Università di Perugia
Dipartimento di Matematica e Informatica

Corso di Laurea in Informatica

A Deep Learning approach for Time Series Imputation on Photovoltaic data

Relatori
Valentina Poggioni
Enrico Bellocchio
Alessandro Devo

Laureando
Nicolò Vescera

Anno Accademico 2022-2023

Contents

Abstract

The growing need for the adoption of tools capable of generating clean energy from renewable and sustainable sources has led to extensive generation and collection of energy production data, especially from photovoltaic panels installed worldwide. However, these data often have gaps and deficiencies due to various factors such as temporary failures, adverse weather conditions, or malfunctions of sensors and data collection instruments. Accurate imputation of these gaps is crucial to ensure the reliability of analyses and predictions based on this data. This thesis aims to address the problem of imputing time series data from photovoltaic panels using advanced deep learning techniques. In particular, a deep learning model based on Fully Connected Neural Networks (FCNN) and Recurrent Neural Networks (RNN) are proposed to capture the complex temporal relationships between the total energy produced (target feature) and various components of the system. The models were trained on a dataset consisting of real data from a photovoltaic system with a power capacity of approximately $1MW$.

Chapter 1

Background

1.1 Problem Definition

The following thesis aims to address the problem of imputing time series data from photovoltaic systems. Specifically, it often happens that data acquisition instruments in a system temporarily fail, causing a period of time, more or less extended, where the curve of the total energy produced is missing. To try to fill this "gap" a simple formula is not sufficient, as various factors such as solar irradiance, ambient temperature, presence of clouds, rainfall, etc., need to be taken into account.

Formally, we can define the problem as follows:

Definition 1.1.1 (**Imputation problem**)**.** Given:

- A set of time series data representing a photovoltaic system, $S = S_1, S_2, \ldots, S_N$, where N represents the number of available time series,
- A target time series $t \in S$ that represent the Total Generated Energy,

where each time series S_i is composed of ordered pairs (t_i, v_i), where t_i is a timestamp representing the moment when the value v_i was recorded, the objective of the imputation problem is to estimate the missing or damaged values in the time series t.

1.2 Photovoltaic Implant

Solar photovoltaic (PV) energy systems are made up of different components. Each component has a specific role. The type of component in the system depends on the type of system and the purpose. For example, a simple PV-direct system is

composed of a solar module or array (two or more modules wired together) and the load (energy-using device) it powers. A solar energy system produces direct current (DC). This is electricity which travels in one direction. The loads in a simple PV system also operate on direct current (DC). A stand-alone system with energy storage (a battery) will have more components than a PV- direct system.

1.2.1 Solar Module

The majority of solar modules available on the market and used for residential and commercial solar systems are silicon- crystalline. These modules consist of multiple strings of solar cells, wired in series (positive to negative), and are mounted in an aluminum frame. The size or area of the cell determines the amount of amperage. The larger the cell, the higher the amperage.

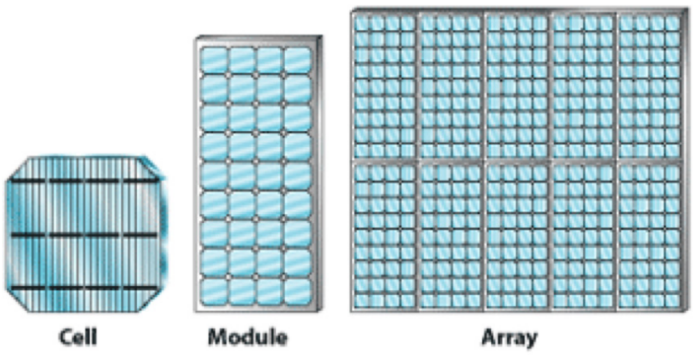


Figure 1.1: The solar cell is the basic component. Cells wired together and mounted in a frame compose a solar module. Several modules wired together form an array.

1.2.2 Solar Array

The solar array is made up of multiple PV modules wired together. Connecting the negative wire of one module to the positive wire of a second module is the beginning of a series string. Wiring modules in series results in the voltage of each of the two modules is added together. A series string represents the summed voltages of each individual module. The negative cable of one module is connected to the positive cable of the next module. In a large system, multiple strings are assembled and the non-connected ends are connected to homerun leads which are landed at the terminals of an enclosure located near the array. The goal is to wire modules in series to build voltage.

1.2.3 Junction Box

A PV system array with multiple strings of modules will have a positive lead and a negative lead on the end of each string. The positive leads will be connected to individual fuses and the negative leads will be connected to a negative busbar in an enclosure. This is called the source circuit. The junction box serves to "combine" multiple series strings into one parallel circuit. For example, an array with three strings of 10 modules wired in series would produce 300 volts (10 modules x 30 volts) per string and 4 amps per string. When the leads are landed in the combiner box, the circuit would produce 300 volts at 12 amps (3 strings x 4 amps/string). Once the circuits are combined, leaving the box it is referred to as the output circuit.

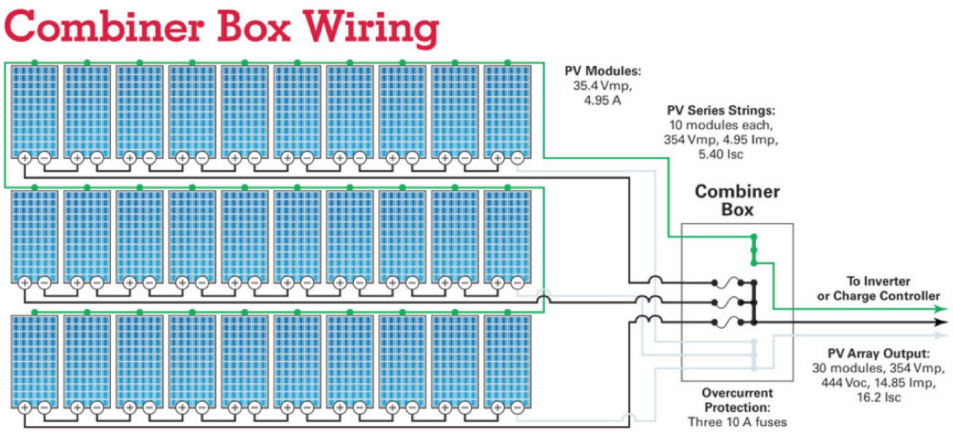


Figure 1.2: This figure represent an output circuit made of 3 string, each one hosts 10 solar modules.

1.2.4 Inverter

Energy from an array or a battery bank is direct current (DC). This will provide for DC loads such a lights, fans, pumps, motors, and some specialty equipment. However, if the energy is to be used to power loads that operate on alternating current (AC), as what is found in a residence, the current needs to be converted. The inverter changes DC energy to AC energy. Inverters are available in many different sizes for various-sized loads. A string inverter is used to convert DC power from a solar array to AC power and can be connected to an AC distribution power panel (service panel) in a residence or facility.

1.2.5 System Metering

Several tools are available to help the solar user to monitor their system. On stand-alone or off-grid PV systems, the battery meter is used to measure the energy coming in and going out of the battery bank. Charging and discharging of batteries, and proper functioning of the charging system is important to alert the user to incomplete charging, battery decline, or possible system shutdown. System monitoring with web-based tools and apps allow the solar user to see system activity using a cell phone or tablet from a location away from their system.

1.3 Open-Meteo

Open-Meteo is an open-source weather data platform and community-driven weather forecasting project. Open-Meteo aims to provide access to weather data and forecasts that are openly accessible to the public and can be used for a variety of applications, including research, development, and personal use. It offers a diverse range of APIs that go beyond traditional weather forecasting such as past weather data, ocean data, air quality, ensemble forecasts, climate forecasts based on IPCC predictions, and even floods. Open-Meteo offers over 80 years of hourly weather data, covering any location on earth, all at a 10 kilometer resolution. This extensive dataset is very useful to delve into the past and analyze historical weather patterns.

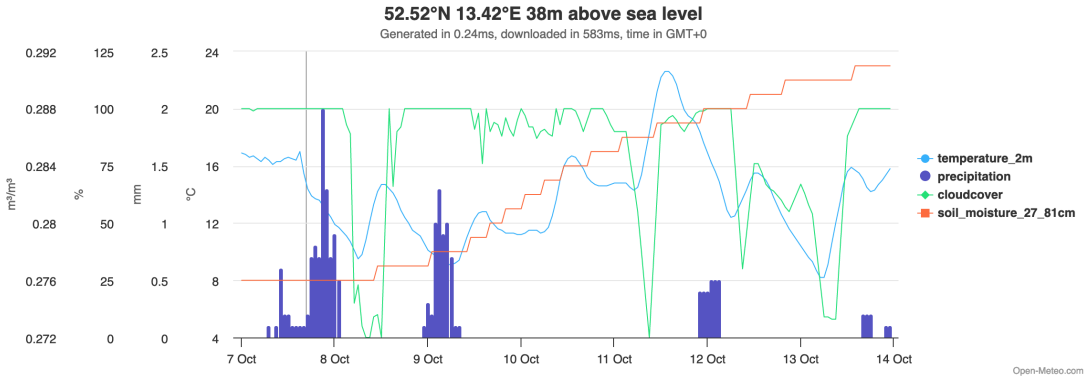


Figure 1.3: A plot about some Open-Meteo data.

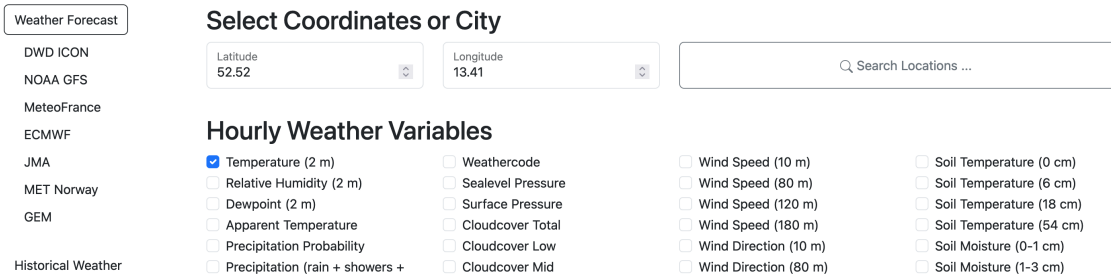


Figure 1.4: Open-Meteo forecast features.

1.4 Multi Layer Perceptron

The Multi Layer Perceptron is the most known and most frequently used type of neural network. On most occasions, the signals are transmitted within the network in one direction: from input to output. There is no loop, the output of each neuron does not affect the neuron itself. This architecture is called feed- forward. Layers which are not directly connected to the environment are called hidden. There are also feed-back networks, which can transmit impulses in both directions, due to reaction connections in the network. These types of networks are very powerful and can be extremely complicated. Introduction of several layers was determined by the need to increase the complexity of decision regions. A perceptron with a single layer and one input generates decision regions under the form of semi planes. By adding another layer, each neuron acts as a standard perceptron for the outputs of the neurons in the anterior layer, thus the output of the network can estimate convex decision regions, resulting from the intersection of the semi planes generated by the neurons. The power of the multilayer perceptron comes precisely from non-linear activation functions. Almost any non-linear function can be used for this purpose, except for polynomial functions. Currently, the functions most commonly used today are the single-pole (or logistic) sigmoid.

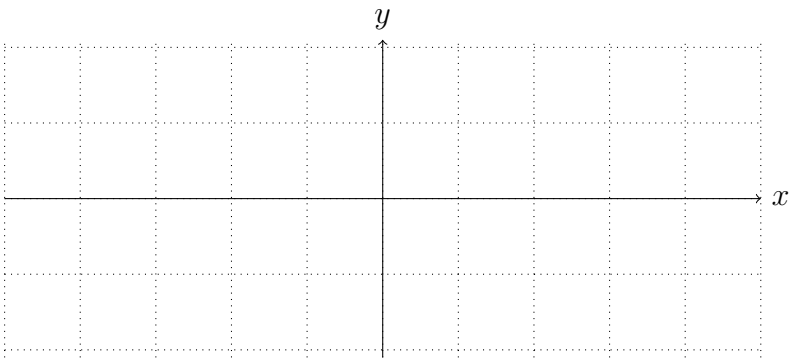


Figure 1.5: Sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$

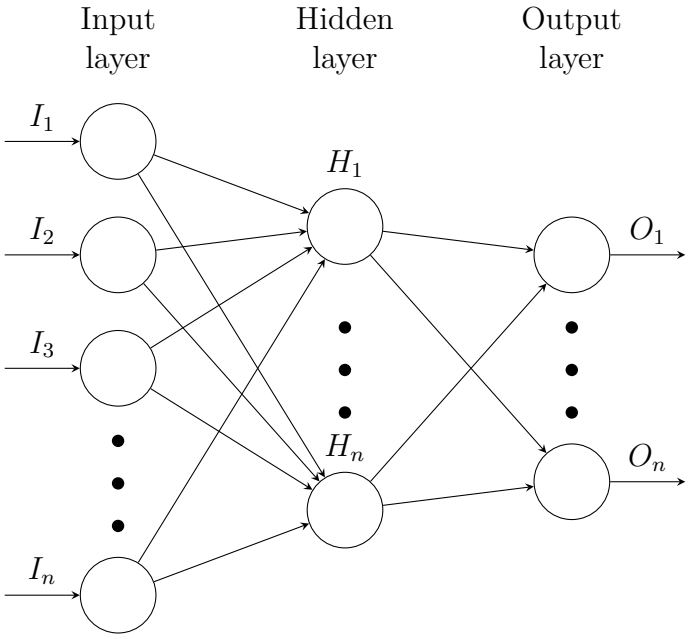


Figure 1.6: Multi Layer Perceprton architecture.

1.5 Recurrent Neural Network

Recurrent Neural Networks (RNNs) are a type of neural network architecture which is mainly used to detect patterns in a sequence of data. Such data can be handwriting, genomes, text or numerical time series which are often produced in industry settings (e.g. stock markets or sensors). However, they are also applicable to images if these get respectively decomposed into a series of patches and treated as a sequence. On a higher level, RNNs find applications in Language Modelling

and Generating Text, Speech Recognition, Generating Image Descriptions or Video Tagging. What differentiates Recurrent Neural Networks from Feedforward Neural Networks also known as Multi-Layer Perceptrons (MLPs) is how information gets passed through the network. While Feedforward Networks pass information through the network without cycles, the RNN has cycles and transmits information back into itself. This enables them to extend the functionality of Feedforward Networks to also take into account previous inputs $X_{0:t-1}$ and not only the current input X_t. This difference is visualised on a high level in Figure 1.7. Note, that here the option of having multiple hidden layers is aggregated to one Hidden Layer block H. This block can obviously be extended to multiple hidden layers.

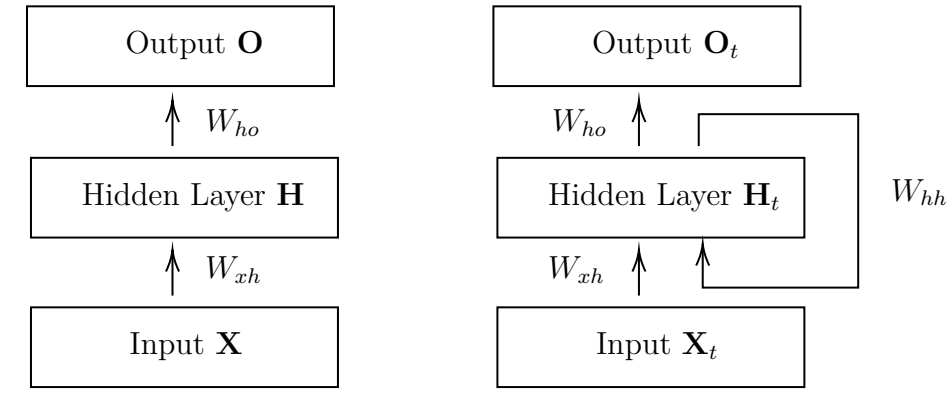


Figure 1.7: Visualization of difference between MLPs and RNN.

1.5.1 Gated Recurrent Unit

Gated Recurrent Units (GRU) are an advanced variation of RRNs (Recurrent Neural Network). GRUs use update gate and reset get for solving a standard RNNs vanishing gradient issue. These are essentially 2 vectors that decide the type of information to be passed towards the output. What makes these vectors special is that programmers can train them to store information, especially from long ago. They do all of this by utilizing its gated units which help solve vanishing/exploding gradient problems often found in traditional recurrent neural networks. These gates are helpful for controlling any information to be maintained or discarded for each step. It is also worth keeping in mind that gated recurrent units make use of reset and update gates.

- **Update Gates Function**: The main function of the update gate is to determine the ideal amount of earlier info that is important for the future.

One of the main reasons why this function is so important is that the model can copy every single past detail to eliminate fading gradient issue.

- **Reset Gate's Function**: A major reason why reset gate is vital because it determines how much information should be ignored. It would be fair to compare reset gate to LSTMs forget gate because it tends to classify unrelated data, followed by getting the model to ignore and proceed without it.

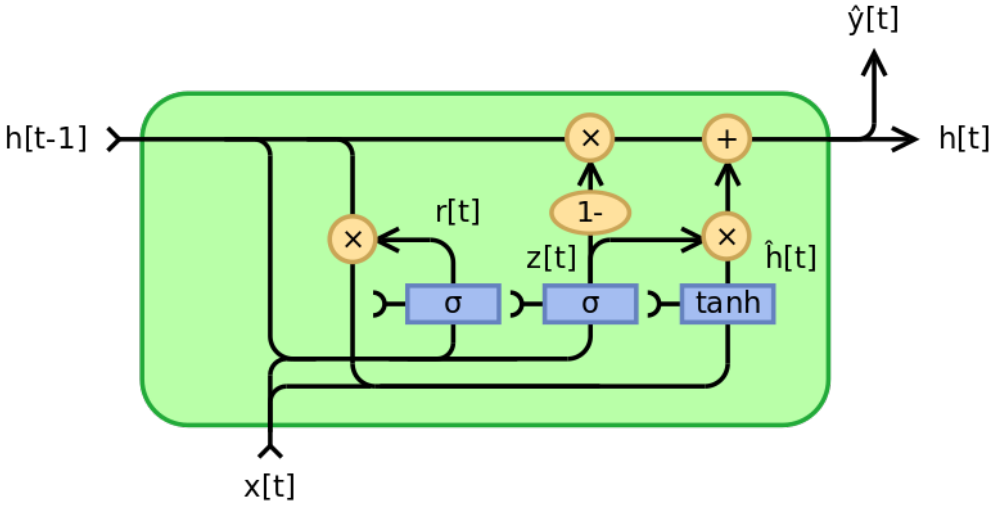


Figure 1.8: Gated Recurrent Unit (GRU) architecture.

1.6 Transformer

Recurrent neural networks, long short-term memory and gated recurrent neural networks in particular, have been firmly established as state of the art approaches in sequence modeling and transduction problems such as language modeling and machine translation. Recurrent models typically factor computation along the symbol positions of the input and output sequences. Aligning the positions to steps in computation time, they generate a sequence of hidden states h_t, as a function of the previous hidden state h_{t-1} and the input for position t. This inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples. Attention mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks, allowing modeling of dependencies without regard to their distance in the input or output sequences. The Transformer allows for significantly more parallelization.

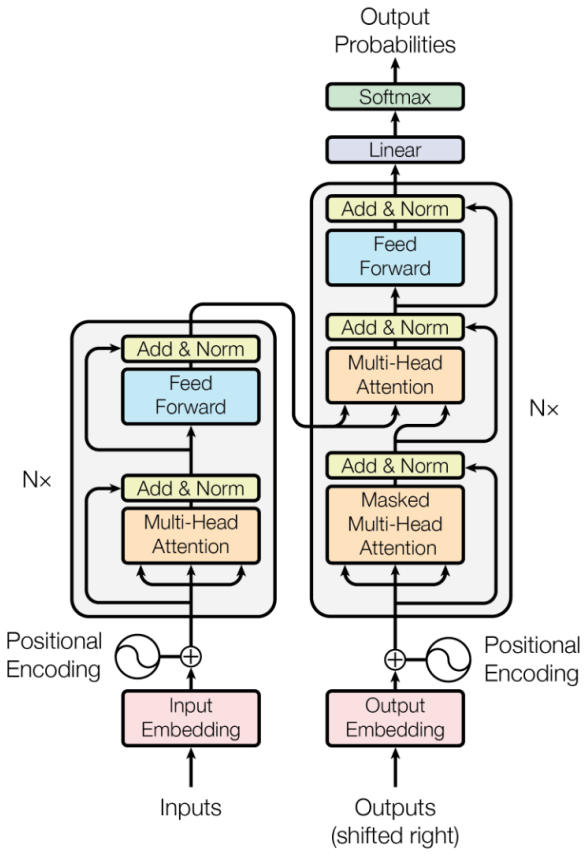


Figure 1.9: This figure shows the Transformers model architecture.

Most competitive neural sequence transduction models have an encoder-decoder structure. Here, the encoder maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a sequence of continuous representations $z = (z_1, \ldots, z_n)$. Given z, the decoder then generates an output sequence $(y_1, \ldots, y_m)$ of symbols one element at a time. At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next. The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1.9, respectively.

- *Encoder*: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position- wise fully connected feed-forward network. Residual connection are employed around each of the two sub-layers,

followed by layer normalization. That is, the output of each sub-layer is LayerNorm(x+Sublayer(x)), where Sublayer(x) is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $dmodel = 512$.

- *Decoder*: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, there are residual connections around each of the sub-layers, followed by layer normalization. The self-attention sub-layer is modified in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

- *Attention*: An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

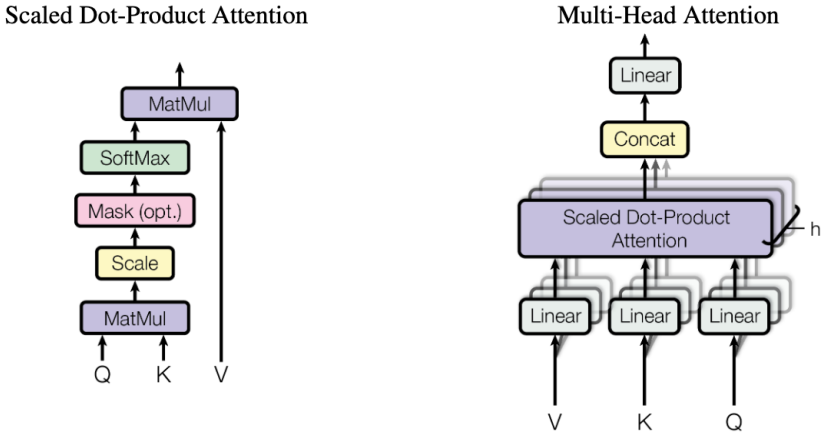


Figure 1.10: Scaled Dot-Product Attention (left). Multi-Head Attention consists of several attention layers running in parallel (right).

1.7 Dataset

The dataset at our disposal describes a period of almost two years (from 02-02-2022 to 16-06-2023) and comes from a 978 kW photovoltaic plant located in the province of Bari. It consists of 3 inverters, 27 field panels, 1 meter, 1 solarimeter, 2 interface protections and 1 "system" device in which data from Solargis are stored. The dataset is organized into files, one for each type of device, representing each individual day, and the data is aggregated every 5 minutes. Each row contains a reference to the device it belongs to (`deviceName` and `deviceId`).

File Name
`2022_02_02_Rofilo_NP00003174_inverter.csv`
`2022_02_02_Rofilo_NP00003174_meteorology.csv`
`2022_02_02_Rofilo_NP00003174_meter.csv`
`2022_02_02_Rofilo_NP00003174_other.csv`
`2022_02_02_Rofilo_NP00003174_plantDevice.csv`
`2022_02_02_Rofilo_NP00003174_stringbox.csv`
`2022_02_03_Rofilo_NP00003174_inverter.csv`
`2022_02_03_Rofilo_NP00003174_meteorology.csv`
`2022_02_03_Rofilo_NP00003174_meter.csv`
`2022_02_03_Rofilo_NP00003174_other.csv`
`2022_02_03_Rofilo_NP00003174_plantDevice.csv`
`2022_02_03_Rofilo_NP00003174_stringbox.csv`
...

Table 1.1: Some files from our dataset.

timestamp	**serial**	...	**TotalEnergy(kWh)**	**Frequency(Hz)**	**deviceid**
2022-10-23 04:30:00	INV01	...	357196.88	50.06	83204
2022-10-23 04:35:00	INV01	...	357196.88	50.06	83204
...	...	...	...	...	...
2022-10-23 13:00:00	INV01	...	357921.16	49.95	83204
2022-10-23 13:05:00	INV01	...	357935.24	49.95	83204
...	...	...	...	...	...
01/02/2023 23:45	INV01	...	431324.36	49.88	83204
01/02/2023 23:50	INV01	...	431324.36	49.88	83204

Table 1.2: Some lines from an Inverter file.

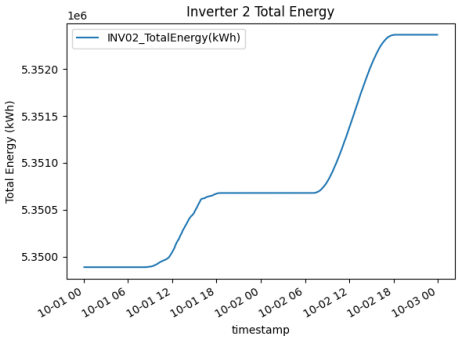
1.7.1 Inverter

Inside the files related to the inverters, we can find data such as the processor's operating temperature (`InternalTemperature`), various Alternating and Direct currents produced (`CurrentAC` and `CurrentDC`), Delivered Power, total energy produced by the individual inverter and other status and control bits that indicate various stages of the inverter's life (boot, reset, ready, etc.).

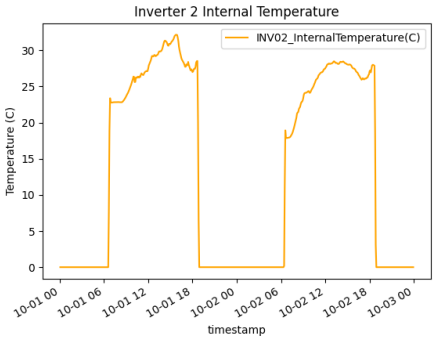
Down below a table with all the available features:

Name	Unit Symbol	Description
CommunicationCode	-	Communication Code
Failure 3	-	Active Alarm
Failure 4	-	Isolation Alarm
CurrentDC	A	Photovoltaic field Current
CurrentAC	A	Network Current
CurrentAC Phase1	A	Line RMS Current Phase R
CurrentAC Phase2	A	Line RMS Current Phase S
CurrentAC Phase3	A	Line RMS Current Phase T
TotalEnergy	kWh	Active Energy Delivered
Frequency	Hz	Network Frequency
PowerAC Phase1	kW	Line PA Phase R
PowerAC Phase2	kW	Line PA Phase S
PowerAC Phase3	kW	Line PA Phase T
PowerAC	kW	Delivered PA
PowerDC	kW	Photovoltaic field Power
Status	-	Inverter State
Failure	-	Network docking PLL State
Failure 2	-	Network State 1
Failure 1	-	Network State 2
InternalTemperature	C	CPU Temp.
HeatSinkTemperature	C	IGBT Temp.
VoltageDC	V	Photovoltaic field Voltage
VoltageAC	V	Network Voltage
VoltageAC Phase1	V	Line RMS Voltage Phase R
VoltageAC Phase2	V	Line RMS Voltage Phase S
VoltageAC Phase3	V	Line RMS Voltage Phase T

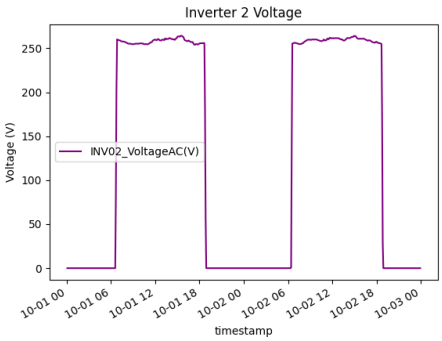
Table 1.3: All available features from an `inverter` file.



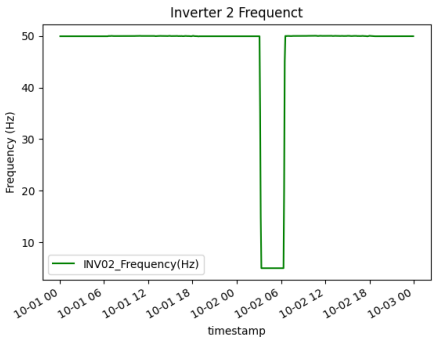
(a) Inverter 2 Total Energy plot.



(b) Inverter 2 CPU Temperature plot.



(c) Inverter 2 Voltage AC plot.



(d) Inverter 2 Frequency plot.

1.7.2 Junction Box

In the files related to the Junction Box or Combiner Box, we find data that describes the current production of the various strings they are connected to (`CurrentString1-7`, in general, each Junction Box manages 7 strings), some temperatures (such as `ModuleTemperature`), and some control bits to check proper operation.

Name	Unit Symbol	Description
CommunicationCode	-	Communication Code
Failure	-	Strings Alarm
CurrentString1	A	Current I1
CurrentString2	A	Current I2
CurrentString3	A	Current I3
CurrentString4	A	Current I4
CurrentString5	A	Current I5
CurrentString6	A	Current I6
CurrentString7	A	Current I7
AverageStringCurrent	A	Average Current
Irradiance	W/m^2	Modules Irradiation
Failure 1	-	Open Strings
Failure 2	-	Not Perform. Strings
EnvironmentTemperature	C	Environment Temperature
ModuleTemperature	C	Modules Temperature
InternalTemperature	C	Board Temperature

Table 1.4: All available features form a `stringbox` file.

1.7.3 Solargis

Solargis is a company specialized in providing solar data and forecasting services for photovoltaic installations and solar energy-related projects. Their main goal is to provide precise and reliable information on solar irradiation and solar weather conditions anywhere in the world. This data is essential for the design, optimization, and management of photovoltaic systems. Solargis collects and provides detailed data on global, direct, and diffuse solar irradiation at every geographical location. This data allows photovoltaic system developers to assess the amount of available solar energy in a given area, which is crucial for properly sizing the system and calculating production forecasts.

timestamp	...	SolargisGHI(W/m2)	SolargisGTI(W/m2)
2022-08-01 11:40:00	...	896	978
2022-08-01 11:45:00	...	896	978
2022-08-01 11:50:00	...	896	978
2022-08-01 11:55:00	...	914	1001
2022-08-01 12:00:00	...	914	1001
2022-08-01 12:05:00	...	914	1001
2022-08-01 12:10:00	...	928	1019
2022-08-01 12:15:00	...	928	1019
2022-08-01 12:20:00	...	928	1019

Table 1.5: Some Solargis data from `2022_08_01_Rofilo_NP00003174_plantDevice.csv` file

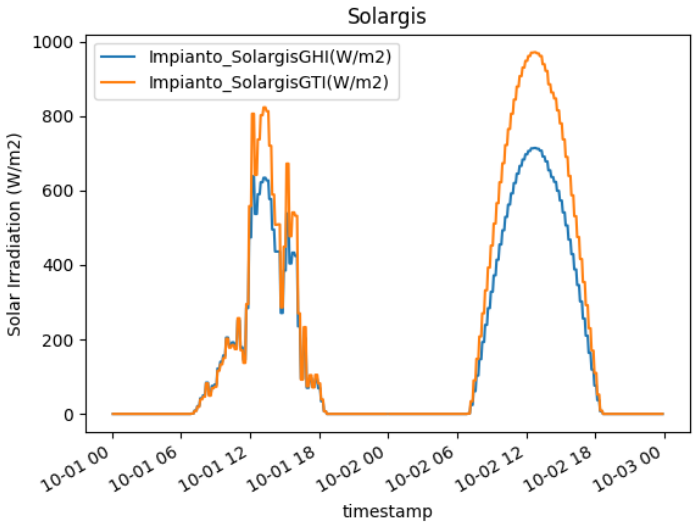


Figure 1.12: Solargis GHI & GTI plot.

Solar radiation takes a long journey until it reaches Earths surface. So when modelling solar radiation, various interactions of extra-terrestrial solar radiation with the Earths atmosphere, surface and objects are to be taken into account. The component that is neither reflected nor scattered, and which directly reaches the surface, is called direct radiation; this is the component that produces shadows. The component that is scattered by the atmosphere, and which reaches the ground is called diffuse radiation. The small part of the radiation reflected by the sur-

face and reaching an inclined plane is called the reflected radiation. These three components together create global radiation.

In solar energy applications, the following parameters are commonly used in practice:

- Direct Normal Irradiation/Irradiance (DNI) is the component that is involved in thermal (concentrating solar power, CSP) and photovoltaic concentration technology (concentrated photovoltaic, CPV).
- Global Horizontal Irradiation/Irradiance (GHI) is the sum of direct and diffuse radiation received on a horizontal plane. GHI is a reference radiation for the comparison of climatic zones; it is also essential parameter for calculation of radiation on a tilted plane.
- Global Tilted Irradiation/Irradiance (GTI), or total radiation received on a surface with defined tilt and azimuth, fixed or sun-tracking. This is the sum of the scattered radiation, direct and reflected. It is a reference for photovoltaic (PV) applications, and can be occasionally affected by shadow.

Name	Unit Symbol	Description
SolargisGHI	W/m^2	Solargis Global Horizontal Irradiation
SolargisGTI	W/m^2	Solargis Global Tilted Irradiation

Table 1.6: All available features from a `plantDevice` file.

1.7.4 Meteorology

In the Meteo files, we can find some environment temperature and solar irradiance data. Is important to mention that these features are not as powerful as weather forecast or Solargis data for the Imputation task.

Name	Unit Symbol	Description
COMMUNICATION_CODE SOL	-	Communication Code
Irradiance SOL	W/m^2	Irradiance
Module Temperature HEX SOL	-	All Registers
Module Temperature SOL	C	Module Temperature

Table 1.7: All available features from a `meteo` file.

1.7.5 Meter

The Meter files contain information about the current injected into and drawn from the network. It is from here that we get the values of our target feature `Cont_TotalEnergy(kWh)`.

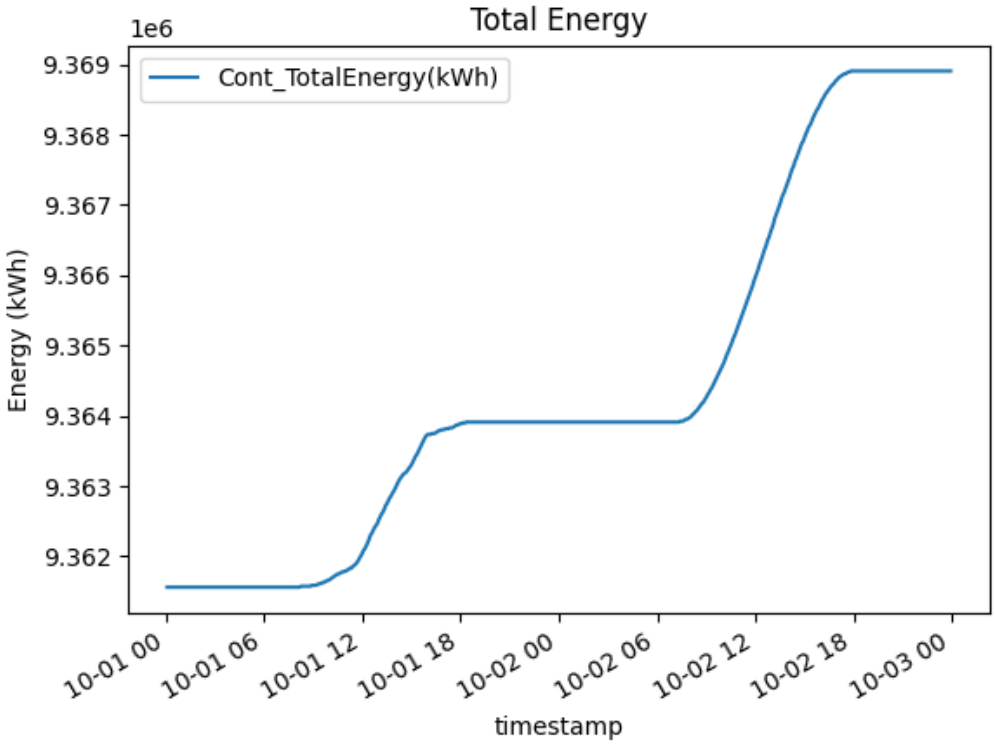


Figure 1.13: Total Energy plot, our target feature.

Name	Unit Symbol	Description
COMMUNICATION_CODE Cont	-	Communication Code
Status Cont	-	Status
Totale Energia Immessa Cont	kWh	Total Energy
Totale Energia Prelevata Cont	kWh	Total Energy Imported

Table 1.8: All available features from a `meter` file.

1.7.6 Other

In the Other type files, we can find the remaining, less relevant, features that describe the behavior and functioning of the leftover elements that make up the photovoltaic plant system.

Name	Unit Symbol	Description
COMMUNICATION_CODE NV10P	-	Communication Code
CB-State NV10P	-	CB State
Frequency NV10P	Hz	Frequency
IN1 NV10P	-	Digital IN 1
IN2 NV10P	-	Digital IN 2
Last Trip Cause NV10P	-	Last Trip Cause
NV10P - Trip BF	-	Digital IN 25
NV10P - Trip f<	-	Digital IN 24
NV10P - Trip f>	-	Digital IN 23
NV10P - Trip U<	-	Digital IN 21
NV10P - Trip U>	-	Digital IN 22
UE NV10P	V	UE
UL1 NV10P	V	Voltage AC Phase 1
UL2 NV10P	V	Voltage AC Phase 2
UL3 NV10P	V	Voltage AC Phase 3
Un NV10P	V	Un
Unp NV10P	V	Unp
COMMUNICATION_CODE NA16	-	Communication Code
CB-State NA16	-	CB State
IE NA16	A	IE
IL1 NA16	A	Current AC Phase 1
IL2 NA16	A	Current AC Phase 2
IL3 NA16	A	Current AC Phase 3
IN1 NA16	-	Digital IN 1
IN2 NA16	-	Digital IN 2
NA16 - Protection Trip	-	Digital IN 25

Table 1.9: All available features from an `other` file.

Chapter 2

Data Preprocessing

Having the dataset divided into different files, one for each day and device (see Section 1.7), with the presence of some periods, ranging from a few minutes to several days, of missing data (likely due to a data collection tool failure), results in many issues during the training phase and makes it almost impossible. In this chapter, we will see how we have addressed these problems by opting for a monolithic tabular structure for the dataset and discussing the approaches we have used for handling missing data.

2.1 Dataset Realization

For the creation of our dataset, we devised a procedure that allowed us to obtain a single file, in CSV format, where for each timestamp, we have all the data for the entire system at that exact moment. Below is the final structure of the dataset and the algorithm for generating it.

timestamp	**DEV.NAME**$_1$**__FEAT**$_1$	...	**DEV.NAME**$_n$**__FEAT**$_n$
01/10/2022 10:00	...	...	...
01/10/2022 10:05	...	...	...
01/10/2022 10:10	...	...	...

Table 2.1: Final dataset feature structure.

Algorithm 1 Dataset aggregation algorithm

Require: data_folder
Ensure: `data_folder` exists
  dev_types ← find all file types inside `data_folder` (e.g. meter, inverter, ...)
  dev_content ← a dictionary with `dev_types` types as *keys* and empty *values*
  for each key **in** dev_conent.keys **do**
    files ← find all file matching type `key` inside `data_folder`
    sort `files` by date (asc.)
    temp_type_aggregate ← and empty csv table
    for each file **in** files **do**
      append all `file` lines to `temp_type_aggregate` table
    end for
    dev_content[key] ← temp_type_aggregate
  end for
  ▷ At this time we have a dictionary mapping a file type with all its available data

  dataset ← an empty csv table
  for each type, data **in** dev_content **do** ▷ `type` is *key*, `data` is *value*
    rename all `data` *columns* to `data.deviceID`_*column*`.name`
    except for 'timestamp' *column*
    dataset ← `dataset` and `data` tables using 'timestamp' column
  end for
  save `dataset` table to file

timestamp	**INV01_PowerAC**	...	**Cont_TotalEnergy**
2022-02-02 00:05:00	NaN	...	NaN
2022-02-02 00:10:00	NaN	...	NaN
...	...	...	...
2022-06-22 10:20:00	175.66	...	8900941.5
2022-06-22 10:25:00	178.29	...	8900995.5
2022-06-22 10:30:00	180.82	...	8901036.0
...	...	...	...
2023-06-16 18:00:00	NaN	...	NaN
2023-06-16 18:05:00	NaN	...	NaN

Table 2.2: Some data from dataset after running Algorithm 1

2.1.1 Timestamp cyclical encoding

To enable the models to learn the alternation of minutes, days, and months as effectively as possible during the training phase, we applied a procedure to transform each individual timestamp into a pair of sine and cosine values, thus performing a cyclic encoding of various seasonalities.

Algorithm 2 Cyclical Encoding Algorithm

Require: dataset table
Ensure: `dataset` **is not** empty
dataset['minute_sin'] $\leftarrow \sin(2\pi(\frac{\texttt{dataset.timestamp.minute}}{60}))$
dataset['minute_cos'] $\leftarrow \cos(2\pi(\frac{\texttt{dataset.timestamp.minute}}{60}))$
dataset['hour_sin'] $\leftarrow \sin(2\pi(\frac{\texttt{dataset.timestamp.hour}}{24}))$
dataset['hour_cos'] $\leftarrow \cos(2\pi(\frac{\texttt{dataset.timestamp.hour}}{24}))$
dataset['day_sin'] $\leftarrow \sin(2\pi(\frac{\texttt{dataset.timestamp.day}}{31}))$
dataset['day_cos'] $\leftarrow \cos(2\pi(\frac{\texttt{dataset.timestamp.day}}{31}))$
dataset['month_sin'] $\leftarrow \sin(2\pi(\frac{\texttt{dataset.timestamp.month}}{12}))$
dataset['month_cos'] $\leftarrow \cos(2\pi(\frac{\texttt{dataset.timestamp.month}}{12}))$

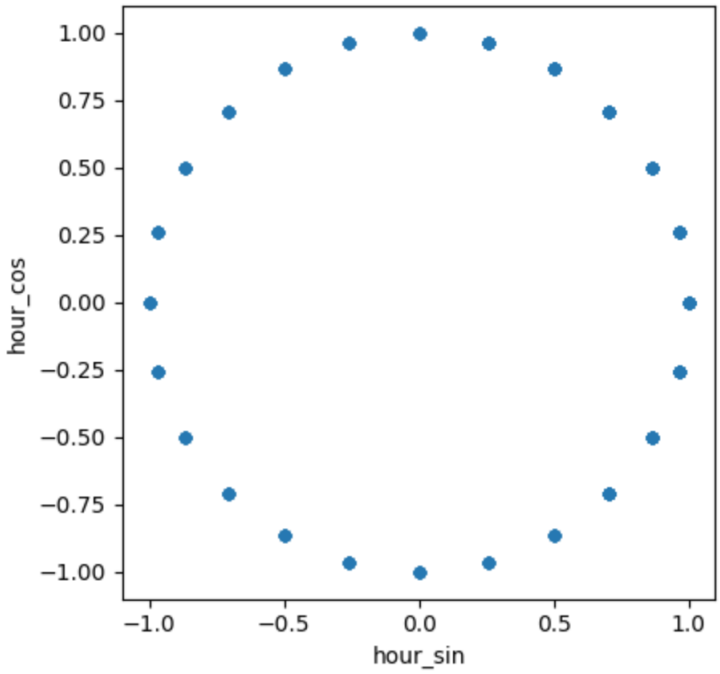


Figure 2.1: Hour cyclical encoding plot.

2.1.2 Historical weather

In addition to the information from Solargis (see Section 1.7.3), the models may require other weather data, such as cloud cover, rainfall, etc. Therefore, we made an API call to Open-Meteo servers (see Section 1.3), using the "Historical Weather" functionality, to obtain the accurate past forecasts for the time period covered by our dataset (from 02-02-2022 to 18-06-2023).

Algorithm 3 Open-Meteo data request Algorithm

Require: dataset table
Ensure: `dataset` **is not** empty
meteo_data ← require meteo feature from Open-Mete public API
dataset ← merge `dataset` and `meteo_data` tables using `timestamp` column

/* At this time we have `dataset` timestamps as 5 minute frequency and `meteo_data` as 1 hour frequency. */

use a *forward fill* method to fill gaps inside `dataset` table only for `meteo_data` columns.

We have obtained most of the information made available by Open-Meteo since the API request is free and computationally and storage-wise inexpensive. These will be filtered at a later time (see Section 2.2). Among these, we find `sunrise` and `sunset`, which indicate the exact time (down to the minute) of when dawn and sunset occurred, respectively. These two features will be used to create a new column in our dataset called `is_day`, which will allow the model to understand whether it is analyzing a period of time during the day or night. This is to ensure that it does not make energy production predictions during the night.

Feature	Unit	Note
temperature_2m	řC	Air temperature at 2 meters above ground.
relativehumidity_2m	%	Relative humidity at 2 meters above ground.
dewpoint_2m	řC	Dew point temperature at 2 meters above ground. The dew point of a given body of air is the temperature to which it must be cooled to become saturated with water vapor. This temperature depends on the pressure and water content of the air.
apparent_temperature	řC	Apparent temperature is the perceived feels-like temperature combining wind chill factor, relative humidity and solar radiation.
precipitation	mm	Total precipitation (rain, showers, snow) sum of the preceding hour. Data is stored with a 0.1 mm precision. If precipitation data is summed up to monthly sums, there might be small inconsistencies with the total precipitation amount.
weathercode	wmo	Weather condition as a numeric code. Follow WMO weather interpretation codes. Weather code is calculated from cloud cover analysis, precipitation and snowfall. As barely no information about atmospheric stability is available, estimation about thunderstorms is not possible.
pressure_msl surface_pressure	hPa	Atmospheric air pressure reduced to mean sea level (msl) or pressure at surface. Typically pressure on mean sea level is used in meteorology. Surface pressure gets lower with increasing elevation.
cloudcover Instant	%	Total cloud cover as an area fraction.
et0_fao_ evapotranspiration	mm	ET_0 Reference Evapotranspiration of a well watered grass field. Based on FAO-56 Penman-Monteith equations ET_0 is calculated from temperature, wind speed, humidity and solar radiation. Unlimited soil water is assumed. ET_0 is commonly used to estimate the required irrigation for plants.

vapor_pressure_deficit	kPa	Vapor Pressure Deificit (VPD) in kilopascal (kPa). For high VPD (>1.6), water transpiration of plants increases. For low VPD (<0.4), transpiration decreases.
windspeed_10m windspeed_100m	km/h	Wind speed at 10 or 100 meters above ground. Wind speed on 10 meters is the standard level.
winddirection_10m winddirection_100m	ř	Wind direction at 10 or 100 meters above ground.
windgusts_10m	km/h	Gusts at 10 meters above ground of the indicated hour. Wind gusts in CERRA are defined as the maximum wind gusts of the preceding hour.
soil_temperature_0_to_7cm soil_temperature_7_to_28cm soil_temperature_28_to_100cm soil_temperature_100_to_255cm	řC	Average temperature of different soil levels below ground.
soil_moisture_0_to_7cm soil_moisture_7_to_28cm soil_moisture_28_to_100cm soil_moisture_100_to_255cm	m^3/m^3	Average soil water content as volumetric mixing ratio at 0-7, 7-28, 28-100 and 100-255 cm depths.

Table 2.3: All feature selected from Open-Meteo with description

The new feature `is_day` will contain two values: 0 and 1, representing *Night* and *Day*, respectively. To create this feature, we add 0 with a frequency of 5 minutes until sunrise, then add 1 until sunset, and finally add more 0s until the end of the day. We repeat this process for each day in the dataset to create a new column that, for each timestamp, will indicate whether it is day or night.

Algorithm 4 is_day feature generation Algorithm

Require: dataset table, sunrise and sunset table
Ensure: `dataset` **is not** empty, `sunrise` and `sunset` tables **match** `dataset` days

```
is_day ← empty table column
for each day in dataset.days do
    temp_is_day ← empty table column
    day_start ← sunrise[day]                    ▷ get sunrise timestamp for that day
    day_start ← find the nearest dataset.timestamp to day_start
    day_end ← sunset[day]
    day_end ← find the nearest dataset.timestamp to day_end
    /* if sunrise[day] = 7:04 then day_start = 7:05 */

    /* 0 means Night, 1 mean Day */
    temp_is_day ← add 0s starting from 00:00 to day_start
     as 5 minute frequency
    temp_is_day ← add 1s starting from day_start to day_end
     as 5 minute frequency
    temp_is_day ← add 0s starting from day_end day end
     as 5 minute frequency
    append to is_day column temp_is_day values
end for
merge is_day column to dataset table
```

2.1.3 Dealing with gaps

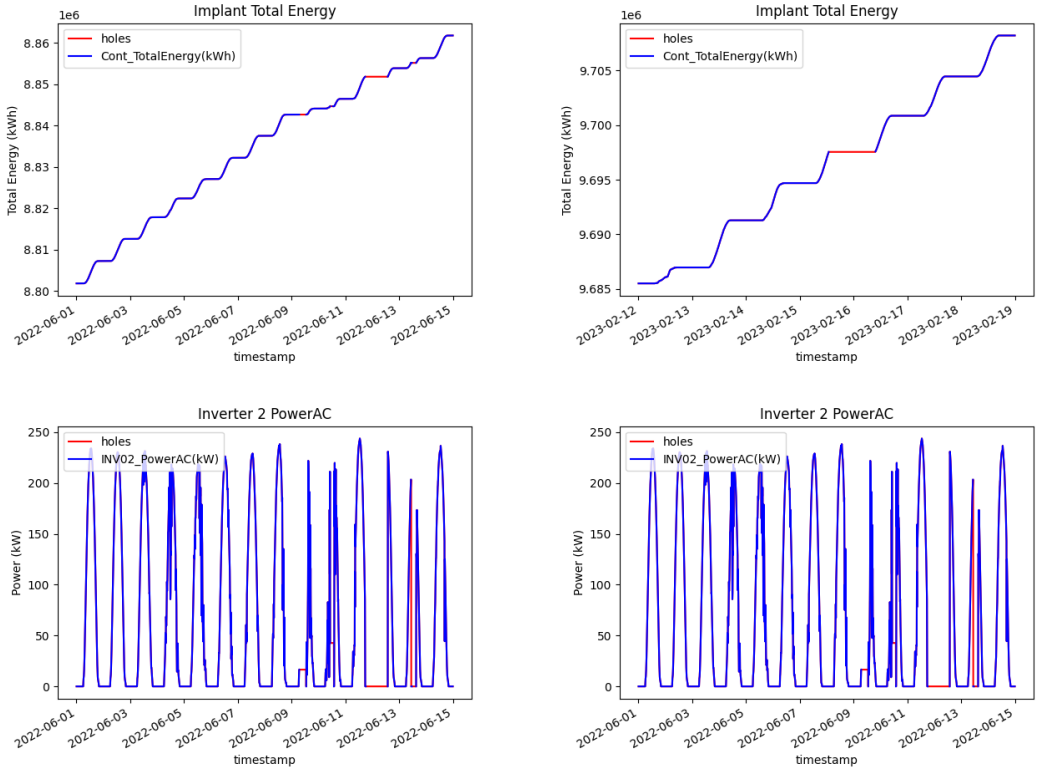


Figure 2.2: Some dataset "holes". The charts at the top refer to the Implant's Total Energy, while those at the bottom refer to the Inverter 2's Power. The charts on the left range from 01-06-2022 to 15-06-2022, while those on the right range from 12-02-2023 to 19-02-2023.

As we can see from Figure 2.2, there are certain periods within the dataset (highlighted in red) where data is missing, which we refer to as "holes". Leaving these gaps in the dataset causes problems during model training (hindering the correct calculation of the loss function), and therefore, they need to be removed. One possible approach for holes removal is to perform a *fill* operation: filling the gap with the first available non-null value. This tactic may be considered acceptable only if the time span involves just a few timestamps. However, if we are talking about several hours or even days, it significantly distorts the overall production and instantaneous power trends, resulting, especially in very unfortunate cases, with extremely abnormal production curves.

The solution we have adopted to address this problem is the removal of the entire day when a hole occurs. For example, if we have a data gap from 12-02-2023

23:00 to 13-02-2023 10:40, the days 12-02-2023 and 13-02-2023 will be completely removed. With this method, we lose some data, but as we will see later, the number of gaps is not extremely high, and this data loss is not debilitating. The following algorithm summarizes what has just been described.

Algorithm 5 Holes Removal Algorithm.

Require: dataset table
Ensure: `dataset` **is not** empty
  holes ← find all timestamp from `dataset` table, where there are some `Nans` inside the columns
  for each row **in** `dataset.rows` **do**
    if row.timestamp **is in** `holes` **then**
      drop `row` from `dataset` table
    end if
  end for

Timestamp
2022-06-09
2022-06-10
2022-06-11
2022-06-12
2022-06-13
2022-06-28
2022-06-29
2022-06-30
2022-08-26
2022-09-23
2022-10-06
2023-02-03
2023-02-15
2023-02-16
2023-03-26

Table 2.4: Timestamps deleted after running the Algorithm 5

2.1.4 Target Feature

For the deep learning models that we will introduce later, it could be very difficult to learn to predict the Total Missing Energy, as it is a curve that grows constantly, potentially to infinity. These models need to have features that vary within a limited range of values. To achieve this, we transformed the variable `Cont_TotalEnergy(kWh)` from cumulative to instantaneous values of produced energy, thereby limiting its values within a finite range. This new feature was added to the dataset with the name `target`.

Algorithm 6 Cumulative Energy to Instant Energy Algorithm.

Require: dataset table
Ensure: `dataset` **is not** empty
  target ← empty column
  for each (prev_val, val) **in** dataset["Cont_TotalEnergy(kWh)"] **do**
    diff ← val − prev_val
    append `diff` value to `target` column
  end for
  /* Now we have target column with first value missing */
  /* We can add a 0 as fist value because its night time */
  append (in head) 0 to `target` column
  merge `target` and `dataset`

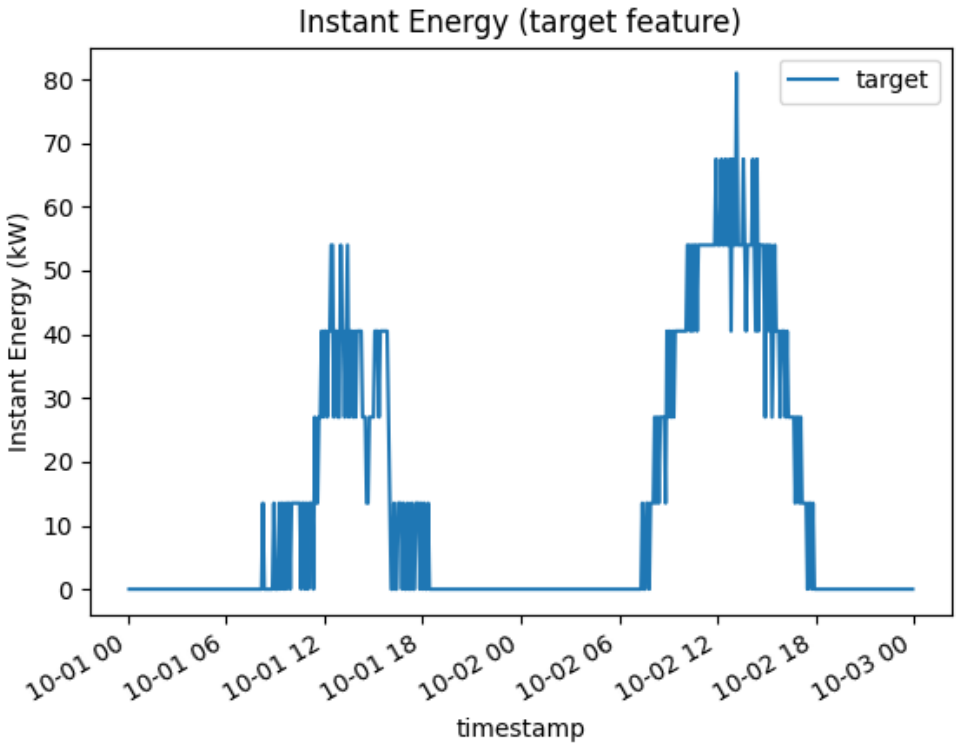


Figure 2.3: Instant Energy (`target` feature) 2 days plot.

2.1.5 Re-Sampling

Finally, after all the operations described above, we have a dataset with samples taken at 5-minute intervals. As we can see from Figure 2.3, this data appears to be quite noisy, which could pose challenges during the training phase. Moreover, most of the reports provided by photovoltaic plant operators are at a 15-minute frequency. To prevent potential issues and align with this standard, we have written and applied a re-sampling procedure.

Algorithm 7 15 minute Re-Sampling Algorithm.

Require: dataset table
Ensure: `dataset` **is not** empty
  for each feature **in** dataset.features **do**
    if feature **is** Cumulative **then**
      feature ← sample `feature` column using 15 min. sampling rate
    else
      feature ← aggregate values every 15 minutes and sum them to form a single value
    end if
  end for

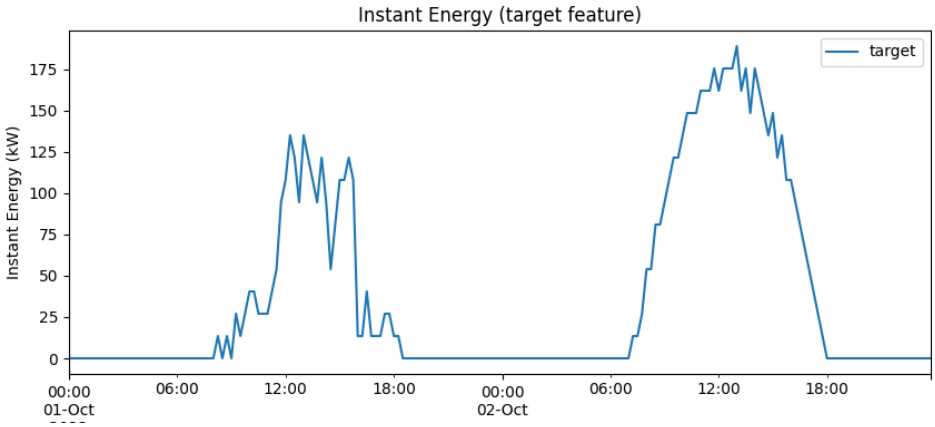


Figure 2.4: Instant Energy feature on 15 minute re-sampled dataset.

2.2 Feature Selection

After applying all the procedures described in the previous section, we find ourselves with a dataset composed of 764 columns. This high number of features would make it nearly impossible to train various models, both from a purely hardware perspective and due to the excessive information redundancy. In an attempt to minimize the number of features as much as possible, we applied and combined the results of two essential tools for feature selection: the *Correlation Matrix* and the *Power Predictive Score*.

Figure 2.5: Correlation Matrix

As we can infer, even with a naive approach, all the information from the various String Boxes and Junction Boxes is perfectly summarized by the data provided by the various Inverters (see Section 1.2); a similar reasoning can be applied to the features coming from *Other* (those starting with `other_`), as they do not contribute any useful information for energy estimation. Consulting the Correlation Matrix (Figure 2.5), we can see that this naive reasoning is confirmed because the features of the junction boxes have a high level of correlation with those of the Inverters. In general, we want to retain features that are minimally correlated with each other and aim to discard highly correlated ones, keeping only those that are more representative.

Applying this approach straightforwardly, however, is not the best choice because it could lead to the loss of important information. Let's consider an example: let's take into account the features `INV01_Power(kW)`, `INV02_Power(kW)`, and `INV03_Power(kW)`; by examining the correlation matrix, these three features are highly correlated, and, therefore, we should retain only one of them. However, all three are important for predicting our target. For instance, if Inverter 1 experiences any issues and is not operating at 100%, this will impact the total energy production, making all three of these features indispensable. Therefore, in conjunction with the correlation matrix, we have utilized the *Power Predictive Score* (PPS): an asymmetric, data-type-agnostic score that can detect linear or non-linear relationships between two or more columns. The score ranges from 0 (no predictive power) to 1 (perfect predictive power).

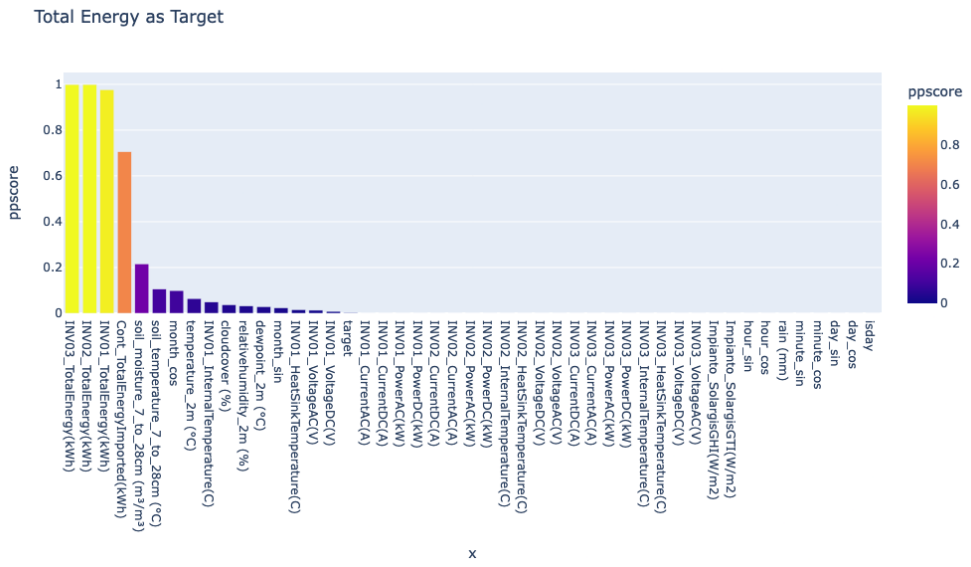


Figure 2.6: All features PPS using `Cont_TotalEnergy(kWh)` as target.

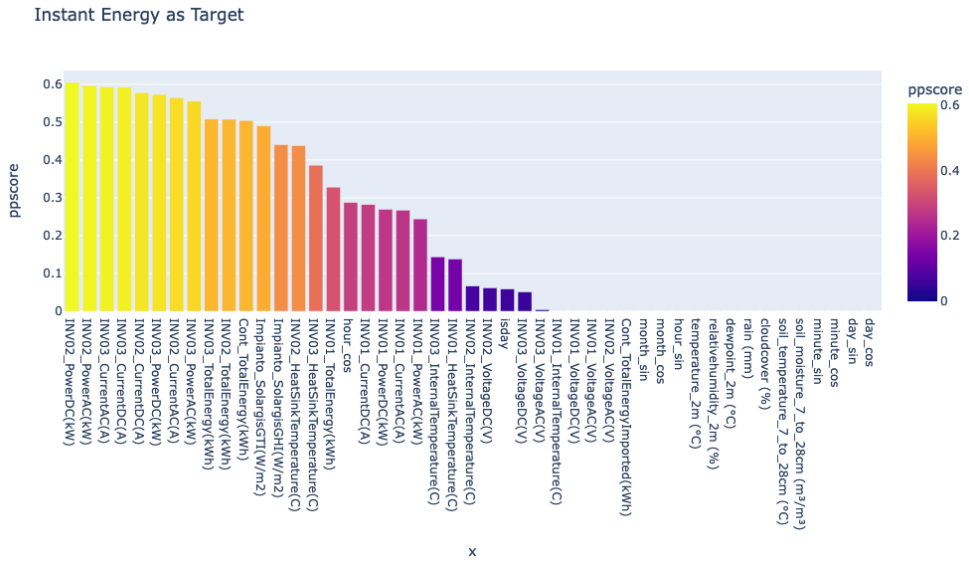


Figure 2.7: All feature PPS using Instant Energy (`target`) as Target.

As we can see in Figures 2.6 and 2.7, the features that perform better in predicting the instantaneous energy production trend are indeed those of the Inverters, confirming the reasoning described earlier. Finally, by combining the results obtained from the Correlation Matrix and the Power Predictive Score, we obtain a final set of 28 features.

INV01_PowerAC(kW)	INV03_TotalEnergy(kWh)
INV01_PowerDC(kW)	INV01_HeatSinkTemperature(C)
INV02_PowerAC(kW)	INV02_HeatSinkTemperature(C)
INV02_PowerDC(kW)	INV03_HeatSinkTemperature(C)
INV03_PowerAC(kW)	Impianto_SolargisGHI(W/m2)
INV03_PowerDC(kW)	Impianto_SolargisGTI(W/m2)
INV01_CurrentAC(A)	rain (mm)
INV01_CurrentDC(A)	cloudcover (%)
INV02_CurrentAC(A)	hour_sin
INV02_CurrentDC(A)	hour_cos
INV03_CurrentAC(A)	day_sin
INV03_CurrentDC(A)	day_cos
INV01_TotalEnergy(kWh)	month_sin
INV02_TotalEnergy(kWh)	month_cos

Table 2.5: All feature selected after this phase.

2.3 Dataset Splitting

After completing the Feature Selection phase (Section 2.2), we decided to divide the obtained dataset into three different sets:

- *Training*: It covers the period from 01-06-2022 to 28-02-2023. This set will be used during the training phase to allow the model to learn the plant's performance and how to predict energy trends during various gaps. It starts in June because there are some issues with the Inverter 1's features before that.
- *Validation*: It spans from 01-03-2023 to 31-03-2023. It will be used in the training phase to assess the presence of overfitting or if this phase might have failed.
- *Testing*: It starts on 01-04-2023 and continues until 30-04-2023. This set will be used in the Evaluation phase to estimate the model's performance on a data period it has never seen before.

	Start	**End**	**Rows**
Training	01-06-2022	28-02-2023	24864
Validation	01-03-2023	31-03-2023	2880
Testing	01-04-2023	30-04-2023	2880

Table 2.6: Summary table of how the dataset was divided.

Chapter 3

Deep Learning Models

In this chapter, we will present three models based on different architectures: the first, our baseline, is based on a Multi-Layer Perceptron, which will be the most basic one. The second model is based on a Recurrent Neural Network, and finally, the last one utilizes the newer and more powerful Transformer layers. We will describe in detail their architecture, the various training phases, and any associated issues along with their strengths and weaknesses for each model.

It's important to note that these models were implemented using only the PyTorch library. Each run was preceded by setting a seed (the same for all runs of each model) to allow for comparisons and ensure the repeatability of the various operations. For training phase, the dataset created in the previous Chapter (2) was used and divided into training, validation, and testing sets as described in Section 2.3. All of these phases were executed on a machine with 64 GB (2x32GB) DDR5 5600MT/s of RAM, an AMD Ryzen 9 7950X3D processor, and an Nvidia 4090 GPU.

3.1 MLP

In this section, we will introduce the first model, which we will refer to as the baseline model, based on a Multi-Layer Perceptron. We will analyze its architecture, the training phase, and its final performance.

3.1.1 Architecture

This neural network is designed to predict the instantaneous energy output over a period of exactly 2 days. To do this, it requires input data that includes the performance of the system (features selected during the Preprocessing phase, see Chapter 2) for exactly one day before and one day after the period in question.

This enables it to understand how the system is performing and, consequently, provide the energy output trend. The model consists of 6 main layers:

- **Input layer**: This is the first layer of the network. It takes two input tensors: *before* and *after*. These tensors represent the day before and the day after the specific period we want to predict. They have the shape `[BATCH_SIZE, 96, 33]`, where 96 is the number of timestamps in our dataset that make up one day, and 33 represents the features obtained from the Data Preprocessing phase. These tensors are then flattened and concatenated to be passed to the subsequent layer. The output of this layer goes through a Batch Normalization layer, and the Rectified Linear Unit (ReLU) activation function is used.
- **Hidden layers**: In total, there are 4 layers, each of which takes the output of the previous layer as input and reduces the number of neurons by half. Batch Normalization is applied to the result, and the Rectified Linear Unit (ReLU) is used as the activation function.
- **Output layer**: The final layer of our network, it takes the result from the previous layer and outputs the value of the Instantaneous Energy Produced during the specific period. It produces a tensor with a shape of `[BATCH_SIZE, 192, 1]`. The SoftPlus function is used as the final activation function.

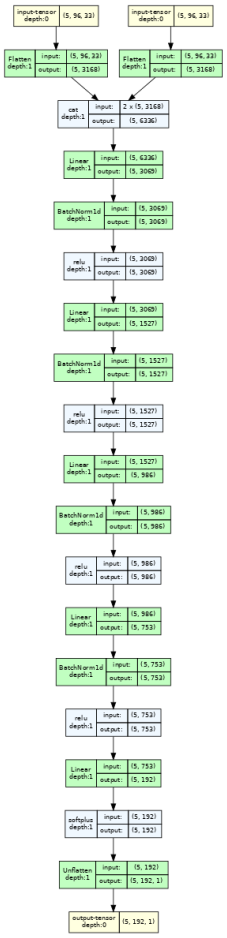


Figure 3.1: Beseline Model architecture visualization.

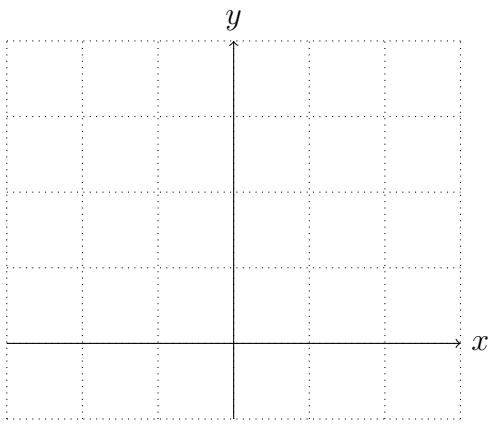


Figure 3.2: Rectified Linear Unit function. $ReLu(x) = \max(0, x)$

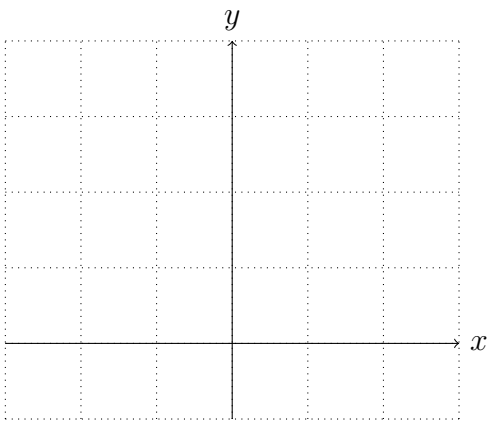


Figure 3.3: SoftPlus function. $SoftPlus(x) = \frac{1}{\beta}\log(1 + e^{\beta x})$

3.1.2 Training

The model was trained by artificially creating gaps of two days in length within the training dataset. These gaps were passed to the network in the format described earlier, and the output result was compared to the actual instantaneous energy produced during the gap. An *Early Stopping* procedure was implemented to prevent training from continuing if the model was not improving its performance. A procedure, called *Save Best*, has also been integrated, which saves the model to a file whenever the Validation Loss improves. The validation dataset was applied in this phase to assess the learning progress at the end of each epoch. A normalization procedure of the area was applied to the model's output in relation to that of the gap to ensure that the output starts exactly from the last value of the instantaneous energy produced *before* and ends exactly with the first value of the energy produced *after*. Adam was used as the optimizer, and L1Loss (Equation 3.1) served as the loss function. We chose to set the batch size parameter to 10, the learning rate λ to 0.01, a maximum of 100 epochs, and a patience value of 20 for Early Stopping.

$$L = \{l_1, \ldots, l_N\}^\top, \quad l_n = |x_n - y_n|, \tag{3.1}$$

$$\ell(x, y) = \text{mean}(L) \tag{3.2}$$

Total Parameters (#)	26543400
Trainable Parameters (#)	26543400
Training Duration (s)	24.0
Model Size (MB)	101.3

Table 3.1: Baseline Model specification.

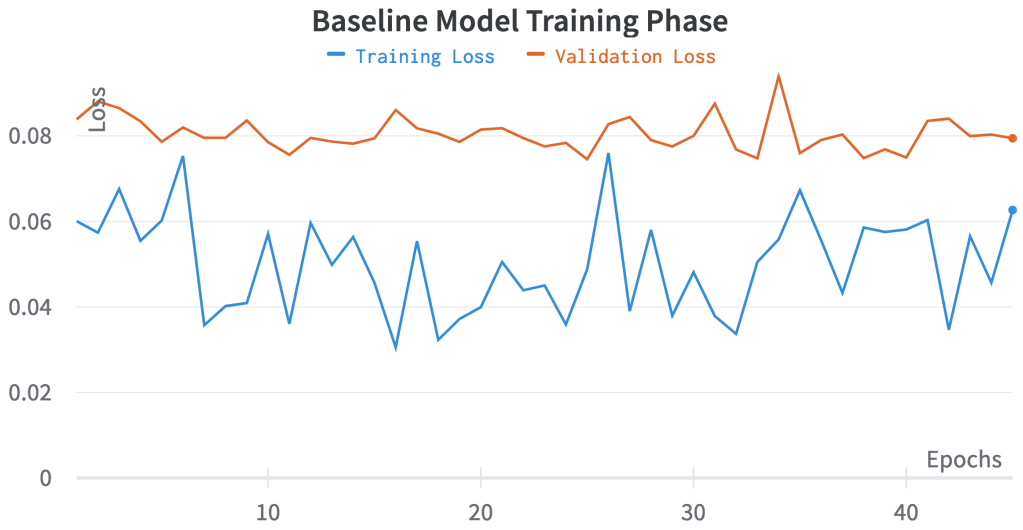


Figure 3.4: The chart displays the loss progression during the training phase. The blue line represents the Training Loss, while the orange line represents the Validation Loss.

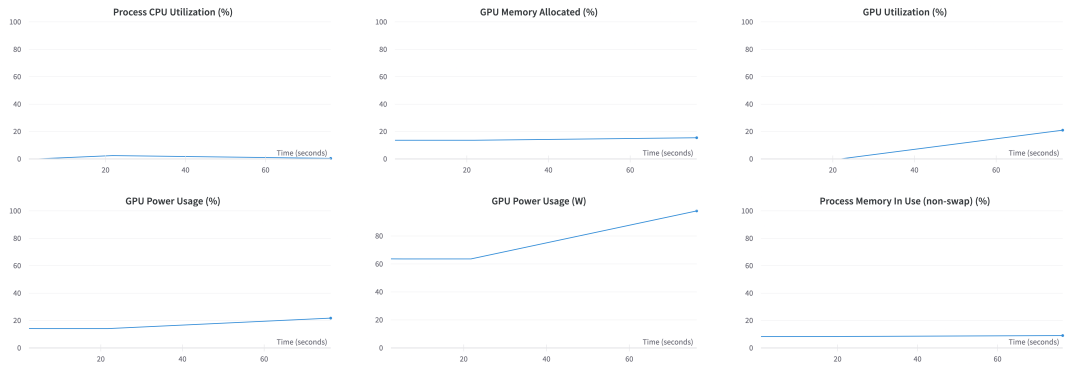


Figure 3.5: System resources utilized during the Training phase.

Algorithm 8 MLP model Training Algorithm

Require: train/validation datasets; Baseline Neural Network Model
  Batch Size ← 10
  Learning Rate $\lambda \leftarrow 0.01$
  Epochs ← 100
  Patience ← 20
  loss ← L1Loss()
  Optimizer ← Adam Optimizer

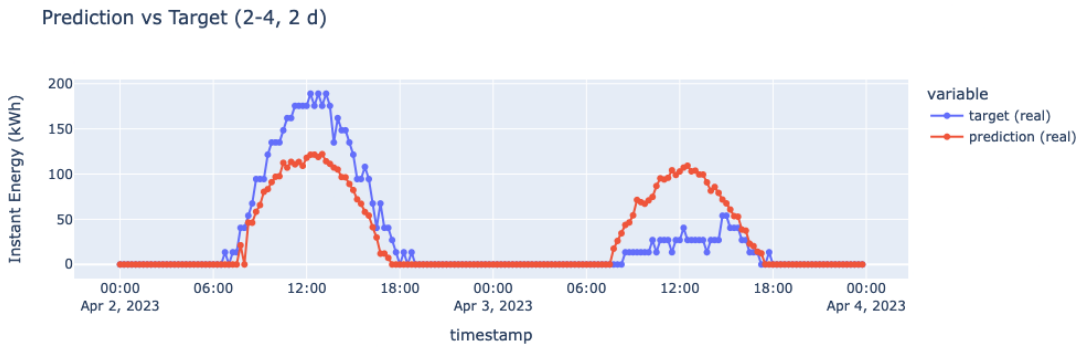
  for each epoch **in** epochs **do**
    for each (batch_id, before, after, target) **in** train.next_batch() **do**
      train_prediction ← model(before, after) ▷ Model inference
      train_prediction ← $\frac{\text{train\_prediction} \cdot \sum \text{train\_prediction}}{\sum target}$ ▷ Area normalization
      train_loss ← loss(train_prediction, target)
      Optimizer step
      Back Propagation
    end for
    stop computing gradient
    for each (batch_id, vbefore, vafter, vtarget) **in** validation.next_batch() **do**
      val_prediction ← model(vbefore, vafter) ▷ Model inference
      val_prediction ← $\frac{\text{val\_prediction} \cdot \sum \text{val\_prediction}}{\sum vtarget}$ ▷ Area normalization
      val_loss ← loss(val_prediction, vtarget)
    end for
    check for Early Stopping
    check for Save Best Result
    start computing gradient
  end for

3.1.3 Evaluation

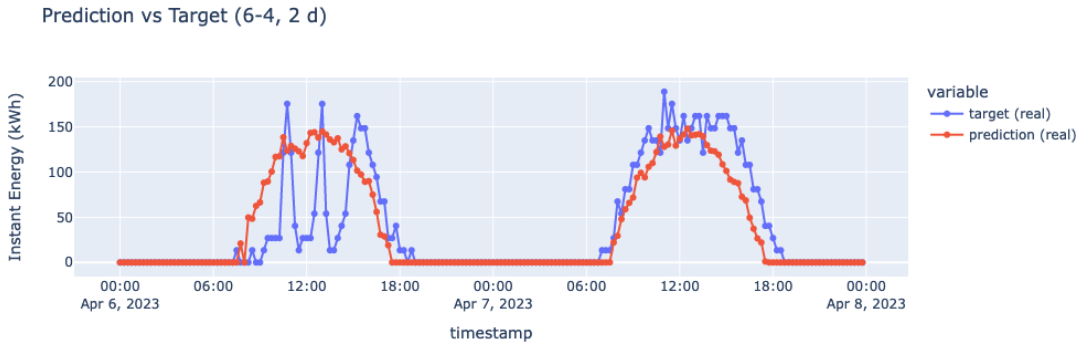
Dal grafico della fase di training mostrato in Figura 3.4 possiamo evicnere come questa sia terminata con successo dal fatto che le due curve (training loss e validation loss) non divergono ma rimangono ad una distanza abbastanza costante e possiamo anche capire che non ci sia presenza di overfitting.

Dai grafici in Figura 3.5 possiamo notare come la macchina su cui è stato addestrato questo modello sia stata sfruttata relativamente poco, in particolare facciamo notare come l'utilizzo della GPU non ha superato il 30% e la sua memoria è stata malapena occupata per il 20%. Questo può portarci ad affermare che il modello può essere anche addestrato su macchino molto meno performanti della nostra. Il modello risulta essere anche relativamente leggero sia in termini di

memoria, soli 100 MB (vedi Tabella 3.2) sia per il tempo di allenamento che non ha superato i 30 secondi. Il tempo di inferenza risulta estremamente veloce non superando il secondo.

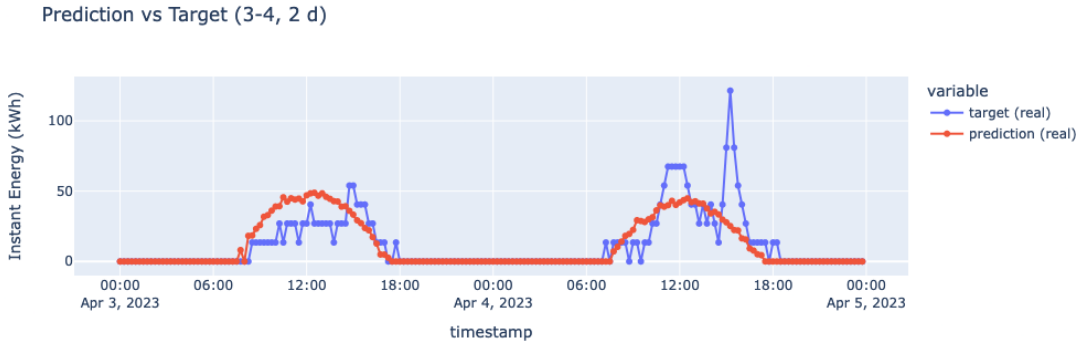


(a)

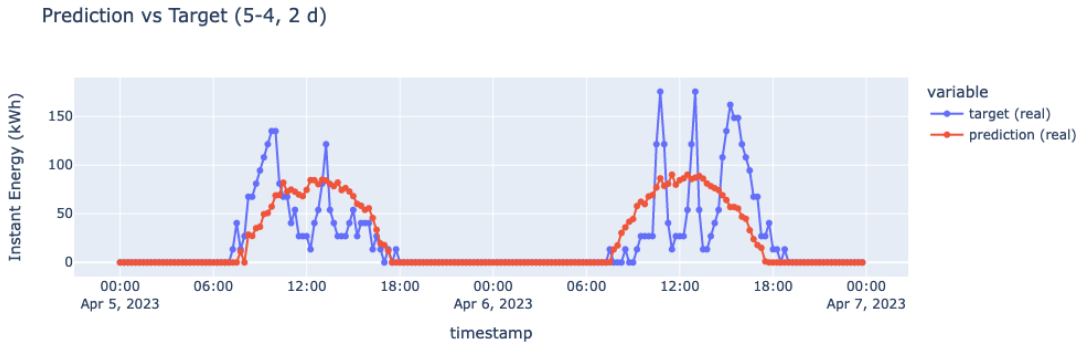


(b)

Figure 3.6: The figure displays two predictions made by the model, where you can observe the network's output (in red) and the ground truth (in blue). The model was tested on two gaps, each covering a period of 2 days. The first gap (a) spans from 02-04-2023 to 04-04-2023, and the second gap (b) spans from 06-04-2023 to 06-08-2023.



(a)



(b)

Figure 3.7: Caption

TODO: completare con un brefe spieghino dove mostro i risultati (i dati) MAE, MAPE e R2. Spieghino sul fatto che termina la giornata prima e alcune riflessioni sul modello

3.2 RNN

We will now introduce a second model based on Recurrent Neural Networks. We will examine its structure, analyze the training phase, and finally evaluate its performance, demonstrating how it manages to outperform the previous architecture, the Baseline model introduced in Section 3.1.

3.2.1 Architecture

This model is designed to make the most of the capabilities of Recurrent Neural Networks to predict the behavior of instant energy production during a variable-length gap period.

The required input format is similar to the previous one: a tensor named *before* containing the plant's status one day before the gap, a tensor named *after* with information about the plant the day after the hole, and an additional tensor named *future* containing features that are *always* available even during blackout periods. These features specifically include `Solargis`, `isday`, and information obtained from OpenMeteo. This last tensor should assist the model in prediction by helping it better understand the state of meteorological conditions and adapt instant production accordingly.

Now, let's display the main elements that compose it.

- **Input**: As described earlier, the model requires 3 input tensors: *before*, *after*, and *future*. The first two should contain information from just one day before and after the gap, respectively, while the last one will have weather features to support predictions, which can vary in length. An example of input could be `[BATCH_SIZE, 96, 33]` for *before*, `[BATCH_SIZE, 96, 33]` for *after*, and `[BATCH_SIZE, 288, 18]` for *future*.

- **Encoder**: This component's role is to identify the most important features described by the *before* and *after* tensors to understand how the plant is operating. These two tensors are passed to two different GRU layers, which will analyze these time series and extract key information. The final output of each GRU will then be extracted and concatenated together to form the Hidden State for the *Decoder*.

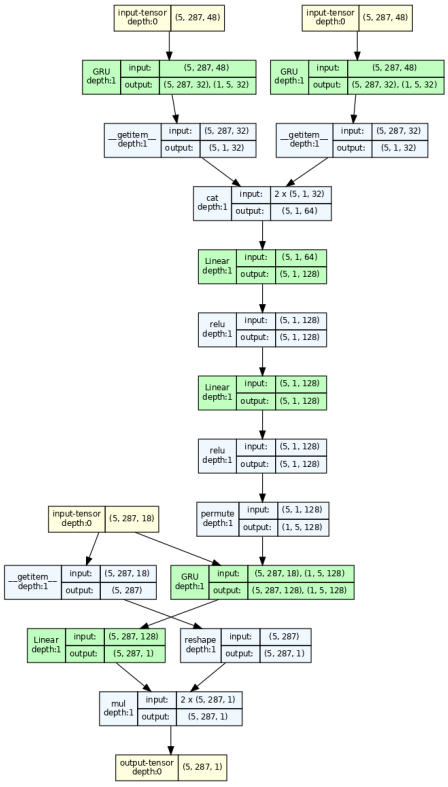


Figure 3.8: Recurren Neural Netowrk Based Model architecture visualization.

- **Middle layer**: This layer consists of 2 Fully Connected Layers. It takes

as input the result of the Encoding phase and reshapes it to the necessary format for input to the *Decoder*. For both layers, the ReLU function is used as the activation function.

- **Decoder**: Its role is to reprocess the information obtained from the Encoder and predict the instant energy produced during the blackout. It consists of a GRU layer that takes the *future* tensor as input and uses the result of the Middle Layer as the hidden state.

- **Output Layer**: This is the final layer of this architecture, and its task is to transform the output from the Decoder into a tensor of shape `[Batch_Size, Gap_Len, 1]`, representing the instant energy produced by the plant during the gap. It consists of a Fully Connected Layer. Its output is then multiplied by the feature `isday` to ensure that the model's prediction is always zero during the night.

3.2.2 Training

The model was trained to learn the trend of the instant energy production curve of the plant during gaps of variable lengths, ranging from a minimum of 1 day to a maximum of 4 days. These gaps were artificially generated in the training dataset and provided to the model as described earlier, with attention to grouping gaps of the same length in batches to avoid complications during training.

In this case, an *Early Stopping* and *Save Best* procedure were implemented as well to ensure that the model always saves the best-performing model and to prevent resource waste.

The validation dataset was applied in this phase to conduct an initial and preliminary evaluation of the training process and highlight potential issues. A normalization procedure was also applied to scale the prediction area with respect to the ground truth area. This allows the model to learn the shape of the curve and not exceed the limits of energy production during the gap.

The Adam optimizer was used, and the L1Loss was applied as the loss function. The batch size was set to 10, the learning rate (λ) was set to 0.01, a maximum of 100 epochs was set, and the patience was set to 20.

Total Parameters (#)	92737
Trainable Parameters (#)	92737
Training Duration (s)	39.0
Model Size (KB)	366.6

Table 3.2: RNN based model specification.

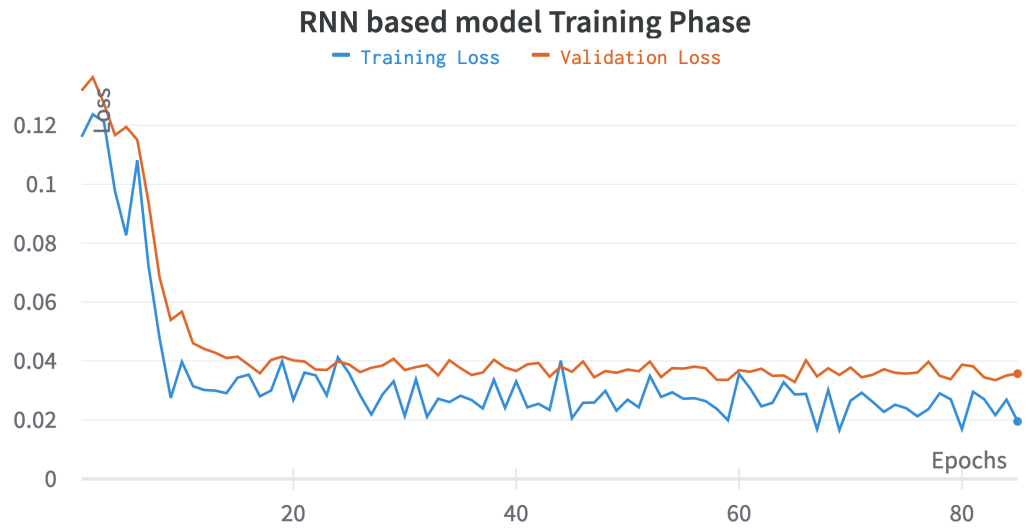


Figure 3.9: The chart displays the loss progression during the training phase.The blue line represents the Training Loss, while the orange line represents the Validation Loss.

Figure 3.10: System resources utilized during the Training phase.

Algorithm 9 RNN model Training Algorithm

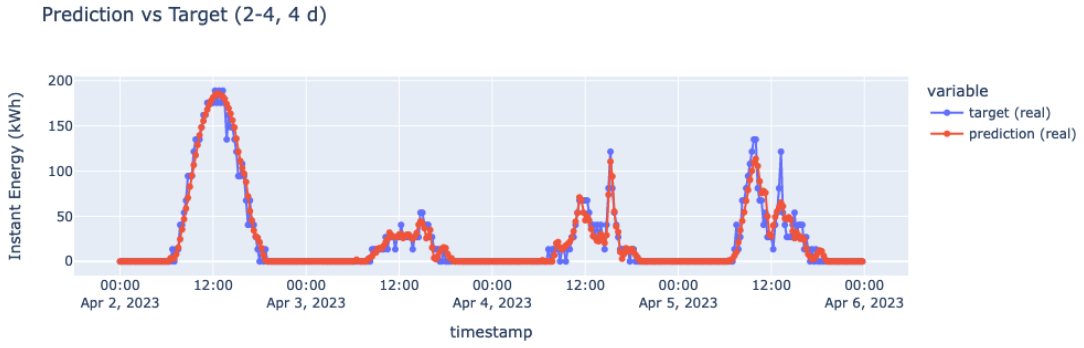
Require: train/validation datasets; Baseline Neural Network Model
  Batch Size ← 10
  Learning Rate $\lambda \leftarrow 0.01$
  Epochs ← 100
  Patience ← 20
  loss ← L1Loss()
  Optimizer ← Adam Optimizer
  Min Gap size ← 1 day
  Max Gap size ← 4 days

  for each epoch **in** epochs **do**
    for each (batch_id, before, after, future, target) **in** train.next_batch() **do**
      train_prediction ← model(before, after, future) ▷ Model inference
      train_prediction ← $\frac{\text{train\_prediction} \cdot \sum \text{train\_prediction}}{\sum target}$ ▷ Area normalization
      train_loss ← loss(train_prediction, target)
      Optimizer step
      Back Propagation
    end for
    stop computing gradient
    for each (batch_id, vbefore, vafter, vfuture, vtarget) **in** validation.next_batch() **do**
      val_prediction ← model(vbefore, vafter, vfuture) ▷ Model inference
      val_prediction ← $\frac{\text{val\_prediction} \cdot \sum \text{val\_prediction}}{\sum vtarget}$ ▷ Area normalization
      val_loss ← loss(val_prediction, vtarget)
    end for
    check for Early Stopping
    check for Save Best Result
    start computing gradient
  end for

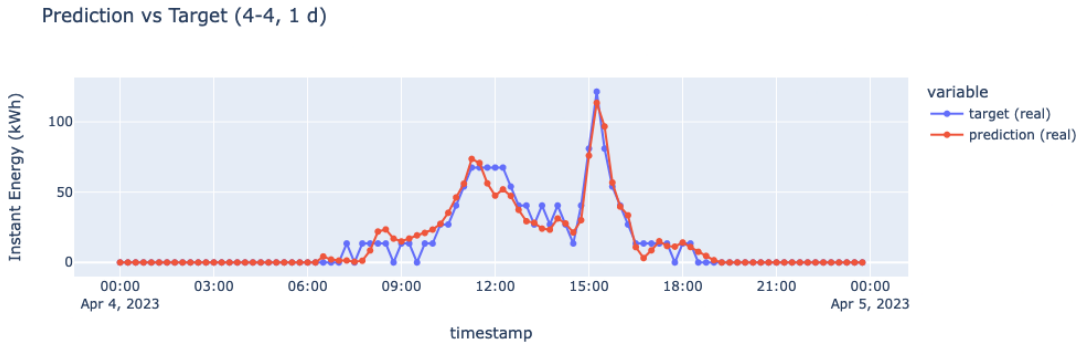
3.2.3 Evaluation

From the graphs shown in Figure 3.9, we can see that the training phase was successful, and after an initial descent, the loss values remained relatively constant without displaying any abnormal trends. The statistics presented in Figure 3.10 reveal that the machine at our disposal was not fully utilized, suggesting that this architecture can be trained and used on less powerful computers than the one we have. Additionally, the model appears to be very lightweight, both in terms of the relatively small number of parameters and its weight, which doesn't exceed 400

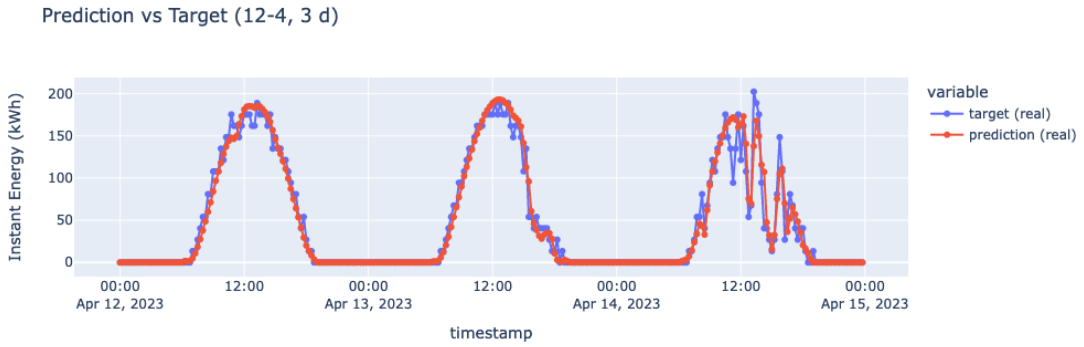
KB.



(a)



(b)

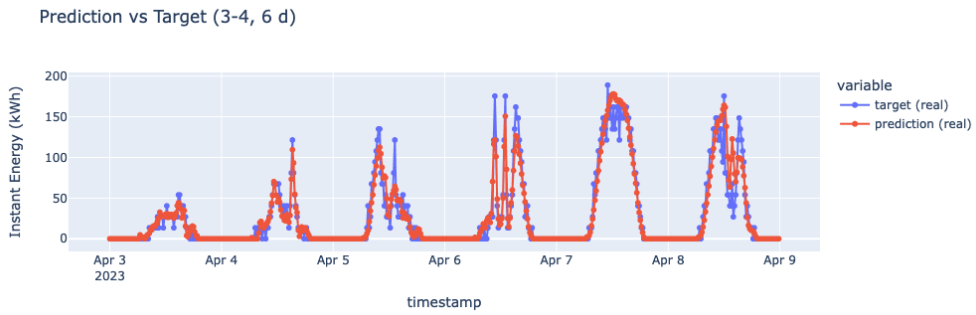


(c)

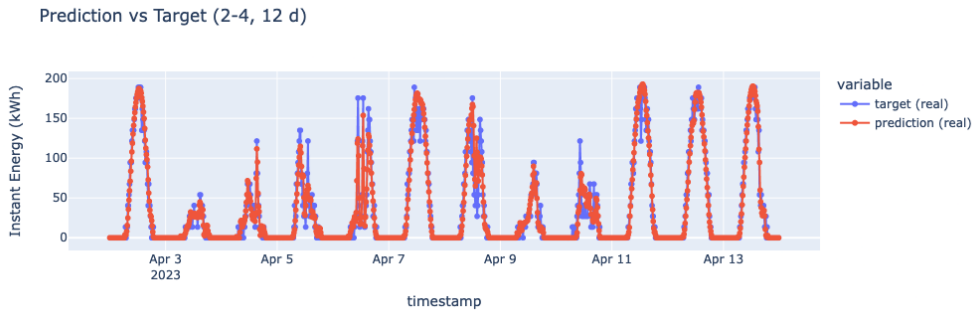
Figure 3.11: In the figure, three model predictions (in red) are shown alongside the ground truth (in blue) for gaps of varying sizes. These predictions were made using data from the testing dataset.

Analyzing some model predictions as shown in Figure 3.11, we can observe how the real instant energy production curves (displayed in blue) are closely approximated by the model (red curves). The model effectively understands the plant's behavior, even managing to predict production spikes. We can also see that energy production is consistently zero during the night in the predictions, and it adeptly captures the day/night cycle by gradually reducing production as sunset approaches.

In graph (a), we can see a 4-day gap from 02-04-2023 to 06-04-2023. The first two days are predicted very well, while in the last part, we can see that a production spike was not detected. In graph (b), we can observe a 1-day gap on 04-04-2023. We notice that the overall trend is almost entirely approximated correctly, except for some time intervals around 12:00. The last graph (c) is related to a 3-day gap, and we can see that the first two days are approximated well, while in the last day, the production spikes are identified but with values not entirely similar to those of the ground truth.



(a)



(b)

Figure 3.12: The graphs depict two model predictions for gaps that exceed the maximum limit of days set during the training phase. The first one (a) shows a 6-day gap, while the second one (b) presents a 12-day gap.

It is interesting to note how the model still performs well even when presented with gaps that exceed the maximum length set during training. In Figure 3.12, two graphs are shown: (a) represents a 6-day gap (two days longer than the maximum length), and (b) a 12-day gap. Given these results, we can conclude that the model can generalize effectively and predict instant energy production trends with considerable reliability.

3.3 Transformers