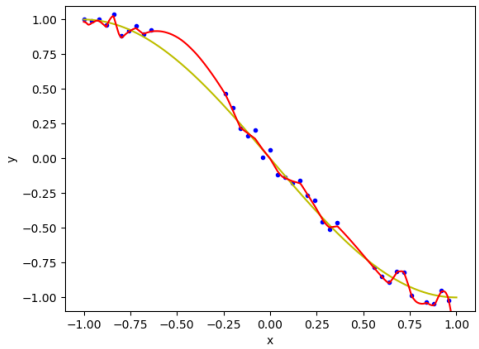
2.1.5 Bias – variance tradeoff

The Bias - variance tradeoff is one of the most significant issues in Machine Learning algorithms. It is a choice that has to be always made, and finding the correct point where to make this tradeoff is not an easy and always objective task.
During the training process, the model starts with randomly initialized parameters. Soon after, the model begins to "learn", and the predictions start to make sense. At this moment in training, if the process is working well, the model is usually in a high variance - low bias situation, where the issue is underfitting the data. The model is not accurate enough to predict well for any element of the dataset. This

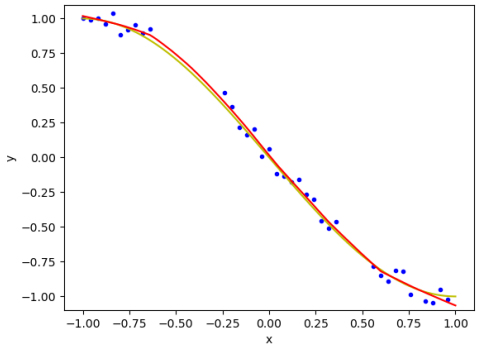
occurrence is shown in figure 2.3a.
If we let the training go on for too long and if the model is complex enough, while the variance will decrease dramatically, an opposite but equally bad situation will take place. Now high bias will be the issue, and overfitting will occur. In this case, the model has learned the data in the training set too well, but this will cause underperformance when the model will have to make predictions based on data that is not similar enough to that of the training set. An example of what may occur in this case is shown in figure 2.3b.
The best model sits in the middle, where the training has allowed it to make guesses that are good enough in any case, buth for the training set and the validation set, as is shown in figure 2.3c.



(a) High variance model.



(b) High bias model.



(c) Well trained model.

Figure 2.3: Different outcomes of training on the same training set.

2.1.6 Neural Networks

Neural Networks are a specific type of Machine Learning algorithm among the many that are available. Figure 2.4 shows the structure of a generic Neural Network, with the inputs shown in green, the "hidden layers" shown in yellow and the output shown in red:

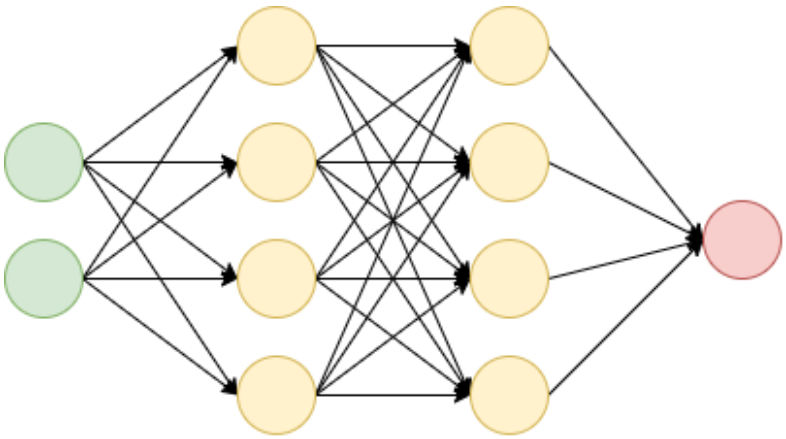


Figure 2.4: A generic Neural Network.

Each node represents a neuron. Neurons are none other than mathematical functions, which may be more or less complex. The transfer function that defines the neuron is applied to the inputs, and the produced output is forwarded to the next neurons or to the neural network output. The function that defines the neuron is also called the activation function, as it resembles how a real neuron works, which "fires" when it gets activated. The connections between the neurons transfer the output of each neuron to those of the next layer or to the output. These connections could be considered as the dendrites of a biological neuron. The similarities are evident as shown on figures 2.5 and 2.6.

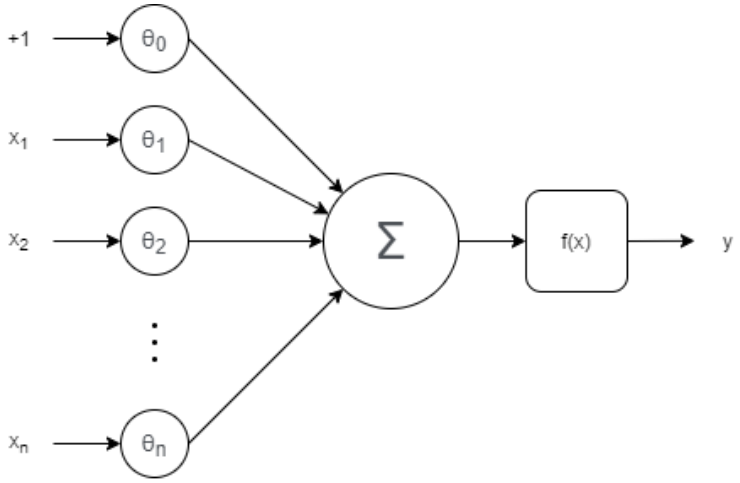


Figure 2.5: Structure of an artificial Neuron.

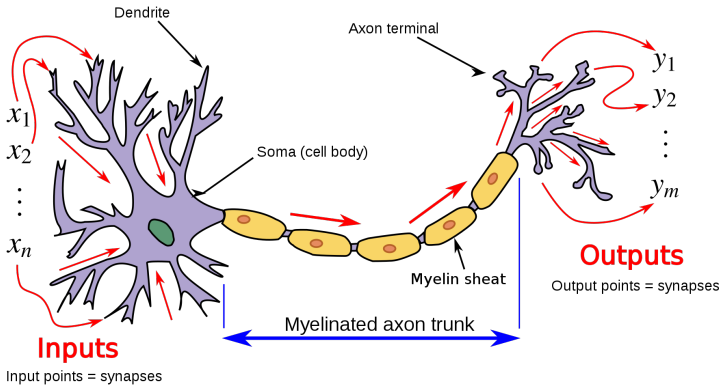


Figure 2.6: Structure of a biological Neuron.
Extracted from [17]

The inputs and the outputs of the artificial model, which are the connections between the neurons in the neural network, can be associated with the synapses of the real neuron, the summation node of the artificial one can be associated with the Soma, and the activation function can be related to the Axon.
Multiple types of activation functions have been developed, and all have different peculiarities. Most transfer functions start by performing a weighted sum of all the inputs plus a bias. The weights of each neuron end up being the parameters of the Neural Network model. The output of the neuron can be left as it is, in which case

it is called a linear activation:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sum_{i=0}^{n} x_i \cdot \theta_i = \boldsymbol{x}\boldsymbol{\theta} \tag{2.4}$$

$\boldsymbol{x}$ and $\boldsymbol{\theta}$ are vectors, where in particular one of the elements of each vector, usually the first one, is reserved for the bias term. So, usually, the x_0 term is always equal to 1 and the θ_0 term defines how much bias should be added. The terms from x_1 to x_n are the actual neuron inputs. Another very important activation function is the sigmoid activation. In this case the same weighted sum is performed, but then the output is supplied to the sigmoid function, which follows the following law:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

The shape of this function is shown in figure 2.8b.
The Neuron activation function becomes the following:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \frac{1}{1 + e^{-\boldsymbol{x}\boldsymbol{\theta}}} \tag{2.6}$$

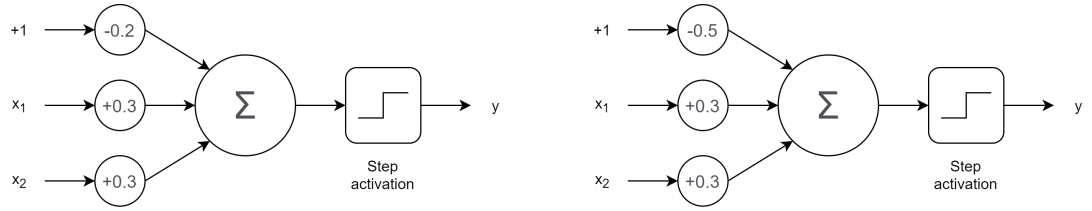
The sigmoid function limits the output range between -1 and +1 and introduces a strong nonlinearity, thus allowing for a much more complex model. However this function is computationally intensive due to the additional exponential function and the division. This might be an issue with many neurons, so simpler models might be better suited. One such model is the simple step function:

$$f(x) = \begin{cases} 0, & \text{if } x < 0. \\ 1, & \text{if } x \geq 0. \end{cases} \tag{2.7}$$

The shape of this function is shown in figure 2.8c.
The Neuron activation function becomes the following:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \begin{cases} 0, & \text{if } \boldsymbol{x}\boldsymbol{\theta} < 0. \\ 1, & \text{if } \boldsymbol{x}\boldsymbol{\theta} \geq 0. \end{cases} \tag{2.8}$$

This function also introduces strong non-linearities but also removes a lot of information as only two values, zero or one, are allowed to pass. With this activation function it is possible to emulate logic gates. This is shown in figure 2.7 where with the appropriate weights and biases the OR and AND gates are presented:



(a) NN based OR gate.
$y = x_1$ OR x_2

(b) NN based AND gate.
$y = x_1$ AND x_2

Figure 2.7: Neural Networks with a single neuron emulating logic gates.

x_1	x_2	Σ	$y = h_{\boldsymbol{\theta}}(\boldsymbol{x})$
0	0	-0.2	0
0	1	+0.1	1
1	0	+0.1	1
1	1	+0.4	1

(a) Truth table of the NN based OR gate.
$y = x_1$ OR x_2

x_1	x_2	Σ	$y = h_{\boldsymbol{\theta}}(\boldsymbol{x})$
0	0	-0.5	0
0	1	-0.2	0
1	0	-0.2	0
1	1	+0.1	1

(b) Truth table of the NN based AND gate.
$y = x_1$ AND x_2

Table 2.1: Neural Networks with a single neuron emulating logic gates.

Another function that is simpler than the Sigmoid but less "aggressive" than the step function is the Rectified linear unit (ReLU). This function can be represented in these two equally valid forms:

$$f(x) = \begin{cases} 0, & \text{if } x < 0. \\ x, & \text{if } x \geq 0. \end{cases} \tag{2.9}$$

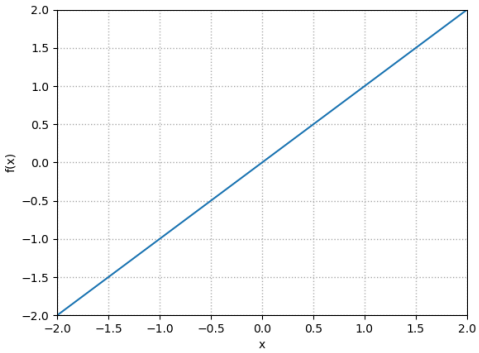
$$f(x) = max\{0, x\} \tag{2.10}$$

The shape of this function is shown in figure 2.8d.
The Neuron activation function becomes the following:

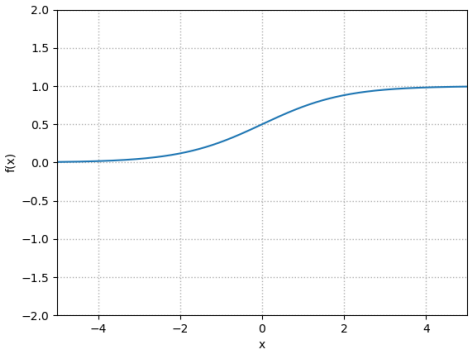
$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \begin{cases} 0, & \text{if } \boldsymbol{x\theta} < 0. \\ \boldsymbol{x\theta}, & \text{if } \boldsymbol{x\theta} \geq 0. \end{cases} \tag{2.11}$$

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = max\{0, \boldsymbol{x\theta}\} \tag{2.12}$$

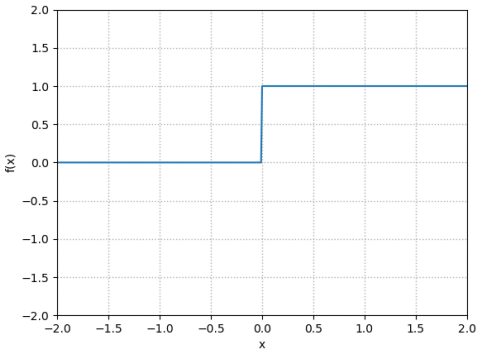
This function is used a lot in many application as it provides a great balance of low complexity and non linearity.



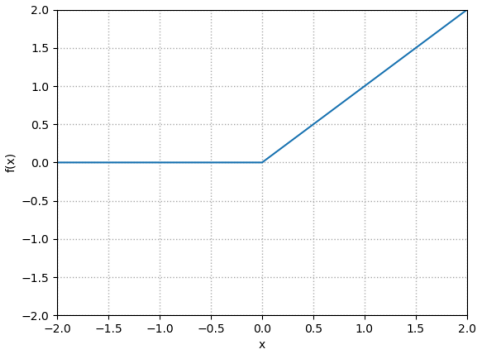
(a) Linear function.



(b) Sigmoid function.



(c) Step function.



(d) Rectified linear unit (ReLU).

Figure 2.8: Comparison of different functions used for activations.

Many other more or less complex activation functions exist. I want to mention just a few notable ones:

- Gaussian activation
- Hyperbolic tangent activation
- Softplus activation

More sophisticated types of Neural Networks have also been created. The following are definitely noteworthy:

- Convolutional neural networks
- Recurrent neural networks
- Support-vector machine

2.2 Machine learning applied to plants

Using ML to predict the plants' status, we must first define what variable we want to predict and how we want to predict it. We have many possibilities to choose from, and every possibility, which describes a different problem, can produce very different results.
Supervised Learning is the most natural choice for plants status, as the focus is not to search for patterns. This being said, the "Plant status" must be defined. In the context of Machine Learning, two very different types of variables can be predicted:

- **Continuous variables**: Can assume any possible value within a range. Examples: air humidity(%), impedance module (Ω), soil moisture (kPa).
- **Discrete or categorized variables**: Can only assume a specific amount of "enumerated" values. Examples: plant appearance (blooming, lively, withered, dry, dead), soil moisture (damp, slightly wet, dry)

For the first type of variables regression is needed, while for the second Classification is employed.
While the Discrete examples may seem a lot more intuitive in the context of plant status, they are either much more subjective, as the plant appearance for the training set would be based on visual inspection of each plant, or would be a simplification of a Continuous variable for the soil moisture case. The appearance variable would be subjective as it would be obtained by manually labeling pictures of the various plants in the dataset. This is a big problem as different labelers may consider plants contradictorily, and bad lighting and low resolution of the pictures could induce many errors. The categorized soil moisture must also be considered carefully as definitions such as damp or dry are subjective, and their relation to the measurable continuous variable moisture is arbitrary. As such, employing the measurable continuous variables directly is a straightforward approach. While their processing requires subjective decisions too, those decisions can be primarily based on the mathematical properties of such variables.
The continuous variables regarding tobacco plants that have been recorded at Politecnico di Torino include:

- Impedance phase of the stem [Ω]
- Impedance angle of the stem [°]
- Soil moisture [kPa]
- Relative ambient humidity [%]
- Ambient temperature [°C]

- Ambient illuminance [lux]
- Time and date of the measurement [YYYY-MM-DD hh-mm-ss]

All these variables can be processed and different parameters associated with them can be extracted and potentially used as inputs or outputs of the plant status prediction algorithm based on machine learning.
The main goal of The Plant Project is to link the stem impedance to the plant status, so clearly the two impedance variables should be employed. Every other variable, while helpful, could be considered as "overhead" when thinking about the end goal of having on-the-field electronic units identifying the status of the plants. The time and date should definitely be included, at least in the form of hour of the day, as the plant life cycle depends on the day-night cycle. On the other hand the other variables should be employed only if absolutely necessary. For example, the temperature could be easily monitorable and can be very useful for predicting the plant status.
Regarding which variable(s) should be used as metrics of plant status, the most related one intuitively is the soil moisture as a plant needs moist terrain to be healthy. Of course, this neglects other dynamics in the lifecycle of plants (like humans need to drink to be healthy but may not be healthy just because they drink), but it is a good initial approximation.
To sum things up, we have a set of environmental and plant extracted variables (such as moisture, impedance, the hour of the day, and temperature) that can be used for our training set. Once the data processing algorithms are fine-tuned and the Neural Network training is complete, the ML-based system could be deployed on the field and fed with real live data to predict the plants' status potentially. This system can then be used to improve the health of plants and so improve the farming processes.

Chapter 3

Software Implementation

3.1 Introduction

This chapter is dedicated to how the entire framework was developed, the ideas behind it, and how it works. The software has been developed in the Python language[18], in particular in version 3.9, under an Anaconda virtual environement[19] to avoid any possible conflict with other scripts or pre-installed modules. This setup was chosen for a multitude of reasons. First, the Python language, being an interpreted language, allows for the framework to be executed on any machine that is running one of many popular Operating Systems such as:

- Windows (multiple versions)
- Linux (source code is provided)
- MacOS

Python ports are also provided for other operating systems; however, the Machine Learning framework that was employed only supports these three most used OSes. The second reason why Python was chosen as the language is this very same ML framework: PyTorch. PyTorch is provided as an importable module and provides many functionalities in a format that is easy to integrate into the software. It also supports GPU accelerated calculations via Nvidia's CUDA, which can be extremely useful to offload much of the workload from the CPU. The third reason is related to the second, and it is about the availability of many public libraries that allow for the implementation of very complex features that would otherwise take much time to be developed. For example, the matplotlib library allows data plotting, and the Pandas module provides complex data structures such as the DataFrame that are very useful in the data science field. Even "simple" utilities like the CSV parser implemented in the Pandas library are incredibly convenient so that they

do not have to be developed from scratch and are already debugged.
As the data to be used for training was already made available through the past work of the MINES Research Group, the software development was initially focused on creating a baseline ML architecture. Initially, two ML architectures were laid out: one named "forecast" and one named "status_now". The first one was not developed due to issues with the PyTorch_forecasting module (derived from the base PyTorch module). The second one instead became the foundation for the whole framework. Initially, the essential ML functions for training and testing the Neural Network and functions dedicated to creating the training dataset were created. These fundamental portions of code kept getting expanded and improved as new features were added. With these main parts of code, the first iteration of "status_now" was also implemented. Initially, the training used hardcoded settings in the script, but the tool soon evolved to accept settings provided via a unique settings file and later via externally provided files. The code of this main script was also later taken advantage of to create the "finder" that lets the user search for the best settings with a sweep over a setting. Finally, a dispatcher has been implemented to completely detach the training and data processing settings from the software core so that multiple instances with different settings can be executed simultaneously.
The latest and most complete version of the software is described.

3.1.1 Framework structure

The software is structured in different files, either Python scripts, csv input files and text and image result files, organized in different folders. Picture 3.1 shows this organization:

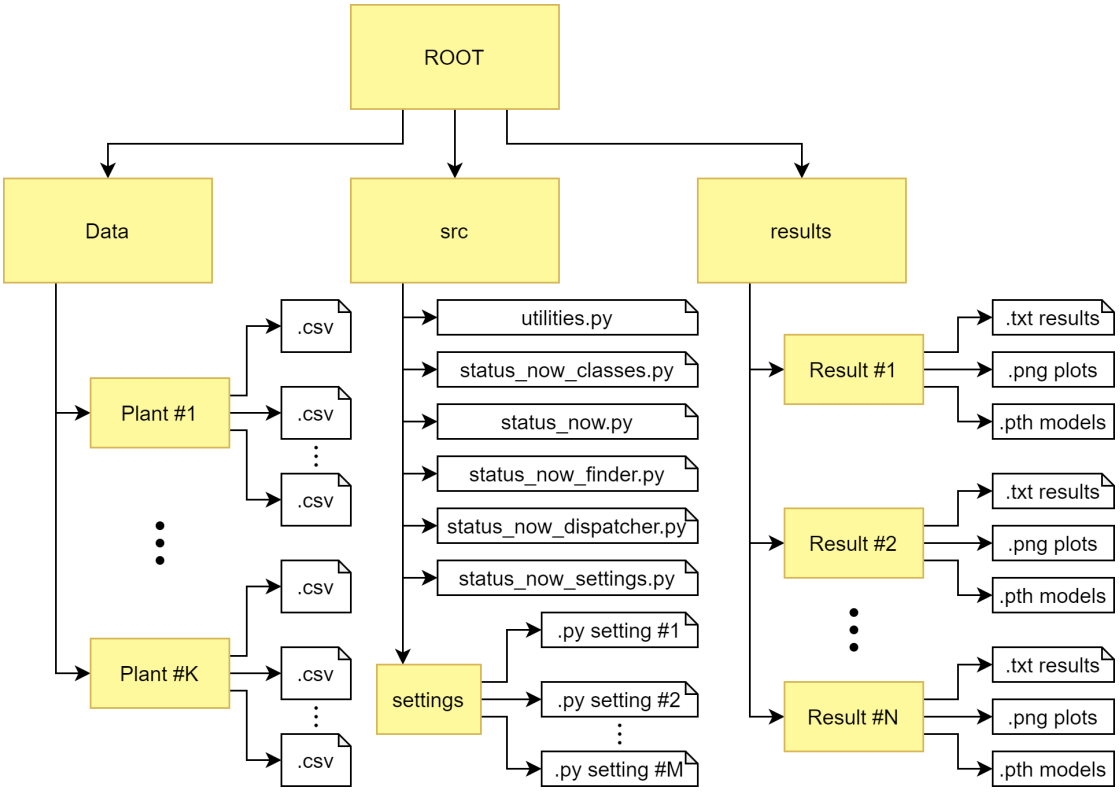


Figure 3.1: Organization of the framework.

Three main folders are present: the "src" folder, the "Data" folder, and the "results" folder. The "src" folder, as the name suggests, is the source files folder. Inside it, there are all the python source files and a settings folder that contains all the setting files. The "Data" folder is used to store the source CSV files that can be used for the training of the Neural Networks. Finally, the "results" directory is used to save the results created after each training or search execution.
The source files in the "src" directory are:

- utilities.py - File that contains utility functions that are used in the other files.
- status_now_classes.py - File that contains "status_now" related classes.
- status_now.py - File that contains the scripts used to perform a single data preparation and Neural Network training routine.
- status_now_finder.py - File that contains the scripts used to perform a sweep over certain variables and find the best settings for data processing and NN training.

- status_now_dispatcher.py - File used to dispatch the setting files to the correct script and run the framework more efficiently on a shell interface.
- status_now_settings.py - File containing default "status_now" settings.

The code contained in these files is described in detail in the following sections.
Within the "src" directory, the "settings" directory is also present. This folder is employed to store all the settings files to be retrieved by the dispatcher. The settings files need to be fully compiled with every necessary setting for the framework to work correctly; for this reason, template files are provided so that only the predefined fields have to be filled in. Two kinds of templates are provided: one for the single training process (to be executed by status_now.py) and another for the sweep process (to be executed by status_now_finder.py). Both files contain similar settings. In fact, settings such as the data processing settings, the training settings, the NN structure, the plants, and the timeframe are common between the two types of settings. The main and only difference is that the sweep settings also include settings related to the particular sweep. For example, a sweep over the number of hidden layers in the Neural Network will have a setting list for the amounts of layers to be swept and another setting list for the number of neurons per layer, as shown in listing 3.1. The total amount of iterations will be equal to the size of the first list multiplied by the size of the second list.

Listing 3.1: Setting example: Number of layers in the NN sweep.

```
1    # List of the number of layers to test
2    finder_setting.n_layers_values = [x for x in range(1, 10 + 1, 1)]
3    # List of the number of neurons per layer to test
4    finder_setting.n_neurons_values = [32, 64, 128, 256, 512]
```

As shown in listing 3.1, the first list for the sweep of the number of layers contains all the integer numbers from 1 to 10, while the list for the number of neurons in each layer to be tested contains five possibilities. A total of 50 training steps will be performed, starting with a training run with only a single hidden layer with 32 neurons and ending with a Neural Network composed of 10 hidden layers with 512 neurons each.
The "Data" folder does not have a strict hierarchy inside it. Currently, the folder is organized with a subfolder for each specific plant that is being used with plant-specific files within each subfolder. However, this is not mandatory as the data files' location can be specified in the settings files.
Finally, the "results" directory contains as many folders as many training and searches have been executed. Each folder has a name that indicates if it was a single training ("status_now") or a sweep ("status_now_finder") and what settings file was used for the first case or the search type for the second, along with the

date the execution started. Inside the results folder for the status_now executions, there will be a "results.txt" file with the employed settings and the error obtained at the end of the training and on the epoch with the lowest error. Plots of the model predictions over the whole dataset at the end of training and on the epoch with the lowest error will be saved, along with the models in the .pth format. A Validation Error plot with the evolution of the errors over time is also going to be saved. The results folder for the sweep contains the same files mentioned before, one for each training in the sweep, and ranking text files that highlight the models with the lowest errors.
Appendix A describes in much more detail how the software can be used.

3.2 Data preparation and processing

The data preparation and processing portions of the software are fundamental for the execution of the whole framework as the outcomes of the training efforts are highly dependent on how the data is handled beforehand. This will also be shown in the final chapter with the results, but, as it is easy to imagine, this portion of the code affects significantly the outcome of the training. Applying different functions or filters to the available data and choosing which and how much data should be given as input to the Neural Network alters the results considerably.
The data used by the framework is provided via CSV files that contain the plants' measured impedance and environmental data. Each CSV file should contain the data of a single plant. The field or column corresponds to a specific variable whose name is standardized among all CSV files. These names are also spelled out on the first line of the CSV file. Listing 3.2 shows an example of a standard CSV file with plants data (The "Temperature [C]", "Air Humidity [RH]", and "Ambient Light [lux]" fields have been omitted due to page space constraints). The first nameless field is the row number.

Listing 3.2: CSV file example.

```
1 ,Moisture [KPa],Date,impedance_modlus,impedance_phase
2 0,-5.652781040304957,2021-03-23 14:37:53,973.782052159,-90.776159869
3 1,-5.624867456369233,2021-03-23 15:37:53,2729.400701413,-87.17937705
4 2,-3.683619830182237,2021-03-23 16:37:53,1892.752663595,-88.013926153
5 3,-3.693476326650445,2021-03-23 17:37:53,1888.156385313,-88.212648753
```

The CSV files used for this thesis work (Of which the listing above is part) have been generated with software developed by the MINES Research Group of Politecnico di Torino. The data they contain refers to measures also performed by the MINES Research Group on tobacco plants under different environmental conditions.

The general flow of the data is shown in figure 3.2, but every step is explained in detail next.

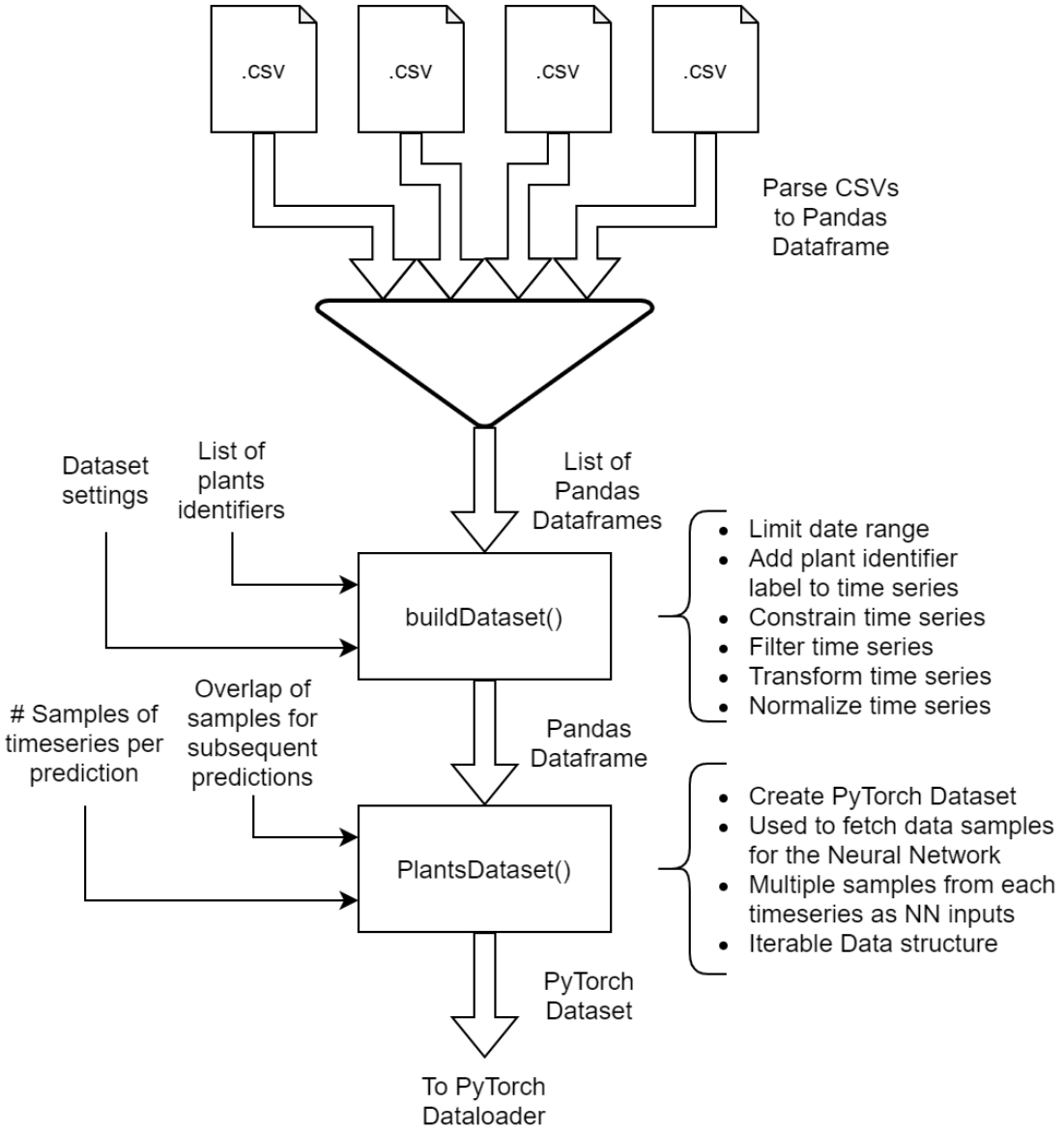


Figure 3.2: Flow of the data during initial preparation.

As represented on figure 3.2, three main steps take place during Data preparation and processing: The CSV files parsing, the full dataset creation with the *buildDataset()* function and the PyTorch Dataset derived *PlantsDataset()* object creation.
The CSV parsing step is the easiest and less complex one. The names of the

CSV files are passed on to the *read_csv()* function of the Pandas library, and the resulting dataframes are added to a list that is then passed on to the next step. At this point, each line in any dataframe corresponds to a series of measurements that occurred at the time specified in the "Date" field. However, these dataframes do not carry information about which plant the data came from, so they cannot just be merged. The example shown before on listing 3.2 would produce a dataframe like the one represented in table 3.1 once parsed.

	Moisture [KPa]	**Date**	**impedance_modlus**	**impedance_phase**
0	-5.652781040304957	2021-03-23 14:37:53	973.782052159	-90.776159869
1	-5.624867456369233	2021-03-23 15:37:53	2729.400701413	-87.17937705
2	-3.683619830182237	2021-03-23 16:37:53	1892.752663595	-88.013926153
3	-3.693476326650445	2021-03-23 16:37:53	1888.156385313	-88.212648753
⋮	⋮	⋮	⋮	⋮

Table 3.1: Dataframe obtained after the CSV parsing.

The next step involves calling the *buildDataset()* function, which is implemented in the *utilities.py* file. The function is used to process the data contained in the dataframes collected in the list obtained before so that:

- The data is merged in a single Pandas DataFrame
- Each row is provided with its unique plant identifier
- The data rows are limited within a Date range
- The data of each time series (column) is:
  - Constrained within a given value range
  - Filtered according to a given filtering function (Example: moving average)
  - Transformed according to a given function (Example: logarithm)
  - Normalized in the $[-1; +1]$ range

To perform all these operations the *buildDataset()* function requires many input parameters: the list of DataFrame from the parsed CSV files, the list of corresponding unique plant identifiers (in the same order as the previous list), a "dataset_settings" list of dictionaries that describes how the new columns will be, a list of the plants that have actually to be used, an optional integer number for removing the last n rows, the starting and the end dates to limit the columns timeframe, the name of the column to pick the Date from, and finally a "testMode" debug flag that, if set true, builds a test dataframe from scratch without using the provided data. A block diagram of how this function works is provided in figure 3.3.

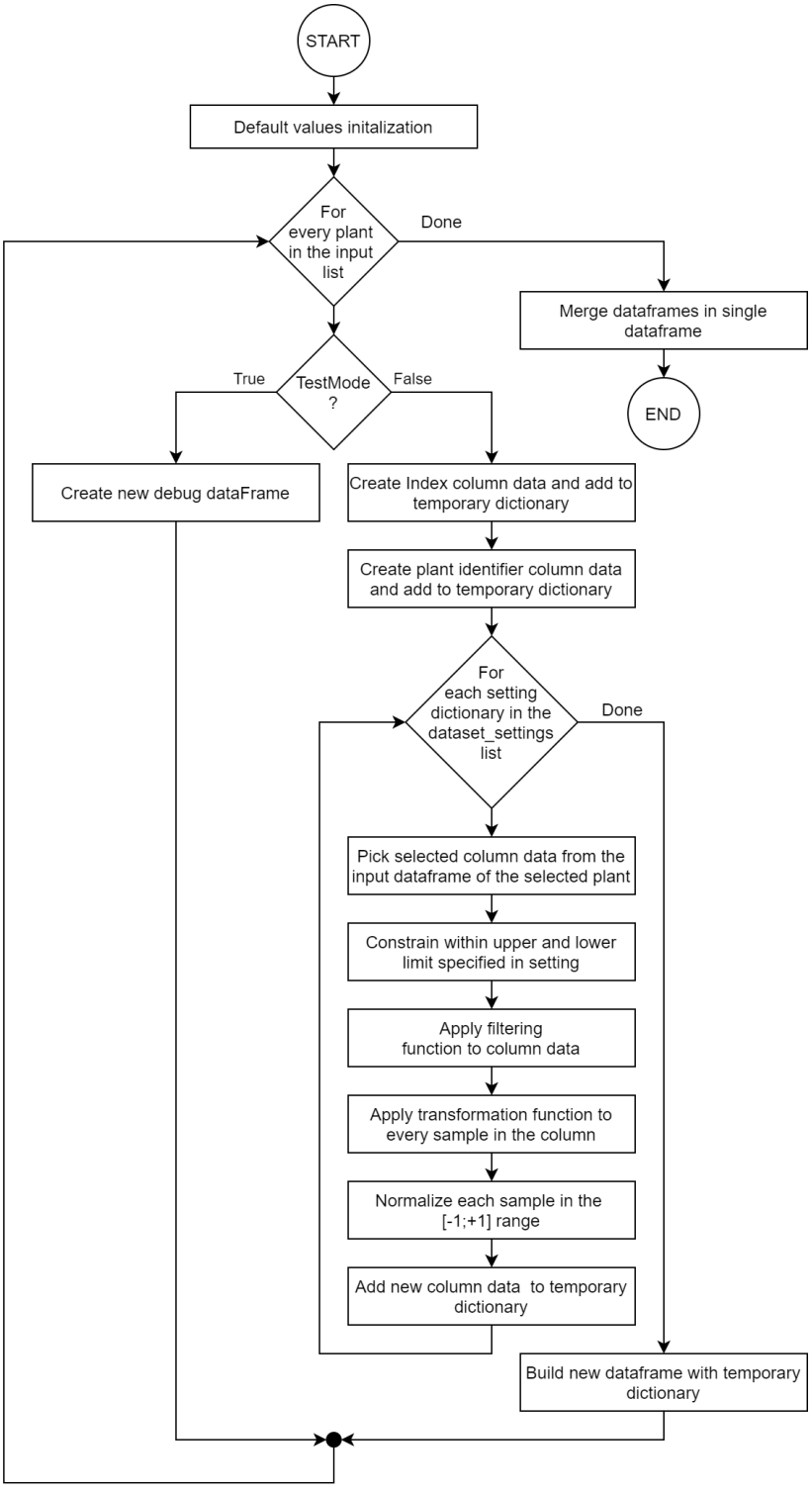


Figure 3.3: Block diagram of the *buildDataset()* function.

As soon as the function is called, it is checked if the optional parameters were passed or not by checking if they are equal to "None". If the case is the latter, these parameters are set to default values: every plant for the list of the plants to use, the earliest date for the start date, and the latest date for the end date. Next, for any input dataframe in the list that should actually be used, unless the testMode is activated, data is extracted according to the passed setting. In particular, a temporary dictionary that will be converted to a dataframe at the end is created. A "time_idx" index field is created with a sequence of integer numbers. Then a "plant" field is created with every element in the column set to the identifier of the current plant. Finally, a column is created for each setting dictionary in the "dataset_settings" list starting from a selected column in the original dataframe. The data in this column is constrained, filtered, transformed, normalized and finally added to the temporary dictionary. Once every new variable has been generated, the dictionary is converted to a dataframe, whose data is merged in another unique dataframe for all the selected plants.
Details on how these "dataset_settings" and the other parameters should be set are provided in Appendix A.
After passing all the required data and arguments to the *buildDataset()* function a dataframe with a structure simular to that shown in table 3.2 may be returned.

time_idx	**plant**	**moisture**	**hour**	**impedance_modulus**	**impedance_phase**
0	"pianta1"	-0.44256	0.21739	-1.00000	-1.00000
1	"pianta1"	-0.44262	0.30435	-0.99706	-1.00000
2	"pianta1"	-0.44733	0.39130	-0.99554	-1.00000
3	"pianta1"	-0.45208	0.47826	-0.99403	-1.00000
⋮	⋮	⋮	⋮	⋮	⋮

Table 3.2: Dataframe obtained from the *buildDataset()* function.

The dataframe created by the *buildDataset()* function contains all the data necessary for the training and the testing of the Neural Network. However, this data is not ready to be provided to the ML-based algorithm. An interface between the two is needed as the Neural Network requires as input a vector with all the data needed to make a prediction. As for a single prediction the Neural Network may need data from multiple time instants, it means that the data needed for a single prediction may be spread over multiple rows of the previous DataFrame. As such, a new function is needed to collect the data. The Neural Network can make predictions about one or more variables. To train and test the model, the correct values of these outputs variable are also needed. The data of these target variables is also present in the previous dataframe, however this output data, while it is also needed in the form of a vector, needs to be separate from the input data. A PyTorch Dataset class was employed to simplify this conversion process, as it is also helpful to load the data during training and testing by passing it to the

Dataloader class, which accepts Dataset derived classes.
The default *Dataset* class is only a template class, and multiple specific methods within it must be implemented for it to work correctly. So a *PlantsDataset* class was created as a specialization of the PyTorch *Dataset* class[20]. These methods are the *___init___()* method for the creation of the private data structure within it and the setup of other internal variables, the *___len___()* method that should return how many "samples" in the dataset are available and the *___getitem___()* method that should return the "samples" (the input and output vector couples to be used for training and testing) provided an address, in the same way items in a Python list would be addressed. The data structure that stores the data inside the class can be of any type, as the method that returns the data also has to be implemented. Two possible implementations have been considered. The first would be to employ the same dataframe obtained with the *buildDataset()* function and dynamically create the input and output vectors as they are indexed by the *___getitem___()* method. The second consists in creating two new dataframes, one for the input vectors and one for the output vectors, where each row in the dataframes corresponds to a specific prediction and all the values of the vectors are set in advance when the class is initialized with the *___init___()* method. While the first option would save memory whenever predictions take data from multiple time instants and there is a time overlap between different predictions, and the initialization would be very fast, the *___len___()* and the *___getitem___()* methods would be very slow. The second option was chosen because while the initialization takes longer and more memory might be used as the data corresponding to a specific time instant might be repeated over multiple input vectors, most of the complexity is offloaded to the *___init___()* method that is executed just once. In such a manner, the *___len___()* and *___getitem___()* can be made as simple as possible in order to optimize the training and testing loops (that call these methods many times).
Heading into finer details, the *PlantsDataset* *___init___()* method, which is called implicitly when a *PlantsDataset* object is instantiated, takes many parameters: a dataframe in the format that is returned by the *buildDataset()* function, a list of plants identifiers to select which plants to use for the Dataset, a parameters list with the names of the variables to use as inputs of the Neural Network, an outputs list with the names of the variables to use as outputs of the Neural Network and four integer parameters. The first, n_samples, is used to specify the time span for the inputs. Its minimum allowed value is 1, which means that a prediction would only be made with the latest measurements. Higher numbers mean that past data is going to be used. For example, with a sampling rate of 1 measurement every hour and n_samples set to 24, the whole day's data would be given as input to the Neural Network. The n_overlap parameter must be lower than n_samples - 1 and higher or equal to 0. It is used to set how much overlap between different samples (or predictions) should be allowed. For example, if n_samples is set to 24 and

n_overlap is set to 12, each prediction would use 12 data points already used for the previous prediction. A higher n_overlap lets *PlantsDataset* have more available samples, but many of these samples are also more likely to be similar to each other. Finally, the "amount" and "amount_start" parameters can be used to take only part of the data so that two Datasets can be created, one for training and one for testing. Both variables must have values between 0 and 1 and have a "percentage" meaning, where 0 means 0% and 1 means 100%. An "amount" parameter set to 0.2, for example, means that 20% of the available data should be used for that Dataset. Meanwhile, an "amount_start" parameter equal to 0.8 means that only the data that comes after the first 80% should be used. So, with this combination of parameters, the last fifth of the data would be used for this Dataset. This *___init___()* method, as mentioned, is executed only on object construction, so it does not have to be called explicitly, and the parameters have to be passed to the class when it is instantiated.
Table 3.3 shows how the dataframe shown in the previous table 3.2 would be elaborated into the internal Inputs dataframe of the *PlantsDataset* class after the*___init___()* method would be called with that Dataframe being passed as parameter along with an n_samples equal to 2 and a n_overlap equal to 1. The moisture is not present as in this example it would be considered as an output. The internal Outputs Dataframe originating from the same input data is provided as an example in table 3.4.

hour0	**hour1**	**imp_mod0**	**imp_mod1**	**imp_pha0**	**imp_pha1**
0.21739	0.30435	-1.00000	-0.99706	-1.00000	-1.00000
0.30435	0.39130	-0.99706	-0.99554	-1.00000	-1.00000
0.39130	0.47826	-0.99554	-0.99403	-1.00000	-1.00000
⋮	⋮	⋮	⋮	⋮	⋮

Table 3.3: Example of the internal Inputs Dataframe of the *PlantsDataset* class.

moisture
-0.44262
-0.44733
-0.45208
⋮

Table 3.4: Example of the internal Outputs Dataframe of the *PlantsDataset* class.

Figure 3.4 shows the block diagram of the *___init___()* method. A few simplifications have been made with respect to the actual code to make the visualization clearer, but the general flow is representative of the real software. The tables on the left show an example of the data as the class constructor processes it, and the numbers associate each table to a certain portion of the algorithm. The data of

tables associated with numbers placed beside a block is produced after the block is executed.

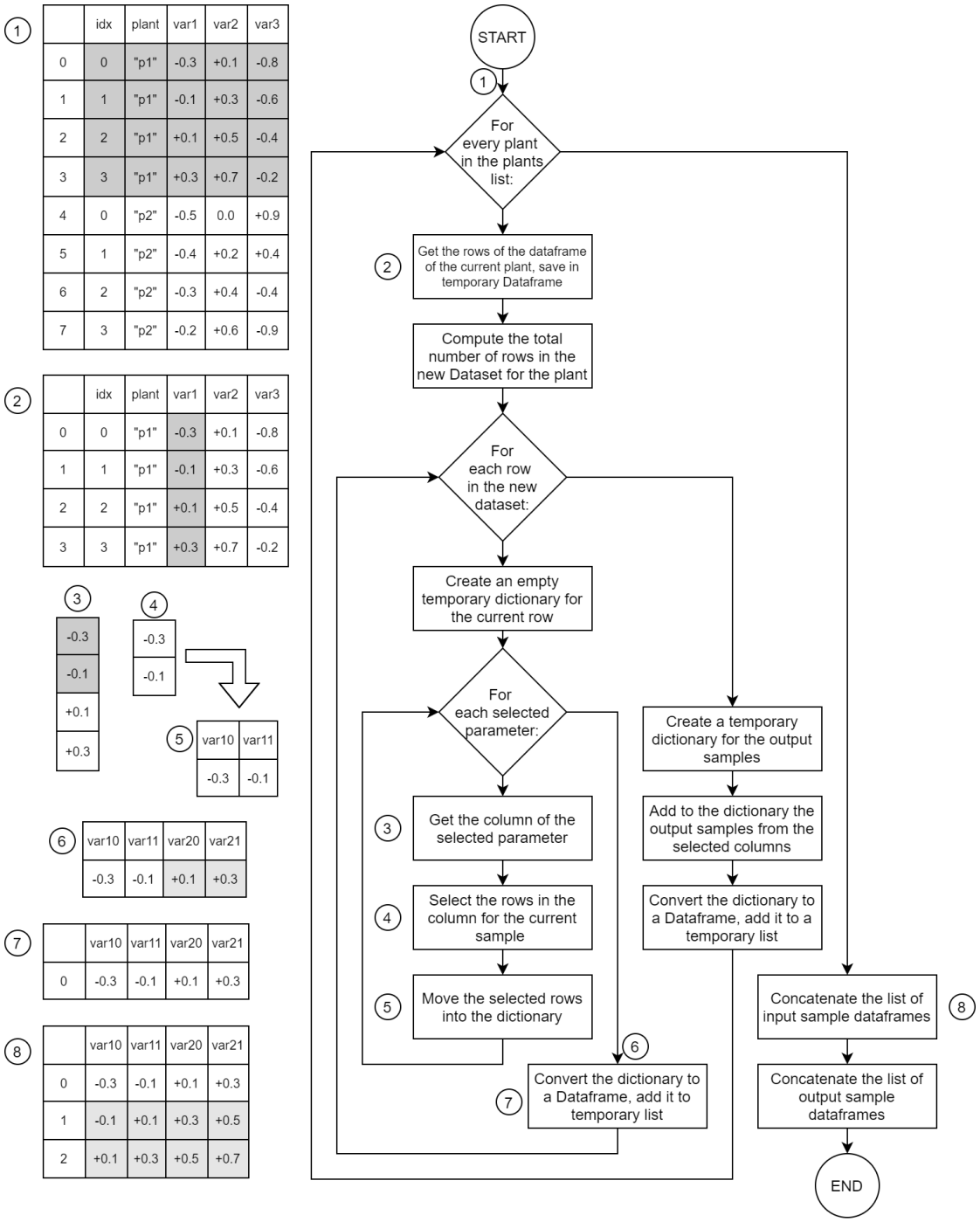


Figure 3.4: Block diagram of the *PlantsDataset* ___*init*___*()* method.

The *___len___()* method is called whenever the *len()* function is applied to the *PlantsDataset* object. Its implementation is very simple and simply returns the number of rows of the internal inputs Dataframe.
Finally, the *___getitem___()* method is called when the object is getting indexed with the [] syntax, by enumeration, or with the for-in construct. The *___getitem___()* method requires one input parameter, which represents the index or the indexes, as it can be a single integer, a list of integers, or a PyTorch Tensor with dimension 1. The method "translates" the indexing so that the data in the internal dataframes is accessed, converted into a NumPy array of 32-bit floating-point numbers, and returned as a two-element Tuple, where the first element is the NumPy array of the inputs and the second is a NumPy array for the outputs. A NumPy array is needed as the data can be unidimensional if a single sample (or row) is being accessed or 2-dimensional if multiple rows are being accessed.
This concludes the presentation of the data processing section, as the PlantDatasets created with the presented methodology would then be provided as input to another type of class, the PyTorch DataLoader, whose usage does not depend on the data itself but on the type of training (with or without K-Fold).

3.3 Machine Learning "foundation"

As the Datasets for training and testing have been obtained, they now can be utilized for the actual Machine Learning portion of the software. As already mentioned before, the PyTorch[21] library for Python has been used as the machine learning framework. PyTorch is a free and open-source software developed primarily by Facebook's AI Research lab and is based on the Torch library[22]. A simplified code based on the PyTorch library of what must be accomplished to train and test the Neural Networks is shown in listing 3.3:

Listing 3.3: Example of basic ML software.

```
train_dataloader = DataLoader(train_Dataset, batch_size=train_batch_size)
test_dataloader = DataLoader(test_Dataset, batch_size=test_batch_size)
model = NeuralNetwork()
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
for t in range(epochs):
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
```

The listed code shows four classes and two functions that are needed to implement the basic Machine Learning algorithm.
These two functions are called *train_loop()* and *test_loop()* and are the routines that train the Neural Network with the training dataset and test the Neural Network with the test dataset, respectively. Both functions are repeated *epochs* times, where the *epochs* variable represents how many times the training-test process has to be

repeated. The *train_loop()* function specifically performs a training step over the *model*, which is the Neural Network, using the data of the training set provided by its dataloader, utilizing the loss function *loss_fn* and the *optimizer* to calibrate the internal parameters of the *model*. The *test_loop()* function, on the other hand, does not change the internal parameters of the *model*, but only applies the test data provided by its dataloader to check how good the model is.
The five objects that have to be passed to the two functions are the following. The *train_dataloader* and *test_dataloader*, both of the class *DataLoader* from the PyTorch library, are classes used to provide the data of the *train_Dataset* and *test_Dataset* respectively to the functions in batches. The size of the batches has to be specified to the *DataLoader*s. The model is a *NeuralNetwork* object in this case, which should be derived from the Module class that can be found in the PyTorch library too. The loss function (or cost function) *loss_fn* is also a class from the PyTorch library and has to be one of the many available loss functions. In this case, the selected function was the Cross Entropy Loss. Finally, the *optimizer* is an object that should contain one of the possible optimization algorithms, like the gradient descent. In the example, the Stochastic Gradient Descent optimizer from the PyTorch library is used. Different optimizers may take as input different parameters (for example, the learning rate is a possible parameter of the SGD). However, one of the parameters that must always be passed is the Tensor corresponding to the model's parameters so that they can be optimized (as they are passed by reference). All the ML-related classes that have been implemented under this thesis work (*NeuralNetwork* and *PlantsDataset*) as well as the *train_loop()* and *test_loop()* functions have been implemented in the *status_now_classes.py* source file. Each component will now be described, and how they are linked to each other will be explained.

3.3.1 Dataloader

The *Dataloader*[23] is a PyTorch class that, as briefly mentioned above, is used to load the data from the *Dataset*[20] and create an iterable that divides the data into different batches so that, for example, the gradient descent does not have to be computed over the whole dataset as it would be too computationally intensive. The batch size, which is the number of samples in the dataset to be provided in a batch, can be set by setting the *batch_size* parameter to the chosen amount. It has a default value of just 1, so it is beneficial to increase it unless the model is remarkably complex. It also has many other parameters that can be passed when the Dataloader is constructed. One of the most important ones, which has been employed for the framework, is the *sampler* parameter.
This sampler parameter should be set to one of the many sampler objects whose parent class is the *Sampler*[24], once again from the PyTorch library. The *sampler*,

as specified in the documentation, "defines the strategy to draw samples from the dataset"[23].
While it is possible to implement a specialization of the *Sampler* class, the PyTorch library provides many different types of samplers:

- *SequentialSampler*: Samples elements sequentially, always in the same order.
- *RandomSampler*: Samples elements randomly.
- *SubsetRandomSampler*: Samples elements randomly from a given list of indices.
- *WeightedRandomSampler*: Samples elements from [0,..,len(weights)-1] with given probabilities (weights).
- *BatchSampler*: Wraps another sampler to yield a mini-batch of indices.
- *DistributedSampler*: Sampler that restricts data loading to a subset of the dataset.

The sampler that has been used for the framework is the *SubsetRandomSampler*, so the focus is going to be placed on it, but more information about the other samplers can be found in the reference of these classes[24]. The example code provided in listing 3.3 doesn't show the usage of any sampler, so default *SequentialSampler* would be used.
The *SubsetRandomSampler*, as mentioned above, samples elements from the Dataset randomly given a given list of indices. So, provided a list of indices the sampler can pick from, it will create the batches by picking from the dataset the samples with the corresponding indexes. Figure 3.5 graphically explains this concept. Step 1 represents the full initial Dataset indexes. Step 2 represents the division into two datasets, one to be used for training (the one on the left) and one for testing (the one on the right). Step 3 shows the randomization of the indexes, which is useful so that during training, the whole dataset is shuffled. As we are training with batches, this is even more important as otherwise "similar" samples close in time would all be in sequence and in the same batch. Finally, Step number 4, divided into three substeps, shows the creation of 3 batches of two indexes each. As evident, the shuffling helps distribute the samples over all the batches, thus making the batches more variegate.

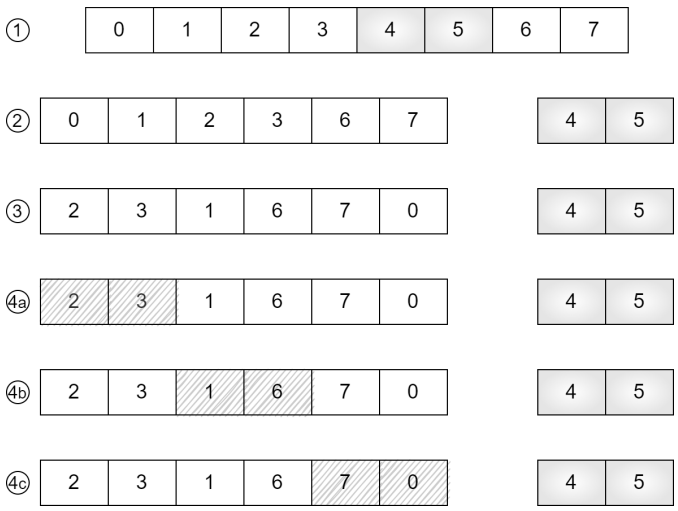


Figure 3.5: Concept behind the *SubsetRandomSampler*.

With the *SubsetRandomSampler*, all that has to be done is compute in advance the indexes valid for training (the list on the left at step 2) and pass them to it when the sampler is instantiated. Then all that is left to be done with it is to pass it to the *Dataloader sampler* parameter.
Returning to the *Dataloader* subject, it is important to explain its actual usage. The dataloader, once created, is at all effects an iterable object whose content (the batches) can be accessed with the usual indexing method of using the square brackets or using a for-in construct to extract every batch. Each batch extracted by the for-if construct would be a tuple, with the first element being a NumPy array for the inputs (the rows being the multiple samples of the batch) and the second element being another NumPy array but this time for the outputs (the rows being the correct outputs given the corresponding input samples). It allows dividing the given training Dataset into multiple batches according to the chosen sampler and batch size. The exact usage of the *Dataloader* will be presented in the Train Loop and Test Loop sections.

3.3.2 Neural Network model

Next in the sample listing 3.3 is the creation of the *model* object by instantiating a *NeuralNetwork* class. This class has been implemented as a specialization of the PyTorch *Module*[25] class, which should be the "base class for all neural network modules" to take advantage of the PyTorch functionalities. The PyTorch *Module* provides an extensive set of already implemented methods used "in the background" to train the model, such as utilities to compute the gradients or utilities to move the data to the GPU for faster computation, which are already debugged and ready

to use. However the *Module* does not fully implement two fundamental methods that are left to be implemented by the developer so that the parent class can be fully general. These methods are the *___init___()* method, which is the object constructor, and the *forward()* method, which defines how the computation of the outputs should be performed given an input provided in a format of our choice. The "default" implementation of a NeuralNetwork class as presentented in the official documentation is the one of listing 3.4.

Listing 3.4: Example of a basic NeuralNetwork class.

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )
    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Both methods can be implented as the developer wishes, however in the *___init___()* method the *super(NeuralNetwork, self).___init___()* function should always be present. The *super()* built in function temparily instantiates the parent class so that its methods can be used, and in this case it is needed to call the already implemented *___init___()* method of the *Module* class that creates and initializes all the internal variables required for the ther functionalities. Moving to the other code, two main variables are created in the *___init___()* method that will then be used in the *forward()* method, the *flatten* and the *linear_relu_stack* objects.
The most important one is the *linear_relu_stack* object, and it is used to define the topology of the Neural network. In this example, its name describes the Neural Network as it is composed of linear layers with ReLU activation functions. However, it is absolutely possible to create much more complex networks by using any of the layers and activation functions provided by the PyTorch library. All that has to be done to create this Neural Network stack is to create a *Sequential*[26] object and pass to it as parameters all the Neural Network layers and activation functions in a structured sequence. As described in the documentation, the *Sequential* class acts as a "sequential container", so it basically creates a sequence of layers. The layers sequence, as said before, is to be passed to the object in order from input to output, making sure that the dimensions of the connections of the layers match. In

the example provided above, the *Sequential* object is used to create a stack of 784 inputs for the input layer, 512 normal neurons with a ReLU activation function for the first hidden layer, other 512 normal neurons with a ReLU activation function for the second hidden layer, 10 normal neurons with their default linear activation for the output layer. This is all that is needed to define the internal structure of the Neural Network.
The other one, *flatten*, is set to the *Flatten()*[27] object of the PyTorch library. It is basically used to remove unneeded dimensions of the passed input when the module is tasked with making a prediction by calling the *Module* instance (more about this will be added next). For example, taking advantage of the listing 3.4, the module is designed to take in input monochromatic pictures of size 28 by 28 pixels. Inside the module, the first layer is hardcoded to take 28*28 inputs, so 784 inputs on a single dimension. However, the images in the dataset may be saved as a 2D array, so it is necessary to flatten the array when the dataset data is provided to the model to adapt the array sizes. As what occurs in the background basically is a series of matrix multiplications, this all resorts to basic linear algebra. Returning to the example, the various layers can be represented by matrices, where the first one has size 784×512, the second 512×512, and the third 512×10. Training data is passed in batches. For example, with a batch size of 32 samples, the array passed to the module has size $32 \times 28 \times 28$. After providing this batch to the model, we would expect an array with size 32×10. However, if we try to check if the series of matrix multiplications works, we may discover that it does not as the last dimension of the batch and the first dimension of the neural network do not match. However, if the *Flatten* function is applied, the input batch size is flattened to 32×784, thus making the whole multiplication process work.
The other method in the *NeuralNetwork* class, *forward()*, is simply to be used to describe how the variables created in the *___init___()* method are to be used when passing some inputs. The code is quite self-explanatory, as what has to be done is to create the relation between the provided input(s) and the output(s). As such, what has been implemented in the example (and also what has been implemented in the framework) is just applying the *flatten* function to the inputs and then applying the model to the flattened inputs.
As a side note, in order to make the framework more versatile, the standard structure of the *NeuralNetwork* *___init___()* method was changed so that the *Sequential* object is not created within the method with a fixed topology, but it is passed a parameter to the *___init___()* method itself, which just copies it. This lets the user define a new Neural Network structure without having to change the Neural Network class code.
Before moving on to the next subject, it is worth mentioning the "components" that PyTorch makes available and that can be used in the *Sequential* class when

creating a Neural Network[28]. More details about each type of layer and non-linear activation listed in tables 3.5, 3.6, and 3.7 con be found in the PyTorch documentation.

Non-linear Activations (weighted sum, nonlinearity)	**Non-linear Activations (other)**
ELU	Softmin
Hardshrink	Softmax
Hardsigmoid	Softmax2d
Hardtanh	LogSoftmax
Hardswish	AdaptiveLogSoftmaxWithLoss
LeakyReLU	
LogSigmoid	
MultiheadAttention	
PReLU	
ReLU	
ReLU6	
RReLU	
SELU	
CELU	
GELU	
Sigmoid	
SiLU	
Mish	
Softplus	
Softshrink	
Softsign	
Tanh	
Tanhshrink	
Threshold	
GLU	

Table 3.5: Available Non-linear Activations from the PyTorch library.

Linear Layers	**Recurrent Layers**	**Dropout Layers**	**Transformer Layers**	**Sparse Layers**
Identity	RNNBase	Dropout	Transformer	Embedding
Linear	RNN	Dropout2d	TransformerEncoder	EmbeddingBag
Bilinear	LSTM	Dropout3d	TransformerDecoder	
LazyLinear	GRU	AlphaDropout	TransformerEncoderLayer	
	RNNCell	FeatureAlphaDropout	TransformerDecoderLayer	
	LSTMCell			
	GRUCell			

Table 3.6: Available Linear Layers, Recurrent Layers, Dropout Layers, Transformer Layers, and Sparse Layers from the PyTorch library.

Normalization Layers	Padding Layers	Pooling layers	Convolution Layers
BatchNorm1d	ReflectionPad1d	MaxPool1d	Conv1d
BatchNorm2d	ReflectionPad2d	MaxPool2d	Conv2d
BatchNorm3d	ReflectionPad3d	MaxPool3d	Conv3d
LazyBatchNorm1d	ReplicationPad1d	MaxUnpool1d	ConvTranspose1d
LazyBatchNorm2d	ReplicationPad2d	MaxUnpool2d	ConvTranspose2d
LazyBatchNorm3d	ReplicationPad3d	MaxUnpool3d	ConvTranspose3d
GroupNorm	ZeroPad2d	AvgPool1d	LazyConv1d
SyncBatchNorm	ConstantPad1d	AvgPool2d	LazyConv2d
InstanceNorm1d	ConstantPad2d	AvgPool3d	LazyConv3d
InstanceNorm2d	ConstantPad3d	FractionalMaxPool2d	LazyConvTranspose1d
InstanceNorm3d		FractionalMaxPool3d	LazyConvTranspose2d
LazyInstanceNorm1d		LPPool1d	LazyConvTranspose3d
LazyInstanceNorm2d		LPPool2d	Unfold
LazyInstanceNorm3d		AdaptiveMaxPool1d	Fold
LayerNorm		AdaptiveMaxPool2d	
LocalResponseNorm		AdaptiveMaxPool3d	
		AdaptiveAvgPool1d	
		AdaptiveAvgPool2d	
		AdaptiveAvgPool3d	

Table 3.7: Available Normalization Layers, Padding Layers, Pooling layers, and Convolution Layers from the PyTorch library.

3.3.3 Loss Function

Returning to the sample code of listing 3.3, after the *model* definition the loss function variable named *loss_fn*, which in that particular case is set to the *CrossEntropyLoss()* object. As mentioned in the second chapter, the loss function, or cost function, is a mathematic function used to compute how well the model can fit the data. So it is possible, by monitoring the cost function, to check if the training process is improving the model or not. Under the PyTorch framework, however, the Loss object is also used to perform more complex operations. The most important of these operations is that it takes an important part in the calculation of the gradient that used for the optimization of the model's parameters.
To use the loss object, the only thing that is needed to do is to pass the outputs of the model and the actual expected results to it: *loss_fn(pred_out, real_out)*. By doing this, the function returns a loss tensor, which has many properties associated with it. The most basic functionality is to call the *item()* method of the tensor in order to obtain the partial value of the cost function (relative to the current batch). Summing all the "items" over the various batches makes it possible to obtain the total loss function or cost function. Another functionality that is much more important is to compute the gradients for the backpropagation process relative to the training of the Neural Nets by calling the *backward()* method of the tensor. Even though there is no explicit link between this obtained tensor and the model parameters, the PyTorch framework takes care of it with an underlying graph that connects all the various tensors when they are generated. In particular, when a

prediction is made by passing an input batch to the model, the returned prediction tensor keeps a link to the model parameters by updating the underlying graph and creating a reference to them. Then the loss tensor is created when the loss function is called, and both the predictions and the correct results are passed to it. This tensor is now the leaf of the underlying graph, which keeps the link to the model, thus maintaining the reference. Finally, when the *backward()* method of the tensor is invoked, it computes all the gradients of the tensors in the graph that require it (a flag marks the tensors that require the gradient calculation) with respect to the loss tensor. So this backpropagation process updates the gradients of the model parameters with respect to the cost function, thus allowing an optimization algorithm, which will be described next, to fine-tune the model itself. Figure 3.6 shows in a graphical form the process described above.

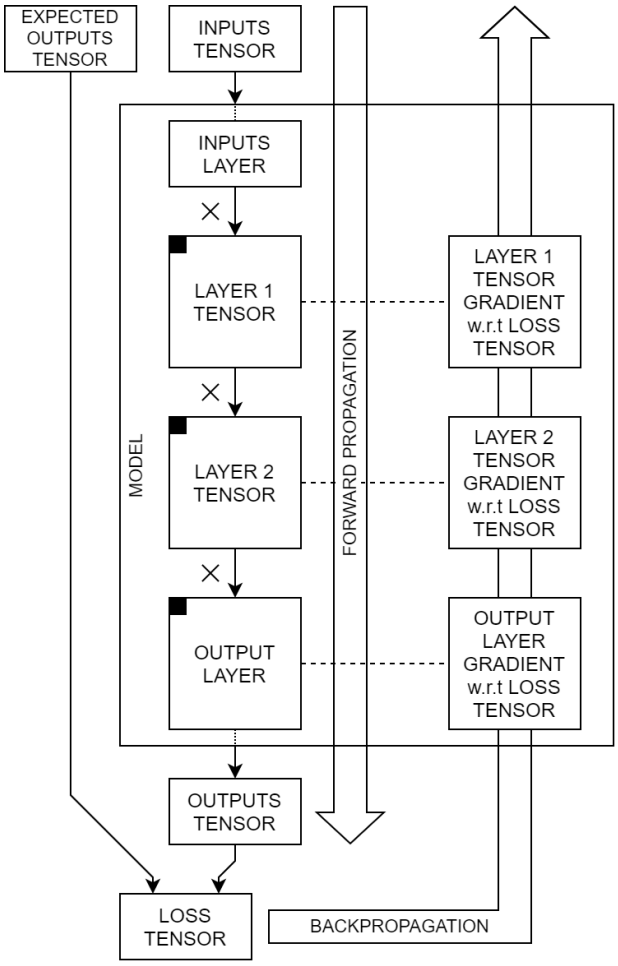


Figure 3.6: Forward propagation and backpropagation.

Following in table 3.8 is a list of all the loss functions available in the PyTorch library[28]. More information about each function can be found in the documentation.

Loss Functions
L1Loss
MSELoss
CrossEntropyLoss
CTCLoss
NLLLoss
PoissonNLLLoss
GaussianNLLLoss
KLDivLoss
BCELoss
BCEWithLogitsLoss
MarginRankingLoss
HingeEmbeddingLoss
MultiLabelMarginLoss
HuberLoss
SmoothL1Loss
SoftMarginLoss
MultiLabelSoftMarginLoss
CosineEmbeddingLoss
MultiMarginLoss
TripletMarginLoss
TripletMarginWithDistanceLoss

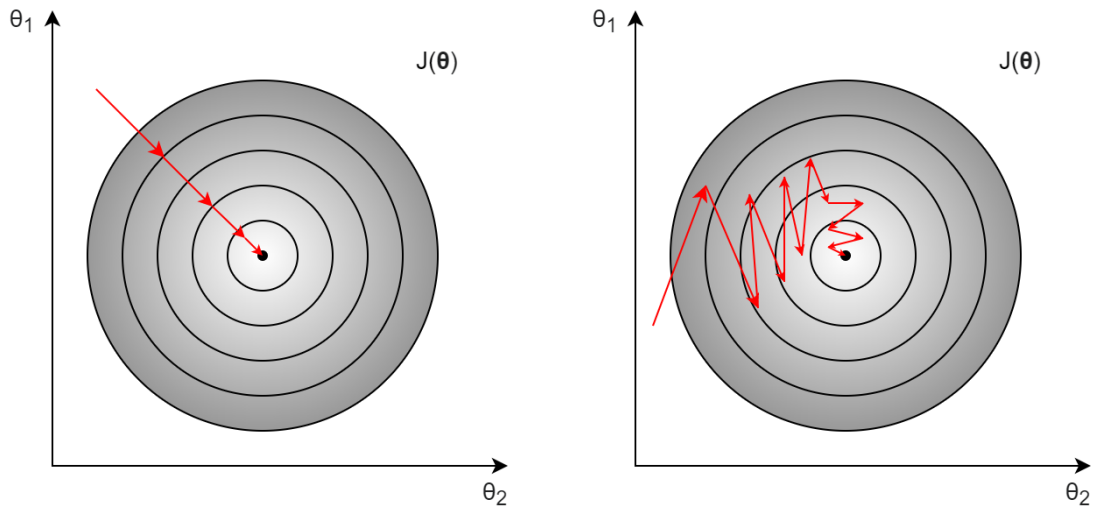
Table 3.8: Available Loss functions from the PyTorch library.

On a final note, while in the sample code the chosen loss function was the CrossEntropyLoss, in the framework the user can select a loss function of choice among those available that are compatible.

3.3.4 Optimizer

The last object in the sample code of listing 3.3 is the *optimizer*. As the name suggests, the purpose of this object is to optimize the model parameters. For this reason, it is provided only to the *train_loop* function as it is not needed during testing. Many types of optimizers with each a different underlying optimization algorithm are available. However, all of them have in common the first parameter to be passed to them when the *optimizer* object is created, which is the model's parameters. As shown in the sample code, to obtain the model's parameters, it is only needed to call the *parameters()* method of the *model* object. So, by passing to the optimizer *model.parameters()*, an iterator is going to be handed out to the optimizer so that it will be able to sweep through all the tensors relative to the model's parameters.
To actually perform the optimization of the model's parameters two methods are provided with the optimizers, *zero_grad()* and *step()*. The *zero_grad()* function,

as the name suggests, sets to zero all the gradients of the tensors that have to be optimized. Using this method by calling *optimizer.zero_grad()* is necessary before computing the gradients with *loss.backward()* as explained before for the correct operation of the optimizer. After the gradients have been cleared and updated, the *step()* method can be used by using the *optimizer.step()* method call to execute an optimization step. While the optimizers always take advantage of the gradients calculated with respect to the loss function, the actual behavior of the optimizer depends on the chosen algorithm. However, after the optimization step is concluded, it is expected that the parameters of the model will be fine-tuned if the correct settings have been employed. It is important to remark that these optimizers do not guarantee the best solution, especially if the cost function has multiple local minimums. With wrong settings, such as too high learning rates, it is actually probable that the optimizers will make the model worse.
In the scope of the framework that has been developed, the Stochastic Gradient Descent[29] algorithm provided by PyTorch has been employed. The stochastic version of the gradient descent is necessary when training occurs over multiple batches. As in any single epoch the optimization occurs as many times as how many batches are available, and each optimization step is computed considering the gradients relative to a small subset of the training set, it becomes obvious why it is called stochastic, which is a synonym for casual, aleatory. Figure 3.7 shows this concept in a graphical form.



(a) Normal Gradient Descent. **(b)** Stochastic Gradient Descent.

Figure 3.7: Different implementations of the Gradient Descent algorithm, the Stochastic Gradient Descent algorithm is necessary with multiple batches.

As it can be seen, when the optimization occurs over the whole training dataset (Normal Gradient Descent) the descent takes the shortest path to the local minimum. However, if the optimization steps do not consider the whole training dataset but only the data of a batch, the descent may be biased towards a specific subset of the dataset. This causes the cost function to "tumble" towards the minimum, but not by taking the shortest path. With the SGD, it is not guaranteed to find the local minimum in a reasonable amount of time, but it can be said that statistically, it is likely to occur if the data in the batches is well shuffled.
The basic algorithm behind the SGD is the following:

$$\begin{cases} v_{t+1} = \mu \cdot v_t + g_{t+1} \\ p_{t+1} = p_t - lr \cdot v_{t+1} \end{cases} \tag{3.1}$$

where p, g, v, μ and lr denote the parameter, gradient relative to the parameter, velocity, momentum, and learning rate respectively. The velocity and the gradient are unique for each parameter. As it can be seen, for every update of each paramter there is a dependance on the previous update via the velocity variable.
By setting the momentum μ to zero, the normal formula with just the learning rate and the gradient can be obtained:

$$p_{t+1} = p_t - lr \cdot g_{t+1} \tag{3.2}$$

In the framework, although it was not made possible to change the optimizer at will, parameters like the learning rate and the momentum of the algorithm were left to the user to set. The Stochastic Gradient Descent algorithm is definitely one of the most used optimizer algorithms in the machine learning field, however, many other optimizers are available. Table 3.9 lists the optimizers made available by PyTorch. More information about each optimizer can be found in the PyTorch documentation[30].

Optimizer algorithms
Adadelta
Adagrad
Adam
AdamW
SparseAdam
Adamax
ASGD
LBFGS
NAdam
RAdam
RMSprop
Rprop
D

Table 3.9: Available NN Optimizer algorithms from the PyTorch library.

3.3.5 Train Loop and Test Loop

The final portion of code in the example code provided in listing 3.3 consists in the iteration of the *train_loop()* and *test_loop()* functions by employing a for loop. The *epochs* variable defines how many times the two functions are executed. The two functions share most of their structure and even most of their code. As a matter of fact, all the code that is present in the *test_loop()* function is also present in the *train_loop()* function with some additional instructions. The idea behind these functions is to have two modular routines where one can perform an optimization step on the model while providing statistics relative to the error and the cost function. At the same time, the other can test the model and produce the same statistics over a different dataset, given a training and a test dataloader respectively, the model, a cost function, and an optimizer.
The *train_loop()* function, of which a block diagram can be found in figure 3.8, takes as input a *dataloader*, the *model*, the *loss_fn* loss function, and the *optimizer*. The first thing that is done is to check whether the math related to the tensors (mostly matrix multiplications) can be performed on the GPU for better performance. If a supported GPU with CUDA is available, the *device* to be used is going to be set to 'cuda' otherwise the 'cpu' will be used. The *device* will be used after to move the data to the correct device. Then all the variables associated with the statistics that have to be returned are initialized to zero. After this initial step, a for loop is executed, and it is used to extract the batches from the dataloader and perform the model optimizations with them. For every batch, the inputs tensor and the expected outputs tensor are provided. Then all the steps related to forward propagation (prediction computation, loss tensor computation) and backward propagation (Reset of the gradients of the tensors to be optimized, computation of the gradients of those tensors, via backpropagation, optimization of the model's parameters) are executed each time. Along with these instructions, the statistics in the form of an average loss and RMSE are partially computed. The loss function and the squared error related to each sample are summed to their respective variable. Only after the for loop has finished, the computation of the statistics is completed by performing the necessary averages, and they are returned. The model does not have to be returned as it was passed by reference, so the original one was actually modified as it was not a copy.
The *test_loop()* function, as said before, is pretty much identical to *train_loop()* and the main difference is that it lacks the instructions related to the backpropagation and optimization. Anyways, its block diagram can be found in figure 3.8.

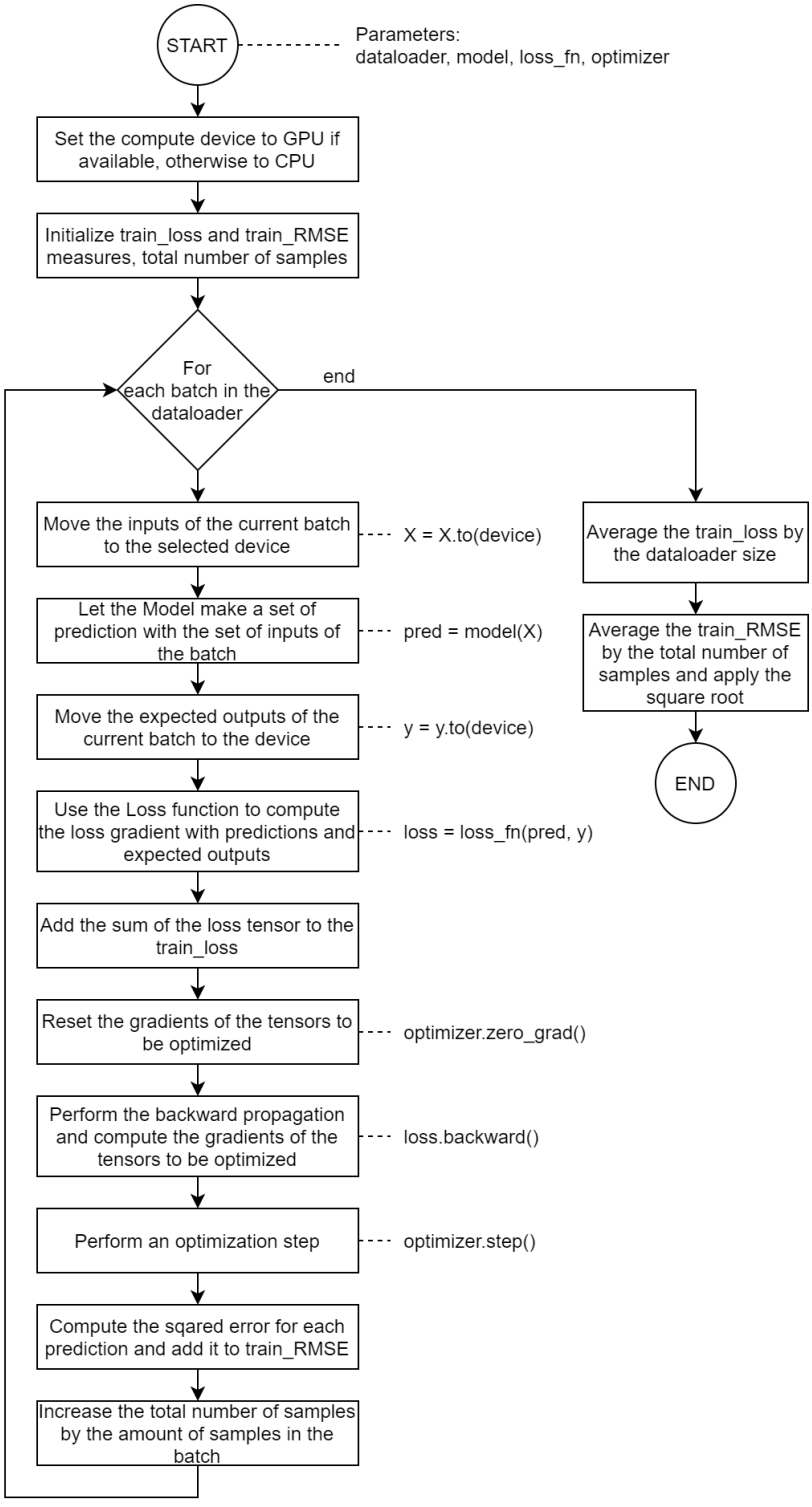


Figure 3.8: Block diagram of the *train_loop()* function.

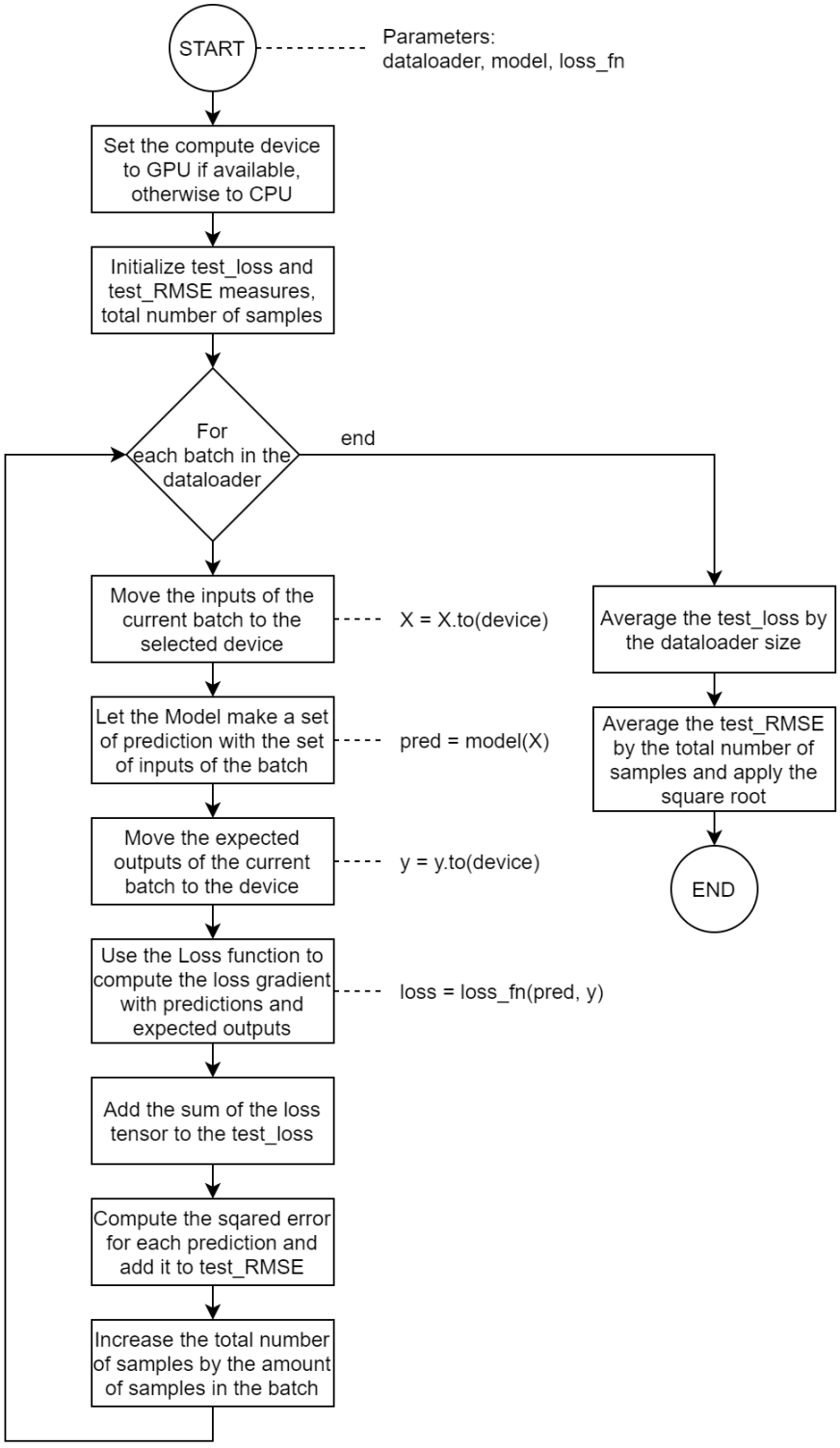


Figure 3.9: Block diagram of the *test_loop()* function.

3.4 Status Now: prediction of the current plant status software implementation

Status Now (or *status_now*) is one of the core functionalities of the framework. The Status Now functionality is implemented in the *status_now.py* source file and can act both as a main executable script or as a module so that its functionalities can be taken advantage of by other scripts. In particular, it contains three implemented functions:

- *main()*:

  Function used when the *status_now.py* script is being run as the main file and not as a module. The function imports the CSV files with the data relative to the plants and also imports the settings from the *status_now_settings.py* file. Then it selects the specific *Setting* object as well as the training type (with or without K-Fold cross-validation) by selecting one of the two functions described below. The selection of the setting and the training type is made by changing the code in the *main()* function. More information about the *Setting* object will be provided later.

- *statusTrain()*

  Function used to train a Neural Network given the settings and the data. It will perform all the data processing and training. It returns the training results with a *ModelData* object for the trained model at the last epoch and at the epoch with the lowest overall or test loss, as well as a picture of the evolution of the errors and the actual model at that time. The ModelData() object contains error data both in numerical and graphical form. The following parameters are needed to execute the function:

  – *plant_df*: List of dataframes of the parsed CSV files.
  – *plant_list*: List of plant identifiers respective to the dataframes.
  – *chosen_setting*: *Setting* object.
  – *batchMode*: Boolean to enable or disable showing plots on screen.

- *statusTrainKFold()*

  Function very similar in function to *statusTrain()* but the training occures using the K-Fold cross validation technique. This function returns the results with the same format as the *statusTrain()* function. The parameters that have to be passed to the function are also the same as *statusTrain()*, which are listed above.

- *statusNowPrint()*

  Function used to save the results produced by either the *statusTrain()* or the *statusTrainKFold()* functions. To execute the function, the following parameters are needed:

  – *save_folder_root*: Path object used to create the save folder.
  – *chosen_setting*: *Setting* object that was used for training.
  – *final_visual_model_data*: *ModelData* object with the best overall data.
  – *best_visual_model_data*: *ModelData* object with the last epoch data.
  – *figure_validation*: *Figure* object with the evolution of the errors.
  – *test_name*: String to be used to give a unique name to the save folder.

At the end of the file, after the functions definitions, a piece of code that is always executed when the file is run is used to check if the *___main___* built-in global variable contains the *"___main___"* string. If that is true, then it means that the file has not been imported as a module and that the settings chosen in the *main()* have to be used. As a consequence, the *main()* is immediatelly called. On the other hand, if the file was imported as a module the *___main___* variable will contain the name of the module (*"status_now"*) so nothing has to be executed and the only thing that should happen is the definition of the functions listed above so that they can be used.
Now more details about the *Setting* object and on the standard implementation of *statusTrain()* and the K-Fold implementation of *statusTrainKFold()* will be provided.

3.4.1 Setting class

The *Setting* class, implemented in the *status_now_classes.py* file, exists to provide a highly parametric setup of all the variables concerning the Data preparation and the Training portions of the framework. The class includes a fairly typical *___init___(self)* method called on object creation that takes no parameters and simply creates the internal private variables that will contain each one of the settings to be stored, as well as two "verification" variables that are employed to check the completeness of the Setting object as its contents are added before it is actually used. All the private variables that contain a setting to be retrieved in either the *"status_now"* or *"status_now_finder"* functionalities include a getter and a setter method (created with the @property and @<variable>.setter decorators). An example of such setter and getter methods is shown in listing 3.5, where the private variable is named *_var* and the public "virtual" variable accessible only via the getter and the setter is named *var*.

Listing 3.5: Example of setters and getters.

```
# _var getter
@property
def var(self):
    return self._var

# _var setter
@var.setter
def var(self, var):
    self._var = var
```

The private variables as well as the public getters setters that can be used to access them are listed in table 3.10.

Private variables	Public getters/setters	Initialization value
_dataset_settings	dataset_settings	[]
_start_date	start_date	""
_end_date	end_date	""
_plants_to_use_list	plants_to_use_list	[]
_remove_n	remove_n	0
_testMode	testMode	False
_params_to_use	params_to_use	[]
_outputs	outputs	[]
_n_samples_per_parameter	n_samples_per_parameter	1
_n_overlap_of_samples	n_overlap_of_samples	0
_learning_rate	learning_rate	1e-3
_momentum	momentum	0
_batch_size	batch_size	32
_model	model	None
_loss_fn	loss_fn	None
_epochs	epochs	100
_folds	folds	None
_trainWithAllData	trainWithAllData	False
_compiled	Not available	-17
_dataset_settings_keys	Not available	***

Table 3.10: All the variables of the *Setting* class.

*** The initialization value of the *_dataset_settings_keys* variable is list with all the keys listed in table 3.11: *["param_name", "input_name", "norm_data_range", "norm_data_median", "transform_function", "transform_function_kwargs", "filter", "filter_kwargs", "constrain_min", "constrain_max"]*
The *_compiled* variable is one of the two variables that are used for completeness checking and is a variable that is incremented whenever a setter is used. Every getter, prior to returning the requested variable, checks if the *_compiled* variable is <=0, and if it is, an error is printed out as most likely the setting object was not fully compiled. The _dataset_settings_keys variable, on the other hand, is used to check if the *dataset_settings* variable provided with the setter was compliant with the standard format. As mentioned in the Data preparation chapter, the *dataset_settings* list is used to convert the initial dataframes obtained by parsing

the CSV files into an initial dataframe where the variables are generated. As a reminder, starting from any column in the CSV files, it is possible to extract a new variable by taking that data and constraining it, filtering it, transforming it, and normalizing it. All these operations require a few settings defined by key-value pairs of multiple dictionaries that are contained in the *dataset_settings* list. The keys of these key-value pairs are listed in table 3.11 along with their usage.

Key name	Value content
"param_name"	Name of the new variable to be created
"input_name"	Name of the column in the CSV files
"norm_data_range"	Range of the data to be used for normalization
"norm_data_median"	Median of the data to be used for normalization
"transform_function"	Function to be applied to every data sample of the series
"transform_function_kwargs"	Arguments of the "transform_function"
"filter"	Filtering function to be applied to the time series
"filter_kwargs"	Arguments of the "filter" function
"constrain_min"	Minimum allowed value of the series
"constrain_max"	Maximum allowed value of the series

Table 3.11: All the keys of the dictionaries in the *dataset_settings* list.

The *_dataset_settings_keys* list is used to check if the dictionaries in the *dataset_settings* list contain all the needed key-value pairs. This is essential as if some pairs are missing, the first dataframe generation will fail. Other checks performed in some of the setters include checking if the passed contents make sense. For example, the passed list must contain at least one item or some of the passed integers must not be negative (such as for the n_samples_per_parameter variable).
Besides all the private variables and their setters and getters, a method called *generateModel()* has also been implemented for the *Setting* class. It was implemented to help create the Neural Network so that the user does not have to change the number of inputs or outputs any time that the settings related to those are changed. For example, the total number of inputs for the first layer of neurons depends on the number of input variables and on how many past samples are used for a new prediction. To avoid this calculation and to make these changes faster to make, the *generateModel()* can be just called with no set parameters, and a default model with the correct amount of inputs and outputs will be created. The method will automatically calculate the number of inputs and outputs from the respective lists, so those lists have to be defined already for the generation to work. Suppose, for some reason, that the user wants to generate a model before having defined those lists. In that case, the number of inputs and outputs can be provided via the *nn_inputs* and *nn_outputs* parameters. However, changing the model structure is also possible so that the user does not have to rely on the default topology. To do that, it is possible to use the *generateModel()* method passing to the *nn_stack* parameter a *Sequential* object (described previously in the Machine

Learning "foundation" section, Neural Network model subsection). The method will take care of creating the *NeuralNetwork* object. To keep the functionality of the auto-updating number of inputs and outputs of the Neural Network, it is possible to retrieve them dynamically at run time by using the *nn_inputs()* and *nn_outputs()* methods of the *Setting* class. As long as the inputs and outputs lists have been compiled, and the n_samples_per_parameter variable has been set before, those two methods will return the appropriate number of inputs and outputs that can be used when creating the *Sequential* object. Listing 3.6 provides an example of the usage of this functionality, where a Neural Network with two linear layers with ReLU activation layers and an output layer with a linear activation is being created (nn is the nn module of the PyTorch library).

Listing 3.6: Example of model generation.

```
setting.generateModel(
    nn.Sequential(
        nn.Linear(setting.nn_inputs(), 32),
        nn.ReLU(),
        nn.Linear(32, 32),
        nn.ReLU(),
        nn.Linear(32, setting.nn_outputs())
    )
)
```

Finally, the *___str___(self)* was also implemented. It is one of the built-in methods that get called whenever a conversion to string is attempted with the *str()* built-in function. This is used when saving the data and the results of a training run, and it simply creates a formatted string with all the data in the *Setting* object.

3.4.2 Standard implementation

The standard implementation of the Status Now algorithm is implemented, as mentioned before, in the *statusTrain()* function. The code strictly related to ML operations is very similar to the one presented in listing 3.3 as it involves the copy of the needed classes (instead of creating them on the spot) and the iteration of the *train_loop()* and *test_loop()* functions. Most of the additional code is related to monitoring the training efforts both in a graphical and numerical form. This is achieved by collecting the loss and error validation data during every epoch (to create a figure about their change during training) as well as collecting the error and the loss over the whole dataset, the error and the loss over the test dataset, an image about the performance of the model over the whole dataset as well as the model itself, with all this data saved both for the last epoch and at the epoch with the best overall error.
The block diagram of the *statusTrain()* function is shown in figure 3.10.

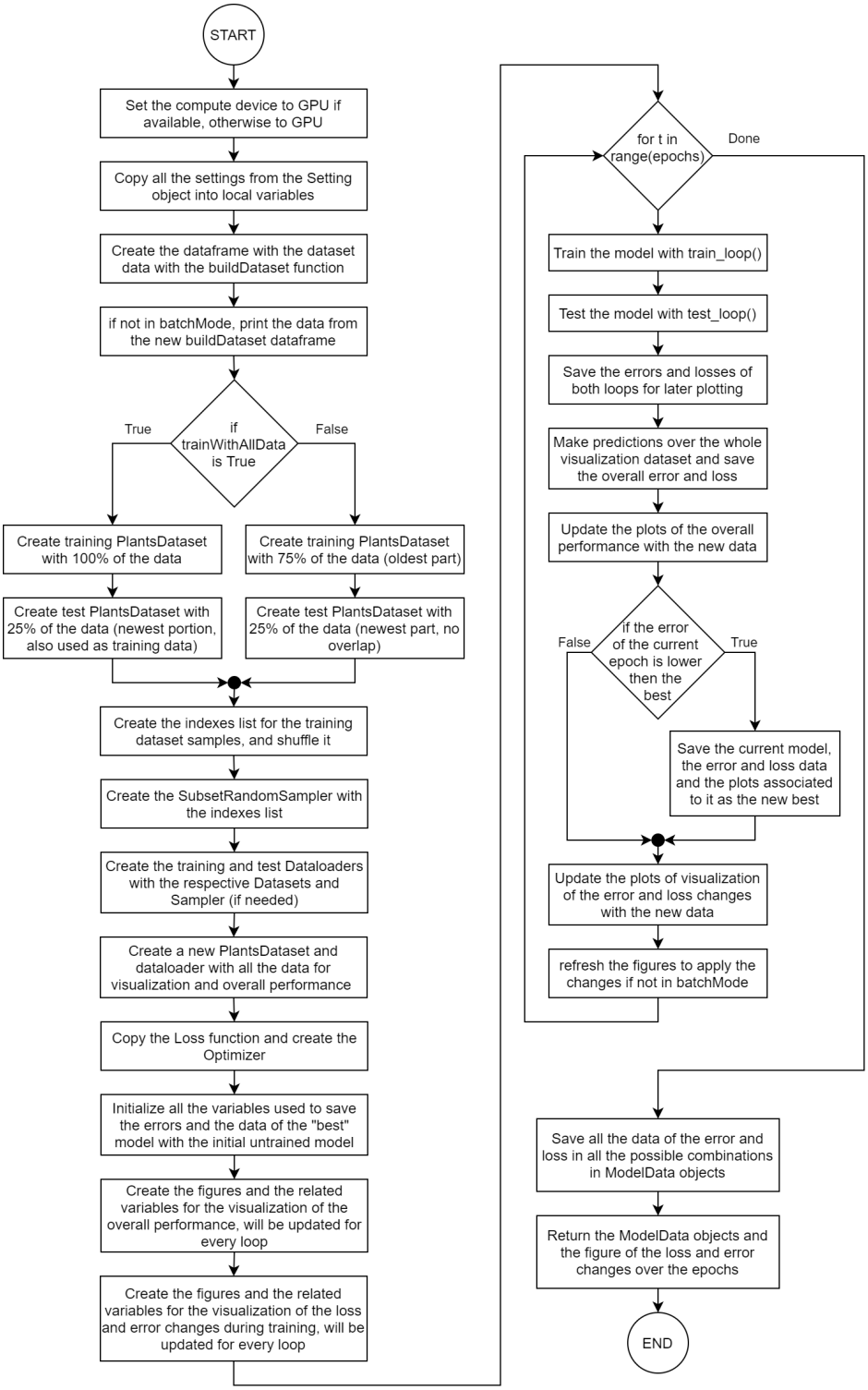


Figure 3.10: Block diagram of the *statusTrain()* function.

The figures were created during the function's execution with the help of the pyplot module of the matplotlib library. The figures mentioned before (the one about the change of the validation error and loss and the ones about the model's performance over the whole dataset) are only the ones that are returned at the end. However, some additional figures are shown before the actual training begins. These other figures show the input and the output time series created with the *buildDataset()* function. This is useful to understand if the time series that were created are correct and behave as expected. This set of figures, however, is not returned by the function and is not saved either. It is also possible to use the *batchMode* parameter to show the figures as usual in an interactive way (as it is possible to move the figures around, zoom them, and use all the pyplot functionalities while the rest of the program is being executed in parallel) by setting it to "False" or disable the functionality by setting it "True".
As mentioned before, the returned data is structured to contain the model as well as the error and loss data and visual information associated with it both at the moment of the last epoch and the epoch where the overall error was the lowest. All this data is passed with two *ModelData* objects, one for the last epoch and one for the one with the lowest error. A figure about the history of the error and loss during the various epochs is also included. The *ModelData* class is a simple container with six private variables in it, each one with setters and getters to access them. These variables, in particular, can be accessed like the following:

- *model*:

  used to store the *NeuralNetwork* object. It will be used to save the model in a *.pth* format with the *torch.save()* PyTorch function. The model can then be reopened at a later time with the *torch.load()* PyTorch function.

- *full_data_RMSE*:

  used to save the RMSE over the full dataset (training plus test datasets).

- *full_data_loss*:

  used to save the average loss over the full dataset (training plus test datasets).

- *test_data_RMSE*:

  used to save the RMSE over the test dataset.

- *test_data_loss*:

  used to save the average loss over the test dataset.

- *visual_data_figure*:

  used to save a *Figure* object of the model applied to the whole dataset (training plus test datasets). The figure represents on a plot the expected values and the values predicted by the model over the whole dataset, so that the model's performance can be evaluated in a much more practical and intuitive way other than just using error numbers.

The *ModelData* object, once instantiated, can be used as a simple container so that the functions using it don't have to return too many variables or accept too many parameters. It is also very useful to organize the resulting data, which is important to avoid accidental swaps of variables with very similar names.
Figure 3.11 shows an example of a *visual_data_figure Figure* object returned as part of a *ModelData* object, while figure 3.12 shows an example of the *figure_validation Figure* object, which represents the change of the avergage loss function and RMSE over the epochs.

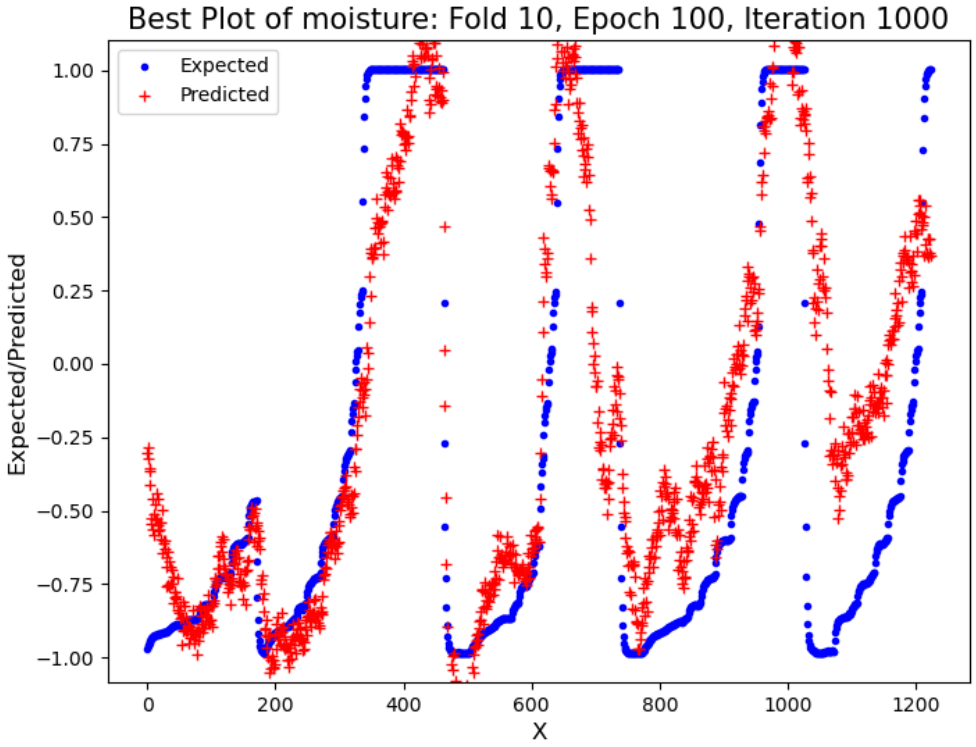


Figure 3.11: "*visual_data_figure*" example figure.

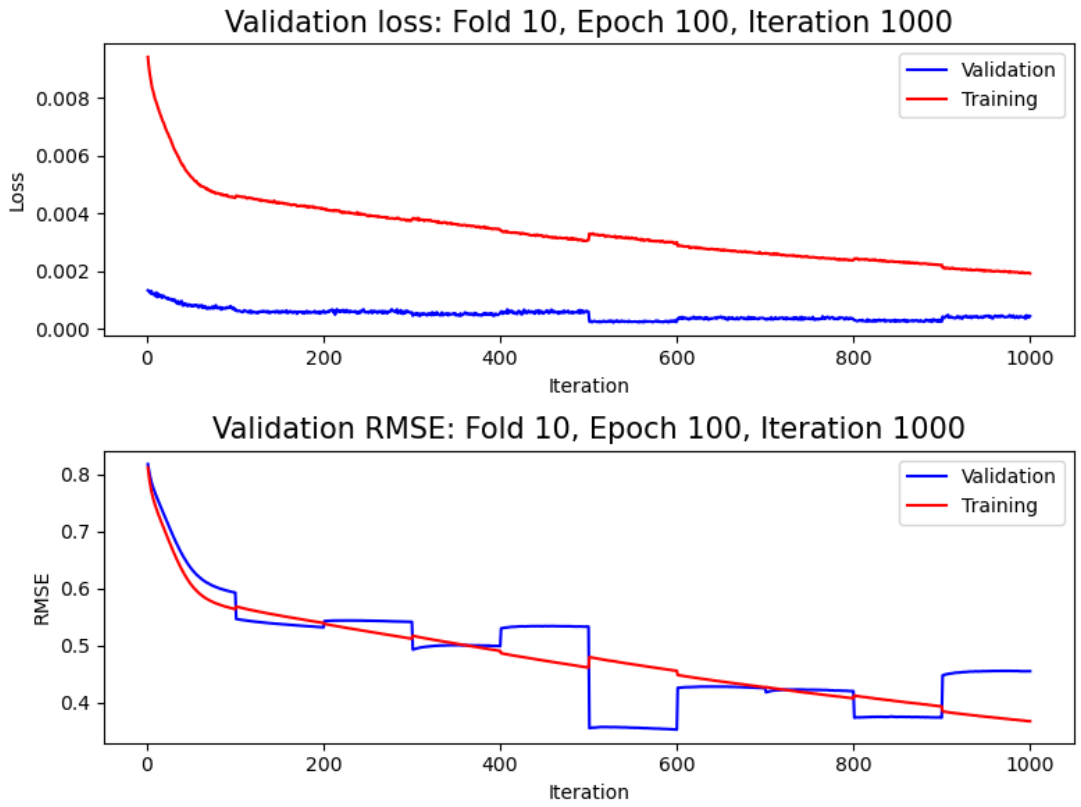


Figure 3.12: "*figure_validation*" example figure.

3.4.3 K-Fold validation implementation

The K-Fold cross-validation variant of the Status Now algorithm, implemented in the *statusTrainKFold()*, is very similar to the standard one, with only a few changes solely related to the K-Fold algorithm. The algorithm was explained in the Training process section of the Machine Learning notions chapter.
Compared to the normal implementation, the most significant change is that there is no more a single large loop where each iteration is an epoch. Now there are two nested loops, where the inner one is, as before, the one that loops through the epochs, while the outer one is used to loop through the folds of the K-Fold algorithm. As a reminder, for each fold, the data used for training and validation changes, as shown in figure 3.13, where the numbers represent the indexes of each Dataset sample.

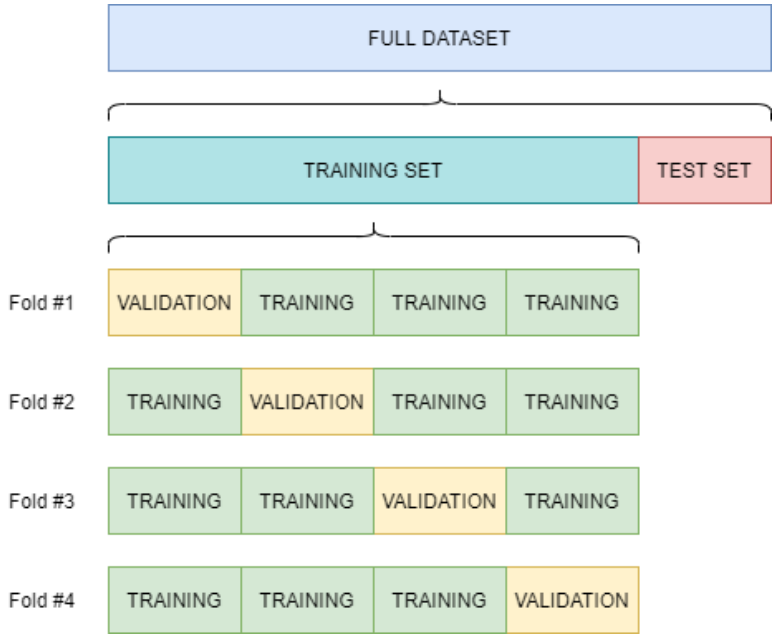


Figure 3.13: K-fold data assignment.

To accommodate this algorithm change, there are no more two *PlantsDataset*s as in the normal implementation. However, there is only a single *train_valid_data PlantsDataset* which is used for both training and validation. The data is then split for the two different usages with two different *DataLoader*s, which use the same *PlantsDataset*, the same batch size, but two different *Sampler*s. The two *Sampler*s are both of the *SubsetRandomSampler* type (see the DataLoader subsection in the Machine Learning "foundation" section of this chapter). However, they are provided with two different sets of indexes to pick from. The indexes for training and validation are updated for each fold by employing the *getKFoldIndexes()* function that returns two lists, the first for the indexes of the *PlantsDataset* samples to be used for training and the second for the indexes to be used for validation. The *getKFoldIndexes()* function takes the following input parameters:

- *fold*:

  number of the fold at the iteration when the *getKFoldIndexes()* function is called. This integer variable should change at each fold iteration, start at zero and never be larger than *total_folds*.

- *total_folds*:

  total number of folds, or the integer number of slices the training / validation Dataset should be divided into.

- *dataset_total_elements*:

  total number of training / validation samples in the Dataset. Integer number.

- *all_indexes*:

  optional parameter. list of the indexes to be used for the training and validation indexes subsets. If not provided a list with the numbers from 0 to *dataset_total_elements* - 1 is used. It is useful when more randomization of the subsets is wanted, as the indexes can be shuffled in advance so that the training and validation indexes are not contiguous numbers.

In reference to figure 3.13, the function returns a list with the indexes of the "green" training indexes and a list with the "yellow" validation indexes, given the fold number. The crux of this problem is identifying how many indexes should be in each one of those "green" and "yellow". While the problem may seem easy (and it is in cases where the total number of samples in the dataset is a multiple of the total number of folds), it is not trivial to generalize the splitting of the indexes so that the groups are always balanced and always have a similar amount of indexes. Let's take for example the indexes of figure 3.14.

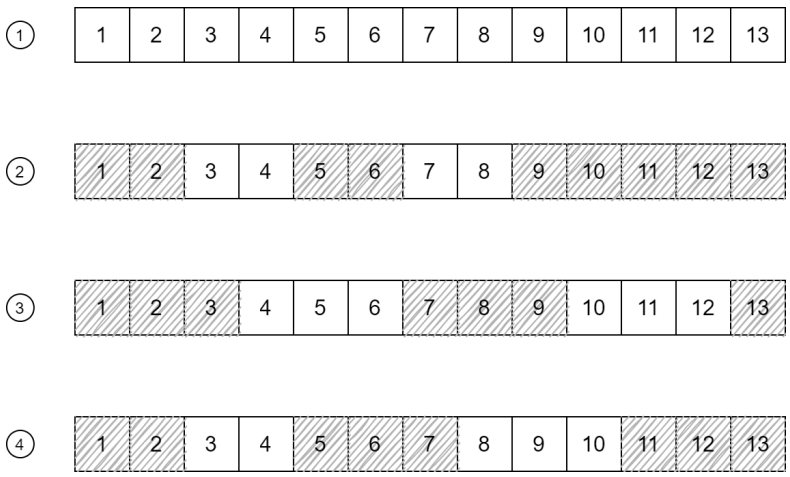


Figure 3.14: K-fold set division example.

Row number ① shows the initial list of indexes, from 1 to 13. This means that the whole Dataset (that includes both the training and the validation data) contains 13 samples. For the sake of this example, suppose that 5-fold cross-validation is selected. As 13 is not a multiple of 5, a naive approach might consist in dividing with an approximation. In the example, row ② shows a division in 5 subsets by creating all the possible subsets with size $floor(13/5) = 2$ and then adding the

remaining indexes to the last subset, however, there is a large imbalance in the subset sizes, with the last one having 5 indexes and other 4 having only 2. Row number ③, on the other hand, shows the opposite approach by creating all the possible subsets with size $ceil(13/5) = 3$, however as one might expect the last subset will have a lower amount of indexes, again with some imbalance between the subset sizes. The ideal solution consists in dividing the indexes set in subsets with a size difference of one at most. This case is shown on row ④, where it can be seen that by having 2 subsets with size 2 and 3 subsets with size 3 it is possible to achieve the best balance. In this way, once these indexes subsets are going to be used in the K-Fold cross-validation algorithm, the validation set will always have the same size with at most a difference of one. This optimal subdivision may be achieved in multiple ways, for example it is possible to take one of the sub-optimal division algorithms (either one of row ② or ③ of figure 3.14) and then move some of the indexes from the larger subset to the lower subsets (for case ②) or from the larger subsets to the smaller subset (for case ③) until the size difference is at most equal to one. However, it would be much more efficient to calculate in advance how many subsets of one size and how many of the other are needed so that the most balanced split can be performed immediately. To do this, a formula was discovered and also demonstrated, as presented next. While it is not guaranteed that this formula was already present in the literature, the search efforts could not provide any results. In the scope of the formula demonstration, the indexes that have to be subdivided into many subsets will be indicated as "elements", while the number of folds will be referred to as the number of "subsets", in order to generalize the formula.

Let n be the size of the original set (or the total number of elements) and k the number of subsets. The subsets should all contain the same amount of elements or, at most, have a size difference equal to one among all of them. Let s be the size of a subset (or how many elements that are part of the subset) and c_1 and c_2 be the quantities of subsets of two different sizes, so that $c_1 + c_2 = k$. It can be demonstrated that:

$$n = s \cdot c_1 + (s+1) \cdot c_2 \text{ with } \begin{cases} s = \left\lfloor \frac{n}{k} \right\rfloor \\ c_1 = k - (n \bmod k) \\ c_2 = n \bmod k \end{cases} \tag{3.3}$$

The mod function, or the modulo operation, represents the reminder of a division, so $x \bmod y$ should return the reminder of the division of x by y. The floor function, represented by the $\lfloor\ \rfloor$ symbols, returns the greatest integer less or equal than the result of the $\frac{n}{k}$ division.

Taking the example of figure 3.14, it can be easily shown that:

$$n = s \cdot c_1 + (s+1) \cdot c_2 \text{ with } \begin{cases} s = \left\lfloor \frac{n}{k} \right\rfloor = \left\lfloor \frac{13}{5} \right\rfloor = 2 \\ c_1 = k - (n \bmod k) = 5 - (13 \bmod 5) = 2 \\ c_2 = n \bmod k = 13 \bmod 5 = 3 \end{cases} \quad (3.4)$$

And making the substitutions we have that $13 = 2 \cdot 2 + (2+1) \cdot 3 = 2 \cdot 2 + 3 \cdot 3$ which means that we can create 2 subsets of size 2 plus 3 subsets of size 3, as shown in the figure. Using the formula that was just presented we can make another example. Imagine having a Dataset of 26 elements, and we want to perform a 10-fold cross validation. We can find that $s = \left\lfloor \frac{n}{k} \right\rfloor = \left\lfloor \frac{26}{10} \right\rfloor = 2$, and then $c_1 = k - (n \bmod k) = 10 - (26 \bmod 10) = 4$ and $c_2 = n \bmod k = 26 \bmod 10 = 6$. We can immediately find that it is possible to create 4 subsets of 2 samples each and 6 subsets of 3 samples each. We can now select a specific subset for each fold iteration to use it for validation and merge the others to make a training subset.
Moving on to the demonstration of the formula 3.3, it is possible to show it by starting with the definition of the division:

$$n = q \cdot k + r \text{ with } \begin{cases} n: \text{ dividend,} \\ q: \text{ quotient,} \\ k: \text{ divisor,} \\ r: \text{ remainder} \end{cases} \quad (3.5)$$

In addition to that, considering the nomenclature used in 3.5, we have by definition of these functions, that:

$$\left\lfloor \frac{n}{k} \right\rfloor = q \quad (3.6)$$

$$n \bmod k = r \quad (3.7)$$

We can substitute 3.6 and 3.7 into 3.3, the formula to be demonstrated:

$$n = s \cdot c_1 + (s+1) \cdot c_2 \text{ with } \begin{cases} s = \left\lfloor \frac{n}{k} \right\rfloor = q \\ c_1 = k - (n \bmod k) = k - r \\ c_2 = n \bmod k = r \end{cases} \quad (3.8)$$

$$n = s \cdot c_1 + (s+1) \cdot c_2 = q \cdot (k-r) + (q+1) \cdot r = q \cdot k - q \cdot r + q \cdot r + r = q \cdot k + r \quad (3.9)$$

Which is the definition of the division presented in 3.5, thus demonstrating the formula.
Finally, the flow chart of the *statusTrainKFold()* function is shown in figure 3.10.

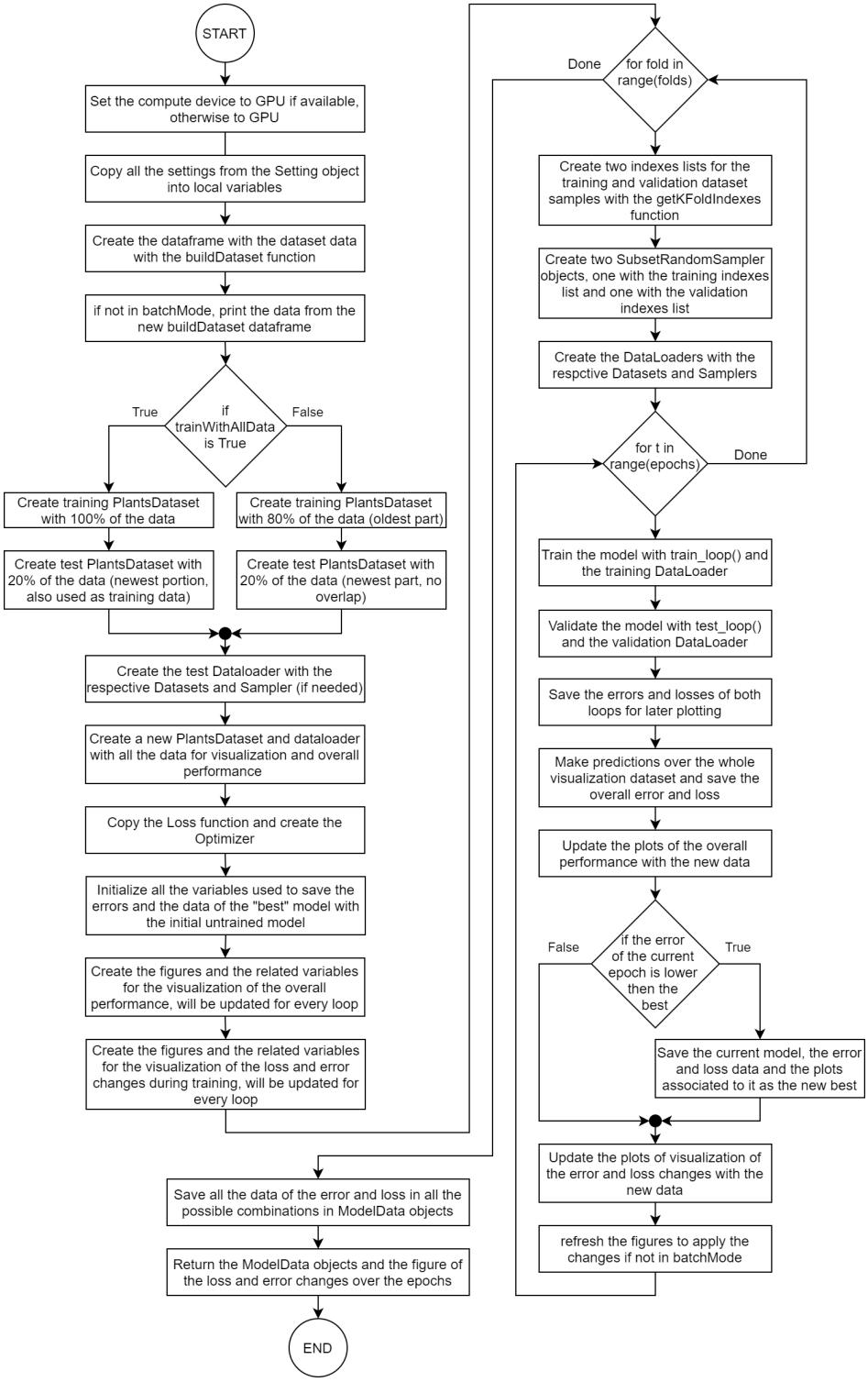


Figure 3.15: Block diagram of the *statusTrainKFold()* function.

3.5 Status Now Finder: finder of the best predictor software implementation

The second core functionality of the framework is Status Now Finder (or *status_now_finder*. While Status Now lets the user make a single training effort with a specific set of settings, Status Now Finder allows the user to run a sweep and search for the optimal value of a specific setting among a few options by just changing a settings file or, if necessary, by adding a new search type to the *status_now_finder* code in the *status_now_finder.py* file. This is necessary as the various settings require different types of sweeps. For example, a sweep over a simple variable like the Number of samples per parameter requires a single loop. In contrast, other types like, for example, the best training time range where the test error is the lowest, may require multiple nested loops. Also, variables like the *dataset_settings* list of dictionaries may require looping over a variable that is internal to a dictionary. At the same time, for sweeps related to the Neural Network model, the whole object has to be rebuilt. Due to this complexity, only a few important search types have been implemented, as implementing any possible type of search would have required too much time.
The *status_now_finder.py* file is structured in the same way as the *status_now.py* file, in the sense that the core of the Status Now Finder code is implemented in the *statusFinder()* function, but there is also a *main()* function that is called if the *___name___* built-in global variable contains the *"___main___"* string. The *main()* function simply selects a sweep type (selectable by changing the code) and parses the input data CSV files, only to pass all these things to the *statusFinder()* function. The Status Now Finder functionality can otherwise be called by the Dispatcher utility (which will be explained later) so that it can be used with much more versatile settings files.
The *statusFinder()* function takes as inputs the following parameters, of which only the first two are mandatory:

- *plant_df*:

  List of Pandas Dataframes, where each Dataframe is obtained by parsing a CSV file with plants data in it. Same format as in *statusTrain()* or *statusTrainKFold()*.

- *plant_list*:

  List of strings, where each string is the identifier of a respective plant in the *plant_df* Dataframe. The order of the identifiers must be the same as the order of the Datframes. Same format as in *statusTrain()* or *statusTrainKFold()*.

- *search_type*:

  String with the name of the search type. Only the strings related to implemented search types are allowed. More search types can potentially be added, but by default they are: *"time1"*, *"nnShape1"*, *"nnShape2"*, *"nSamplesPerParameter1"*

- *save_folder_root*:

  Path object from the Pathlib library. It contains the root folder where the "results" directory is present. A new folder will be created on the "results" directory to save the sweep results.

- *useKFold*:

  Boolean value, if set to True *statusTrainKFold()* will be used, otherwise *statusTrain()* will be used.

- *decideWithFullData*:

  Boolean value, if set to True the rankings files and the decision whether to save or not the data of a training iteration will be based on the overall error data relative to the whole Dataset (Training plus Test Datasets). If False only the error relative to the Test Dataset will be employed.

- *finder_setting*:

  FinderSetting object. Acts as a struct, where the kinds of public variables in it depend on the *search_type* parameter (same as the one described before) that is passed to the *FinderSetting* class when it is created. It has three settings that are common to all the search types: a *"common_settings"* that is a *Setting* object to be passed to a statusTrain function; two threshold variables called *"RMSE_threshold_to_save"* and *"loss_threshold_to_save"* that can contain integer or floating point values. The error and loss selected with the *decideWithFullData* value are compared to these values. *"inf"* is an allowed value, and it allows to save everything. The other public variables are created only depending on the *search_type* and are described later in the respective subsections.

- *batchMode*:

  Boolean value, is forwarded to the *statusTrainKFold()* function or to the *statusTrain()*, depending on which one is used. If true, the graphical plots are not shown on screen, while if false, they are shown in interactive mode. The plots are saved as .png files regardless of this setting.

The *statusFinder()* function has an initial and a final part that are common to any sweep type and a central part that is divided into many portions that, albeit similar, allow for the sweep of very different types of variables. Figure 3.16 shows the block diagram of the *statusFinder()* function.
The first common part simply sets up what is needed next: first, the folder for the files to be saved is created within a *"results"* folder that is located in the passed path, then a *results.txt* file is also created and opened. This file will be used as a sort of log file where information about each iteration will be saved in a format similar to the Status Now utility presented previously, but with additional info related to the sweep iterations. An index is also initialized, and a ranking list is created. The index will be used as a unique number identifying every iteration of the sweep. For example, for a sweep with 100 steps, there will be 100 different models to be trained, and each one will get its unique identifying index depending on the training order. The ranking list will contain a dictionary for each trained model, where each dictionary will have five key-value pairs. One of the pairs is dedicated to the index, while the other will contain error and loss data about the best model obtained during training and the last model obtained during training. These error and loss values will be referred to the whole Dataset (Training and Test) or only the Test Dataset depending on the *decideWithFullData* flag. Besides this detail, these dictionaries will be used to sort the elements in the list multiple times, each time by a different key. In a way, it sort of acts like a Pandas Dataframe.
The central part is a series of cascading if statements that let the program select the correct search type, like in a switch environment. Each option behaves in a very similar way and has pretty much the same structure with some nuances related to the specific variables of each search type. This repeated sweeping portion of the code is shown in the block diagram of figure 3.17. As it can be seen in the block diagram, the first thing that is done is to check if a *FinderSetting* object has been passed. If it has not, the algorithm will use default sweep settings and default fixed settings that are included in the file. In the opposite case, the passed settings will be copied over. As mentioned before, this *FinderSetting* object is a simple container with many public variables, with a few common ones and then some variables specific to the search type. As mentioned before, this portion of the code has some parts specific to the search type, like in this case, where only the variables relative to the search type are copied. After printing the information relative to the fixed settings on the results file, a for loop or a series of nested for loops is used to perform the sweep. This can occur by providing a start value, an end value, and a step, or by directly providing a list of the values to be swept. While the details on this are search-type specific, like applying the swept variable to the setting that has to be passed to the *statusTrain()* or *statusTrainKFold()* function, what occurs next is pretty much the same for all the search types, and differences reside mostly in the names of the files that are created and in what

is written in the files. First the *statusTrain()* or the *statusTrainKFold()* is called depending on the *useKFold* flag. All the necessary parameters are passed, and the results are obtained. Then, depending on the other flag *decideWithFullData*, the error and loss variables relative to the whole Dataset or only to the test Dataset are extracted and added to a dictionary that is appended to the ranking list, which will be used for the rankings that will be generated at the end. At the end of this portion of the code, it is checked if the RMSE and loss data of the best model that has been obtained during the various epochs are better than the necessary thresholds passed via the *FinderSetting* object. If this is the case, all the data that has been returned by either the *statusTrain()* or the *statusTrainKFold()* function is saved in the *results.txt* file (if it is in numerical form) or as *.png* for the figures or as *.pth* files for the actual models This is done so that later it is possible to check everything that was obtained besides the rankings. These thresholds are available if it is wanted to avoid saving data about models that are not good enough and not worth saving. This can be useful for large sweeps over hundreds and hundreds of variations, where if everything were to be saved in every iteration with no exceptions, too much storage memory would be occupied.
The final common portion of the *statusFinder()* function is related to the creation of the rankings files. Each file will contain two columns, the first with the index of the model so that it is possible to find the pictures and the model saved in the .pth format (as these have a prefix with the index in their filename), and the second column with the value of the error or the loss of the respective ranking. Four rankings are created: one for the prediction RMSE associated with the model that has been obtained during training that behaves the best (regardless of the epoch), one for the prediction RMSE associated with the last model obtained during training (at the last epoch), and two other rankings associated to the average loss values in the same conditions as the RMSE. As said before, it is possible to choose to use the RMSE and average loss values relative to the whole Dataset or to the test dataset only by using the *decideWithFullData* variable. The rankings are generated by using the list of dictionaries mentioned before, which is sorted for each ranking type by using the built-in *sorted()* function and using the *"key"* parameter. By passing to this parameter a lambda function that returns the value to be used for sorting in the dictionary, it is possible to sort the dictionaries in the list by a key-value couple of choice without manually implementing a sorting algorithm. For example, by writing *sorted(ranking, key=lambda k: k["last_loss"])*, a sorted list with the same items contained in *ranking* will be returned, where the sorting will be performed according to the values associated to the *"last_loss"* key of each dictionary in the list.
Finally, the *statusFinder()* function simply closes all the opened files and returns with now return value.

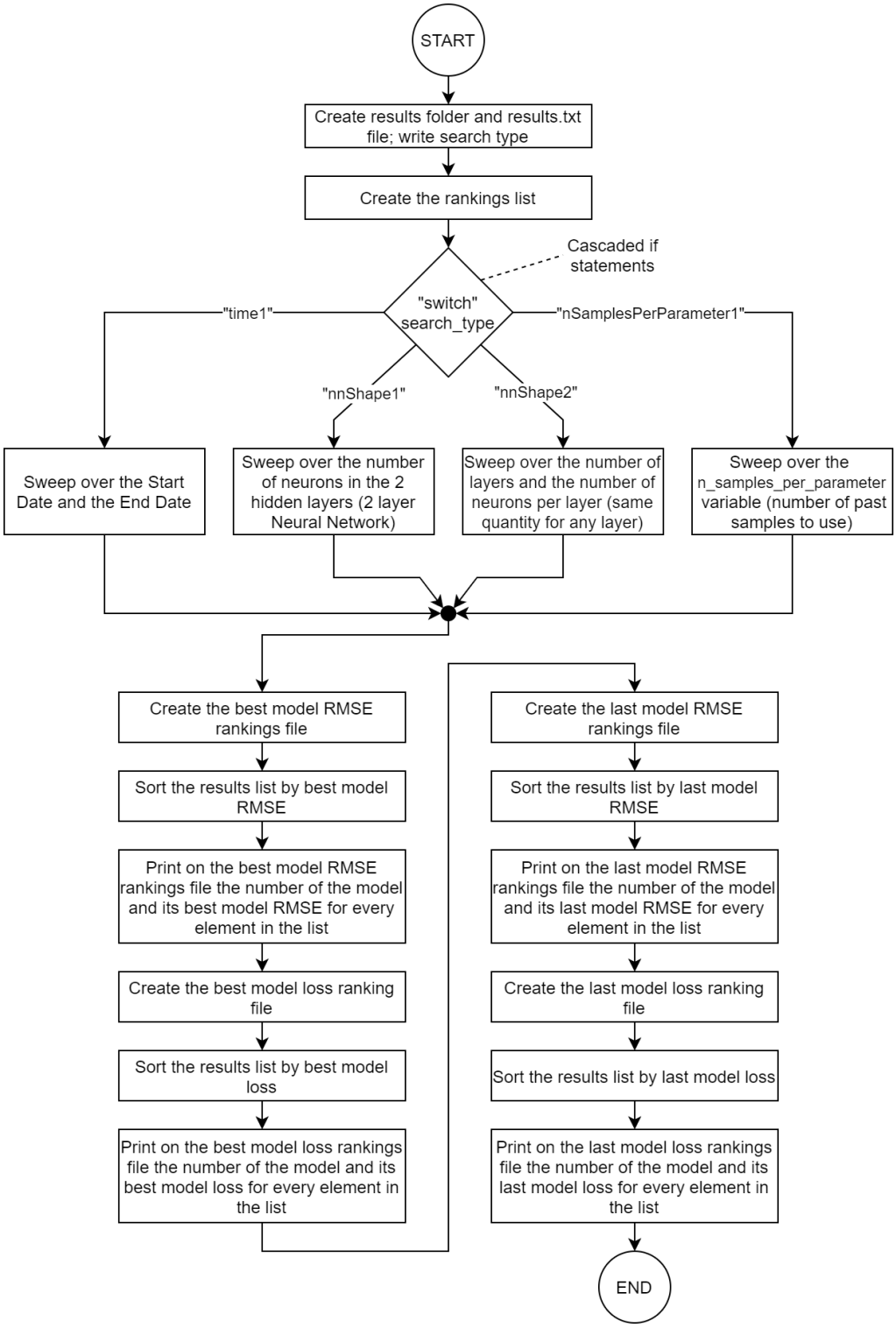


Figure 3.16: Block diagram of the *statusFinder()* function.

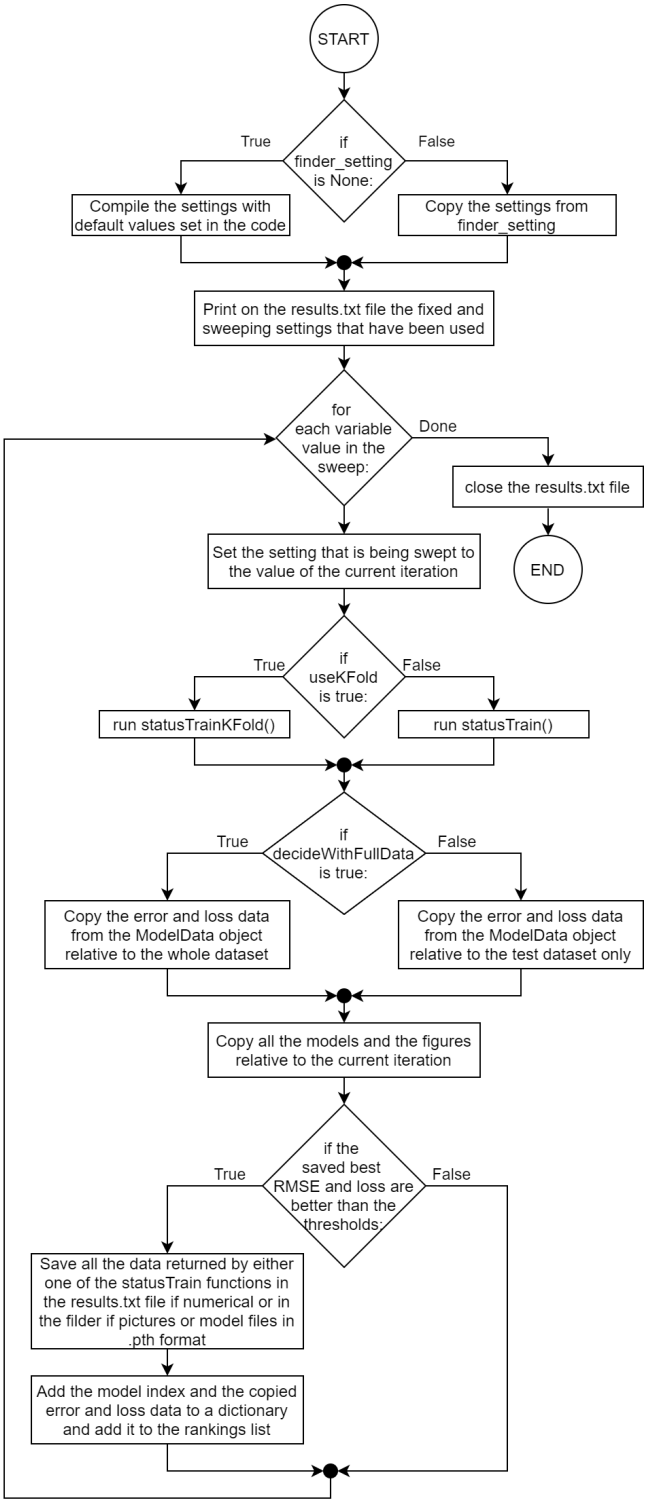


Figure 3.17: Block diagram of the sweeping portion of the *statusFinder()* function.

To end this subject, the search types that have been implemented are going to be described briefly. It is important to note that while these search types have an intrinsic value for searching for the best settings to be used for training, they also serve as an example of similar search types that may be implemented in the future. As such, a few words in this regard will also be spent.

3.5.1 Time search: search for the data that gives the best predictions

The first search type that has been implemented can be used to search for timeframes where the training is more effective. This can be used to check if some events (like watering events) affect the training positively or negatively. The string that identifies this search type is *"time1"*.
This search consists of three nested loops. The outermost loops over a list of plants on which the search should be performed. This is done as this type of search makes sense only when performed on a single plant at a time, as the events mentioned before may not be the same for every plant and may not occur in the same timeframe. The central one loops over the window size given a minimum time window length, a maximum time window length, and a step to move from the minimum to the maximum. For example, with a minimum of 3 days, a maximum of 9 days, and a step of 3 days, the combinations that would be tested would be 3, 6, and 9 days windows. Finally, the innermost loop cycles through the time window position. It is used to move the window through the available days, acting as an offset from the first available days. This offset starts at zero and can reach a maximum equal to the available days minus the current window length. A variable used to set the step indicating how many days the window should be moved further for each iteration is also made available. The variables specific to this search type are:

- *start_date*:

  String in the "YYYY-MM-DD hh:mm:ss" format representing the earliest date from where to start the sweep.

- *end_date*:

  String in the "YYYY-MM-DD hh:mm:ss" format representing the latest date where the sweep should end.

- *window_min*:

  Integer number representing the minimum number of days that should be used as time window size.

- *window_max*:

  Integer number representing the maximum number of days that should be used as time window size.

- *window_change_step*:

  Integer number representing the step in days to change a time window from a size to the next in subsequent iterations.

- *window_move_step*:

  Integer number representing by how many days the window should be moved to the right in the the timeframe enclosed by *start_date* and *end_date* in each iteration of the innermost loop.

- *min_samples_for_training*:

  Integer number setting a minimum number of samples that should be available in a timeframe for a training effort to start. This is needed in cases where there are holes in the data as otherwise a training could be commanded over a timeframe with no samples in it.

- *plants_to_cycle*:

  List of plants identifiers. Used to select the plant or the plants on which to run the time search.

Figure 3.18 shows an example where the timeframe from *start_date* to *end_date* is 10 days long, *window_min* is set to 2, *window_max* is set to 3, *window_change_step* is set to 1, and *window_move_step* is set to 2. The curved arrows show the movement of the window set by the *window_move_step* variable.

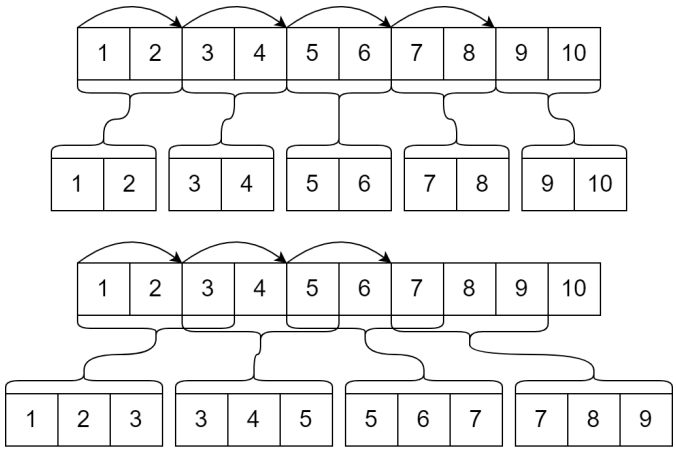


Figure 3.18: Example of timeframes generated for training.

3.5.2 Neural Net Shape search: search for the best Neural Net topology

The second type of search is about the shape of the Neural Network. This is very important as the number of neurons and the number of hidden layers can substantially affect the system's performance. For this reason, the search types have been developed, both with the focus on the shape of the Neural Network: *"nnShape1"* and *"nnShape2"*.
"nnShape1" is used to find the best configuration of a Neural Netowork with two hidden layers by sweeping the number of layers in each one of the hidden layers. This is accomplished with two nested loops, the outermost for hidden layer 1 (the one nearest to the inputs) and the innermost for hidden layer 2 (the one nearest to the outputs). These for loops cycle through the only two variables that are specific to this search type:

- *layer1_neurons_list*:

  List of amounts of neurons for hidden layer 1 that have to be tested.

- *layer2_neurons_list*:

  List of amounts of neurons for hidden layer 2 that have to be tested.

In this case, no minimum, maximum, and step are provided, but a list is provided so that the user can define precisely the amounts of neurons in each layer to be tested. This is especially useful as, for example, a change from 4 to 8 is substantial while a change from 508 to 512 is not so much. With this setup, the user can manually list the amounts, use a generating function, or even join different lists to test complex lists of amounts of neurons. As every combination of amounts of neurons is tested, the total amount of training cycles that will be executed in the sweep is equal to the product of the sizes of the two lists. This is important to keep in mind to avoid running sweeps that may take too long to complete as too many combinations have to be tried. On a side note, it is worth mentioning that it is also possible to make the *layer2_neurons_list* list an empty list, in which case a Neural Network with only a single hidden layer tested with the amounts of neurons specified in the *layer1_neurons_list*list.
The other search type in this category is *"nnShape2"*. This one offers a more powerful capability as it allows for sweeping the number of neurons in the hidden layers as well as the number of the hidden layers. In this case the there are also only two variables that are specific to this search type:

- *n_layers_values*:

  List of amounts of neurons in the hidden layers that have to be tested.

- *n_neurons_values*:

  List of amounts of hidden layers in the Neural Network that have to be tested.

In this case, too, there are two nested for loops, with each one dedicated to each variable. Once the amount of a specific iteration is picked, a new Neural Network model is generated with the correct amount of hidden layers and the correct amount of neurons for all the hidden layers. While in *"nnShape1"* it was possible to have different amounts of neurons in each layer, this is not possible in *"nnShape2"*. This was not implemented as the complexity would have been too large as the total amount of iterations would have grown out of control pretty quickly, even at just three or four hidden layers. For example, if it was wanted to test just ten different amounts of neurons for each hidden layer, and every combination were to be tested, while with two layers there would be just 100 iterations, with four layers that amount would rise already to 10000. As the growth of the complexity would be exponential in that case, it was decided that the sweep over the number of neurons in the hidden layers would be uniform over all the layers so that the total amount of iteration for the *"nnShape2"* search type would be equal, as before, to the product of the sizes of the two passed lists.

3.5.3 Past samples search: search for how long in the past the samples are useful

The last search type that has been implemented, *"nSamplesPerParameter1"*, is also the simplest one. The objective of this search type is to find the best amount of past samples to use for the input parameters. This sweep tests how far in the past is worth getting the samples, allowing us to find out the time window that should be used to make a single prediction. As such, this sweep type acts on variables of the *Setting* object such as *n_samples_per_parameter*, *n_overlap_of_samples*, and also on the *NeuralNetwork* as the number of input changes with each iteration. As an example, a *n_samples_per_parameter* value of x means that the last x samples are used to make a prediction, so if one sample is captured every hour, a value of x means that x hours worth of data are being employed to make a prediction.
In this case, a min-max-step setup is employed, with three variables associated with this task:

- *n_samples_per_parameter_min*:

  Minimum value of *n_samples_per_parameter*. The absolute minimum value that should be used for this variable is 1, which corresponds to taking only the latest sample, thus not using past data for any new prediction.

- *n_samples_per_parameter_max*:

  Maximum value of *n_samples_per_parameter*. There is no absolute maximum value for this variables, but a too big value may make the model too complex and also cause overfitting.

- *n_samples_per_parameter_step*:

  Integer value that represents the step in the number of samples used to increase the *n_samples_per_parameter* from one iteration to the next.

On a final note, it is worth noting that for most other variables that it could be worth sweeping, this final search type is probably the best starting point to implement a new search type, as most variables do not require nested loops to be swept. More details on implementing a new search type for a specific variable will be presented in Appendix A.

3.6 Utilities

As part of all the work described up to this point, numerous functions and classes not strictly related to the machine learning algorithms were developed. Most of these functions and classes have been implemented in the *utilities.py* file, which contains, as the name suggests, most of the generic utilities that have been developed. Some of these functions and classes have already been mentioned or explained before as they were crucial for a specific portion of the framework. The *buildDataset()* and the *getKFoldIndexes()* functions as well as the *ModelData* class fall under this category. Most other functions are quite simple and straightforward, but it is still worth mentioning them and their usage. Before listing all these functions, it must also be mentioned the fact that the *multipledispatch* library has been employed to perform a sort of function "overload" as in C++ so that multiple functions with the same name but with different input parameters and return types could coexist. This library works by adding a decorator *@dispatch(<variable type>)* in the line before the "overloaded" functions, specifying inside the decorator the type of variable that should be passed to the specific overloaded function. This is useful for cases where we want to implement a function that acts differently with distinct parameter types without having to check for the type of the parameter inside the function itself.

This being said, here is a comprehensive list of the functions that have been implemented in the *utilities.py* file:

- *getHour(date_list)*:

  Function that given a list of dates in the "YYYY-MM-DD hh:mm:ss" string format returns which hour of the day it is for each provided date in a list format.

- *getHour(date)*:

  Function that given a date in the "YYYY-MM-DD hh:mm:ss" string format returns which hour of the day it is for that date in an integer format.

- *getDayOfYear(date_list)*:

  Function that given a list in the "YYYY-MM-DD hh:mm:ss" string format returns which day of the year from 1 to 365 it is for each provided date in a list format. Leap years are taken into account.

- *getDayOfYear(date)*:

  Function that given a date in the "YYYY-MM-DD hh:mm:ss" string format returns which day of the year from 1 to 365 it is in an integer format. Leap years are taken into account.

- *getYear(date_list)*:

  Function that given a list in the "YYYY-MM-DD hh:mm:ss" string format returns which year it is for each provided date in a list format.

- *getLatestDate(date_list)*:

  Function that returns the latest date in the date_list list. "YYYY-MM-DD hh:mm:ss" string format must be correct as no checks are performed.

- *getEarliestDate(date_list)*:

  Function that returns the earliest date in the date_list list. "YYYY-MM-DD hh:mm:ss" string format must be correct as no checks are performed

- *pause()*:

  Function that pauses the execution of the program until the enter key is pressed.

- *norm(data, median=None, data_range=None)*:

  Function that normalizes between -1 and +1 the elements in a list. The function uses the median and data_range paramters if provided to perform the normalization, otherwise the median and data range are calculated from the data list.

- *constrain(data, min_val=None, max_val=None)*:

  Function that constraints each element in the list within a minimum and a maximum value, both passed as paramters. If a limit is not passed, that side of the range is unconstrained.

- *transform(data, function, **kwargs)*:

  Function used to transform a list of elements in another list of elements according the *function* function. The function is applied to each element individually.

- *filter_f(data, filter_function, **kwargs)*:

  Function used to filter a list of elements in another list of elements according the *filter_function* function. Each element may change also depending on the neighboring elements.

- *log10abs(x)*:

  Function that returns the logarithm in base 10 of the absolute value of x.

- *getPlantDatasetNRows(tot_plant_samples=1, n_samples_per_row=1, n_samples_overlap=0)*:

  Given a total number of samples, the number of samples per row, and the overlap of samples between multiple rows, the function returns the total number of obtainable rows. The function is used in the *PlantsDataset* class initialization method to compute in advance how many rows in the Dataset will be created in order to iterate over every row.

- *makeRamp(data, noise=0.0)*:

  Function that builds a ramp-like vector given a coefficient and some noise amount to add to the ramp.

- *printData(dataset, params, plants)*:

  Function that prints plots of a plants dataset (obtained with *buildDataset()*), selecting the parameters to print and of which plants.

- *simpleMovingAverage(data, N=24)*:

  Function that computes the simple moving Average over a chosen period of samples.

- *exponentialMovingAverage(data, N=24, alpha=0.5)*:

  Function that computes the exponential moving Average over a chosen period of samples with the chosen α coefficient.

Another class named *Date* was also created in order to handle more efficiently various dates, especially in the time search of the Status Now Finder. In that search type, it was necessary to increase the dates given in a "YYYY-MM-DD hh:mm:ss" string format by a certain amount of days. So this class was created to handle these increases of dates, like adding 10 days to a specific date. Class methods to convert to and from the "YYYY-MM-DD hh:mm:ss" string format were also implemented, as well as methods that simply extract a specific component of a date, like the hour. This *Date* class has six private variables, all accessible via setter and getter methods, which contain the integer components of a date (the year, the month, the day, the hour, the minute, and the second). All the setter functions include assertions about the correctness of such values. For example, the second value must be ≥ 0 and < 60 like the minute, and so on. The assertion for the day is for sure the most complex one, as moths have to be taken into account. Besides all these getters and setters, the *Date* has the following methods:

- *___init___(self, date_str)*:

  Class constructor. Splits the input string *date_string* in the "YYYY-MM-DD hh:mm:ss" format into the year, month, day, hour, minute, second integer variables.

- *___str___(self)*:

  String conversion built-in function redefinition. Just calls the *toString()* method.

- *toString(self)*:

  Converts the Date into a string in the "YYYY-MM-DD hh:mm:ss" format.

- *toStringCompact(self)*:

  Converts the Date into a string in the "YYYYMMDD_hhmmss" format, which is better when used in filenames as it doesn't have any space or colon symbol in it.

- *increaseYear(self, increase)*:

  Increase the year by "increase" years.

- *increaseMonth(self, increase)*:

  Increase the month by "increase" months. The year is increased too if necessary.

- *increaseDay(self, increase)*:

  Increase the day by "increase" days. Increase the month too if necessary. Differences of day counts of different months and leap years are taken into account.

- *increaseHour(self, increase)*:

  Increase the hour by "increase" hours. Increase the day too if necessary.

- *increaseMinute(self, increase)*:

  Increase the minute by "increase" minutes. Increase the hour too if necessary.

- *increaseSecond(self, increase)*:

  Increase the second by "increase" seconds. Increase the minute too if necessary.

- *diffDays(self, date2)*:

  Get the absolute difference in days between the two dates. Hours, minutes, and seconds are discarded. Initial and final days are included in the count.

On a note regarding the *increase*()* functions, all of them are implemented in such a way that a "chain" increase can happen. For example if the original date is "2021-12-31 23:59:50" and the *increaseSecond()* function is used to increment the date by 15 seconds, the Date will be updated to "2022-01-01 00:00:05".
While this concludes the presentation of the Utilities, it is worth noting that each function, class, and method implemented in the *utilities.py* file includes comments and Type Hints as specified in the Python standard, so the type that each variable should assume is specified, along with a brief explanation of the function, class, or method.

3.7 Dispatcher: running the scripts from a shell

The last part of the framework that is left to be talked about is the dispatcher, implemented in the *status_now_dispatcher.py* file. The dispatcher is what truly makes this project a framework and not just a compilation of scripts. The dispatcher allows the user to:

- Launch the Status Now or the Status Now Finder functionalities from a CLI.
- Use settings files so that the core source files do not have to be modified to run different training efforts.
- Run multiple Status Now or Status Now Finder processes simultaneously with different settings
- Select at runtime the setting to be used and whether to launch the functionalities in a Batch mode or not.

The dispatcher has been implemented by taking advantage of the Python *argparse* library, which simplifies the creation of a CLI interface that is similar to other commands that can usually be called from a Linux shell, for example. Following is listing 3.7 with text taken from the output of the dispatcher when launched with the *python status_now_dispatcher.py -h* command, which prints the help utility. This help text is generated automatically by the *argparse* library.

Listing 3.7: Dispatcher help.

```
usage: status_now_dispatcher.py [-h] [-f FILE] [-b BATCH] script

Run the Status Now scripts from a Shell.

positional arguments:
  script                Select the script to run. Available scripts:
   status_now, status_now_finder

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  Run the selected script with the settings
   from the passed file. Pass the name of the chosen setting file
   located in the settings folder.
  -b BATCH, --batch BATCH
                        Choose to run the script in Batch mode if
   True (Interactive plots and graphs disabled) or not if False (
   Interactive plots and graphs enabled).
```

As it is shown in the listing 3.7, four arguments are allowed, of which three are optional. The compulsory argument is the *script*, which should be either *status_now*

or *status_now_finder*, and is used to select the functionality of the framework to run. Then there are three optional arguments of different importance. The *-h* or *–help* argument simply prints what is shown in the listing and is used to remind to the user of the usage of the dispatcher. On normal operation, it should not be used. The *-f FILE* or *–file FILE* argument is an argument used to specify a setting file to be used by Status Now or Status Now Finder. The filename should be passed instead of the *FILE* text, and it should refer to a file with that name present in the settings *directory* within the *src* folder. If the file is not found, an error is given. However, suppose the argument is not used. In that case, the selected functionality will simply be executed with the default settings as if the file was launched standalone (the main code of Status Now or Status Now Finder would be executed). The last argument is the *-b BATCH* or *–batch BATCH* argument and is used to run the functionalities in Batch mode or not. If Batch mode is selected, the figures will not be shown on screen during execution, and they will only be saved to file if necessary. In contrast, if disabled, the interactive mode on the windows relative to the figures will be active. To enable batch mode, it is sufficient to substitute the *BATCH* with other text like "true" or "yes" or a related abbreviation, while to disable the mode, the opposite has to be done by writing "false" or "no" or, again, a related abbreviation. If the batch argument is not used, the batch mode will be inferred: it will be enabled by default whenever a setting file is passed, and it will be disabled by default in the opposite case. Examples of the actual usage of the dispatcher are provided in Appendix A.
This utility allows the user also to run the framework much more efficiently remotely, which has also been demonstrated working on a Linux server with the tensor-related code running on the GPU.

3.8 On the Field

Before moving on to the final chapters with the results and the conclusion, this section is needed to understand how what has been developed under this thesis work could be deployed on the field to be used as intended. As explained in the introduction, the objective is to perform the machine learning related computation on the field, possibly on a microcontroller or even on an integrated device that is self-sufficient and that can transmit the results. However, to do this, quite a few steps have to be performed so that the ML algorithm developed thanks to the framework can be deployed. If used as intended, the Status Now Finder utility would be used to find the best data processing and ML architectures. Then the Status Now utility would be used to fine-tune the small details that don't require a large sweep. When executed, the Status Now utility saves to file the model and the settings used, both to train the model and process the data. All this information

could be used to convert these results to a C program, for example. While this process could certainly be performed manually, in theory, it would also be possible to automate it as long as the C core functions have been developed beforehand. For example, it could be possible to make a python script that can convert the .pth Neural Network model to a C function that takes in input an array with the input samples and returns as output a prediction. It could also be possible to make a script that can convert the Dataset settings written as a list of Python Dictionaries to a series of C functions that perform the same operations, as long as the functions used in the dictionary have also been developed in C beforehand and can just be linked to the rest of the code. Once these conversions will have been performed, the data processing code and the ML-related code could be used alongside the measuring related code and the transmission related code.

Chapter 4

Capabilities and Results

This chapter is dedicated to presenting the main results of this thesis work. These results include both the presentation of the capabilities of the framework and also actual training results. Of this last case, some results of single training efforts obtained with the Status Now functionality are presented, as well as some findings of some of the best settings obtained with the Status Now Finder functionality.

4.1 Framework capabilities

The framework, as developed, offers a large number of capabilities, especially in the data processing and in the Neural Network creation portions. The framework also has a substantial margin for improvement, especially for the Finder capability, where possibly the sweeping of every variable could be implemented.
Regarding the Data Processing capabilities, the possibility to define the features to use in the training and test dataset with a simple list of standard dictionaries and a few variables is undoubtedly the most valuable one. Each of the dictionaries allows one to choose one of the time series from the provided CSV files and use that data to create another time series. This new time series can be fully customized by the user that can apply a considerable amount of manipulations to the original series. First, the user can choose to limit the range of the time series values by setting constraints, which is extremely useful in cases where the full range does not make physical sense. Then the user can apply a filtering function of choice: a moving average and an exponential moving average have already been implemented (with parametric windows and coefficients), but the user is free to define other types of filters like IIR and FIR filters that may be used to filter out specific frequency components. This allows for creating multiple time series of a certain quantity, where each one contains only values in a specific frequency range. Next, the user can also decide to apply a function to transform the time series. This means that

a function is applied to every sample in the series, in addition to the filtering onen. This function should only take into account one sample at a time. For example, such functions may be the absolute value, the squaring function, the square root, and much more complex trigonometric or polynomial functions. Any function that can be implemented in software, takes one input, returns one output, and does not have memory is allowed. Finally, normalization is applied, but the user may decide which range and with which mid-range the series must be normalized. This is useful as an automated normalization algorithm may not consider outliers. After this processing has been performed and the new time series have been generated, the framework creates the features to be used by the Neural Network. At this step, the user can decide, by setting a simple variable, how large the time window should be when making a prediction, i.e., how many past samples should be provided to the Neural Network besides the most recent ones. The minimum value is one, where the Neural Network uses only a single value of each time series. However, this number can be scaled as desired, for example, to 24, where the data of a whole day is provided to the Neural Network from the time series.
The Neural Network structure and also how the training is performed can be changed easily. The model structure can be changed externally on the settings files, and there is no restriction in the number of layers, number of neurons per each layer, types of neuron layers, or types of activation layers. The user can also easily choose whether to perform a regular training and testing process or perform K-fold cross-validation, choosing the preferred amount of epochs and folds.
Finally, the Finder functionality allows, in addition to what has been mentioned, to perform sweeps on variables of choice to identify the best values to use for training. While the sweeping has not been implemented for every variable, the software is modular enough to add new sweep types to the software quickly.
Appendix A explains in detail how to take advantage of all these capabilities.

4.2 Status Now results

After performing many training efforts and obtaining mixed results, two many findings have been obtained: It is possible to obtain a good fit over the training data, especially if the K-Fold cross-validation is employed over the whole dataset, but it is tough to understand if overfitting may be occurring or not. It is especially hard to understand as the plants may react differently over time to similar environmental variables depending on the history of the plant itself.
Let us take an example of the training of a Neural Network with three hidden layers with 512 neurons each. Four input time series have been used: the impedance modulus of the plant, the impedance phase of the plant, the ambient temperature, and the hour at which each measurement was taken. The impedance time series

have been filtered with an exponential moving average with a 24 hour time window and a smoothing factor α equal to 0.4 to remove most of the noise. The temperature has been filtered with a simple moving average also with a window of 24 hours. The terrain moisture data has been used as the output to be predicted. The time series relative to the moisture has been filtered too with an exponential moving average with the same settings used for the impedance data. A transformation function has also been used, in particular, a function that applies the absolute value and then the logarithm in base 10. The model has been trained with data of the "pianta4" plant in four different cases, where the differences for the four cases are the end date and the option to train the model with all the data and not hold out any data for final testing.
The first configuration to be analyzed is the one with the earliest end date and some data being held out of the training dataset for testing purposes. As such, the last 20% portion of the dataset was held for testing and was not used during training. Figure 4.1 shows the visualization over the whole dataset of the model trained with 80% of the dataset, with the last 20% portion held out for testing. In this case, the earlier end date was used, so only a portion of the entire data provided in the original CSV file has been used for training. As it can be seen, the portion to the right (which is the portion held for testing) does not follow precisely the expected values. From what can be observed, it seems like there is some lag like the model expected the changes to take more time. Figure 4.2 shows the variation of the RMSE and the loss values during the training with these settings.
The next set of figures, 4.3 and 4.4 refer instead to the model also trained with the earlier end date, but with no data held for testing. In this case, it can be noticed that the model can predict accurately in that last portion, too.
Figures 4.5 and 4.6 shows instead the results of training over the full date range (additional data points are visualized on the right), while holding out once again 20% of the data. The situation now is different with respect to the first case: The model cannot again predict correctly over the test portion of the dataset, however only the general trends of the data can be considered correct, and no "lag" effect can be noticed this time.
The final case, shown with figures 4.7 and 4.8, is the one where training occurred over the whole datset, with no time range limitation. The model is once again able to make correct predictions over the whole dataset.

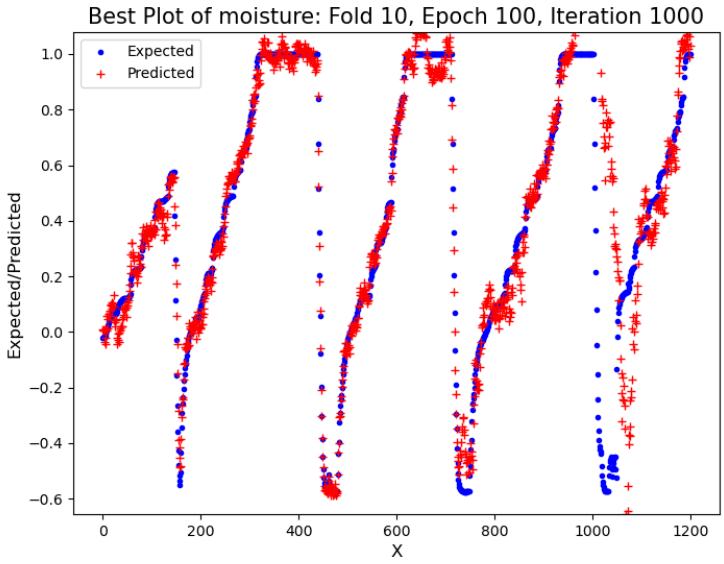


Figure 4.1: Model visualization over the whole dataset. Trained with 80% of the data, early end date.

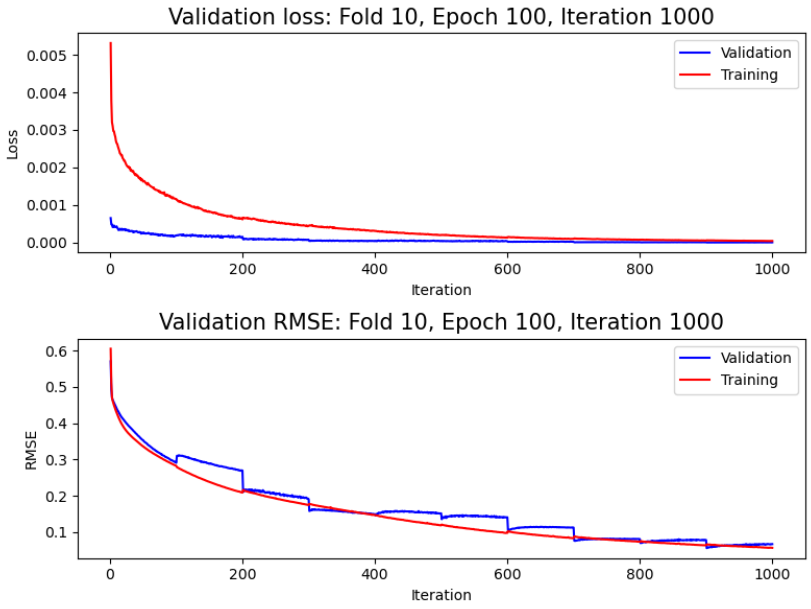


Figure 4.2: RMSE and loss variations. Model trained with 80% of the data, early end date.

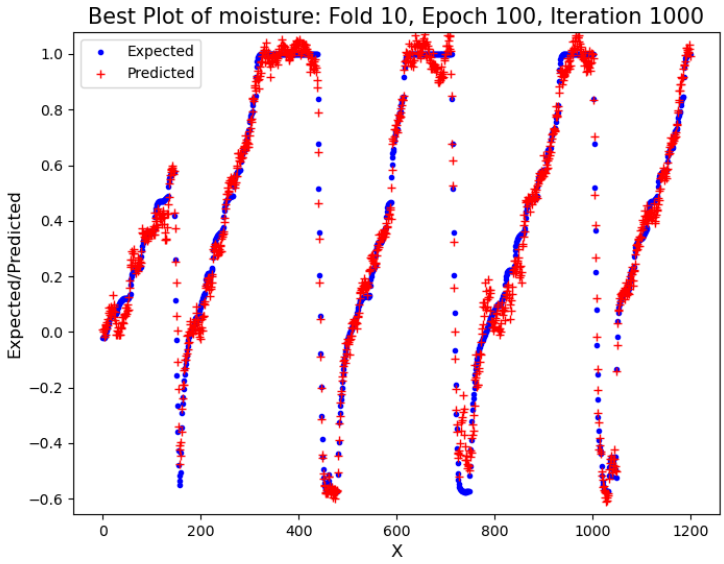


Figure 4.3: Model visualization over the whole dataset. Trained with 100% of the data, early end date.

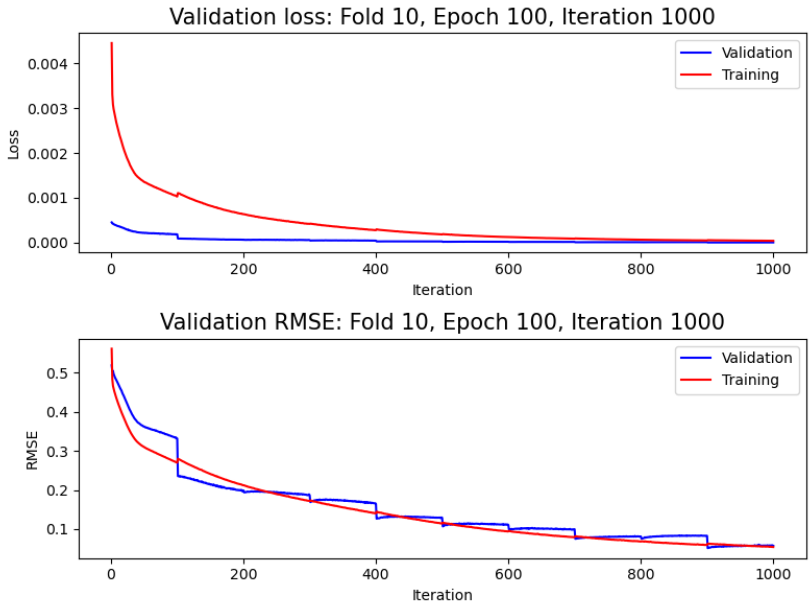


Figure 4.4: RMSE and loss variations. Model trained with 100% of the data, early end date.

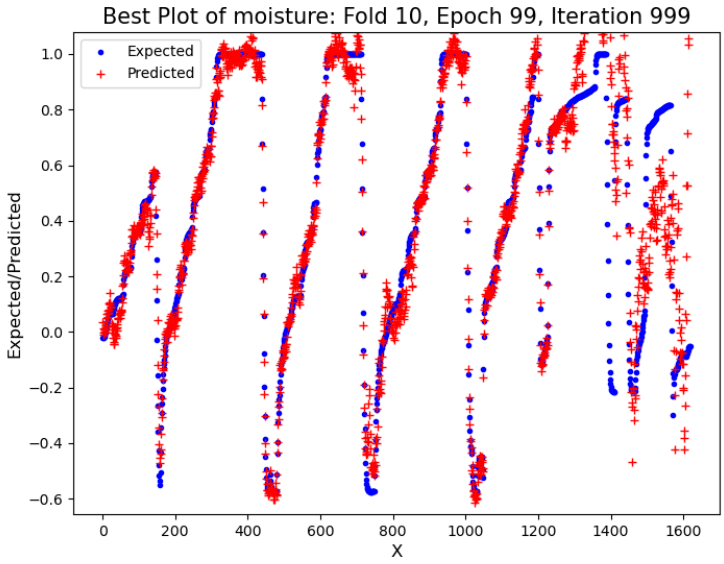


Figure 4.5: Model visualization over the whole dataset. Trained with 80% of the data, full date range.

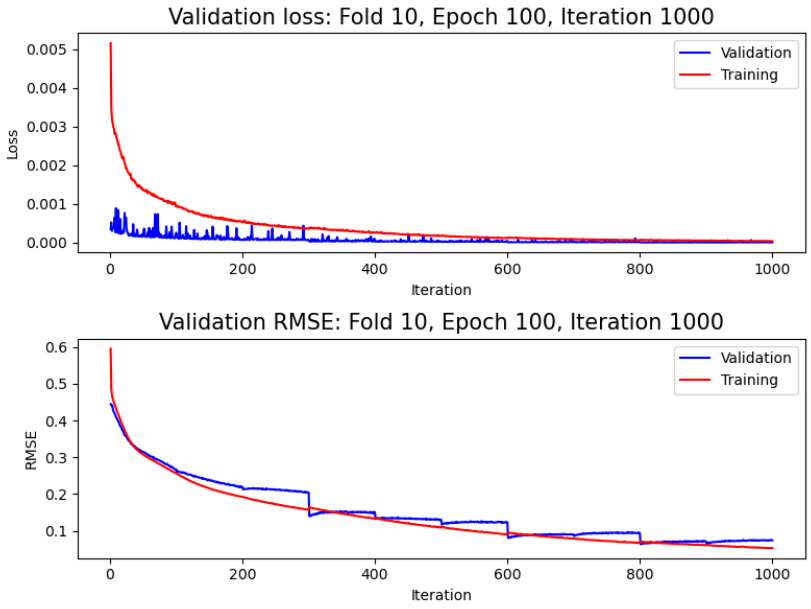


Figure 4.6: RMSE and loss variations. Model trained with 80% of the data, full date range.

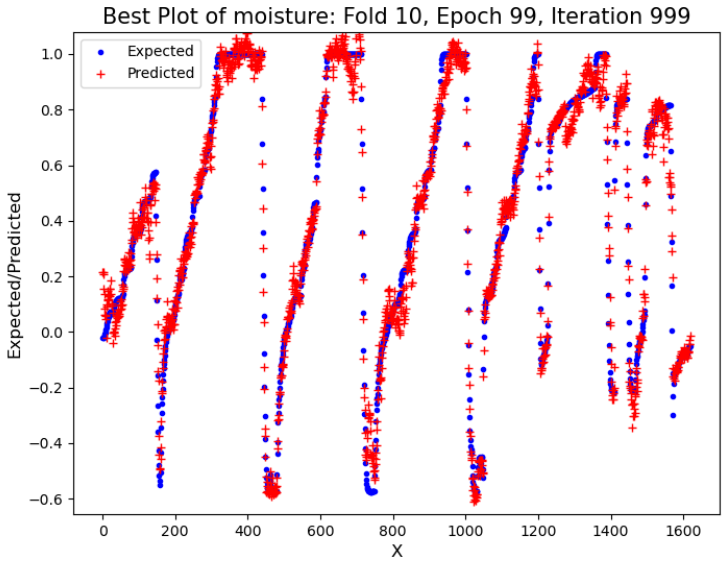


Figure 4.7: Model visualization over the whole dataset. Trained with 100% of the data, full date range.

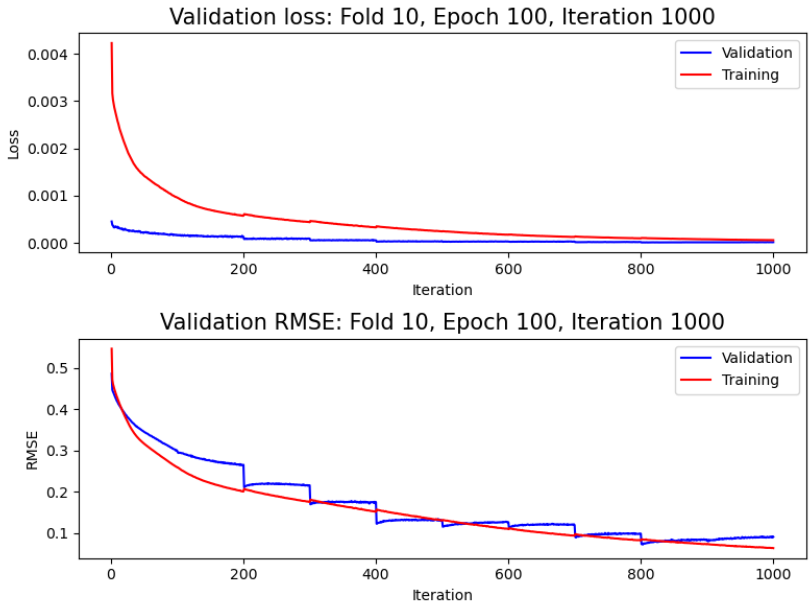


Figure 4.8: RMSE and loss variations. Model trained with 100% of the data, full date range.

Comparing the first two cases and the last two cases, it is hard to understand what is causing the effects that can be noticed. Overfitting may be at play, especially in the last two cases, as simply training over the whole dataset seems to fix the problem even if the model was performing awfully in the last portion of the data. The fact that the validation RMSE tends to grow (even if imperceptibly) while the training RMSE keeps decreasing seems to confirm this thought. On the other hand, the first two cases may suggest that additional and more variegate training data may be needed so that the model can learn behaviors of the data that are not otherwise always present, like the "lag" effect characteristic of the first case.

Another example is now provided to demonstrate that it is possible to create more advanced and exotic Neural Networks and data processing techniques. The data preparation is identical to the previous examples with only one main difference, the moisture parameter generation. The dictionary that generates the new moisture time series is the one of listing 4.1:

Listing 4.1: Dictionary used for the creation of the thresholded moisture paramter.

```
{
    "param_name": "moisture",
    "input_name": "Moisture [KPa]",
    "norm_data_range": 2,
    "norm_data_median": 0,
    "transform_function": thresholdlog10abs,
    "transform_function_kwargs": {"threshold": util.log10abs(500)
    /1.5},
    "filter": util.exponentialMovingAverage,
    "filter_kwargs": {"N": 24, "alpha": 0.4},
    "constrain_min": -500,
    "constrain_max": -1
}
```

As it can be seen, a *thresholdlog10abs* function is applied to the filtered data series. This function is defined in the following listing (4.2):

Listing 4.2: Function used to cap the data in a $[-1 \div +1]$ range with the selected threshold.

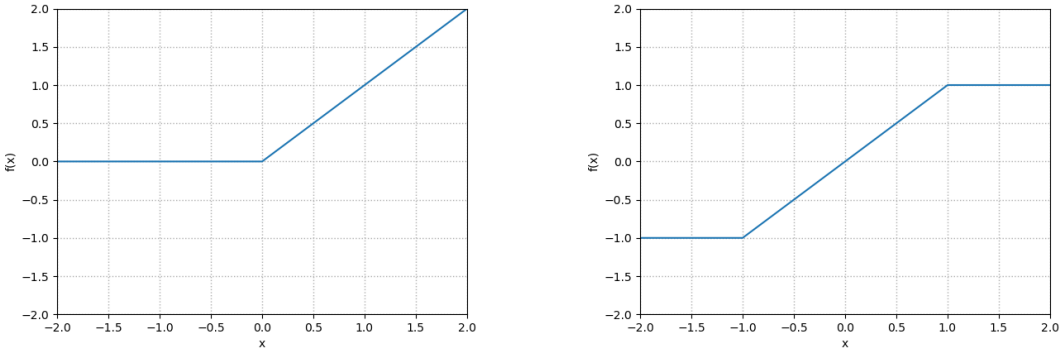
```
def thresholdlog10abs(data, threshold=249.5):
    return -1 if util.log10abs(data) < threshold else +1
```

The effect is that the time series is quantized, where only two levels are allowed, +1 and -1. The threshold, if not specified, is set to the middle value of the input data series if only the absolute value were used to transform the series. However, in the dictionary, as it can be seen above, the threshold is set to $\frac{2}{3} \cdot \log_{10} |500|$, where 500 is the maximum absolute value allowed by the constraint. The idea behind this threshold is to create a binary variable where, in this case, +1 means

that the plant is stressed while -1 means that the plant is healthy. This obviously is a significant approximation, but it is good enough as the first approach. The problem now is that the Neural Network used before is not really suited to make predictions that should not exceed the $[-1 \div +1]$ range, preferably only at the two extremes of the range. This is an issue because the *ReLU* activation function does not have an upper limit, so the output values are improbable to reach values near the extremes of the range with a small enough margin. For this reason, a new Neural Network structure has been created, this time taking advantage of another activation function: *HardTanh*. This function is defined as follows:

$$f(x) = \begin{cases} 1, & \text{if } x > 1. \\ x, & \text{if } -1 \geq x \geq 1. \\ -1, & \text{if } x < -1. \end{cases} \tag{4.1}$$

This activation function allows the output to cap at either -1 or 1 if the input is large enough in modulo, this allowing the Neural Network to assume the correct values even if the inputs were not quantized. Figure 4.9 shows a comparison between the ReLU function (4.9a) and the HardTanh function (4.9b).



(a) ReLU function. **(b)** HardTanh function.

Figure 4.9: Comparison of the ReLU and HardTanh functions used for activations.

The new Neural Network has been created with 3 hidden layers. The first two are the same as the previous examples: two layers with *ReLU* activation functions with 512 neurons each. The additional layer is a 128-neurons layer with *HardTanh* activations. The output layer with a single neuron also has the *HardTanh* activation function. This concept behind this structure is having a two-part Neural Network, where the first makes the predictions (with a structure that is known to perform well) while the second performs the capping in the $[-1 \div +1]$ range.

The generated Neural Network is the following shown in listing 4.3:

Listing 4.3: Neural Network used for the prediction of data capped in a $[-1 \div +1]$ range.

```
single_test_setting.generateModel(nn.Sequential(
    nn.Linear(single_test_setting.nn_inputs(), 512),
    nn.ReLU(),
    nn.Linear(512, 512),
    nn.ReLU(),
    nn.Linear(512, 128),
    nn.Hardtanh(),
    nn.Linear(128, single_test_setting.nn_outputs()),
    nn.Hardtanh()
))
```

This model has been trained in two different ways, in similar ways as the previous examples. The model was always trained with the *"pianta4"* data, processed as explained before with the new capping and threshold for the moisture data, with the dataset ending on the same "early" date used in the first two examples presented before. The difference is, once again, whether some of the data was held out for testing or not. Figures 4.10 and 4.11 contain the usual model performance and error/loss history graphs for the case where 80% of the data was used for training and 20% of the data (the last part of dataset, chronologically) was held for testing. As it can be seen in the first figure, the model cannot accurately predict the status when there is a transition in the test dataset, while it can make correct predictions once the data has settled. IF the model is trained instead with the whole dataset, so with no data being held out for testing, the model is then able to more correct prediction in that timeframe too, as it can be seen in figure 4.12. This is the same dilemma presented before, where it is hard to understand if by including also the test data for training, we are adding data useful for training so that the model can learn new behaviors, or if the model is simply overfitting, in which case by adding the test data to the test dataset we are simply making it overfit on that data too. In this case, it is even harder to understand this problem, as the graph that shows the change of the RMSE and the loss over each iteration for the second training type (figure 4.13) shows that the validation error (the error over the portion of data being used for validation during each fold) is constant and does not tend to increase. At the same time, the error over the training set also stops decreasing. This happens as many of the predictions get exactly to their target of -1 or +1, thus making the error over most of the predictions actually zero. Previously, on the other hand, there was pretty much always a margin for improvement as the output could have assumed any value in the floating-point range, making it virtually impossible to achieve an error of exactly zero.

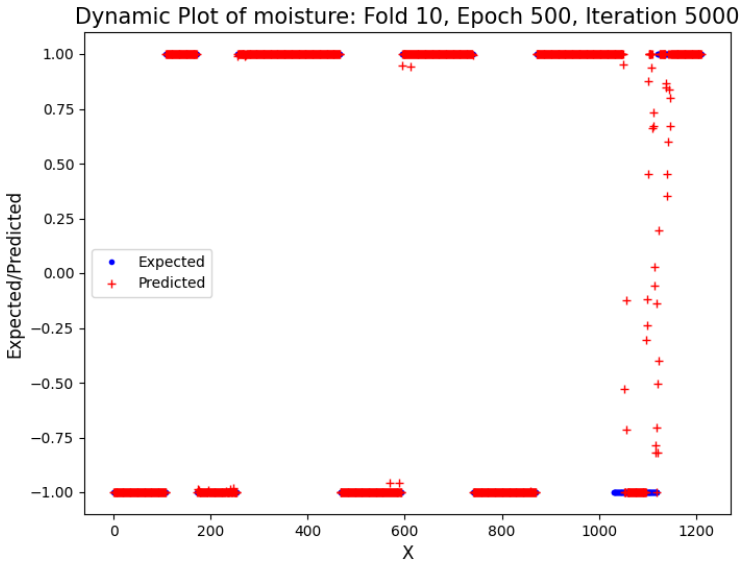


Figure 4.10: Advanced model visualization over the whole dataset. Trained with 80% of the data, full date range.

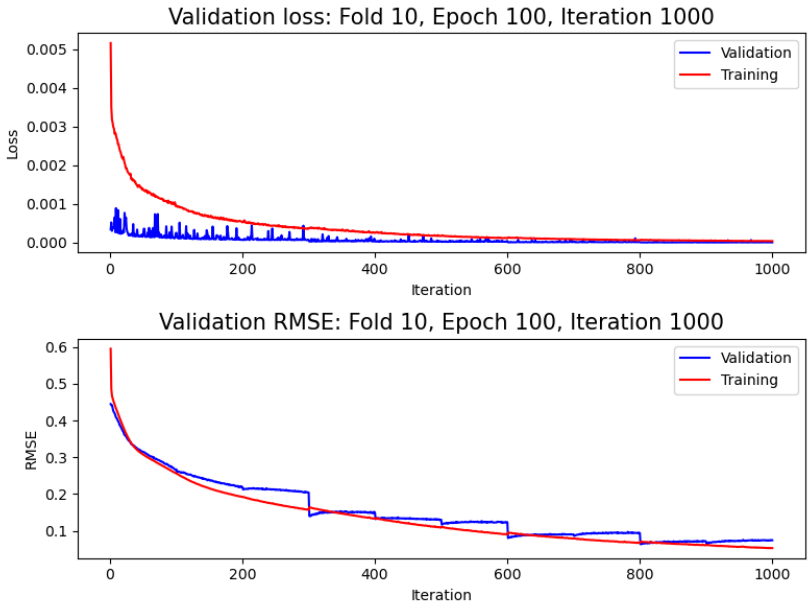


Figure 4.11: RMSE and loss variations. Advanced model trained with 80% of the data, full date range.

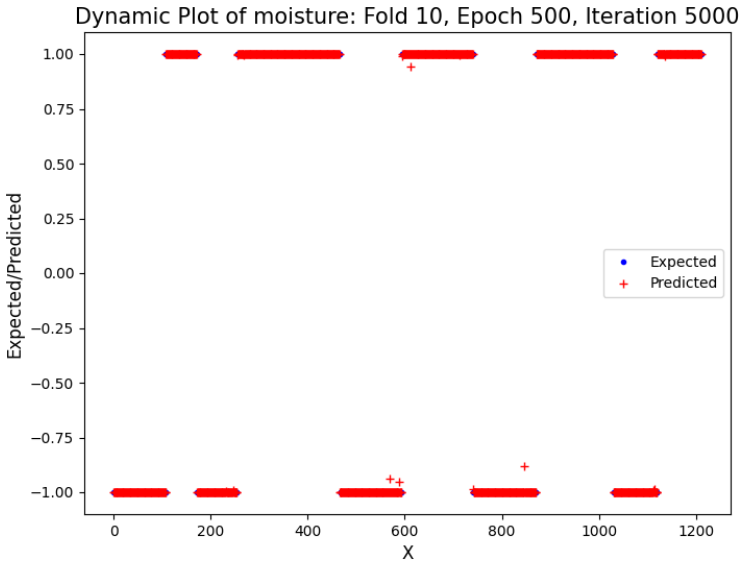


Figure 4.12: Advanced model visualization over the whole dataset. Trained with 100% of the data, full date range.

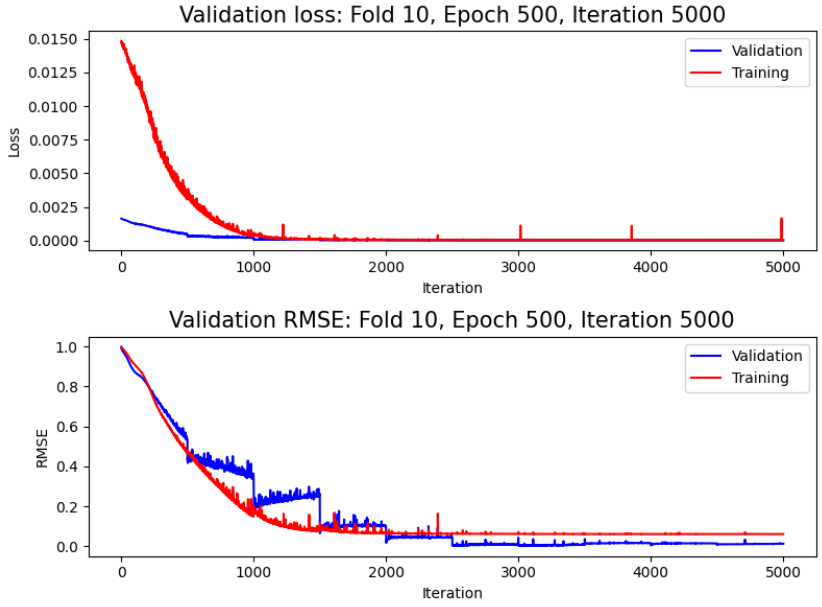


Figure 4.13: RMSE and loss variations. Advanced model trained with 100% of the data, full date range.

4.3 Status Now Finder results

Regarding the results of the finder, the results obtained with the implemented search type can be discussed.
By employing the search in time, so the search of the time windows that offer the best training results, it was found that time series that include distinguishable watering events, especially after the plant has been stressed by keeping it dry, provide the best training results. While it is not possible to know for sure the reason, it is possible that the large fluctuations of the measured variables caused by the rare watering events may ease the predictions as the noise in the data is less predominant.
Using the Neural Network shape sweep type instead, it has been found that the obtained performance is better by having more than ~100 neurons in the hidden layers and by having one or two hidden layers, depending on the situation. Increasing the number of layers of neurons per layer can improve the performance of the neural network. However, it is evident that there are diminishing returns by increasing the number of neurons or the number of layers too much. For example, employing over ~1000 neurons or over 4 layers is not worth the additional complexity. In addition to the complexity problem, overfitting is also more likely to occur. The following graphs show the results of a sweep over 1 to 5 hidden layers and over 8 to 4096 neurons per layer with a power of 2 steppings. The used data processing and training settings were identical to the ones used in the first examples of the Status Nowe section of the results (the full range of the moisture is allowed). Figure 4.14 shows the improvement of the RMSE by adding more neurons per each layer and by increasing the number of layers. However, it must be considered that the horizontal axis is actually logarithmic as the number of neurons grows as a power of 2. As such, it is clear that the improvement for a large number of neurons becomes slow with increasing numbers of neurons. It is also to be noticed that the RMSE improves with more layers, but the performance is very similar with one and two layers. However, this is not a result that is valid for every case, so it should always be checked if a single hidden layer provides enough performance. Figure 4.15 shows that the estimated complexity of the Neural Network grows a lot with high numbers of neurons and higher numbers of layers increase even more this effect. As the lines are compressed one onto the other on the left portion of the graph, figure 4.16 is provided. It is the same plot as 4.15, but the vertical axis is logarithmic. It is clear that the lines never cross, and the complexity always increases at the same rate. Finally, figure 4.17 shows a complexity-error product to compare the different combinations. For this particular case, it can be noticed that the models with a single hidden layer perform better than the others at similar complexity. The models with higher numbers of layers tend to perform more or less the same with similar complexity.

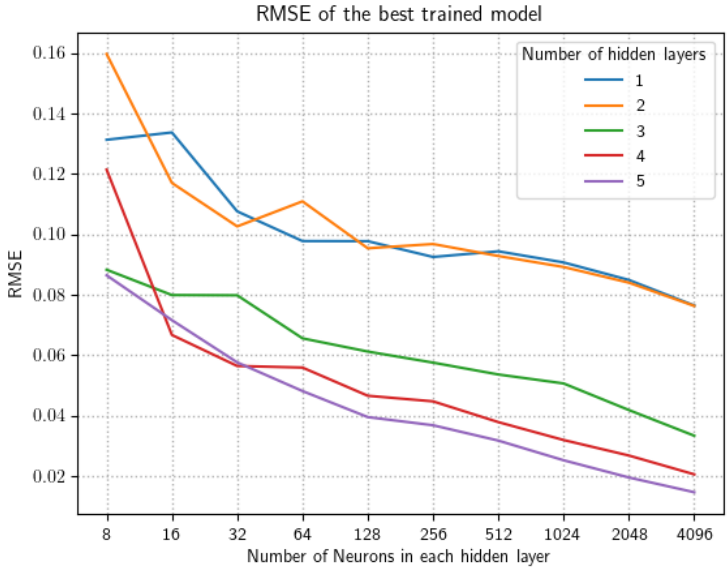


Figure 4.14: RMSE of the best trained model over the whole dataset depending on the number of layers and neurons.

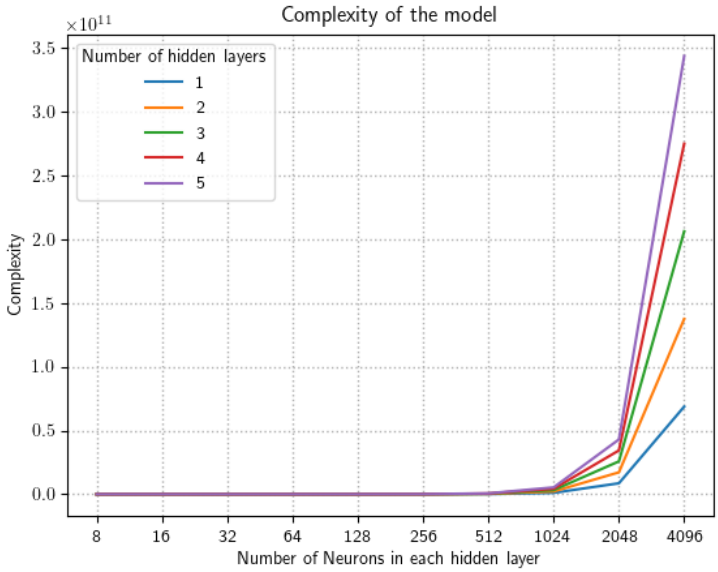


Figure 4.15: Complexity of the model depending on the number of layers and neurons.

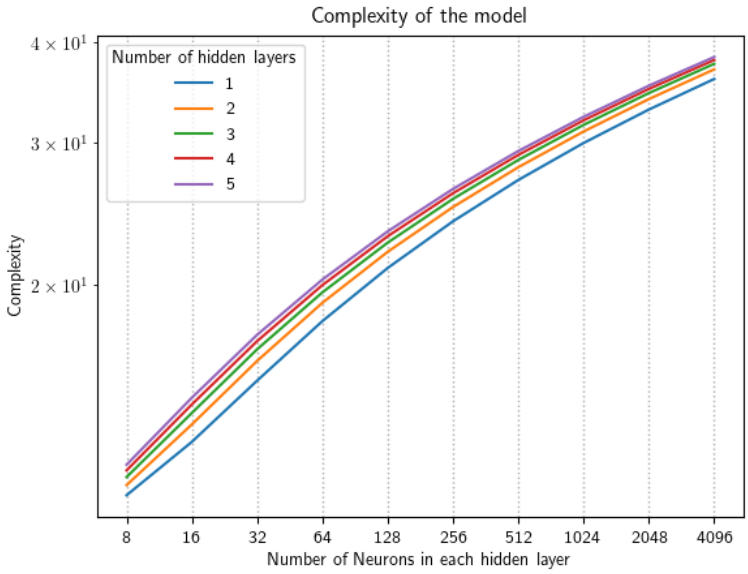


Figure 4.16: Complexity of the model depending on the number of layers and neurons. Log scale.

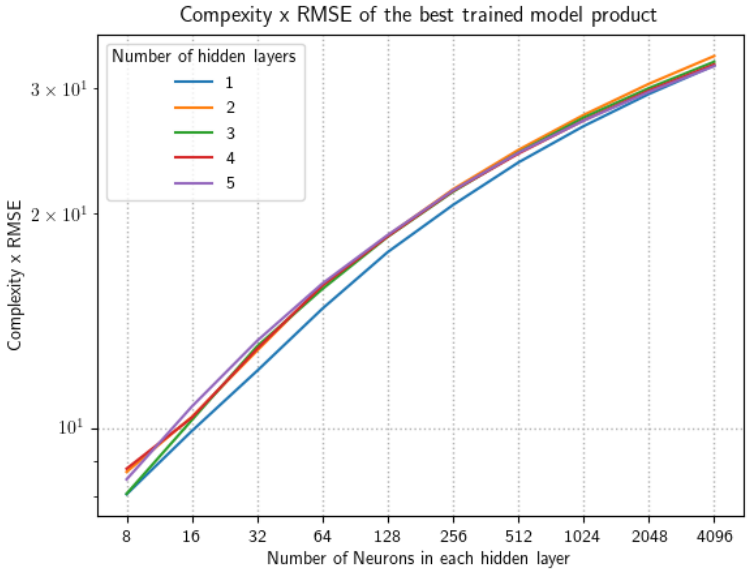


Figure 4.17: Complexity x RMSE of the best trained model over the whole dataset product depending on the number of layers and neurons.

These are all very useful plots suggested being created whenever such a choice has to be made.
Finally, by using the search type over the number of samples per parameter, it was found how many past samples should be picked approximately from each time series to predict and obtain the best results. It has been discovered that values that provide the best results are in the $20 \div 50$ range. In particular, the improvement is substantial when employing at least 20 past samples. Then the improvement becomes slower but constant up to 50 past samples. After about 50 past samples, no additional improvement has been recorded. The model actually starts to perform worse when using more than 50 past samples. Assuming that a sample is recorded every hour, by taking at least 20 hours worth of data, performance improves as the fluctuations due to the day-night cycles are taken into account. As the improvements stop after about 50 samples, i.e., about 50 hours, it seems like data older than two days is not useful anymore for predicting the plants' status. By taking too much past data, the model will also tend to memorize the samples, thus producing overfitting and worsening the model's performance.
The next figures show these concepts with experimental data. As before, the data processing and training settings employed for this sweep were the same that were used in the Status Now section of this chapter (not the advanced case). Figure 4.18 shows the behavior described above: fast decent up to 20 past samples, lower descent up to 50, and then no more improvements over 50. Figure 4.19 shows that in this case, the complexity of the model increases linearly (as an offset, actually) with the number of past samples being used. The final graph, figure 4.20, shows once again the complexity-error product, where the error is the RMSE of the best-trained model over the whole dataset. As it can be seen, besides some noise due to the randomness of the model's initial conditions, there is now a local minimum in the plot. This local minimum is located at a number of past samples of about 50. As such, in this case, using a value of about 50 would be the best choice unless a lower complexity is required due to other requirements.
It must also be reminded that these results are not generic and are only verified with a select amount of settings. However, it is suggested to use the data provided in the ranking files to make similar considerations and make the appropriate decisions and choices. Plots like the one presented are very useful to visualize these results, and all of these graphs have been, in fact, created by using the ranking files.

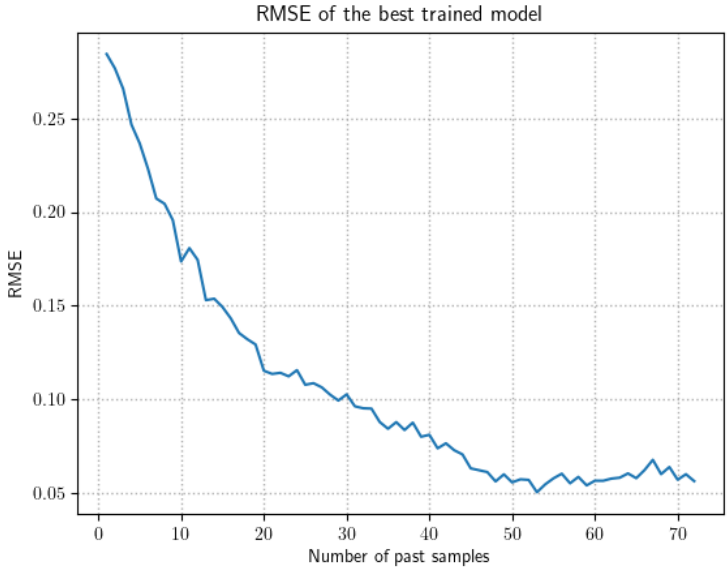


Figure 4.18: RMSE of the best trained model over the whole dataset depending on the number of past sampled.

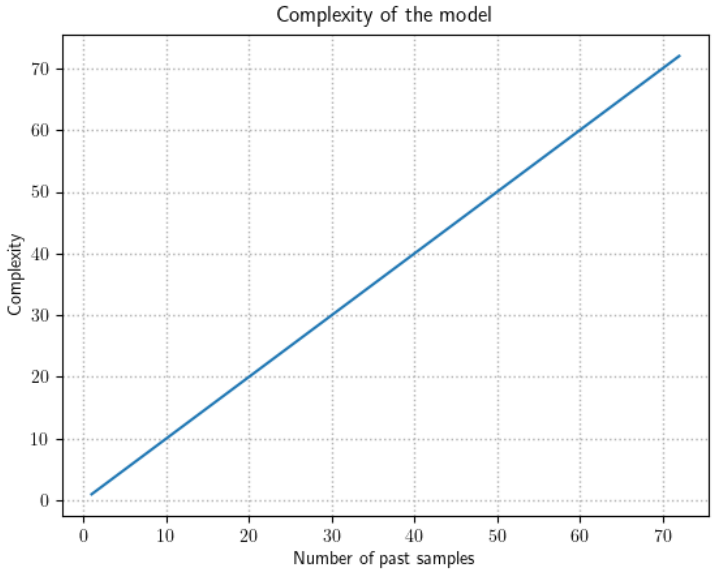


Figure 4.19: Complexity of the model depending on the number of past sampled.

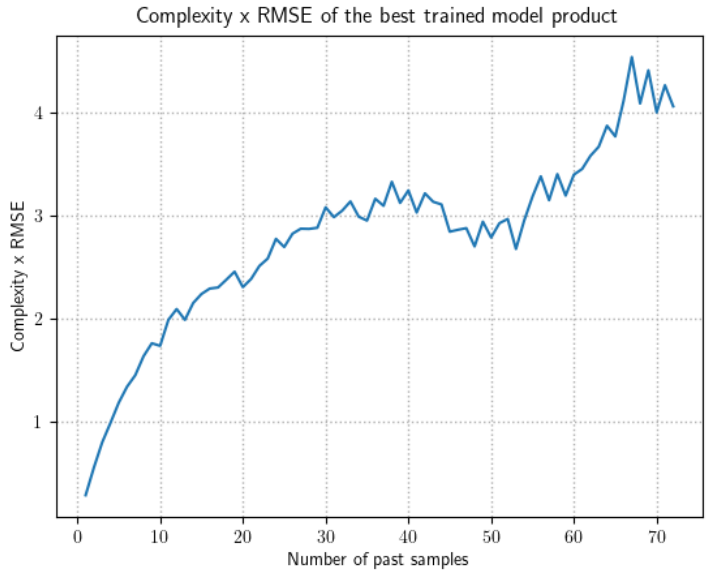


Figure 4.20: Complexity x RMSE of the best trained model over the whole dataset product depending on the number of past sampled.

Chapter 5

Conclusions

This thesis dissertation has described the Neural Network development framework for plant monitoring applications in all its facets. The introductory chapter has provided the reasons why plant monitoring is important, the work that is being done about it, and how this framework can be helpful for the objectives of the Plant Project of the MINES research group of Politecnico di Torino. While much work remains to be done for this project, this framework may help future developments and speed up the development of the necessary Machine Learning based plant monitoring software. The second chapter has provided insights into the Machine Learning field and its basics, which are necessary to understand the work performed to develop the framework. The third and most crucial chapter details how the framework has been structured, how its software has been implemented, and how it works. The final chapter, before this conclusive one, summarizes what can be accomplished by using this framework. The design choices have been presented through this document, but what has always been the focus during the development was the intention of creating this framework as a solid basis for future work. The software was designed to be modular and parametric to be a strong foundation for additional features and improved functionalities. While it would have been gratifying to implement many other features and start the work to move the ML-based software to the actual hardware, the work left to be done is still a lot, and it could not be performed under this thesis work.
Regarding the actual framework, there is a lot that remains open work. For example, implementing the sweeping algorithms for every variable used for training is possible with no necessary design work to be performed in advance. However, most of the remaining work is about actually using the framework. While many training efforts have been performed, there are still many possibilities to find the best settings both for the data processing and the Neural Network structure. Other work that may be done to improve the framework is about modifying the model generation so that it would be possible to employ not only Neural Networks but

also other types of models based on machine learning. From SVMs to Anomaly Detection algorithms, many possibilities are left to be explored.
On the work that is left to be done outside of the framework, that is even more. First, once a satisfactory model has been obtained, both the model and the data processing algorithms have to be implemented on a much better performing language, such as C, so that they can be executed on a microcontroller. The measurement software also has to be interfaced with the data processing algorithms, and the Machine Learning based software has to be interfaced with the transmission software. It would also be exciting to develop a software tool that can generate both the data processing algorithms and the model C code using the setting file that generated the satisfactory model and the saved file with trained model parameters. This "conversion" software would not be simple and would have to take advantage of functions to be developed in C separately. However, it would simplify enormously the deployment of the software into the microcontroller systems.
On a final remark, I would like to say that while there is still an enormous amount of work to be done, the path forward is clear. Hopefully, this framework will prove helpful for the end goal of developing the plant status monitoring systems for the years to come. A time where food production may become predictable and secure might still be off in the future, but, optimistically, the work that is being done now in the agri-food sector will have a positive impact on society and the planet as a whole.

Appendix A

Software usage

This appendix is dedicated to explaining how to use the framework and where to act in case it is needed to add some functionalities. Three main use cases will be explained:

- Using the Status Now functionality
  - Without the Status Now settings file
  - With the Status Now settings file
- Using the Status Now Finder functionality
  - Without the Status Now Finder settings file
  - With the Status Now Finder settings file
- Adding a new Status Now Finder search type

The framework is supposed to be used with the dispatcher that interfaces with all the functionalities. The dispatcher also allows the user to use the settings files without changing the code of the main source files, which is very useful to run multiple training or search efforts simultaneously. The dispatcher also allows the use of the functionalities without employing settings files, but in this case, the source code has to be changed unless it is wanted to use the default settings. It can still be useful to test if everything works fine on a new installation of the framework and if all the required libraries have been successfully installed. In this regard, a *reqirements.txt* file is also provided in the *src* folder so that the required libraries can be installed or updated with the *pip* utility. Each of the two following sections will explain how to use each functionality and the commands that can be used to launch them with or without the dispatcher. In the end, it will also be explained how to add a new search type to the Status Now Finder functionality.

Dispatcher help

Before moving on to the actual functionalities of the framework, this is just a reminder that by running the following command while in the *src* folder, the dispatcher will provide hints about its usage:

```
python status_now_dispatcher.py -h
```

The help text that is provided is shown in listing A.1.

Listing A.1: Dispatcher help.

```
usage: status_now_dispatcher.py [-h] [-f FILE] [-b BATCH] script

Run the Status Now scripts from a Shell.

positional arguments:
  script                Select the script to run. Available scripts:
    status_now, status_now_finder

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  Run the selected script with the settings
    from the passed file. Pass the name of the chosen setting file
    located in the settings folder.
  -b BATCH, --batch BATCH
                        Choose to run the script in Batch mode if
    True (Interactive plots and graphs disabled) or not if False (
    Interactive plots and graphs enabled).
```

A.1 Using the Status Now functionality

The Status Now functionality is the easiest one to use. It can be used with or without a setting file. In case it is wanted to use it without the setting files, there are two ways to launch it. The first is to simply launch the *status_now.py* file with the Python interpreter, for example, by executing the following command in a shell:

```
python status_now.py
```

This will run Status Now functionality with the settings selected in the *main()* function of the *status_now.py* file and with the files also selected in that function. *main()* function imports a series of default settings implemented in the *status_now_settings.py* file. These settings employ the *Setting()* object described in the Software implementation chapter. Its usage will be explained later along with the explanation of the settings file, as the same parameters that are present in the settings files are also used in the *status_now_settings.py* file. In the *main()* function then it is also needed to select the correct training function, *statusTrain* or *statusTrainKFold* if the K-Fold cross-validation is preferred, and to list the input CSV files in the *files* list.
It is also possible to run the Status Now functionality in the same "default" fashion using the dispatcher. This can be done with the following command:

```
python status_now_dispatcher.py status_now
```

The Batch mode described in the previous chapters is also available here if there is no need to visualize the graphs at run time. To do this, it is possible to use the following commands:

```
python status_now_dispatcher.py status_now -b true
python status_now_dispatcher.py status_now -b yes
python status_now_dispatcher.py status_now -b y
```

By default, the Batch mode is disabled for the Status Now functionality if the default settings are used and no separate setting file is used. However, it is still possible to specify that the Batch mode should be disabled and to allow the interactive visualization of all the plots with one of the following commands:

```
python status_now_dispatcher.py status_now -b false
python status_now_dispatcher.py status_now -b no
python status_now_dispatcher.py status_now -b n
```

Moving on to the usage of the setting file, let us first see the typical content of such a file. In particular, the following listing A.2 shows the content of the available template setting file *"single_test_template.py"*. For the sake of simplicity, all the

import statements at the beginning and the comments have been removed. The real file includes instead many comments with suggestions about each parameter in the setting file so that it is easier to compile.

Listing A.2: Template of the settings file for the Status Now functionality.

```
root = Path(__file__).parent.parent.parent
files = [root / "Data" / "pianta1" / "data_export.csv",
         root / "Data" / "pianta2" / "data_export.csv",
         root / "Data" / "pianta3" / "data_export.csv",
         root / "Data" / "pianta4" / "data_export.csv"]
plant_list = ['pianta1', 'pianta2', 'pianta3', 'pianta4']
useKFold = True
single_test_setting = Setting()
single_test_setting.start_date = "2021-03-23 09:30:00"
single_test_setting.end_date = "2021-07-30 09:30:00"
single_test_setting.dataset_settings = []
single_test_setting.plants_to_use_list = []
single_test_setting.remove_n = 0
single_test_setting.testMode = False
single_test_setting.params_to_use = []
single_test_setting.outputs = []
single_test_setting.n_samples_per_parameter = 24
single_test_setting.n_overlap_of_samples = 23
single_test_setting.learning_rate = 1e-3
single_test_setting.momentum = 0.9
single_test_setting.batch_size = 32
single_test_setting.generateModel()
single_test_setting.loss_fn = nn.MSELoss()
single_test_setting.epochs = 50
single_test_setting.folds = 10
single_test_setting.trainWithAllData = False
```

First, it is important to remark that it is advised to leave each statement in the order presented above. While not strictly necessary for all the parameters, a few of them require that sorted for them to work correctly. The first line defines the *root* folder of the framework, taking advantage of the *pathlib* library. It will be used to get the paths of the CSV data files. This line should be left as is; however, it is also possible to define other folders in case an external data file is needed.

The *files* parameter is a list where each element should be a path to a particular CSV file with Plants data in it, with the format specified in the previous chapters (first line with the names of the tame series, each subsequent line with a set of measurements obtained every hour). The suggested format to pass these paths is the one used for the four default files. The "Data" folder of the framework is used to contain all the plants' data. For better management, the "Data" directory should contain a folder for each plant (with a specific plant name identifier as the folder name), and

each folder should contain the CSV file related to that plant. It is suggested to use the pathlib way of providing paths so that it can manage the paths whether the framework is being run on a Linux system or a Windows system. As shown with the default files, the "/" division symbol can be used to define the subfolders and the files it contains, with the string containing only the names of the files and the folders.

The *plant_list* parameter is another list that should have the same size of the *files* parameter and should contain a string for every CSV file in the *files* list. The strings should be the identifiers of the plants to be associated with the data in the files. Each string will be associated with the file in the same position in the *files* list. There should not be more than one file associated with each plant, otherwise ordering and Dataset generation issues could occur. The identifiers specified in this list are the same that should be used in any other parameter that requires a plant identifier.

The *useKFold* parameter is a simple boolean parameter used to choose whether to perform training with the normal validation or use K-Fold cross-validation instead. To unable K-Fold cross-validation *useKFold* should be set to *"True"*, otherwise if normal validation is preferred it should be set to *False*.

The *single_test_setting* parameter is for creating a *Setting* object, and this line should not be modified. Every subsequent line of the template is related to the setup of every parameter in the *Setting* object. Refer to the Setting Class subsection of the Status Now section in the Software Implementation chapter for additional information about this object.

The *start_date* parameter of the *Setting* object is a string that should contain the chosen start date of the dataset, in a *"YYYY-MM-DD hh:mm:ss" format*. If any of the CSV files contain rows with dates antecedent to the one specified with this parameter, they are discarded.

The *end_date* parameter of the *Setting* object is another string that, however, should contain the chosen end date of the dataset, in a *"YYYY-MM-DD hh:mm:ss"* format. If any of the CSV files contain rows with dates subsequent to the one specified with this parameter, they are discarded.

The *dataset_settings* parameter of the *Setting* object is a list of standardized dictionaries, where each dictionary has a predefined set of key-value pairs. Each of these dictionaries is used to generate a time series from a column on the input CSV files. Multiple operations are performed on a selected data column of each CSV file depending on the key-value pairs. These operations are, in order, constraint, filtering, transformation, and normalization. The values in the column

are constrained first so that values too high or too low are capped to known limits. Then a filter can be applied to the whole data series. Examples of such filters may be moving averages or FIR/IIR filters. The successive step is the transformation. At this step, a function of choice can be applied to every sample in the data series. For example, a logarithm or a square function can be applied. The final step is the normalization in the $[-1 \div 1]$ range, which is an important step so that all the features are on the same scale. The following list shows all the keys and what their values should contain:

- *"param_name"*:

  String with the name of the new time series to be created. It should be a unique name, so it should not be repeated in any of the other dictionaries.

- *"input_name"*:

  String with the name of the column of the CSV files that should be used to generate the new time series. Different time series can use the same *"input_name"*, so it can be used in other dictionaries.

- *"norm_data_range"*:

  Integer or float value with the range of the data to be used for normalization to the $[-1 \div 1]$ range. If set to *"None"* it will be computed automatically; however, only the data of the plant that is being processed will be used, so different plants with different ranges will all be normalized to the same $[-1 \div 1]$ range even though the initial ranges were different. For this reason, it is suggested to check the range and set it here manually. The range to be provided should be the one obtained after all the other steps, so after the constraint, the filtering, and the transformation. For example, if the max value in the series is 25 and the minimum value is -15, this value should be set to 40.

- *"norm_data_median"*:

  Integer or float value with the mid-range of the data to be used for normalization to the $[-1 \div 1]$ range. If set to *"None"* it will be computed automatically; however, only the data of the plant that is being processed will be used, so different plants with different mid-ranges will all be normalized to the same $[-1 \div 1]$ range even though the initial mid-ranges were different. For this reason, it is suggested to check the mid-ranges and set it here manually. The mid-ranges to be provided should be the ones obtained after all the other steps, so after the constraint, the filtering, and the transformation. The mid-range is defined as the maximum value in

the data series plus the minimum value in the time series, all divided by two. So, for example, if the max value in the series is 25 and the minimum value is -15, this value should be set to 5.

- *"transform_function"*:

  Python function to be applied to every data sample of the series. One of the functions defined in the *utilities.py* source file or any of the built-in functions may be used, but it is also possible to pass lambda functions or functions defined elsewhere in the file with the def keyword. For example, if it is wanted to apply the absolute value function to the time series, the value corresponding to this key should be set to *abs*.

- *"transform_function_kwargs"*:

  Python dictionary with the keyworded arguments of the function passed with the "transform_function" key-value couple. Each key in this dictionary should be the name of a keyworded argument of the function, while each value should contain the content of that parameter. For example if we want to apply a function named *funct* that takes two parameters named *a* and *b* besides the data (function defined as *funct(data, a, b*), if we want to set a to 5 and b to -3, we should set the *"transform_function_kwargs"* paramter to: *{"a":5, "b":-3}*.

- *"filter"*:

  Filtering function to be applied to the time series. Unlike the transformation function, this function should take into account the neighboring values of each sample in the series. The *simpleMovingAverage* and the *exponentialMovingAverage* functions have been implemented in the *utilities.py* source file, but, as for the other function, it is also possible to pass lambda functions or functions defined elsewhere in the file with the def keyword.

- *"filter_kwargs"*:

  Python dictionary with the keyworded arguments of the "filter" function. As in the other case, each key in this dictionary should be the name of a keyworded argument of the function, while each value should contain the content of that parameter. For example, if the *exponentialMovingAverage* function is employed, a possible value for this *"filter_kwargs"* key is *{"N":24, "alpha":0.5}*, where *"N"* represents the window and *"alpha"* represents the smoothing factor, and should be set in a range between 0 and 1.

- *"constrain_min"*:

  Integer or float value used to constrain the time series values. This value will replace any value present in the time series that is lower than this limit.

- *"constrain_max"*:

  Integer or float value used to constrain the time series values. This value will replace any value present in the time series that is higher than this limit.

Finally, here is an example of the content of a *dataset_settings* parameter:

```
[
    {
        "param_name": "moisture",
        "input_name": "Moisture [KPa]",
        "norm_data_range": util.log10abs(500),
        "norm_data_median": util.log10abs(500) / 2,
        "transform_function": util.log10abs,
        "transform_function_kwargs": {},
        "filter": util.simpleMovingAverage,
        "filter_kwargs": {"N": 24},
        "constrain_min": -500,
        "constrain_max": -1
    },
    {
        "param_name": "hour",
        "input_name": "Date",
        "norm_data_range": 23,
        "norm_data_median": 11.5,
        "transform_function": util.getHour,
        "transform_function_kwargs": {},
        "filter": None,
        "filter_kwargs": {},
        "constrain_min": None,
        "constrain_max": None
    },
    {
        "param_name": "impedance_phase",
        "input_name": "impedance_phase",
        "norm_data_range": -37.5 - (-52.5),
        "norm_data_median": -45,
        "transform_function": None,
        "transform_function_kwargs": {},
        "filter": util.exponentialMovingAverage,
        "filter_kwargs": {"N": 24, "alpha": 0.4},
        "constrain_min": -52.5,
```

```
        "constrain_max": -37.5
    }
]
```

The *plants_to_use_list* parameter of the *Setting* object is a list used to select which plants should be used for training. It should be a subset of the *plant_list* parameter. If can contain from one up to all of the plant identifiers specified before. While this variable may seem redundant, this is done so that the *files* list and the *plant_list* list can be left untouched and contain all the files that may be needed. This is pretty much only done to prevent errors so that only a list should be changed to add or remove plants instead of two.

The *remove_n* parameter of the *Setting* object is an integer used to remove some rows at the end of a dataset manually. This feature has been kept only for backward compatibility purposes and should not be actively used as the *end_date* parameter can be used for the same purpose in a much more intuitive way.

The *testMode* parameter of the *Setting* object is a boolean that should be used only for testing purposes, so it should be set to *False*. It should be set to *True* only to test if the framework works when no plants data is available. If set to *True*, a "fake" dataset is created with two input variables named *"testin1"* and *"testin2"* and two output variables named *"testout1"* and *"testout2"*.

The *params_to_use* parameter of the *Setting* object is a list of the names of the input features to use among all the time series created with the *dataset_settings* list of dictionaries. Names used in this list should not be used in the *outputs* list.

The *outputs* parameter of the *Setting* object is a list of the names of the outputs to use among all the time series created with the *dataset_settings* list of dictionaries. Names used in this list should not be used in the *params_to_use* list.

The *n_samples_per_parameter* parameter of the *Setting* object is an integer used to specify how many past samples should be used for making a current prediction. A *n_samples_per_parameter* value of one would use only the latest measured data of all the input features, while a value of 24, for example, would use the data of the past 24 hours (assuming measurements occur every 60 minutes). This means that there would be 24 input values provided to the Neural Network for each input feature. The minimum allowed value is one, but no upper limit is set.

The *n_overlap_of_samples* parameter of the *Setting* object is an integer used to specify the data overlap between multiple training and test Dataset rows. As the *n_samples_per_parameter* parameter allows to take past data for any

prediction, two subsequent predictions could share a lot of data. For example, with a *n_overlap_of_samples* value of 24 and with no overlap checking, the subsequent prediction would reuse 23 hours of data. The *n_overlap_of_samples* parameter allows the user to adjust that by specifying how many samples of overlap for each feature are allowed. The allowed range of this parameter is $0 \leq n\_overlap\_of\_samples < n\_samples\_per\_parameter$, where if it is set to zero no overlap is allowed and if set to $n\_samples\_per\_parameter - 1$ the subsequent samples are allowed to fully overlap. Higher numbers of this value allow for many more training and test samples; however, the samples may be quite similar.

The *learning_rate* parameter of the *Setting* object is a floating-point number that should be in the $0 < momentum < 1$ range. The learning rate controls the change rate of the parameters, depending on their gradient calculated with respect to the training set inputs. A good starting point usually is $1.0e - 3$.

The *momentum* parameter of the *Setting* object is a floating-point number that should be in the $0 \leq momentum \leq 1$ range. If it is wanted to just use the learning rate, the momentum can be set to 0. Otherwise, a *momentum* of 0.9 is a good starting point.

The *batch_size* parameter of the *Setting* object is an integer used to define the batch used to be used during training and testing. It is essential during training as the Stochastic Gradient Descent algorithm trains the model with batches of data of this size. A starting point of 32 is suggested, but higher numbers can also be used. While higher numbers can provide better results, the complexity can increase a lot.

The *generateModel()* method of the *Setting* object should be used, as name suggest, to generate the Neural Network model. How this method works in detail was explained in the Setting Class subsection of the Status Now section in the Software Implementation chapter. So now it will only be explained how to use it. While it is possible just to call the *generateModel()* method with no parameters (a default Neural Network will be created with two hidden layers with 16 neurons each, with ReLu activation functions for the first hidden layer and linear activation functions for the second) this is not the best way to use it. The suggested way to use it is to pass a *nn.Sequential* object with the internal structure of the Neural Network. Each parameter of this object should be a layer in the Neural Netowork, from inputs to outputs. So let us take the example also provided as a comment in the template:

```
single_test_setting.generateModel(
    nn.Sequential(
        nn.Linear(single_test_setting.nn_inputs(), 32),
```

```
            nn.ReLU(),
            nn.Linear(32, 32),
            nn.ReLU(),
            nn.Linear(32, single_test_setting.nn_outputs())
        )
)
```

In this case, a Neural Network with 3 hidden layers with 32 neurons is generated. The first two hidden layers have ReLU activation functions, while the last one only has a linear activation to get the full output range. If, for example, we wanted to change the number of neurons in the first hidden layer, we would have to change the second parameter of the first *Linear* object and the first parameter of the second *Linear* object. As it can be seen, the *nn_inputs()* and *nn_outputs()* methods of the *Setting* object are used so that they are computed automatically depending on what was provided previously in the other settings. This is one of the reasons why there should be an order with the setting of all the parameters. The available layers as well as the activation function layers can be found in the online documentation[28] and have also been listed in the Neural Network model subsection of the Machine Learning "foundation" section of the Software Implementation chapter.

The *loss_fn* parameter of the *Setting* object should be used to provide a loss function to be used during training and testing of the model. A Loss function among those provided by PyTorch should be used. These functions are available in the online documentation[28] and are also listed in the Loss Function subsection of the Machine Learning "foundation" section of the Software Implementation chapter. The suggested Loss functions (that have also been tested) are the *nn.MSELoss()*, *nn.CrossEntropyLoss()*, *nn.L1Loss()* provided in the *nn* module of the *torch* library. In the example *nn.MSELoss()* is used to show how to pass one of these Loss function objects.

The *epochs* parameter of the *Setting* object is an integer used to define for how many epochs should the training be performed. Higher numbers do not always provide better results. However, it is possible to check the validation error plots to see if a higher epoch number can provide better results or if the error is already stable with a lower epoch count. If K-Fold cross-validation is not used, this is the total number of epochs, while if K-Fold cross-validation is employed, this is the number of epochs for each fold, so the total number of iterations will be equal to the value of this parameter multiplied by the *folds* parameter, explained next.

The *folds* parameter of the *Setting* object is an integer used to define in how many folds should the Dataset be split in. It basically defines the K of the K-Fold algorithm. Suggested values of K are in the 5 to 10 range. In case K-Fold is not

used, this parameter can be set to *None*

The *trainWithAllData* parameter of the *Setting* object is a boolean parameter that lets the user decide if the training should occur with all the available data or if some of it should be held exclusively for the test dataset. In case this parameter is set to *True*, 100% of the data is used for training, but a small portion at the end of the dataset is also reused for the test Dataset. In case it is set to *False*, the test Dataset will have the same size as with the other case, but that data will not be used for training.

After every parameter in the setting file has been filled, it is possible to launch the training. To do that, it is important to remember that the settings files have to be placed in the *settings* folder within the *src* directory for the dispatcher to find them. It is also where the templates are stored.
That being said, assuming that the new setting file name is *"file_name.py"*, it is possible to use either of the following commands, as the extension of the file can also be omitted. The real file, however, must have a *".py"* extension.

```
python status_now_dispatcher.py status_now -f file_name.py
python status_now_dispatcher.py status_now -f file_name
```

The Batch mode described previously is now enabled by default when a file is passed.
However, it is also possible to make sure that it is enabled by forcing it with any of the following commands:

```
python status_now_dispatcher.py status_now -f file_name.py -b true
python status_now_dispatcher.py status_now -f file_name.py -b yes
python status_now_dispatcher.py status_now -f file_name.py -b y
```

If it is wished to disable batch mode and check the graphs interactively at run time, that can be done with either of the following commands:

```
python status_now_dispatcher.py status_now -f file_name.py -b false
python status_now_dispatcher.py status_now -f file_name.py -b no
python status_now_dispatcher.py status_now -f file_name.py -b n
```

While the file names were always passed in the previous examples with their extension, they are still not strictly necessary, of course.

A.2 Using the Status Now Finder functionality

The Status Now Finder functionality is the second functionality of the framework and is used to search for the best values of specific settings. As with Status Now, it can be used with or without a setting file. In case it is wanted to use it without the setting files, there are two ways to launch it in this case too. The first is to simply launch the *status_now_finder.py* file with the Python interpreter by executing the following command in a shell while in the *src* directory:

```
python status_now_finder.py
```

This will run Status Now Finder functionality with the settings selected in the *main()* function of the *status_now_finder.py* file and with the files also selected in that function. In this *main()* it is possible to choose the search type by setting the *search_type* variable to the correct search identifier, whether to use K-Fold cross validation or not (with the *useKFold* variable, and whether to create the rankings with the data of the whole dataset or of only the the test database (with the *decideWithFullData* variable.
It is also possible to run the Status Now functionality in the same "default" fashion using the dispatcher. This can be done with the following command:

```
python status_now_dispatcher.py status_now_finder
```

The Batch mode described in the previous chapters is also available here if there is no need to visualize the graphs at run time. To do this, it is possible to use the following commands:

```
python status_now_dispatcher.py status_now_finder -b true
python status_now_dispatcher.py status_now_finder -b yes
python status_now_dispatcher.py status_now_finder -b y
```

By default, the Batch mode is disabled for the Status Now Finder functionality if the default settings are used and no separate setting file is used.
However, it is still possible to specify that the Batch mode should be disabled and to allow the interactive visualization of all the plots with one of the following commands:

```
python status_now_dispatcher.py status_now_finder -b false
python status_now_dispatcher.py status_now_finder -b no
python status_now_dispatcher.py status_now_finder -b n
```

Moving on to the settings file for the Status Now Finder functionality, listings A.3 and A.4 show the content of the the available template setting file *"finder_template.py"*. The import statements, as well as the comments, have been removed for simplicity. The first part of the setting shown in listing A.3 shows an

initial part similar to the setting file for the Status Now functionality and then a series of settings related to the search. The second and last part shown in listing A.4 contains instead the same settings used in the other type of setting file.

Listing A.3: First part of the template of the settings file for the Status Now Finder functionality.

```
root = Path(__file__).parent.parent.parent
files = [root / "Data" / "pianta1" / "data_export.csv", root / "Data" / "pianta2" / "data_export.csv",
         root / "Data" / "pianta3" / "data_export.csv", root / "Data" / "pianta4" / "data_export.csv"]
plant_list = ['pianta1', 'pianta2', 'pianta3', 'pianta4']
search_type = "" #"time1", "nnShape1", "nnShape2", "nSamplesPerParameter1"
useKFold = True
decideWithFullData = True
finder_setting = FinderSetting(search_type)
finder_setting.RMSE_threshold_to_save = float("inf")
finder_setting.loss_threshold_to_save = float("inf")
if search_type == "time1":
    finder_setting.start_date = ""
    finder_setting.end_date = ""
    finder_setting.window_min = 0
    finder_setting.window_max = 0
    finder_setting.window_change_step = 0
    finder_setting.window_move_step = 0
    finder_setting.min_samples_for_training = 1
    finder_setting.plants_to_cycle = []
if search_type == "nnShape1":
    finder_setting.layer1_neurons_list = []
    finder_setting.layer2_neurons_list = []
if search_type == "nnShape2":
    finder_setting.n_layers_values = []
    finder_setting.n_neurons_values = []
    finder_setting.n_samples_per_parameter_min = 0
    finder_setting.n_samples_per_parameter_max = 0
    finder_setting.n_samples_per_parameter_step = 1
if search_type == "nSamplesPerParameter1":
    finder_setting.n_samples_per_parameter_min = 0
    finder_setting.n_samples_per_parameter_max = 0
    finder_setting.n_samples_per_parameter_step = 1
```

As with the settings files for Status Now, it is advised to leave each statement in the order presented above. While not strictly necessary for all the parameters, a few require that sorted to work correctly. The first line defines the *root* folder of the framework, taking advantage of the *pathlib* library. It will be used to get the paths of the CSV data files. This line should be left as is; however, it is also possible to define other folders in case an external data file is needed.

The *files* parameter is a list where each element should be a path to a particular CSV file with Plants data in it, with the format specified in the previous chapters (first line with the names of the tame series, each subsequent line with a set of measurements obtained every hour). The suggested format to pass these paths is the one used for the four default files. The "Data" folder of the framework is used to contain all the plants' data. The "Data" directory should contain a folder for each plant (with a specific plant name identifier as the folder name) for better management. Each folder should contain the CSV file related to that plant. It is suggested to use the pathlib way of providing paths, so that it can manage the paths whether the framework is being run on a Linux system or a Windows system. As shown with the default files, the "/" division symbol can be used to define the subfolders and the files it contains, with the string containing only the names of the files and the folders.

The *plant_list* parameter is another list that should have the same size of the *files* parameter and should contain a string for every CSV file in the *files* list. The strings should be the identifiers of the plants to be associated with the data in the files. Each string will be associated with the file in the same position in the *files* list. There should not be more than one file associated with each plant, otherwise ordering and Dataset generation issues could occur. The identifiers specified in this list are the same that should be used in any other parameter that requires a plant identifier.

The *search_type* parameter is a string that should contain the identifier of the search type. The default available search type identifiers are *"time1"* for the search of the best timeframe, *"nnShape1"* for the search of the best combination of the amounts of neurons in a Neural Network with one or two hidden layers, *"nnShape2"* for the search of the best number of hidden layer and the best amount of neurons per layer, and *"nSamplesPerParameter1"* for the search of the best amount of past data that should be used to make a prediction. If a new search type is implemented, it will be possible to set this variable to the new identifier.

The *useKFold* parameter is a boolean value used to choose which type of valdiation should be used for training and testing. If set to *True* K-Fold cross-validation will be used (the *statusTrainKFold()* function will be called), otherwise if set to *False* the normal validation will be employed (the *statusTrain()* function will be called).

The *decideWithFullData* parameter is a boolean value used to choose how to compute the rankings. If set to *True* the rankings will be based on the RMSE and the loss values computed over the whole dataset (training plus test datasets). In contrast, if set to *False* the rankings will be based solely on the RMSE and the

loss values computed over the test dataset.

The *finder_setting* parameter is a *FinderSetting* object constructed with the specified *search_type*. The *FinderSetting* is a simple container that acts as a struct. It is used to pass to the statusFinder() function the sweep settings as well as the constant training settings. The line of code where this object is instantiated should not be changed.

The *RMSE_threshold_to_save* parameter of the *finder_setting* object is a threshold used in combination with the *loss_threshold_to_save* parameter to not save the data relative to models that do not reach the specified performance level. If the RMSE of the model is larger than the threshold, no files about it are saved, and also, it is not added to any of the rankings. Its value can be either an integer or a float value. It can also be set to infinity by setting it to *float("inf")*, in which case any RMSE value is allowed.

The *loss_threshold_to_save* parameter of the *finder_setting* object is a threshold used in combination with the *RMSE_threshold_to_save* parameter to not save the data relative to models that do not reach the specified performance level. If the average loss value of the model is larger than the threshold, no files about it are saved, and also, it is not added to any of the rankings. Its value can be either an integer or a float value. It can also be set to infinity by setting it to *float("inf")*, in which case any average loss value is allowed.

Next there are a set if *if* statements that check which *search_type* was selected. Depending on the type, onlt the variables related to that search type are set. This is important as the *FinderSetting* object only contains the variables needed for the search type that was chosen. For the unused variables of the other search types, it is possible to just leave them uncompiled or to just remove them so that, for example, if *"nnShape1"* is chosen only the *finder_setting.layer1_neurons_list* parameter and the *finder_setting.layer2_neurons_list* parameter are left in the setting file.

Regarding the specific parameters necessary for each search type, to avoid a long repetition, please refer to the Time search subsection of the Status Now Finder section of the Software Implementation chapter for the *"time1"* search type, the Neural Net Shape search subsection of the Status Now Finder section of the Software Implementation chapter for the *"nnShape1"* and the *"nnShape1"* search types, and the Past samples search subsection of the Status Now Finder section of the Software Implementation chapter for the *"nSamplesPerParameter1"* search type.

Regarding instead the "constant" settings reported next in listing A.4, please refer to the previous section of this Appendix, as the paramters are exactly the same.

Listing A.4: Second part of the template of the settings file for the Status Now Finder functionality.

```
finder_setting.common_settings = Setting()
finder_setting.common_settings.start_date = "2021-03-23 09:30:00"
finder_setting.common_settings.end_date = "2021-07-30 09:30:00"
finder_setting.common_settings.dataset_settings = []
finder_setting.common_settings.plants_to_use_list = []
finder_setting.common_settings.remove_n = 0
finder_setting.common_settings.testMode = False
finder_setting.common_settings.params_to_use = []
finder_setting.common_settings.outputs = []
finder_setting.common_settings.n_samples_per_parameter = 24
finder_setting.common_settings.n_overlap_of_samples = 23
finder_setting.common_settings.learning_rate = 1e-3
finder_setting.common_settings.momentum = 0.9
finder_setting.common_settings.batch_size = 32
finder_setting.common_settings.generateModel()
finder_setting.common_settings.loss_fn = nn.MSELoss()
finder_setting.common_settings.epochs = 50
finder_setting.common_settings.folds = 10
finder_setting.common_settings.trainWithAllData = False
```

After every parameter in the setting file has been filled, it is possible to launch the search. To do that, it is important to remember that the settings files have to be placed in the *settings* folder within the *src* directory for the dispatcher to find them. This folder is also where the templates are stored. That being said, assuming that the new setting file name is *"file_name.py"*, it is possible to use either of the following commands, as the extension of the file can also be omitted. The real file, however, must have a *".py"* extension.

```
python status_now_dispatcher.py status_now_finder -f file_name.py
python status_now_dispatcher.py status_now_finder -f file_name
```

The Batch mode described previously is now enabled by default when a file is passed.
However, it is also possible to make sure that it is enabled by forcing it with any of the following commands:

```
python status_now_dispatcher.py status_now_finder -f file_name.py -b true
python status_now_dispatcher.py status_now_finder -f file_name.py -b yes
python status_now_dispatcher.py status_now_finder -f file_name.py -b y
```

If it is wished to disable batch mode and actually check the graphs interactively at run time, that can be done with either of the following commands:

```
python status_now_dispatcher.py status_now_finder -f file_name.py -b false
python status_now_dispatcher.py status_now_finder -f file_name.py -b no
python status_now_dispatcher.py status_now_finder -f file_name.py -b n
```

While the file names were always passed in the previous examples with their extension, they are still not strictly necessary, of course.

A.3 Adding a new Status Now Finder search type

This final section of the appendix is dedicated to roughly explaining how the *status_now_finder.py* source file should be modified in order to add a new search type. First, it should be clear how the functionality works in general. The Status Now Finder section of the Software Implementation chapter explains in detail the algorithm and how it works. In that section, it was explained that a sort of switch-case process was created using cascaded if and elif statements. If a new search type has to be added, everything that needs to be added is a new "case" of this switch-case, in the form of a new elif statement and its block of code to has to be executed if the condition turns out to be true. The if statement should check if the *search_type* parameter corresponds to the identifier chosen for the new search type. Then the rest of the code should be, in theory, very similar to one of the other search types. As such, it is suggested to copy one of the other search types and modify them as needed. In particular, if the sweep should be performed over a single variable, it is suggested to copy the *"nSamplesPerParameter1"* search algorithm as it is based on a single loop. If the sweep should instead be performed over two variables, it is suggested to copy either the *"nnShape1"* search algorithm or the *"nnShape2"* algorithm as they are based on two nested loops. Finally, if the sweep should instead be performed over three variables, it is suggested to copy the *"time1"* algorithm as it includes three nested loops. It is also recommended not to go higher than three inner loops to keep complexity not too high.
To provide a practical example on how a new search type should be structured, listing A.5 shows a Python based pseudocode with a single sweeping loop based on the "nSamplesPerParameter1" search type.

Listing A.5: Second part of the template of the settings file for the Status Now Finder functionality.

```
elif search_type == "newSearchType":
    # if no setting for the FinderSetting object has been passed
    if finder_setting is None:
        # Create the default sweep setting
        ...

        # Choose the thresholds to save the data to file
        RMSE_threshold_to_save = float("inf")
        loss_threshold_to_save = float("inf")

        # Create the "constant" settings
        current_setting = Setting()

        # Compile the "constant" settings
        current_setting.start_date = "..."  # YYYY-MM-DD hh:mm:ss
    format
```

```
16          current_setting.end_date = "..."  # YYYY-MM-DD hh:mm:ss
    format
17          current_setting.plants_to_use_list = [...]
18          current_setting.dataset_settings = [
19              {
20                  "param_name": ...,
21                  "input_name": ...,
22                  "norm_data_range": ...,
23                  "norm_data_median": ...,
24                  "transform_function": ...,
25                  "transform_function_kwargs": {...},
26                  "filter": ...,
27                  "filter_kwargs": {...},
28                  "constrain_min": ...,
29                  "constrain_max": ...
30              },
31              {
32                  "param_name": ...,
33                  "input_name": ...,
34                  "norm_data_range": ...,
35                  "norm_data_median": ...,
36                  "transform_function": ...,
37                  "transform_function_kwargs": {...},
38                  "filter": ...,
39                  "filter_kwargs": {...},
40                  "constrain_min": ...,
41                  "constrain_max": ...
42              }
43          ]
44          current_setting.remove_n = 0
45          current_setting.testMode = False
46          current_setting.params_to_use = [...]
47          current_setting.outputs = [...]
48          current_setting.learning_rate = 1e-3
49          current_setting.momentum = 0.9
50          current_setting.batch_size = 64
51          current_setting.loss_fn = nn.MSELoss()
52          # Set the epochs and the number of folds depending on the
    validation method
53          if useKFold is True:
54              current_setting.epochs = ... # 50
55              current_setting.folds = ... # 10
56          else:
57              current_setting.epochs = ... # 500
58              current_setting.folds = None
59          current_setting.trainWithAllData = False
60      else:
61          # else if the FinderSetting object has been passed, copy its
    content to local variables
```

```
62          ...
63
64      # If needed, compute additional support variables to be used by
        the loops here
65       ...
66
67      # Print to the results.txt file the common search settings and
        info about the search parameters
68       out_fp.write("Search settings:\n " + ... + "\n")
69       out_fp.write("..." + ... + "\n")
70       out_fp.write("..." + ... + "\n")
71       out_fp.write("RMSE threshold to save: " + str(
        RMSE_threshold_to_save) + ", Loss threshold to save: " + str(
        loss_threshold_to_save) + "\n")
72       out_fp.write("Initial settings:\n")
73       out_fp.write(str(current_setting) + "\n")
74       out_fp.write("\n\n\n\t\t_____SEARCH
        STARTING_____\t\t\n\n\n")
75
76       for variable in sweep_variable_list:
77          # update the current_setting parameter to sweep with the
        current variable value
78          # Other related variables may also need to be changed, but
        not always
79           current_setting.variable = variable
80          ...
81
82          # Increase the training index
83          index += 1
84
85          # Train
86          if useKFold:
87              final_visual_model_data, best_visual_model_data,
        figure_validation = statusTrainKFold(plant_df, plant_list,
        current_setting, batchMode)
88          else:
89              final_visual_model_data, best_visual_model_data,
        figure_validation = statusTrain(plant_df, plant_list,
        current_setting, batchMode)
90
91          # Choose whether to use the entire dataset or the test
        dataset only
92          # for the threshold comparisons and the ranking.
93          if decideWithFullData:
94              best_RMSE = best_visual_model_data.full_data_RMSE
95              best_loss = best_visual_model_data.full_data_loss
96              RMSE = final_visual_model_data.full_data_RMSE
97              loss = final_visual_model_data.full_data_loss
98          else:
```

```
99              best_RMSE = best_visual_model_data.test_data_RMSE
100             best_loss = best_visual_model_data.test_data_loss
101             RMSE = final_visual_model_data.test_data_RMSE
102             loss = final_visual_model_data.test_data_loss
103         # Save the  model and the figures
104         best_model = best_visual_model_data.model
105         best_figure = best_visual_model_data.visual_data_figure
106         model = final_visual_model_data.model
107         figure = final_visual_model_data.visual_data_figure
108
109         # If the error and the loss values are within the thresholds
110         if best_RMSE <= RMSE_threshold_to_save and best_loss <=
    loss_threshold_to_save:
111             # Save all the error and loss results to the results.txt
    file
112             out_fp.write("Model #" + str(index).zfill(4) + "\n")
113             out_fp.write("...variable... value: " + str(...variable
    ...) + "\n")
114             out_fp.write(str(current_setting) + "\n")
115             out_fp.write("\nFinal training results:\n")
116             out_fp.write("\tFull Data RMSE: " + str(
    final_visual_model_data.full_data_RMSE) + "\n")
117             out_fp.write("\tFull Data loss: " + str(
    final_visual_model_data.full_data_loss) + "\n")
118             out_fp.write("\tTest Data RMSE: " + str(
    final_visual_model_data.test_data_RMSE) + "\n")
119             out_fp.write("\tTest Data loss: " + str(
    final_visual_model_data.test_data_loss) + "\n")
120             out_fp.write("Best training results:\n")
121             out_fp.write("\tFull Data RMSE :" + str(
    best_visual_model_data.full_data_RMSE) + "\n")
122             out_fp.write("\tFull Data loss :" + str(
    best_visual_model_data.full_data_loss) + "\n")
123             out_fp.write("\tTest Data RMSE :" + str(
    best_visual_model_data.test_data_RMSE) + "\n")
124             out_fp.write("\tTest Data loss :" + str(
    best_visual_model_data.test_data_loss) + "\n")
125             out_fp.write("\n\n\n\t\t
    ----------------------------------------------------------
    ---------------------------------------------------------- \t\t\n\n\n")
126
127             # Save the performance visualization figures, the models
    in pth format and the error/loss change plot
128             i = 1
129             for fig in best_figure:
130                 fig.savefig(out_file_folder / (str(index).zfill(4) +
    "_best_model_...name.of.the.variable..._" + str(...variable...) +
    "_" + str(i) + ".png"))
131                 i += 1
```

```
132             torch.save(best_model, out_file_folder / (str(index).
    zfill(4) + "_best_model_...name.of.the.variable..._" + str(...
    variable...) + ".pth"))
133             i = 1
134             for fig in figure:
135                 fig.savefig(out_file_folder / (str(index).zfill(4) +
    "_last_model_...name.of.the.variable..._" + str(...variable...) +
    "_" + str(i) + ".png"))
136                 i += 1
137             torch.save(model, out_file_folder / (str(index).zfill(4)
    + "_last_model_...name.of.the.variable..._" + str(...variable...)
    + ".pth"))
138             ranking.append({"index": str(index).zfill(4), "best_rmse"
    : best_RMSE, "best_loss": best_loss, "last_rmse": RMSE, "last_loss
    ": loss})
139             figure_validation.savefig(out_file_folder / (str(index).
    zfill(4) + "_validation_error_...name.of.the.variable..._" + str
    (...variable...) + ".png"))
140         plt.close('all')
```

After adding this code to the *status_now_finder.py* source file in the correct location, the search type can be used as explained in the previous section of the appendix.

Bibliography

[1] *ONU Sustainable Development Communications material*. ONU. URL: https://www.un.org/sustainabledevelopment/news/communications-material/ (cit. on p. 1).

[2] *ONU Sustainable Development Goal 2: Zero Hunger*. ONU. URL: https://sdgs.un.org/goals/goal2 (cit. on p. 1).

[3] 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 p. 5).

[4] 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)*. 2019, pp. 178–180. DOI: 10.1109/ICECS46596.2019.8964804 (cit. on p. 5).

[5] 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 p. 5).

[6] C. W. J. Granger. «Investigating Causal Relations by Econometric Models and Cross-spectral Methods». In: *Econometrica* 37.3 (1969), pp. 424–438. ISSN: 00129682, 14680262. URL: http://www.jstor.org/stable/1912791 (cit. on p. 5).

[7] 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. 5).

[8] Simon Heller and Peter Woias. «Microwatt power hardware implementation of machine learning algorithms on MSP430 microcontrollers». In: *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2019, pp. 25–28. DOI: 10.1109/ICECS46596.2019.8964726 (cit. on p. 6).

[9] Kyong Ho Lee and Naveen Verma. «A Low-Power Processor With Configurable Embedded Machine-Learning Accelerators for High-Order and Adaptive Analysis of Medical-Sensor Signals». In: *IEEE Journal of Solid-State Circuits* 48.7 (2013), pp. 1625–1637. DOI: 10.1109/JSSC.2013.2253226 (cit. on p. 6).

[10] Nesreen K. Ahmed, Amir F. Atiya, Neamat El Gayar, and Hisham El-Shishiny. «An Empirical Comparison of Machine Learning Models for Time Series Forecasting». In: *Econometric Reviews* 29.5-6 (2010), pp. 594–621. DOI: 10.1080/07474938.2010.481556. eprint: https://doi.org/10.1080/07474938.2010.481556. URL: https://doi.org/10.1080/07474938.2010.481556 (cit. on pp. 6, 7).

[11] Francisco Martínez, María Pilar Frías, María Dolores Pérez-Godoy, and Antonio Jesús Rivera. «Dealing with seasonality by narrowing the training set in time series forecasting with kNN». In: *Expert Systems with Applications* 103 (2018), pp. 38–48. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2018.03.005. URL: https://www.sciencedirect.com/science/article/pii/S0957417418301441 (cit. on p. 6).

[12] Saeid Niazmardi, Saeid Homayouni, Heather McNairn, Jiali Shang, and Abdolreza Safari. «Multiple kernels learning for classification of agricultural time series data». In: *2014 The Third International Conference on Agro-Geoinformatics*. 2014, pp. 1–4. DOI: 10.1109/Agro-Geoinformatics.2014.6910640 (cit. on p. 7).

[13] Georgios Makridis, Dimosthenis Kyriazis, and Stathis Plitsos. «Predictive maintenance leveraging machine learning for time-series forecasting in the maritime industry». In: *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. 2020, pp. 1–8. DOI: 10.1109/ITSC45102.2020.9294450 (cit. on p. 8).

[14] Jordan J. Bird, Jhonatan Kobylarz, Diego R. Faria, Anikó Ekárt, and Eduardo P. Ribeiro. «Cross-Domain MLP and CNN Transfer Learning for Biological Signal Processing: EEG and EMG». In: *IEEE Access* 8 (2020), pp. 54789–54801. DOI: 10.1109/ACCESS.2020.2979074 (cit. on p. 8).

[15] Shurouq Hijazi, Alex Page, Burak Kantarci, and Tolga Soyata. «Machine Learning in Cardiac Health Monitoring and Decision Support». In: *Computer* 49.11 (2016), pp. 38–48. DOI: 10.1109/MC.2016.339 (cit. on p. 8).

[16] Taylor Hall and Nishant Kumar. *Why Machine Learning Models Often Fail to Learn: QuickTake Q&A;* Bloomberg. URL: https://www.bloomberg.com/news/articles/2016-11-10/why-machine-learning-models-often-fail-to-learn-quicktake-q-a (cit. on p. 9).

[17] Egm4313.s12 (Prof. Loc Vu-Quoc) - Own work. *Neuron and myelinated axon, with signal flow from inputs at dendrites to outputs at axon terminals.* No changes made. Utilized under CC BY-SA 4.0 license. Wikipedia. URL: https://commons.wikimedia.org/w/index.php?curid=72801384 (cit. on p. 17).

[18] *Python.* Available at https://www.python.org/. Python Software Foundation (cit. on p. 23).

[19] *Andaconda Virtual Environment.* Available at https://www.anaconda.com/. Anaconda Inc. (cit. on p. 23).

[20] *PyTorch Dataset types.* Available at https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset. Facebook's AI Research lab (FAIR) (cit. on pp. 32, 36).

[21] *PyTorch.* Available at https://pytorch.org/. Facebook's AI Research lab (FAIR) (cit. on p. 35).

[22] *Torch.* Available at http://torch.ch/ (cit. on p. 35).

[23] *PyTorch Dataloader.* Available at https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader. Facebook's AI Research lab (FAIR) (cit. on pp. 36, 37).

[24] *PyTorch Dataloader Sampler.* Available at https://pytorch.org/docs/stable/data.html#torch.utils.data.Sampler. Facebook's AI Research lab (FAIR) (cit. on pp. 36, 37).

[25] *PyTorch Module.* Available at https://pytorch.org/docs/stable/generated/torch.nn.Module.html. Facebook's AI Research lab (FAIR) (cit. on p. 38).

[26] *PyTorch Sequential.* Available at https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html. Facebook's AI Research lab (FAIR) (cit. on p. 39).

[27] *PyTorch Flatten.* Available at https://pytorch.org/docs/stable/generated/torch.nn.Flatten.html. Facebook's AI Research lab (FAIR) (cit. on p. 40).

[28] *PyTorch Layers, Activation functions and Loss functions.* Available at https://pytorch.org/docs/stable/nn.html. Facebook's AI Research lab (FAIR) (cit. on pp. 41, 44, 112).

[29] *PyTorch Stochastic Gradient Descent (SGD)*. Available at `https://pytorch.org/docs/stable/generated/torch.optim.SGD.html`. Facebook's AI Research lab (FAIR) (cit. on p. 45).

[30] *PyTorch optimizers*. Available at `https://pytorch.org/docs/stable/optim.html`. Facebook's AI Research lab (FAIR) (cit. on p. 46).