POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering
(Control Technologies for Industry 4.0)

Master's Degree Thesis

HPC Energy Consumption Optimization

Supervisors

Prof. Carlo NOVARA

Eng. Mario BONANSONE – Modelway S.r.l.

Candidate

Luca ROSMARINO

OCTOBER 2024

Contents

List of Tables

List of Figures

Chapter I
Introduction

1.1 Thesis Outline and Contributions

Datacenters play an increasingly crucial role in supporting a great variety of services, from cloud computing and data storage, to streaming services and artificial intelligence. Their growing high computing resources utilization, along with all the cooling system mechanisms to manage the generated heat, have led to significant environmental impacts and CO_2 emissions.

Looking at the state of the art, many studies focused their attention on minimizing datacenters energy consumption by optimizing the frequency and voltage at which the processors run or designing predictive control mechanisms to optimally manage the cooling systems. However, few studies proposed solutions which leverage the possibility of shutting down the idle nodes and allocating jobs to the nodes with the lowest increment of predicted temperature.

This thesis manages the jobs scheduling of the "Marconi 100" datacenter considering temperature forecasts and availability of all nodes. In other words, this optimizer is a thermal-aware job scheduler, with the addition of being able to turn off nodes that are not required for current workloads, thus reducing unnecessary energy waste, and then reactivate nodes when jobs demand increases. This approach led to 38% of energy consumption reduction through a one-day simulation.

In Chapter 1 we give a general introduction to datacenters' architectures and working principles followed by environmental impacts and some proposed solutions to mitigate power consumption.

Then, in Chapter 2 we give an overview of the "Marconi 100" datacenter history, architecture, plugins and tools, followed by a deepen statistical analysis of available data.

Chapter 3 introduces the mathematical formulations of the system identification models which will be used to make power and temperature predictions.

System identification training results are reported in Chapters 4, followed, in Chapter 5, by a rapid introduction in Genetic Algorithm working principles, and then a detailed analysis of the formulation and implementation of the optimization problem.

In Chapter 6, several tests are performed to verify the correct functioning of the optimization algorithm and then conclusions and future work are reported in Chapter 7.

The job scheduling functioning, which will be further deepened in Chapter 5, can be briefly summarized in the following way:

- **Job submission:** Users can submit jobs specifying the maximum number of CPU/GPU cores requested and a time limit within which the job must be completed.
- **Job power consumption prediction:** All job submission info is fed into a Random Forest Regressor which, as described in Chapter 4, is trained to estimate the mean power that a submitted job will require during its execution.
- **Temperature prediction of nodes:** The aforementioned power consumption prediction, along with the measured/approximated nodes' ambient temperature, are fed into a LSTM Neural Network (more info in Chapter 4). For each node, we estimate the temperature increment in case of job allocation and execution on that specific node. For our purposes, a 3-step ahead prediction is sufficient, in this way, knowing that each step corresponds to twenty seconds, we are predicting temperature behavior one minute in the future.
- **Optimization problem:** At this point we have an array containing estimated temperature increments for each available node. This array, as it will be discussed in Chapter 5, is the key part of the Genetic Algorithm objective function.

The minimization of the objective function returns an optimization matrix which states:

- To which node the submitted job must be allocated to maintain the lowest temperature increment.

- Which nodes are ON, and among them which are $Busy$ or $Idle$.
- If it is necessary to turn on new nodes.
- If it is possible to power off idle nodes.

1.2 Understanding datacenters and their operations

A datacenter is a physical room, building or facility that houses IT infrastructure for building, running and delivering applications and services.

Datacenters started out as privately owned facilities, for the exclusive use of one company. Nowadays their usage is shared among multiple organizations and customers and supports a great variety of services, from cloud computing and data storage to video streaming and artificial intelligence.

There exist different types of data center facilities depending on workload and business needs.

- In **Enterprise Datacenters** all IT infrastructure and data are kept on premises. This means that the datacenter is dedicated exclusively to the need of one company, allowing the management of private information.
- **Public Cloud Datacenters** share computational resources to multiple customers through an internet connection. They are larger than conventional datacenters, contain thousands of servers and miles of connection equipment.
- In a **Managed Datacenter**, a client company leases dedicated servers and storage from the provider, and the provider handles the client company's administration, monitoring and management of the servers. This option is ideal for companies who do not prefer to host their infrastructure by using the shared resources of a public cloud datacenter.

The core part of datacenters is represented by servers. They are high-performance computers that deliver applications, services and data to end-users' devices.

In **rack-mount architecture**, servers are stacked on top of each other in a rack. Each rack has its own power supply, switches, ports, cooling fans, processors, memory and storage.

Most servers include some local storage capability to enable the most frequently used data to remain close to the CPU. Other data is transmitted over a standard Ethernet connection on dedicated servers with hard disk and/or solid-state drives.

Processors, memories and networking equipment generate heat. For this reason, most datacenters are equipped with a combination of air and liquid cooling systems to keep servers within the proper temperature ranges. Air cooling is air conditioning, while liquid cooling consists in pumping liquid directly to hardware components.

1.3 Energy consumption and environmental impact

In the last years, datacenters growing computing resources utilization, along with all the cooling systems mechanisms to manage the generated heat, have led to significant environmental impacts and CO_2 emissions.

According to the International Energy Agency (IEA), since 2010, the number of internet users worldwide has more than doubled, while global internet traffic has increased 25-times. As a consequence, significant expansions in datacenters computing capability have been made to manage the growing demand for data processing, storage and cloud computing.

The IEA estimates global datacenter electricity consumption in 2022 was 240-340 TWh, or around 1-1.3% of global final electricity demand. These data exclude energy used for

cryptocurrency mining, which was estimated to be around 110 TWh, accounting for 0.4% of annual global electricity demand.

An astonishing result is that only 16 nations in the world consume more than the sum of all the existing datacenters. Looking at the graph below, we notice that datacenters, all together, absorb more energy than the majority of industrialized countries such as Italy.

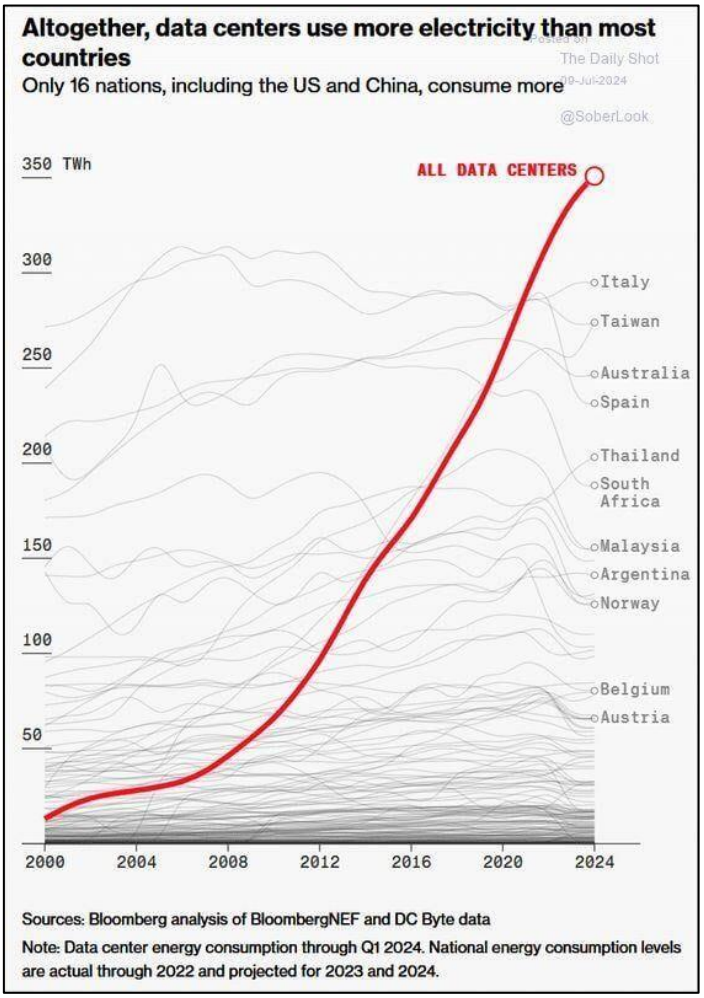


Figure 1.1: Data centers energy consumption

From 2015 to 2022, datacenters workloads increased by 340%, and consequently energy usage by 20-70%, with a growing by 20-40% annually. Talking about cryptocurrency

mining energy consumption growth, the change between 2015 and 2022 is estimated to be about 2300-3500%.

These numbers have a direct impact on greenhouse gas (GHG) emissions. In fact, datacenters accounted for around 330Mt CO_2 emissions in 2020, equivalent to 0.9% of energy-related GHG emissions.

In addition to their operational energy usage, datacenters are also responsible for "embodied" life cycle emissions, including raw materials extraction, manufacturing, transport and end-of-life disposal or recycling.

1.4 State of the art in Datacenter Energy Optimization

Several papers have been published regarding datacenter energy optimization, and each of them proposes different approaches. In fact, the energy consumed depends on multiple sources (CPUs P-state, cooling systems, workload distribution etc.) and identifying them and their behavior can help developing optimization strategies.

Daniele Cesarini and Andrea Bartolini, from the University of Bologna propose one of the most interesting approaches based on CPU frequency control. The paper is named: "*COUNTDOWN Slack: A Run-time Library to Reduce Energy Footprint in Large-scale MPI Applications*".

The targets of this paper are HPC scientific applications, composed of parallel processes running on a cluster of compute nodes interconnected with a high-bandwidth low-latency network. Processing can exchange data using a message-passing interface (MPI) library that can send explicit messages.

There exist two main policies for managing power consumption of HPC facilities and both aim at scaling down the P-state of the computation units.

Proactive policies always execute a newly encountered code region at the highest available P-state to measure the performance of that region. Performance evaluation is based on the collection of data such as execution time, number of retired instructions etc. These parameters are used to compute a new P-state for the next time the code region is encountered.

In **Reactive policies**, when specific events occur, the runtime triggers well-defined actions. In COUNTDOWN Slack, a timeout counter is implemented to ignore portions of code which are too short to cause a P-state transition. In this case, there is not a history table to predict the next P-state to be assigned, since the runtime only needs to intercept specific events.

COUNTDOWN Slack approach is based on scaling down the P-state in slack times of the application reducing the frequency but leaving unaltered the performance for both computation and data copy regions. For this purpose, every time an application calls a collective primitive, so when an application must wait for the execution of parallel tasks, its relative P-state is reduced to the minimum available one. When all processes reach the collective primitive, the maximum frequency is restored.

Chapter II

The “Marconi 100” Datacenter

2.1 History and applications

The “Marconi 100” project owes its birth to European funding to Cineca in 2015, and it is part of a series of investments that Europe is making in the supercomputing area. This project was won by Lenovo, which acquired the x86 server business from IBM in 2014.

The first step was the “Marconi” project, in which the computing power was mainly produced by x86 processors. But the high demand of computing resources in scientific research and deep learning models led to the necessity of equipping the servers with graphics accelerators. This allowed Cineca to significantly increase the performance of the single server.

This gave birth to the “Marconi 100”, which name comes from the Nvidia Volta 100 graphics accelerators mounted in the new servers. This means that the Marconi 100 is a continuation of the Marconi project.

Much of the Marconi 100’s computing power comes from NVIDIA chipsets. They are designed exclusively for supercomputing, and in particular for artificial intelligence. They allow concurrent processing for the hundreds of cores that are part of the GPU with a clock typically from 1 to 1.3 GHz. This means that in the unit of time given by a clock cycle the machine is able to do a certain number of operations simultaneously.

The Marconi 100 datacenter enables high performance computing to simulate complex scenarios in healthcare, space exploration and weather forecasting sectors. Additionally, it supports scientific research for university and industry, AI projects and big data analysis.

2.2 System architecture and racks spatial distribution

The Marconi 100 datacenter consists of 980 compute nodes and 8 login nodes, connected with a network architecture called DragonFly.

The login nodes and the computed nodes are the same. Each node consists of 2 Power9 sockets, each of them with 16 cores and 2 Volta GPUs. In this sense each node owns 32 cores and 4 GPUs.

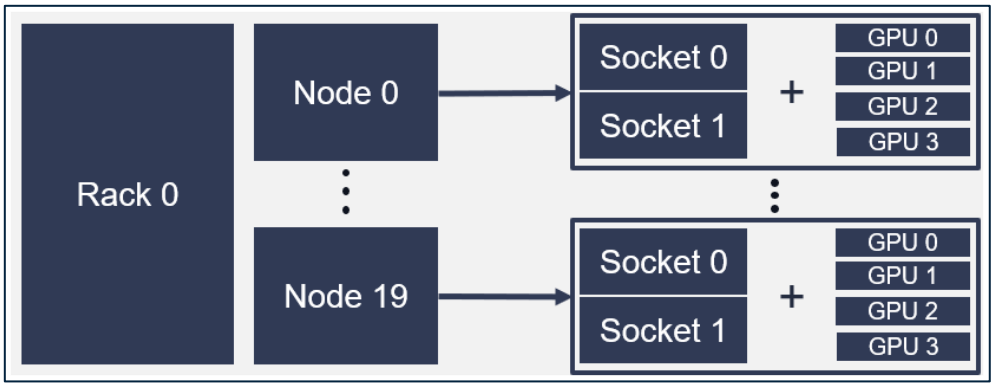


Figure 2.1: System Hardware Architecture

Nodes are stacked on top of each other in a rack. Each rack consists of 20 nodes, for this reason the datacenter room hosts 49 computing racks and one rack for login nodes. Each rack has its own power supply, switches, ports, cooling fans, processors, memory and storage.

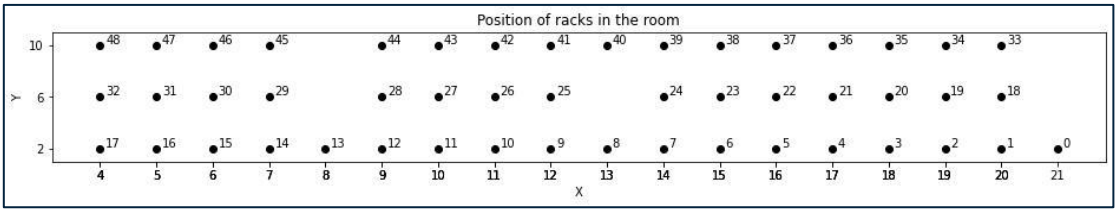


Figure 2.2: Racks spatial distribution

The theoretical peak performance of the single node is about 32 TFlops. This value is due to 0.8 for the CPU part and 7.8x4 for the four GPUs.

The theoretical peak performance for the entire datacenter is therefore 988*(0.8+4*7.8) = 31.6 PFlops.

2.3 Plugins and data acquisition

Marconi 100 offers different plugins for data extraction, providing the tools to monitor datacenter operations. They are designed to collect info related to hardware performance, environmental conditions and energy usage.

The **IPMI** and **Ganglia** plugins collect all the sensor data provided by cluster nodes, and most of their features are represented in figure [2.1].

Each feature is sampled with a sampling time of 20 seconds, which is particularly precise in IPMI features, and less accurate in Ganglia ones. For this reason, to synchronize the timestamps it is necessary to perform some linear interpolation.

Most of the data that these two plugins provide will be used during the training, validation and testing of the node's temperature model in Chapter 4.

Another fundamental plugin is the **Job Table**, which collects information regarding the jobs executed on the cluster. The information collected is provided by users at submission time. This tool is fundamental to train the Random Forest regressor for the prediction of the mean power consumption of the submitted jobs, as will be discussed in Chapter 4.

Table 2.1: Available features

Features	Description	Sampling time	Unit
ambient	Temperature at the node inlet	20 s (per node)	°C
pX_coreY_temp	Temperature of core n. Y in the CPU socket n. X (X = 0..1 , Y = 0..23)	20 s (per node)	°C
gpuX_core_temp	Temperature of the core for the GPU id X (X = 0,1,3,4)	20 s (per node)	°C
gpuX_mem_temp	Temperature of the memory for the GPU id X (X=0,1,3,4)	20 s (per node)	°C
dimmX_temp	Temperature of DIMM module n. X (X = 0..15)	20 s (per node)	°C
PCIE_temp	Temperature at the PCIExpress slots	20 s (per node)	°C
gpuX_power_usage	Power usage for the device	~20 s (per node)	W
pX_power	Power consumption for the CPU socket n. X (X=0..1)	20 s (per node)	W
pX_mem_power	Power consumption for the memory subsystem for the CPU socket n. X (X=0..1)	20 s (per node)	W
pX_io_power	Power consumption for the I/O subsystem for the CPU socket n. X (X=0..1)	20 s (per node)	W
total_power	Total node power consumption	20 s (per node)	W

Table 2.2: Job Table

Features	Description	Unit
User_ID	User ID for a job	Integer
Job_ID	Job ID	Integer
Start_time	Time execution begins	Timestamp
Time_limit	Maximum runtime	Seconds
Nodes_list	Nodes allocated to the job	List of integers
Num_GPUs	Maximum number of GPUs usable by job	Integer
Max_CPUs	Maximum number of CPUs usable by job	Integer
Job_power	Node power consumption during job execution	Watts

2.4 ExaMon and ExaMon-X tools

The continuous growth of datacenters scale, along with the increment of the number of hardware components, brought to a bigger complexity in maintenance, requiring more sophisticated tools and strategies. Here comes to help the predictive maintenance approach, which consists in a set of practices to intervene on Industrial Internet of Things (IioT) systems before critical conditions might arise.

In particular, ExaMon and ExaMon-X, developed by the University of Bologna, are two tools which are responsible for collecting and processing data from datacenters. Recently, they have been deployed on two supercomputers hosted at CINECA: D.A.V.I.D.E. and **MARCONI**.

They are based on monitoring a set of HW and SW resources to characterize datacenters behavior and create digital twins for predictive maintenance tasks.

ExaMon has been designed to handle big data from many heterogeneous sources. It is based on "**collector**" components which read data from several sensors scattered across the system and deliver them to the upper layers of the software stack. There are collectors with direct access to HW resources and collectors that sample data from SW applications.

The implemented communication protocol is the **MQTT**, which is based on a publish-subscribe pattern. It requires three different agents: the **publisher**, that sends data on a specific topic; the **subscriber**, that receives data from the appropriate topic; the **broker**, which handles the communication between publishers and subscribers. In ExaMon the collectors assume the role of publishers.

ExaMon-X provides a set of functionalities for predictive maintenance, which can exploit both the data stored in the database, and the online data stream. Each of these functionalities could be used independently from the others or in combination to perform macro activities.

In the case of predictive maintenance models, ExaMon is responsible for collecting data, while ExaMon-X is used to process this data. In particular, machine learning models can be trained to predict hardware failures.

To make an example, a single hard disk (HD) has a very low failure probability, but in datacenter context, with a high number of HDs, failure probability is not negligible. Current solutions to this problem are based on redundancy, but this approach is expensive and in case of fault, lead to a temporary shutdown of services.

For this reason, the health status of HDs can be monitored through a machine learning algorithm, which is fed by disk temperature, failures in reading/writing and many others.

Figure 2.3: Marconi 100 Datacenter

2.5 Features Data Analysis and Normalization

As described in Section [2.3], the "Marconi 100" provides different tools to retrieve data. IPMI and Ganglia plugins are the ones used for our purposes.

Table 2.3: Features of interest

Features	Description	Sampling time	Unit
ambient	Temperature at the node inlet	20 s (per node)	°C
pX_coreY_temp	Temperature of core n. Y in the CPU socket n. X (X = 0..1 , Y = 0..23)	20 s (per node)	°C
dimmX_temp	Temperature of DIMM module n. X (X = 0..15)	20 s (per node)	°C
gpuX_core_temp	Temperature of the core for the GPU id X (X = 0,1,3,4)	20 s (per node)	°C
gpuX_mem_temp	Temperature of the memory for the GPU id X (X=0,1,3,4)	20 s (per node)	°C
PCIE_temp	Temperature at the PCIExpress slots	20 s (per node)	°C
pX_power	Power consumption for the CPU socket n. X (X=0..1)	20 s (per node)	W
GpuX_power_usage	Power usage for the device	~ 20 s (per node)	W
total_power	Total node power consumption	20 s (per node)	W

In table [3.1] we report a list of features that may have a physical connection with temperature. In fact, our objective can be resumed in this way: given the power consumption and ambient temperature of a node, we shall train a model capable of estimating the future node temperature behavior.

For these reasons, we retrieved all those features which represent temperature and/or power consumption of different hardware components of the datacenter. Thinking about it, the most energy-intensive devices are the GPUs, followed by the CPU cores. For this reason, we concentrate our attention on them, highlighting in table [3.1] the corresponding features.

The data is sampled every 20 seconds, with high time precision, but this is not always the case. The exception is the "*GpuX_power_usage*" feature, which sampling frequency is sometimes imprecise. To address this problem, we opted for linearly interpolating the data, and to extract the timestamps of interest. In this way all features are synchronized.

All these data will be used to train the models illustrated in Chapter 3. For this reason, it is crucial to normalize them in order to improve accuracy and efficiency of the system identification procedures. **Z-score** normalization is a technique used to rescale data so that it has zero mean and unitary variance. In formula:

$$z_i = \frac{x_i - \mu}{\sigma} \tag{2.1}$$

Where z_i is the normalized data point, μ is the mean value of the entire dataset, σ is the standard deviation of the entire dataset, and x_i is the real value of the data point.

The advantage of normalization is to give the same scale to all input data of a system identification model, to center data around a null mean value and with unitary variance, and to help machine learning algorithms to perform better.

2.6 Data aggregation

Our thermal system identification of the nodes of the datacenter needs the aggregation of the pX_coreY_temp, $gpuX\_core\_temp$ and $gpuX\_power\_usage$ features.

To make it clearer, let's take the cores temperature data of a single CPU. For each node, we have 2 CPU sockets, composed of 16 cores each. What we have at our disposal is a time series CSV file for each of these cores, with data from January to September 2022.

To train a model, we need to feed it with input and target features which are physically consistent with each other. Our objective is to predict the CPU socket temperature given

the ambient temperature and the CPU socket power consumption. But the available features are the temperature behaviors of the single cores, and the power usage of a group of cores (a socket). For this reason, it is necessary to aggregate in some way the temperatures to obtain something which represents the temperature behavior of each socket.

The solution adopted was to compute, for each timestamp, the average temperature of all the cores belonging to the same CPU socket. In this way, giving to the model the socket power consumption data, along with the ambient temperature behavior, we are able to estimate the target. The target is the aggregated temperature computed before.

Following the same reasoning, we aggregated the GPU core temperatures, to get the temperature behavior of the entire GPU chipset. At the same time, we aggregated the single GPU cores power consumptions to approximate the overall GPU mean power consumption over time.

2.7 Job table

The “Job table” is a CSV file which stores information regarding the jobs executed by the datacenter. The information collected is provided by users at submission time.

Table 2.4: Job features

Features	Description	Unit
User_ID	User ID for a job	Integer
Job_ID	Job ID	Integer
Start_time	Time execution begins	Timestamp
Time_limit	Maximum runtime	Seconds
Nodes_list	Nodes allocated to the job	List of integers
Num_GPUs	Maximum number of GPUs usable by job	Integer
Max_CPUs	Maximum number of CPUs usable by job	Integer

As we can observe in the above table, each user and each job are univocally identified by an ID. In this sense, it is possible to verify which jobs are in execution or have been requested by which users.

Users can specify the maximum number of CPU and/or GPU cores necessary for the execution of their jobs.

Each job is characterized by a start time, that in general does not correspond to the submission time. In fact, the job scheduler can delay the execution of a process considering the time limit constraint. This is because the user, when querying the datacenter, can specify the maximum runtime of the job.

The job scheduler decides where to allocate running jobs, and this information is stored into a list of integers, where each number corresponds to a specific node of the datacenter. Notice that it is possible that more jobs are allocated simultaneously to the same node,

but to different partitions of it. At the same time, it is also possible that more nodes are executing different parts of the same job.

2.8 Jobs statistics

In this paragraph we are going to discuss statistics representing the behavior of jobs submitted to the Marconi 100. This is fundamental in understanding which type of jobs are executed and their intrinsic characteristics.

The first aspect that we want to analyze is the average number of CPU and GPU cores required by the users for the execution of their jobs. From table [3.3] and table [3.4], we notice that a low number of resources are allocated to most of the jobs. Typically, the ranges of CPU cores vary from $[1, 8]$ to $]32, 64]$. From the GPU point of view the trend is similar, in fact most jobs require only one GPU. But it is evident that a non-negligible number of jobs need all the four GPUs of a single node.

Table 2.5: Number of CPU cores requested

# CPUs	# Jobs	%
[1, 8]	1366386	57.08%
]8, 16]	175376	7.33%
]16, 32]	65135	2.72%
]32, 64]	93562	3.91%
]64, 128]	557187	23.28%
]128, 256]	53711	2.24%
]256, 512]	27658	1.16%
]512, 1024]	13073	0.55%
]1024, +inf)	41695	1.73%

Table 2.6: Number of GPU cores requested

# GPUs	# Jobs	%
0	430226	17.98%
1	1252728	52.33%
2	40594	1.70%
3	16926	0.70%
4	653309	27.29%

Regarding the execution time, most of the jobs are executed in less than 30 minutes, with 45% of them which are completed in less than 10 minutes. This means that jobs submitted to the datacenter require simple calculations or small datasets, which reduces the time needed for data input, processing and output. Another possible motivation could be that the system has a great number of available resources, so processes do not wait in queue for shared hardware. At the same time, a great role could have the parallelization, which means that multiple tasks can be run simultaneously, leading to faster overall execution.

Table 2.7: Jobs execution time

Execution time	# Jobs	%
≤ 10 min	1078999	45.08%
≤ 20 min	1917441	80.10%
≤ 30 min	2017360	84.28%
≤ 40 min	2038210	85.15%
≤ 50 min	2050479	85.66%
≤ 60 min	2068391	86.41%
> 60 min	325341	13.59%

As we said in Section [3.3], users during submission shall specify the time limit within which jobs must be completed. To verify the reliability of the Marconi 100 datacenter, we compared the real execution times with the maximum required ones. The result is that more than 97% of jobs succeed in being executed before their deadline.

Table 2.8: Deadline accomplishment

	# Jobs	%
Real execution time > execution time limit	58426	2.44%
Real execution time <= execution time limit	2335357	97.56%

Another interesting statistic is knowing how much in advance deadline are fulfilled. From table [3.7] we notice that nearly the 90% of jobs, considering the submission time and their execution, are terminated in half of their time limit.

Table 2.9: Jobs execution time (deadline percentage)

Job completed in (%) of the max execution time	# Jobs	%
10%	1012724	42.31%
20%	1828386	76.38%
30%	1955222	81.67%
40%	2019000	84.34%
50%	2113059	88.27%
60%	2181877	91.15%
70%	2219978	92.74%
80%	2239480	93.55%
90%	2258201	94.34%
100%	2335357	97.56%

One more interesting statistical information to consider is for how much time the datacenter nodes have been idle from January to September 2022. This detail is useful to understand for how much time those nodes could have been turned off to save energy.

Looking at figure [2.4] we notice that the datacenter was idle for the 24% of the time (on average). In other terms, considering an observation window of 273 days (9 months), datacenter nodes have been powered on and idle for about 65 days.

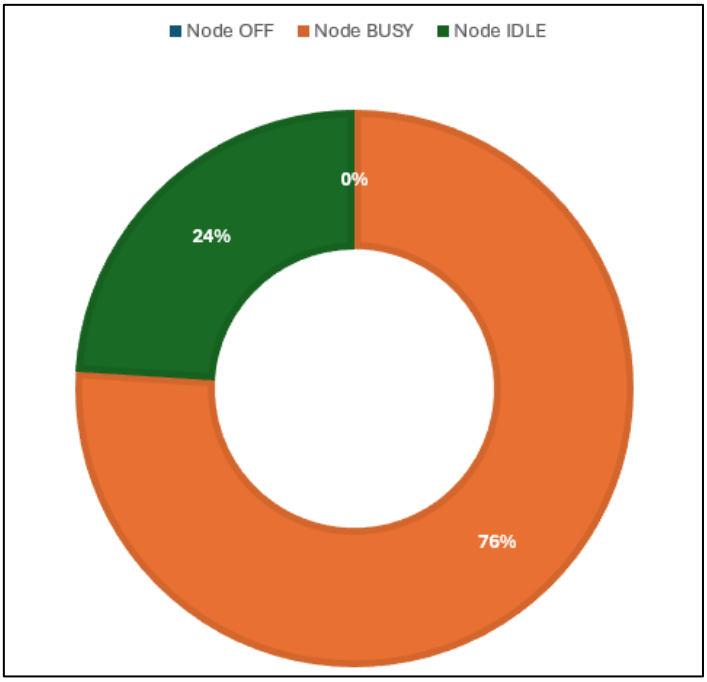


Figure 2.4: Nodes Busy/Idle time

Chapter III

System Identification

3.1 Introduction

System identification is the process of creating mathematical models that describe how a dynamic system behaves, based on observed data. These data are typically collected by measuring, from the system, inputs and corresponding outputs over time. The goal is to capture the relationship between inputs and outputs, in order to predict future or actual behavior of the system's dynamics.

Choosing a model structure is a crucial step. There are three main types of models based on how much we know about the system: white-box, grey-box and black-box models.

White-box models assume that all the physical phenomena involved in the system are exactly known, for this reason the structure of the model is obtained by applying the principles of physics.

Grey-box models are obtained by applying the principles of physics, but the physical parameters are not exactly known and need to be estimated from experimental data.

Black-box models are used when very little is known about the system. In this case the structure of the equations describing the model is selected based on "general" a-priori information. The parameters involved in the equations do not have, in general, any physical meaning.

3.2 Linear models

When working with linear models the three most used approaches are ARX (AutoRegressive with eXogenous input), OE (Output Error), and ARMAX (AutoRegressive Moving Average with eXogenous input) models. Each method offers a different way to handle noise and capture the system's dynamics.

3.2.1 ARX (AutoRegressive with eXogenous inputs)

The ARX model captures the relationship between the current output and past outputs and inputs in the following way:

$$y(t) + a_1 y(t-1) + \cdots + a_{n_a} y(t-n_a) = b_1 u(t-1) + \cdots + b_{n_b} u(t-n_b) + e(t) \quad (3.1)$$

$$with\ n_a \geq 0, n_b \geq 1$$

The term $a_1 y(t-1) + \cdots + a_{n_a} y(t-n_a)$, represents the autoregressive component, showing how past outputs influence the current one.

The term $b_1 u(t-1) + \cdots + b_{n_b} u(t-n_b)$, represents the exogenous inputs, showing how past inputs influence current output.

The error term $e(t)$, which enters in the equation as a white noise with zero mean and unitary variance, represents the unmodeled dynamics.

Introducing the following polynomials in z^{-1} (unitary delay operator):

$$A(z) = 1 + a_1 z^{-1} + \cdots + a_{n_a} z^{-n_a} \quad (3.2)$$

$$B(z) = b_1 z^{-1} + \cdots + b_{n_b} z^{-n_b} \quad (3.3)$$

the ARX equation can be mathematically written as:

$$A(z)y(t) = B(z)u(t) + e(t) \tag{3.4}$$

$$y(t) = \frac{B(z)}{A(z)}u(t) + \frac{1}{A(z)}e(t) \tag{3.5}$$

$$y(t) = G(z)u(t) + H(z)e(t) \tag{3.6}$$

Or can be represented with the following block diagram:

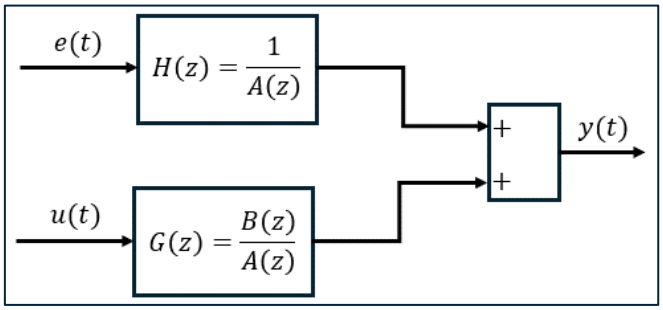


Figure 3.1: ARX block diagram

If $n_a = 0$, then $A(z) = 1$ and $y(t)$ is modeled as a FIR (Finite Impulse Response), in which the output signal is represented as a finite sum of past input values, without involving any feedback from past outputs.

3.2.2 OE (Output Error)

The OE model separates the dynamics of the system from the noise affecting the output:

$$y(t) + f_1 y(t-1) + \cdots + f_{n_f} y\left(t - n_f\right) = b_1 u(t-1) + \cdots + b_{n_b} u(t - n_b) \quad (3.7)$$
$$with\ n_f \geq 0, n_b \geq 1$$

where the real output $y(t)$ is corrupted by a white measurement noise:

$$\tilde{y}(t) = y(t) + e(t) \quad (3.8)$$

Introducing the following polynomials in the z^{-1} variable:

$$F(z) = 1 + f_1 z^{-1} + \cdots + f_{n_f} z^{-n_f} \quad (3.9)$$

$$B(z) = b_1 z^{-1} + \cdots + b_{n_b} z^{-n_b} \quad (3.10)$$

the undisturbed OE equation can be written as:

$$F(z)y(t) = B(z)u(t) \quad (3.11)$$

$$\tilde{y}(t) = y(t) + e(t) = \frac{B(z)}{F(z)} u(t) + e(t) \quad (3.12)$$

$$\tilde{y}(t) = G(z)u(t) + e(t) \quad (3.13)$$

Or can be represented with the following block diagram:

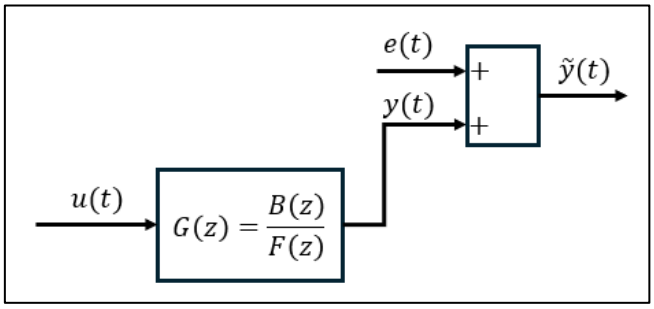


Figure 3.2: OE block diagram

If $n_f = 0$, then $F(z) = 1$ and $\tilde{y}(t)$ is modeled as a FIR (Finite Impulse Response).

3.2.3 ARMAX (AutoRegressive Moving Average)

The ARMAX model is similar to ARX, but the error term $e(t)$ is substituted by a linear combination of past samples of the white noise.

$$y(t) + a_1 y(t-1) + \cdots + a_{n_a} y(t-n_a) = b_1 u(t-1) + \cdots + b_{n_b} u(t-n_b) + \\ + e(t) + c_1 e(t-1) + \cdots + c_{n_c} e(t-n_c) \tag{3.14}$$

By introducing the following polynomials in the z^{-1} variable:

$$A(z) = 1 + a_1 z^{-1} + \cdots + a_{n_a} z^{-n_a} \tag{3.15}$$

$$B(z) = b_1 z^{-1} + \cdots + b_{n_b} z^{-n_b} \tag{3.16}$$

$$C(z) = 1 + c_1 z^{-1} + \cdots + c_{n_c} z^{-n_c} \tag{3.17}$$

the ARMAX equation can be written as:

$$A(z)y(t) = B(z)u(t) + C(z)e(t) \tag{3.18}$$

$$y(t) = \frac{B(z)}{A(z)}u(t) + \frac{C(z)}{A(z)}e(t) \tag{3.19}$$

Or can be represented with the following block diagram:

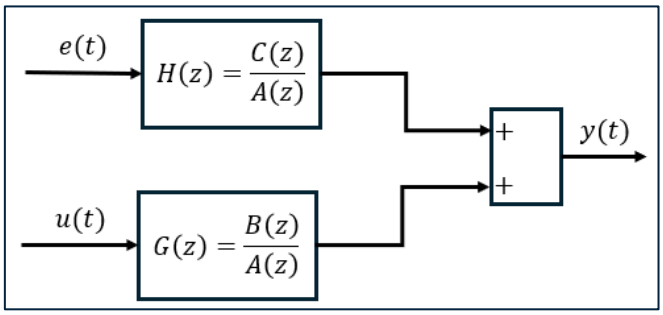


Figure 3.3: ARMAX block diagram

3.3 Nonlinear Models

Unlike linear models, which assume a proportional relationship between inputs and outputs, nonlinear models can describe more complex and intricate interactions. In this sense, the effect of a change in the input variables on the output is not constant.

3.3.1 NLARX (Nonlinear ARX Model)

A Nonlinear ARX (AutoRegressive with eXogenous inputs) model consists of two main components: model Regressors and an Output Function.

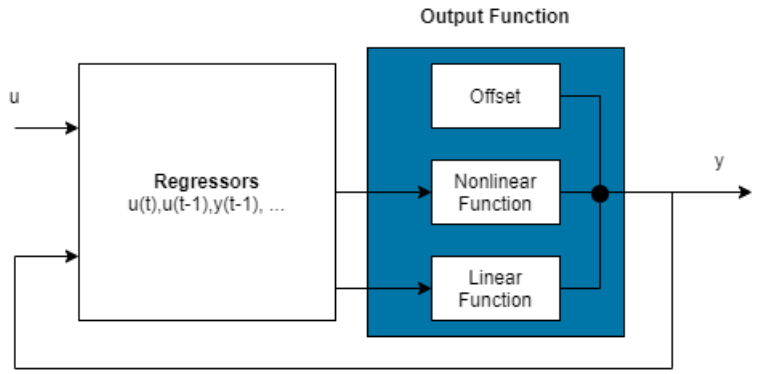


Figure 3.4: Nonlinear ARX block diagram

The computation of Regressors involves using current and past inputs, as well as past outputs. Several types of regressors can be arbitrarily chosen, based on the specific data and application:

- Linear Regressor: $[u(t), u(t-1), u(t-2), y(t-1), y(t-2)]$
- Polynomial Regressor: $[u(t-1)^2, y(t-1)^4, y(t-2)^2]$
- Periodic Regressor: $[\cos(u(t-2)), \sin(y(t-1)), \cos(y(t-1))]$
- Nonlinear Regressor: $[u(t-1) * y(t-2), y(t-1) * y(t-3)]$

The output function is composed of one or more mapping objects, which can include linear, nonlinear, and offset components, all operating in parallel. These mapping objects, applied to the model regressors, produce the output.

$$F(x) = L^T(x - r) + g(Q(x - r)) + d \tag{3.20}$$

In this general equation, x is the vector of the regressors, and r is the mean of x. The structure of $F(x)$ and g are chosen a-priori, while L, r, d, Q parameters are estimated during the optimization process.

$L^T(x - r)$ is the output of the linear function block.

$g\big(Q(x-r)\big)$ represents the output of the nonlinear function block. In particular, Q is a projection matrix that makes the calculations well-conditioned, and g is a chosen nonlinear function.

d is a scalar offset added to the combined outputs of the linear and nonlinear blocks.

In general, the form of $F(x)$ depends on the chosen output function, which could be a Wavelet Network, a Sigmoid Network, a Neural Network, or a customized one.

3.3.2 LSTM (Long Short-Term Memory)

Neural Networks are a key component of machine learning, inspired by the human brain's structure and functioning. These networks consist of multiple layers of interconnected nodes, each of them performing simple computations. The structure typically includes an input layer, one or more hidden layers, and an output layer. Each connection between nodes has an associated weight, which adjusts as the network learns, to minimize errors in the output predictions.

The working principle is conceptually easy. Initially, input data is fed into the network, and passes through each layer of neurons. As data moves through these layers, neurons activate based on the weighted sum of their inputs and apply an activation function (Sigmoid, Tanh, ReLU) to introduce non-linearity. The output from the last layer compared with the actual desired output, gives an output error. To reduce this error the network uses backpropagation, which means that the error propagates back through the layers, and the weights are adjusted using optimization techniques like gradient descent. This iterative process continues over time, until a stop criterion is met, or the entire dataset has been fed into the network a certain number of times (epochs).

The training process relies on large datasets, for this reason instead of using the entire dataset at once, which is computationally expensive and inefficient, the data is split into

smaller subsets known as batches. Each batch is processed individually, and the network's weights are updated after each batch.

In this thesis we are managing time series data, which are characterized by the fact that the order of data points matters: future values are often dependent on past values. Traditional neural networks lack the capability to remember past inputs, which means they cannot capture temporal dependencies effectively.

Recurrent Neural Networks (RNNs) address this issue. They are networks with internal loops, allowing information to persist. One main drawback is the so-called Gradient Vanishing/Exploding Problem, which occurs when the gradients become exceedingly small/big as they are propagated back through the layers of the network.

Long Short-Term Memory neural networks are special types of RNNs designed to mitigate the vanishing/exploding gradient problem.

Similar to RNNs, LSTMs are projected to find patterns and long-term dependencies in the data. They have the form of a chain of repeated modules, where the same weights' values are used.

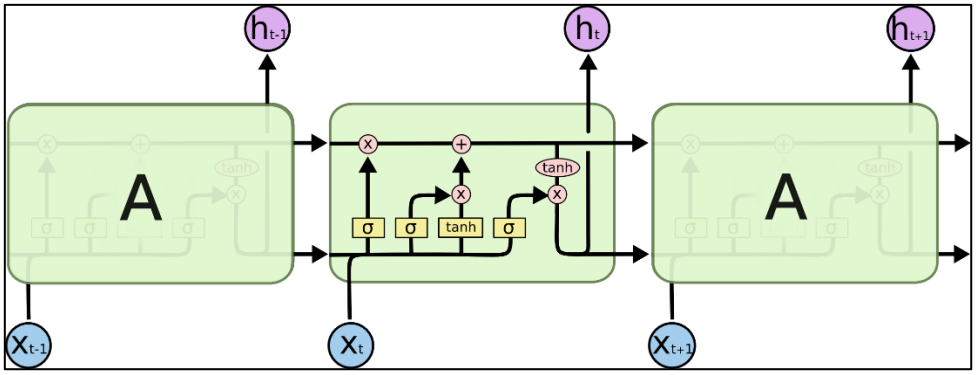


Figure 3.5: LSTM NNs modules

The core component of LSTMs is the cell state, the horizontal line running through the top of the diagram, which retains information over time.

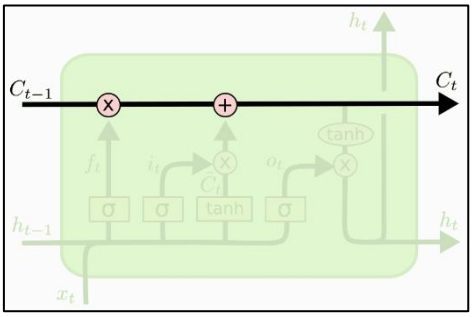


Figure 3.6: The Cell State

Three structures called “gates” control the cell state value along the LSTM unit chain.

The forget gate determines which information from the previous cell state should be forgotten or kept. It consists of a sigmoid layer which gives a number between 0 and 1 based on the values of the past hidden state h_{t-1} and the current input x_t. The result f_t is a vector that determines which parts of the previous cell state C_{t-1} are forgotten.

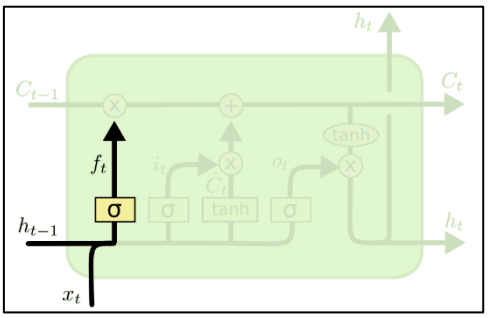


Figure 3.7: The Forget Gate

The input gate controls how much new information must be stored in the cell state. It is composed of two parts: a sigmoid layer which decides which values of the old cell state we will update, i_t, and a tanh layer which creates a vector of new candidate values $\tilde{C}_t$.

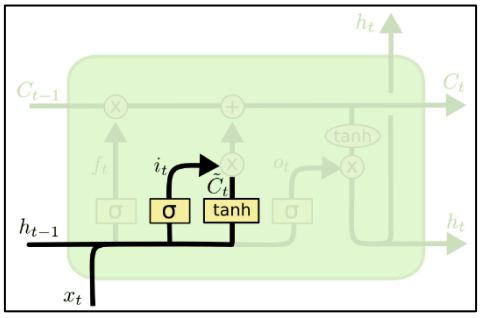


Figure 3.8: The Input Gate

In formula, we multiply the old state $\tilde{C}_{t-1}$ by f_t, forgetting what we decided to forget earlier, and then we add $i_t * \tilde{C}_t$, which represents the new information.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

The output gate decides what part of the cell state will go to the output. This work is done by a Sigmoid layer, while a tanh layer is used to push the values of the cell state to be between -1 and 1.

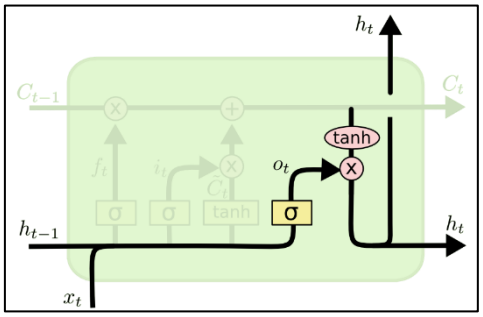


Figure 3.9: The Output Gate

This process allows the LSTM to output information relevant to the task while keeping other information in memory for future time steps.

3.3.3 Random Forest Regressor

Random Forest Regressor can be seen as a collection of decision trees, combined to improve overall accuracy of the model. A decision tree is a model used for both classification and regression tasks, which splits data into subsets based on feature values. Each node in a decision tree represents a feature in an instance to be classified, each branch represents a decision rule, and each leaf represents an output.

One main drawback of decision trees is their high sensitivity to training data, which implies low accuracy in predicting output from unseen data. In this sense, Random Forest Regressor improves its prediction accuracy applying several techniques such as bootstrapping, feature selection, and aggregation.

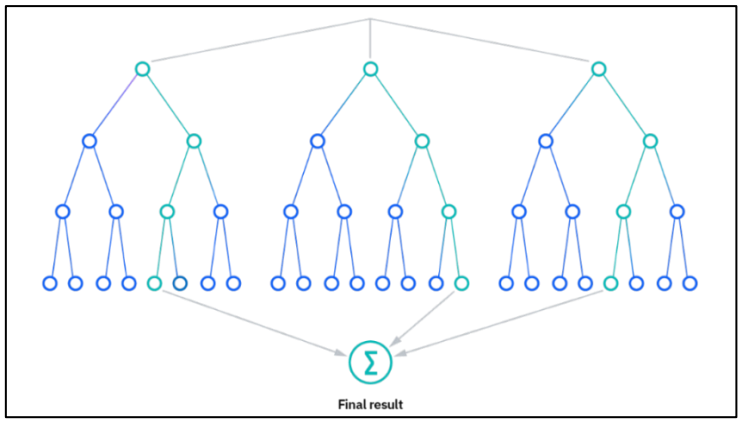


Figure 3.10: The Random Forest Regressor structure

Bootstrapping involves creating multiple subsets of the original training dataset by sampling with replacement. Each of these subsets is then used to train an independent decision tree. Training on different datasets, the regressor can better generalize.

Feature selection is used to take a random subset of features for each decision tree. This reduces correlation between trees and prevents overfitting by not relying too much on any particular feature.

Aggregation consists in averaging the predictions of all the decision trees. In this sense, aggregation helps to filter out the errors that individual decision trees models may make.

Chapter IV

Temperature and Job-Power Models

4.1 Linear models (ARX, OE)

Linear models capture a linear relationship between inputs and outputs. This means that changes in input variables correspond to proportional changes in the output variable. Before training a linear model, we must specify the number of lags, which is the number of past data which affect the predicted output.

A first approach was to train a ARX model fed by features which are most relevant for predicting CPU cores temperature:

$$\begin{aligned} T_{CPU}(t) + a_1 T_{CPU}(t-1) + \cdots + a_{n_a} T_{CPU}(t-n_a) \\ = b_{11} P_{p0}(t-1) + \cdots + b_{1n_1} P_{p0}(t-n_1) + b_{21} P_{p1}(t-1) + \cdots \\ + b_{2n_2} P_{p_1}(t-n_2) + b_{31} T_{amb}(t-1) \end{aligned} \tag{4.1}$$

where T_{CPU} is the mean CPU core temperature, P_{p0} and P_{p1} are the power consumption for the CPU sockets, T_{amb} is the inlet node temperature, n_1 and n_2 are the inputs lags, n_a is the output autoregression lag. The choice of n_1, n_2 and n_a is arbitrary, starting from low values and, consequently, evaluating performance with higher values.

Training is performed in simulation mode, which means that at each iteration we are feeding into the model the predicted lagged values of the output, not the measured ones. In this way, the model cannot rely on real measurements of the temperature. This is important because our objective is to make multi-step predictions, and obviously we cannot measure future temperatures values. Results are shown in figure [4.1].

A second approach was to train an OE model. The difference with the above method is that error is no longer considered as unmodeled dynamics, but it enters as an output error.

Keeping the same features and the same lags we obtained the results shown in figure [4.2].

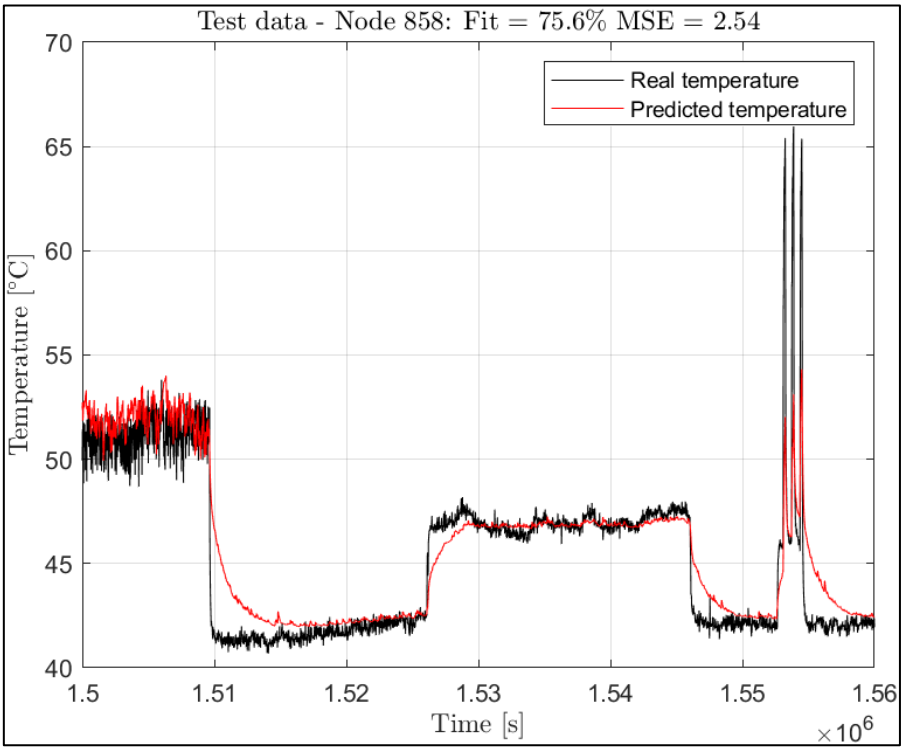


Figure 4.1: CPU ARX temperature model

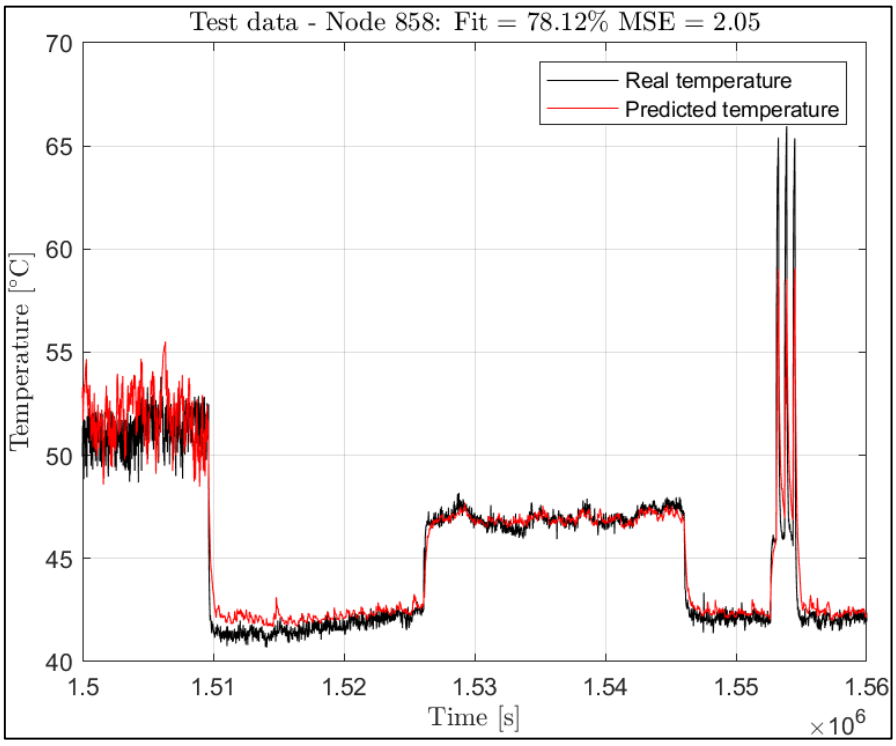


Figure 4.2: CPU OE temperature model

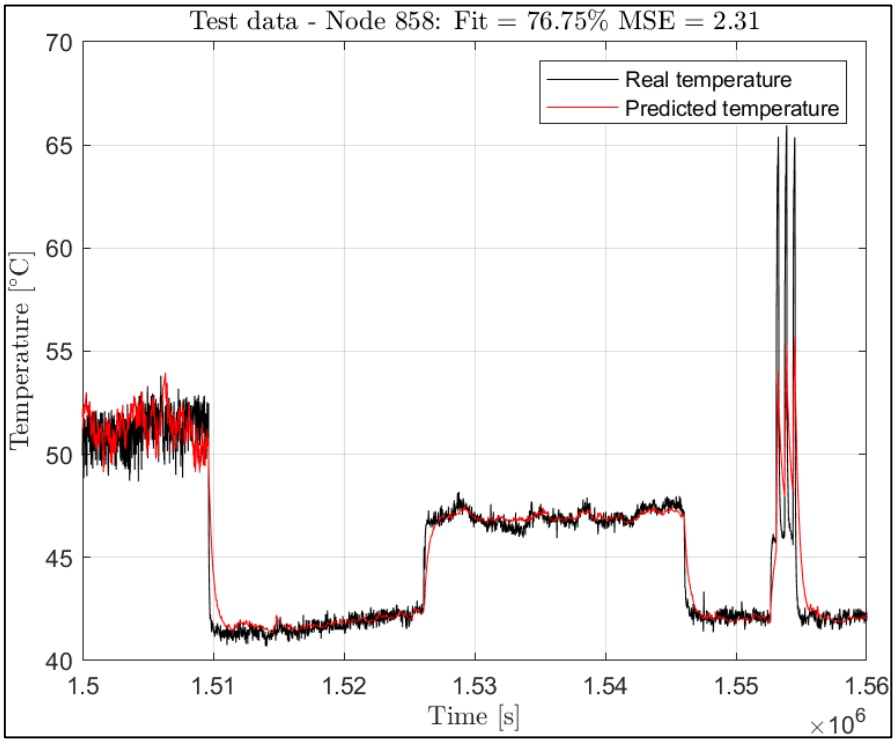


Figure 4.3: CPU NLARX temperature model

4.2 Nonlinear models (NLARX)

Nonlinear models try to predict system dynamics which cannot be captured by linear functions. In particular, the objective is to find a nonlinear relationship between inputs and a target output.

As mentioned in Chapter 3, in the training phase we need to select the type of regressors (linear, polynomial, periodic, or customized) and a nonlinear output function, which in general is a Sigmoid Network or a Wavelet Network, and the corresponding number of units.

Initially, we created two $iddata$ structures, one for the training and one for the test data. In both cases we used normalized data, with null mean value and unitary variance.

```
% Create iddata structures
train_data = iddata(y_train, u_train, 20);
train_data.TimeUnit = 's';

test_data = iddata(y_test, u_test, 20);
test_data.TimeUnit = 's';
```

Figure 4.4: MATLAB data structures

Starting from the model orders chosen for the ARX, we trained a NLARX.

```
model = nlarx(train_data, [na nb nk], net);

% Make predictions
[y_train_pred, fit_train, ~] = compare(train_data, model, inf);
y_train_pred = y_train_pred.OutputData;
[y_test_pred, fit_test, ~] = compare(test_data, model, inf);
y_test_pred = y_test_pred.OutputData;
```

Figure 4.5: MATLAB training and testing

In particular, the net structure is an $idSigmoidNetwork$. The best number of units is automatically chosen by the NLARX optimizer.

Running the $compare$ function, in simulation mode, and using the testing subset we obtain the results shown in figure [4.3].

4.3 Temperature LSTM Neural Network

LSTM (Long Short-Term Memory) neural networks not only find nonlinear relationships between inputs and outputs, but also try to capture short and long-time data dependencies and patterns. With respect to classical neural networks, they are particularly efficient in handling time series sequences.

For this reason, given an arbitrary sequence length, data was subdivided in this way:

```
% Number of sequences
num_sequences = length(u_train) - sequence_length + 1;

overlap = 1;
% Loop through the data to create sequences
for i = 1:overlap:num_sequences

    % Create an input sequence
    sequence = u_train(i:i+sequence_length-1,:);

    % Store the input sequence
    X_train{i} = sequence;

    % Create the corresponding target for the sequence
    Y_train(i) = y_train(i+sequence_length-1);

end
```

Figure 4.6: NLARX temperature model

We created two structures X_{train} and Y_{train}, containing inputs and target output sequences, respectively. Each cell has an overlap equal to one, meaning that adjacent sequences share *sequence_length-1* time steps.

The structure of the LSTM neural network implemented is the following:

```
layers = [...
    sequenceInputLayer(numInput)
    lstmLayer(50, OutputMode='sequence')
    dropoutLayer(0.2)
    lstmLayer(50, OutputMode='last')
    dropoutLayer(0.2)
    fullyConnectedLayer(25)
    fullyConnectedLayer(numOutput)];
```

Figure 4.7: LSTM neural network structure

We use an input layer with a number of nodes equal to the number of input features, two LSTM hidden layers and two Dense layers to produce the predicted output.

The Dropout layers are used to disable an arbitrary fraction (20%, in this case) of neurons during each training iteration. This is useful in preventing overfitting and improving model generalization.

```
options = trainingOptions("adam", ...
    MaxEpochs=50, ...
    LearnRateSchedule="piecewise", ...
    Plots="training-progress", ...
    VerboseFrequency=500, ...
    Verbose=true, ...
    ValidationData={X_val', Y_val'}, ...
    ValidationFrequency=250, ...
    Shuffle="every-epoch", ...
    MiniBatchSize=miniBatchSize, ...
    ValidationPatience=20, ...
    OutputNetwork="best-validation-loss" ...
    );
```

Figure 4.8: LSTM neural network training options

We trained the neural network with a maximum number of epochs equal to 50, and a validation patience of 20. This means that when evaluating the loss in the validation data, if the estimate does not improve for 20 consecutive evaluations, the training stops.

When training stops, the “best-validation-loss” model is saved.

We chose to use the “Adam” optimizer, with a “piecewise” schedule, which means that the learning rate is not constant and changes over epochs.

Moreover, the input sequences are given to the network in minibatches, and at each epoch they are shuffled, to prevent overfitting and improve model generalization.

In figure [4.9] we display the training process for temperature prediction of the CPU cores, where in blue we highlight the training loss, while in red the validation one. In figure [4.10] we show the results relative to the testing set and it is possible to notice that the predicted temperature succeeds in following the real one behavior.

In figure [4.11] and [4.12] the same plots are shown for the temperature prediction of the GPU cores.

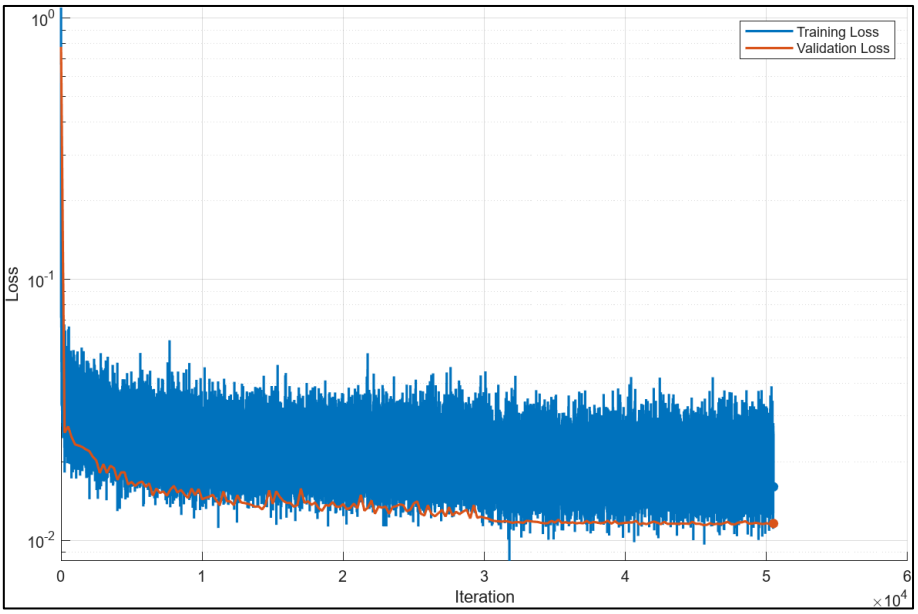


Figure 4.9: CPU LSTM neural network training and validation loss

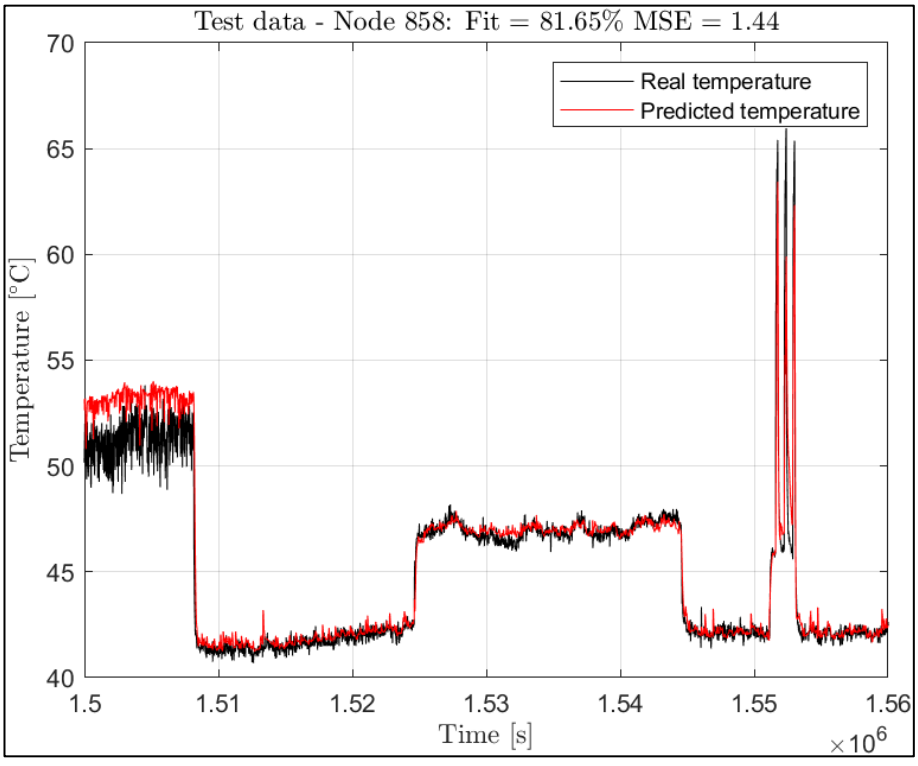


Figure 4.10: CPU LSTM neural network model

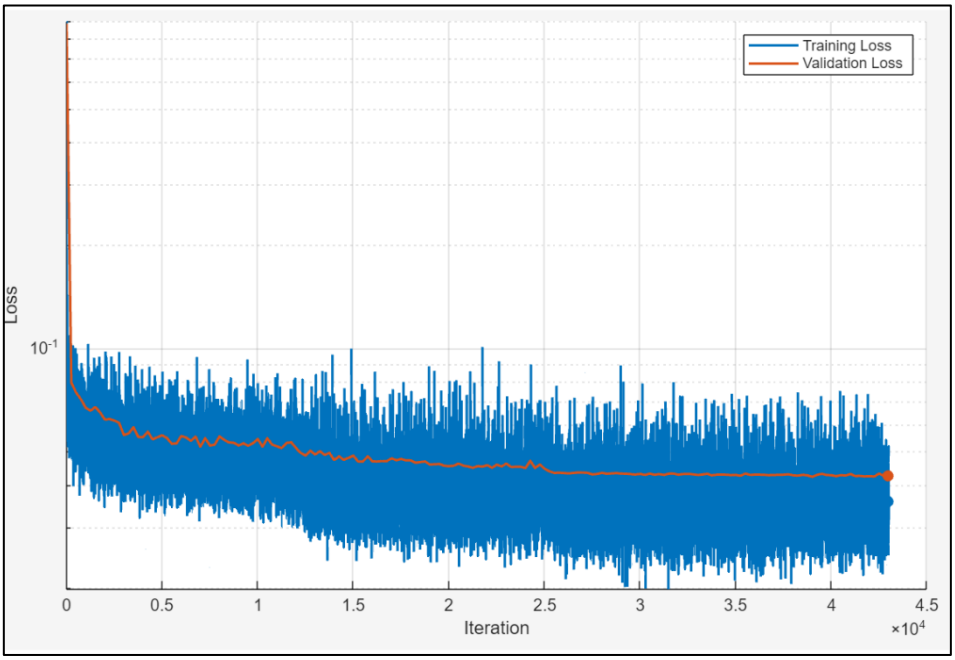


Figure 4.11: GPU LSTM neural network training and validation loss

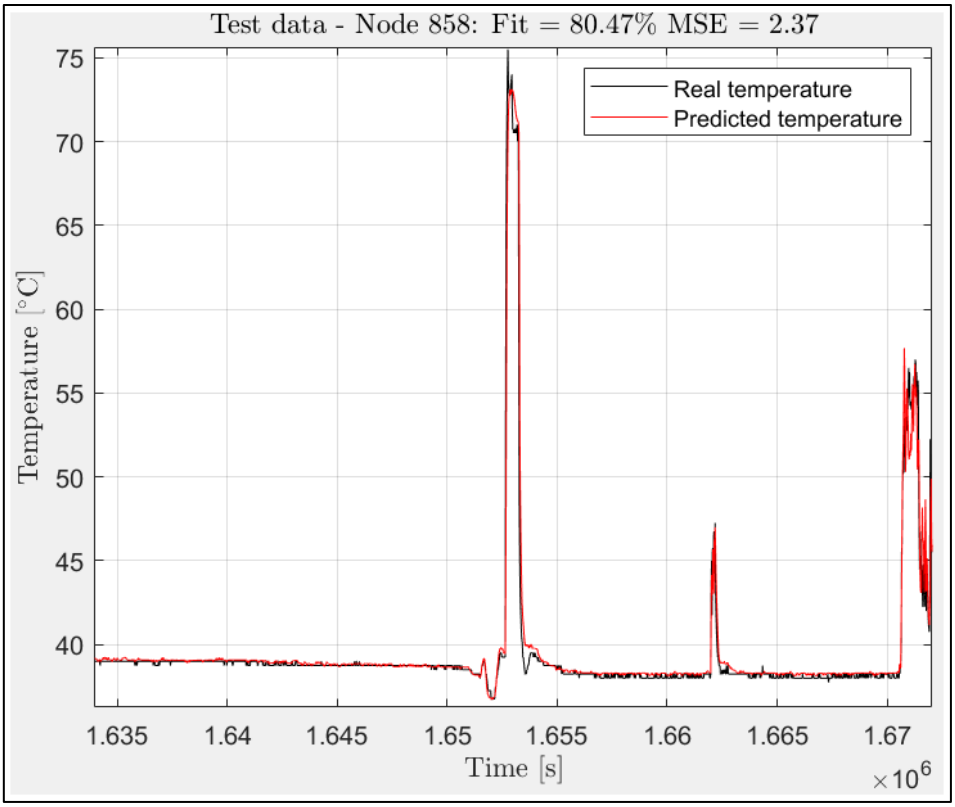


Figure 4.12: GPU LSTM neural network model

4.4 Random Forest Regressor

The Random Forest Regressor is a fundamental part of the optimization algorithm, that will be described in Chapter 5.

As illustrated in Chapter 3, it consists of a certain number of independent decision trees whose outputs are averaged to get a final estimation of the target. It uses bootstrapping, feature selection and aggregation techniques to not rely too much on training data and to better generalize on unseen data.

For our purposes, we are interested in getting an estimate of the CPU and GPU mean power that a submitted job will consume, based on its submission resources request.

The target power is obtained in the following way:

- From the Job Table, where old jobs information is stored, we filter and select only the jobs which execution time was greater than a sampling time. In fact, for each node, power consumption is read every twenty seconds.
- For each job we create a time window constrained by the start and end execution time.
- For each node which was allocated to a job, we select the corresponding time window, read the CPU and GPU power consumptions and compute the mean value.

Our target is the power mean value, and not its temporal behavior because we are using static features, which do not evolve over time.
The input data can be directly obtained from the Job Table, in particular we will use $user\_id$, $time\_limit$, num_nodes, num_cpus and num_gpus features.

Table 4.1: Random Forest Regressor input features

user_id	job_id	start_time	end_time	time_limit	nodes_list	num_nodes	num_cpus	num_gpus
678	91470	2021-12-31 00:25:50+00:00	2022-01-01 00:02:49+00:00	86340	[258]	1	4	4
1077	5430475	2021-12-31 00:25:50+00:00	2022-01-01 00:11:34+00:00	86400	[858]	1	128	4
964	4709582	2021-12-31 00:32:16+00:00	2022-01-01 00:32:38+00:00	86400	[882]	1	16	1
548	2863129	2021-12-31 00:53:41+00:00	2022-01-01 00:33:35+00:00	86400	[879]	1	128	4
964	3835502	2021-12-31 01:00:26+00:00	2022-01-01 01:00:48+00:00	86400	[766]	1	16	1
964	3765226	2021-12-31 01:00:26+00:00	2022-01-01 01:00:48+00:00	86400	[886]	1	16	1
964	3074591	2021-12-31 01:02:46+00:00	2022-01-01 01:02:48+00:00	86400	[889]	1	16	1
964	5034321	2021-12-31 01:02:46+00:00	2022-01-01 01:02:48+00:00	86400	[930]	1	16	1

In particular, 80% of the data was used for the training phase, and the remaining 20% for the testing. We trained both the CPU and GPU Random Forest Regressors on Python importing the $sklearn$ library.

Regarding the training dataset, as can be seen in figure [4.13] and [4.14] we obtained a mean relative error of -1.8% for the CPU mean power consumption and of -2.27% for the GPU one.

We notice a gaussian distribution centered on zero, which means that for most of the jobs the predicted power consumption relative error is almost null.

To verify the generalization capabilities of the model, we fed it with unseen data from the testing set, and we obtained a mean relative error of -4.35% for the CPU mean power consumption and of -2.89% for the GPU one.

As expected, the testing relative mean errors are a little higher with respect to the training ones, but its value guarantees accurate predictions.

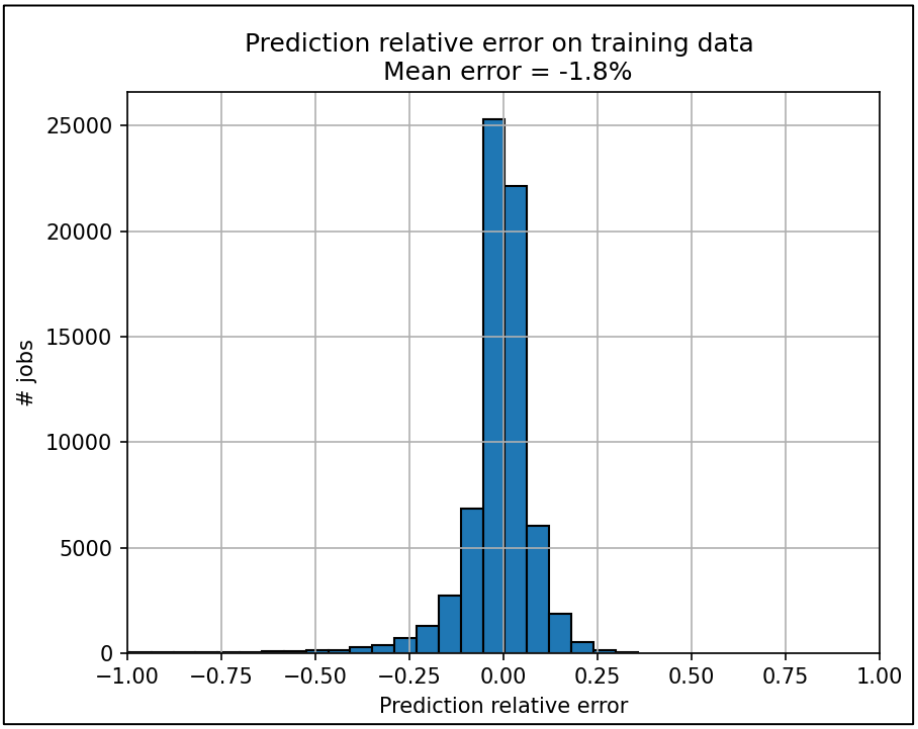


Figure 4.13: CPU Random Forest Regressor training prediction relative error

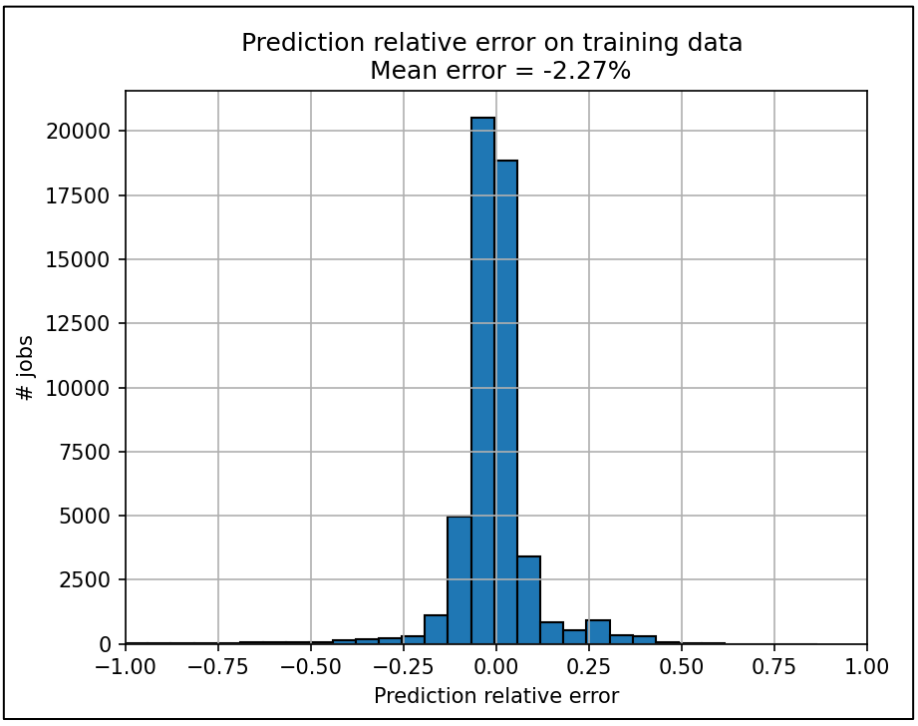


Figure 4.14: GPU Random Forest Regressor training prediction relative error

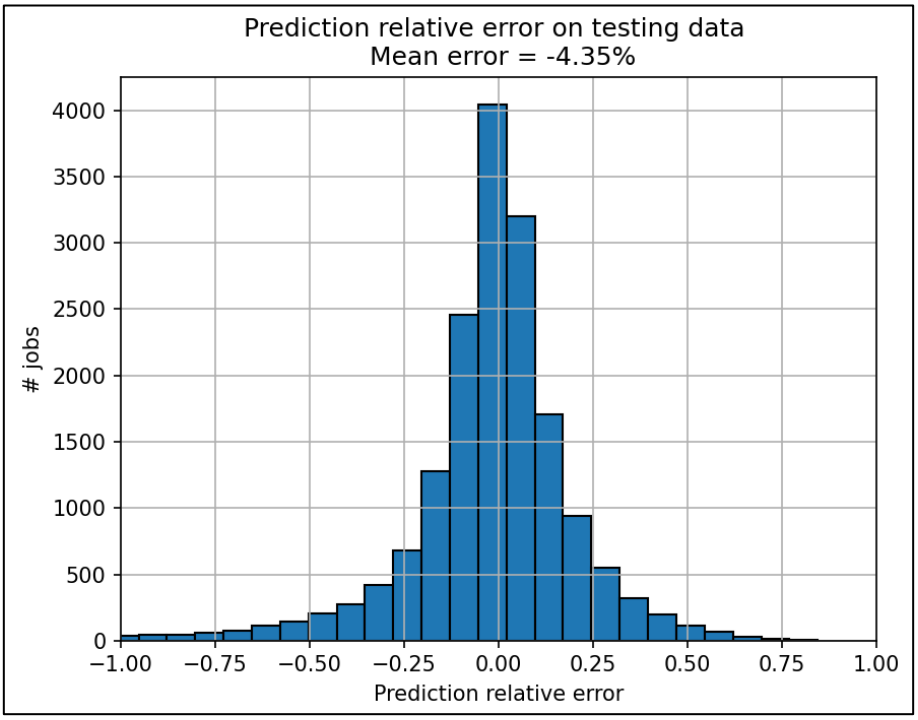


Figure 4.15: CPU Random Forest Regressor testing prediction relative error

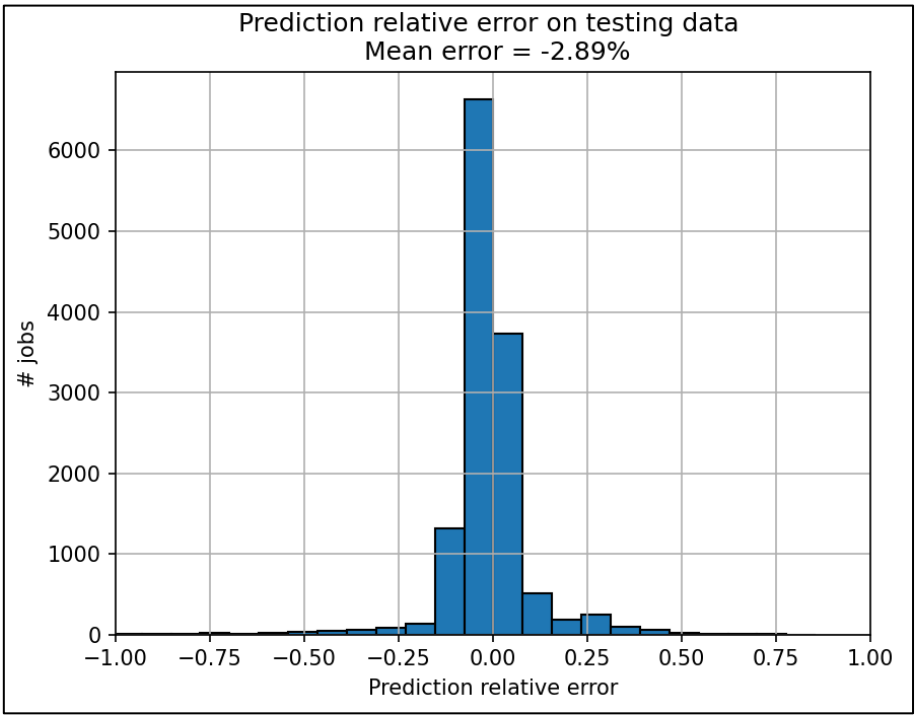


Figure 4.16: GPU Random Forest Regressor testing prediction relative error

4.5 Power LSTM Neural Network

As will be mentioned in Chapter 5, we do not have the plant of the datacenter at our disposal. This aspect implies two important consequences:

- We cannot measure the nodes' ambient temperatures; we can only approximate their values considering them as constants.
- We cannot measure the nodes' power consumption; therefore, it is not possible to compare our job scheduling energy consumption with the existing one.

To overcome this second issue, we propose a LSTM Neural Network that given the nodes' temperatures coming from the temperature neural network models, outputs the total power consumption of each node.

With the term *total power consumption,* we intend the power consumed by all the processes and hardware components internal to each node such as: power supply, switches, ports, cooling fans, processors, memory and storage.

The implemented LSTM has the same structure of the LSTM described in Section 4.3, the only difference is that in this case the inputs are the CPUs and GPUs temperatures, while the target output is the total power consumption.

In figure [4.17] we show the testing set results, with a FIT of about 42%. Even if this does not seem a good result, it is acceptable as we are interested in computing the total energy consumption, which is obtained by integrating the power over time.

Moreover, we do not require highly accurate results. Our concern is to understand the energy trend, so whether the optimization algorithm described in Chapter 5 is able to reduce the energy consumption of the datacenter or not.

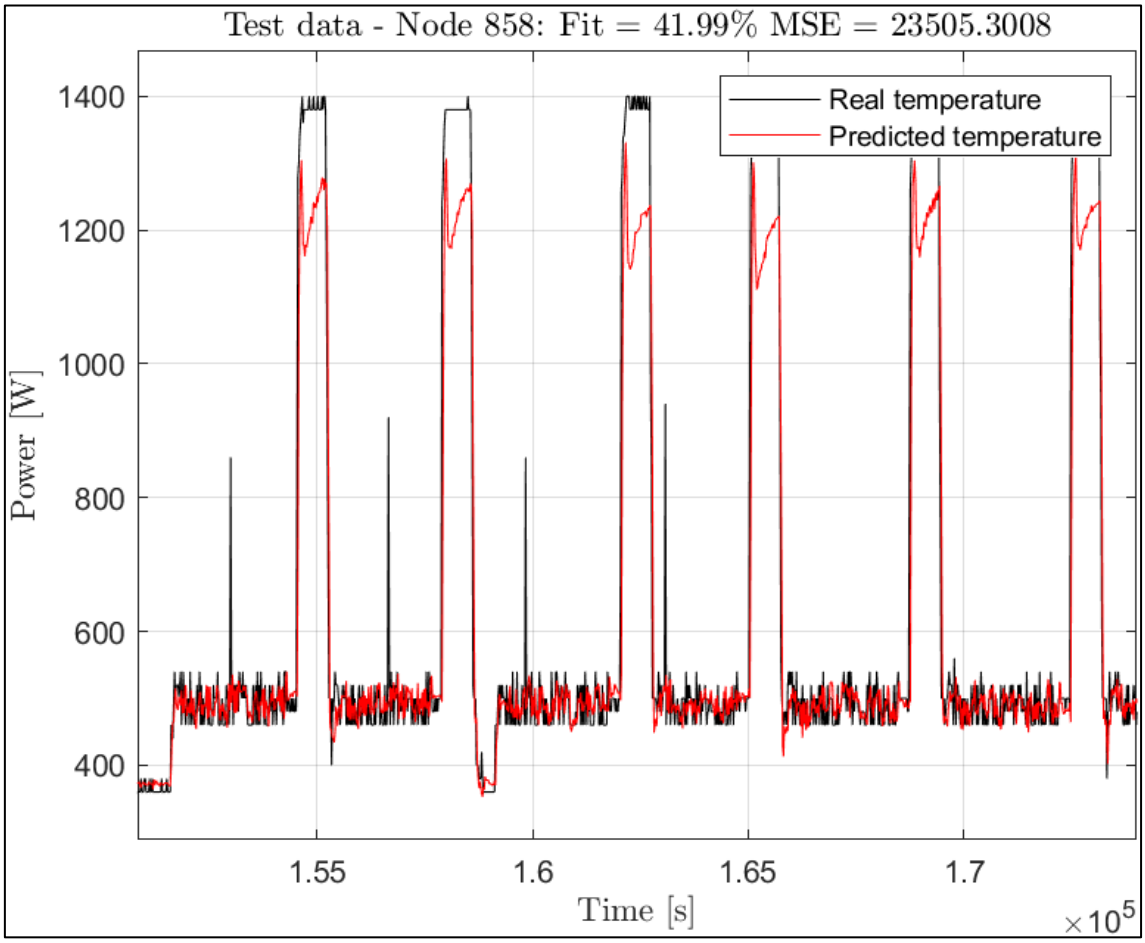


Figure 4.17: Total power consumption LSTM neural network

Chapter V
Optimization Problem

5.1 Genetic Algorithm functioning

Genetic Algorithms (GAs) are adaptive heuristic search algorithms inspired by the process of natural selection and genetics. This analogy comes from the process of the "survival of the fittest": only species that can adapt to changes in their environment can survive and reproduce and go to the next generation.

The search for the optimal solution starts from an initial population of individuals, each of which represent a point in the search space and a possible solution. They are coded into finite length vectors (**chromosomes**), composed of several variable components (**genes**).

Figure 5.1: Genetic Algorithm: population, chromosomes and genes

The evolution of the population during the algorithm execution is based on three operators:

- **Selection Operator:** Each individual of a population is evaluated with a fitness score, the ones with the better results (**parents**) will pass their genes to successive generations.
- **Crossover Operator:** Iteratively, two individuals are chosen, and their genes are exchanged randomly creating completely new individuals (**offspring**).

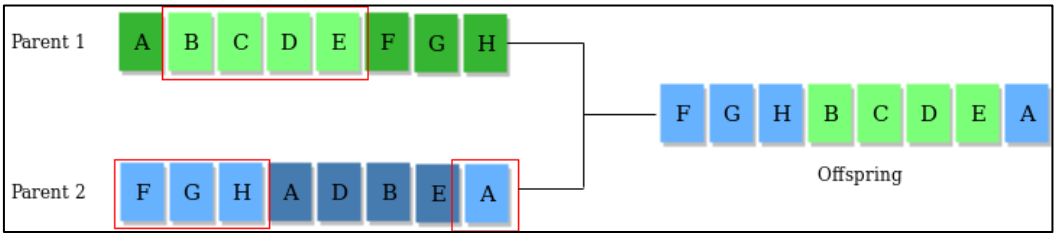


Figure 5.2: Crossover Operator

- **Mutation Operator:** Random genes are inserted in offsprings to maintain diversity and avoid premature convergence.

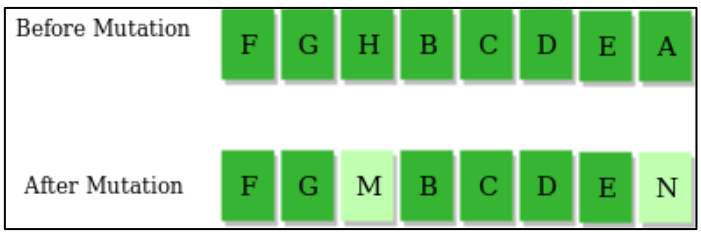


Figure 5.3: Mutation Operator

The algorithm process can be summarized as follows:

1) Randomly initialize population
2) Determine fitness score of the population
3) Until convergence repeat:
    a. Select parents from population
    b. Crossover and generate new population
    c. Perform mutation on the new population
    d. Calculate fitness score for new population

From what explained above, we notice that GAs, differently from traditional optimization techniques, do not rely on gradient information. They work with a population of possible solutions, allowing, at the same time, the exploration of multiple areas of the search space.

GAs are based on randomness, in fact mutation and crossover operators randomly select which of the "fittest" individuals shall mate and so exchange genes. This randomness brings to non-deterministic behaviors, meaning that different runs of the same optimization problem may yield different solutions.

Moreover, these kinds of algorithms are highly sensitive on parameters choice. Among them, the most important are the following:

- **Population Size:** dimension of the population, which is constant during all evolution.
- **Population Type:** type of individuals which constitute the population (double vector, bit string).
- **Initial Population Matrix:** initial population from which to start the algorithm evolution.
- **Initial Population Range:** range of values that individuals can assume.
- **Max Generations:** maximum number of iterations before algorithm stops.

The tuning of the above parameters is not trivial and depends on the difficulty and specific optimization problem. In general, GAs are less likely to get stuck into local optima with respect to traditional gradient-based methods, but they require more computational effort and could risk to prematurely converge, especially if the population diversity is lost too early. For these reasons, several attempts shall be made to find the best parameters tuning.

5.2 Why Genetic Algorithm?

Genetic Algorithms are powerful search algorithms, in fact they are also appropriate to solve nonlinear and complex optimization problems. They use mechanisms like penalty functions to penalize the fitness score of the individuals during "parents" selection. For this reason, GAs direct individuals towards a feasible search space.

For our purposes, we need an optimization algorithm capable of finding optima results observing equality and inequality linear constraints and capable of handling binary optimization variables.
Different approaches were taken into consideration, but the best results in terms of computational effort and solution optimality were given by the Genetic Algorithm optimizer.

5.3 Optimization scheme

As described in Chapter 2, users can submit jobs specifying the maximum number of CPU/GPU cores and the time limit constraint in which the job must be completed. Moreover, each user is associated with an ID, and this info is important for power consumption predictions, because typically the same users submit similar jobs.

All these job submission info is fed into the "$Job - Power\ Model$", trained as illustrated in Chapter 4. In a few words, it is a Random Forest Predictor, capable of estimating the CPU and GPU mean powers that a submitted job will require for its execution.

The predicted mean power consumption, together with the measured node's inlet temperature, are fed into the "$Power - Temperature\ model$". This model, as mentioned in Chapter 4, for each node outputs an estimate of the temperature increment that would occur in case of job allocation and execution on that specific node.

For our purposes, a 3-step ahead prediction is sufficient. In this way, knowing that each step corresponds to twenty seconds, we are predicting temperatures one minute in the future.

At this point, we have an array T, which entries are the estimated temperature increments for each available node. This array, as will be described in the next paragraph, is part of the objective function.

The minimization of the objective function returns an optimization variable which states:

- where the submitted job must be allocated to maintain the lowest temperature increment.
- Which nodes are ON, and among them which are $Busy$ or $Idle$.
- If it is necessary to turn on new nodes.
- If it is possible to power off idle nodes.

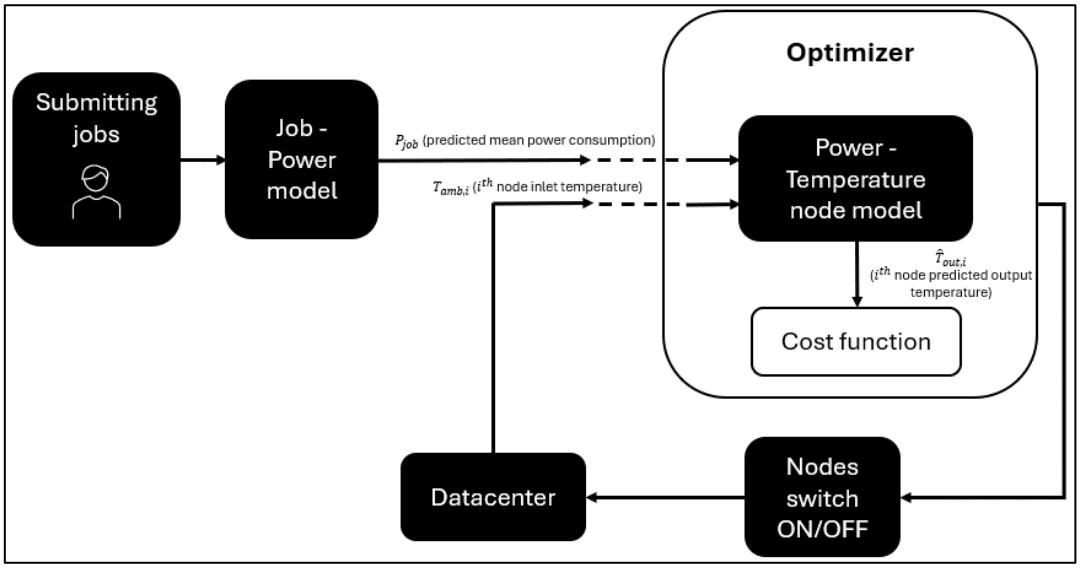


Figure 5.4: Closed-loop optimization scheme

5.4 Constraints and Objective function: first approach

The optimization algorithm must keep track of which nodes are active, and among them which are busy or idle. Therefore, we need to use two data structures.

Let's define an array of length equal to the number of nodes N:

$$Y = [y_1, \dots, y_N] \tag{5.1}$$

where each element can assume values:

$$y_i = \begin{cases} 1 \rightarrow if\ i^{th}\ node\ is\ ON \\ 0 \rightarrow if\ i^{th}\ node\ is\ OFF \end{cases} \tag{5.2}$$

And let's define a matrix with number of rows equal to the number of nodes N, and number of columns equal to the number of jobs n submitted to the datacenter:

$$X = \begin{bmatrix} x_{1,1} & \cdots & x_{1,n} \\ \vdots & \ddots & \vdots \\ x_{N,1} & \cdots & x_{N,n} \end{bmatrix} \tag{5.3}$$

where each element can assume values:

$$x_{i,j} = \begin{cases} 1 \rightarrow if\ j^{th}\ job\ is\ allocated\ to\ i^{th}\ node\ (BUSY) \\ 0 \rightarrow if\ j^{th}\ job\ is\ not\ allocated\ to\ i^{th}\ node \end{cases} \tag{5.4}$$

If $x_{i,j} = 0\ \forall j$, the i_{th} node is considered $IDLE$.

For ease of use, it is possible to merge these two data structures in a unique matrix:

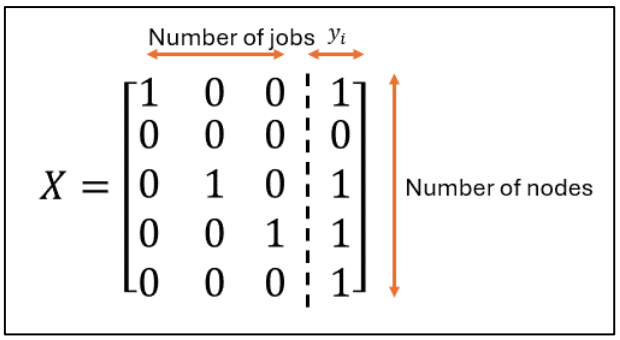


Figure 5.5: Matrix of optimization variables: First Approach

In this example:

- Nodes $1, 3$ and 4 are ON because $y_i = 1$, and are $BUSY$ because $\sum_{j=1}^{3} x_{i,j} = 1$, for $i = 1,3,4$;
- Node 2 is OFF as $y_2 = 0$;
- Node 5 is ON, as $y_5 = 1$, but it is $IDLE$ because there are not jobs allocated.

To correctly control the datacenter jobs allocation, the optimization algorithm must respect some constraints:

$$\sum_{j=1}^{n} x_{i,j} \leq 1 \,, \forall i = 1, \dots, N \tag{5.5}$$

$$\sum_{i=1}^{N} x_{i,j} = 1 \,, \forall j = 1, \dots, n \tag{5.6}$$

$$\mathrm{y_i} \geq \sum_{j=1}^{n} x_{i,j} \,, \forall i = 1, \dots, N \tag{5.7}$$

$$\sum_{i=1}^{N} y_i \geq n_{jobs} + threshold \tag{5.8}$$

Constraint (8.5) imposes a simplification: each node can execute maximum one job at a time.

Constraint (8.6) sets to one the maximum number of nodes to which a job can be allocated. This is a first assumption; we are considering only jobs which require only one node for their execution.

Constraint (8.7) states that busy nodes cannot be turned off. In this sense, it is not possible to set $y_i = 0$ until the job being executed on node i is terminated.

Constraint (8.8) specifies how many nodes must be ON beyond those that are executing jobs. This is because we want to guarantee a minimum number of available nodes for the execution of newly submitted jobs.

To make an example, if one job arrives at a moment in which all active nodes are busy, before allocation and execution, the job must wait for the power on of a new node, which typically takes minutes. This amount of time is too much for jobs which execution is on the order of seconds.

For this reason, it is convenient to keep active and idle a certain number of nodes, specifying arbitrarily a $threshold$ value.

Finally, we need to specify an objective function which optimization gives the best job allocation based on temperature predictions.

$$f = \sum_{j=1}^{n}\sum_{i=1}^{N} T_i * X_{i,j} + \sum_{i}^{N} y_i + \sum_{i}^{N} (y_i \neq y_{old,i}) \tag{5.9}$$

The term $\sum_{j=1}^{n}\sum_{i=1}^{N} T_i * X_{i,j}$ considers the total datacenter predicted temperature.

The term $\sum_{i}^{N} y_i$ tends to minimize the number of active nodes.

The term $\sum_{i}^{N} (y_i \neq y_{old,i})$ avoids repeated turning node on and off.

5.5 Constraints and Objective function: second approach

In this second approach we will use an optimization matrix with a number of rows equal to the number of nodes N, and only two columns.

While in the above paragraph matrix X columns reflected the number of submitted jobs, with this alternative method the number of columns is kept fixed. In particular, the second column tells us if i^{th} node (given by i^{th} row) is ON or OFF, while the first one indicates if the same node is BUSY or IDLE.

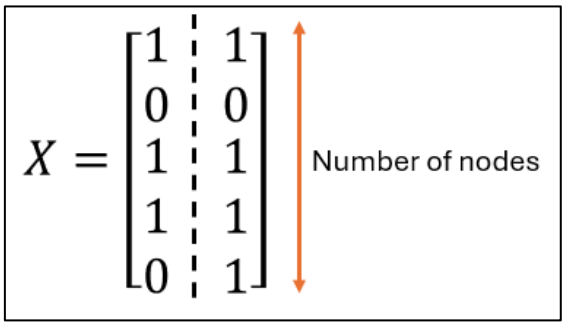


Figure 5.6: Matrix of optimization variables: Second Approach

To make an example:

- Nodes 1, 3 and 4 are ON because their corresponding rows in the second column are set to 1 and are BUSY because also in the first column, they assume value 1.
- Node 2 is OFF.
- Node 5 is ON, but in this case it is IDLE.

To correctly control the datacenter jobs allocation, the optimization algorithm must respect some constraints:

$$\sum_{i=1}^{N} x_{i,1} = n_{jobs} \tag{5.10}$$

$$x_{i,2} \geq x_{i,1} \quad \forall i = 1, \dots, N \tag{5.11}$$

$$\sum_{i=1}^{N} x_{i,2} \geq n_{jobs} + threshold \tag{5.12}$$

Constraint (8.10) imposes that the number of BUSY nodes shall be equal to the number of submitted jobs. This means that each job has been allocated and in execution on the available nodes.

Constraint (8.11) specifies that the optimizer cannot turn off nodes which are executing a job. Trivially, it states that if in a certain row and in the first column we have a 1, in the second column of the same row we cannot have a 0. If this happens, it means that the optimizer has shut down a node which was Busy.

Constraint (8.12) expresses how many nodes must be ON beyond those that are executing jobs. This is because we want to guarantee a minimum number of available nodes for the execution of newly submitted jobs, as mentioned in the previous paragraph.

Regarding the objective function it remains the same of the first approach, the only thing that changes is the implementation to adapt it to a 2-columns matrix.

This second approach is much more efficient than the previous one in terms of optimal solution computation time. In fact, in the first method, for each submitted job, we add a column to the X matrix and consequently we are adding an additional number of optimization variables equal to the number of nodes (rows).

To make an example, at the submission of the first job, the X matrix will have dimension $Nx2$, and this means that we will have $2 * N$ optimization variables. After a while, if the number of jobs increases to 20, the matrix will be $Nx21$, with a number of optimization variables equal to $21 * N$. Considering that $N = 980$, the Genetic Algorithm should be able to minimize the objective function in a problem with $980 * 21 = 20580$ optimization variables! Obviously, the computation time will increase exponentially.

With the second approach this problem is mitigated as the matrix has a constant size, so the number of the optimization variables will be the same for all the time.

5.6 MATLAB implementation and Simplified Optimization

The above optimization algorithm was implemented in MATLAB using the ga function. It refers to the Genetic Algorithm function, and its choice comes from the possibility to use binary optimization variables.

The inequality constraint functions must be specified in the form:

$$c(x) \leq 0 \tag{5.13}$$

By default, the ga function does not allow the possibility to implement equality constraints when the decision variables are integers. To overcome this issue, they were expressed in this alternative way:

$$c(x) = a \leftrightarrow \begin{cases} c \leq a \\ c \geq a \end{cases} \leftrightarrow \begin{cases} c - a \leq 0 \\ -c + a \leq 0 \end{cases} \tag{5.14}$$

We created a MATLAB function named $constraintsFunction()$ which output is:

```
c = [sum(x(:,1),1)'-nJobs; % sum of elements of first column <= nJobs
    -sum(x(:,1),1)'+nJobs; % sum of elements of first column >= nJobs
    x(:,1)-x(:,2); % it cannot turn off busy nodes
    threshold+nJobs-sum(x(:,2));
    -threshold-2-nJobs+sum(x(:,2))]; % sum of nodes ON >= nJobs + threshold
```

Figure 5.7: MATLAB constraints implementation

Regarding the objective function to minimize, it is given by $objectiveFunction()$:

```
f = sum(dot(T(end,:)',x(:,1))) + sum(x(:,end)) + sum(x(:,end) ~= x_old(:,end));
```

Figure 5.8: MATLAB objective function implementation

To set the decision variables as binary, first we set them as integers using the option $IntCon$ and then we assign lower and upper bounds to zero and one, respectively.

Moreover, it is possible to set some options such as the $PopulationType$ and $PopulationSize$:

```
options = optimoptions('ga', ...
    'PopulationType', 'doubleVector', ...
    'PopulationSize', 500);
```

Figure 5.9: MATLAB GA options

Finally, the optimizer x and the corresponding minimized objective function $fval$ are given by:

```
% GA optimization problem
[x, fval] = ga(objective, nVars, [], [], [], [], ...
    lb, ub, constraints, IntCon, options);
```

Figure 5.10: MATLAB GA function

As in the second approach we are using a fix sized matrix, it cannot be used to keep track of where new submitted jobs are allocated. For this reason, we implemented the job_info

matrix, which is a table used to store for each submitted job its ID, node allocated, and execution time.

At this point we need to make an assumption: as we do not have at disposal the datacenter, we cannot simulate our optimization algorithm on the real plant, for this reason we cannot know the real execution time of the jobs on the assigned nodes. To handle this problem, we assume that the execution times will be the same specified in the Job Table.

With a *for loop,* we simulate the flow of time, where each increment of the loop variable corresponds to a second. Following this reasoning, at each iteration we decrease of one second the remaining execution time of all running jobs. When a job terminates, its corresponding node state from BUSY becomes IDLE, and then the node is ready to be assigned to new jobs.

The fundamental part of the optimizer and of the objective function is the temperature model: we need to forecast nodes temperature increments 3-steps in the future. To accomplish this task, we use the Temperature LSTM neural network described in Chapter 4, which needs to be fed by the job mean power consumption estimates and the actual ambient temperature.

The problem is that we do not have a real plant on which run our simulation, for this reason we do not have at disposal real time data regarding ambient temperature behavior. In this sense, we need to assume the ambient temperature to be constant. These assumptions simplify the optimization scheme of figure [5.4] as represented by figure [5.11]. We notice that our predictive control algorithm is now in open loop, because we do not have plant feedback.

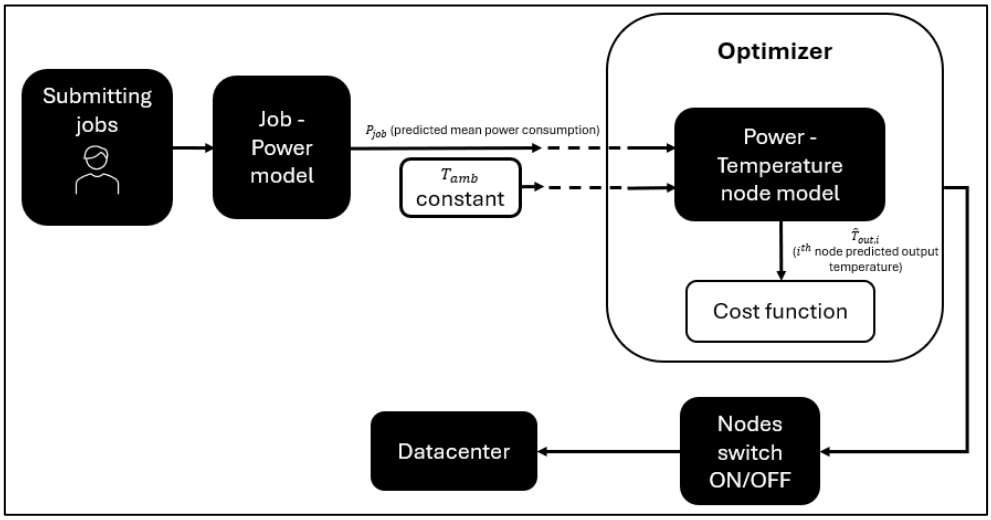


Figure 5.11: Open-loop optimization scheme

After considering this aspect, we need now to underline one last thing. The LSTM model was trained to accept in input a sequence of the last 50 values of the power and of the ambient temperature. For this reason, we need to keep track of these values.

When a node is OFF, its power consumption is null, when it is IDLE the power is assumed to be constant, while in the BUSY state, the power is given by the job mean power estimate. Therefore, in this last case, we are assuming that the job, during its execution on the assigned node, is consuming a power equal to the one estimated by the Random Forest.

Regarding the ambient temperature, as said before it will be considered constant for the entire simulation time, as we do not have datacenter feedback data.

Chapter VI

Simulation Results

6.1 First Test: Allocation

To understand the results, it is important to describe how they are represented. Looking at figure [6.1], we have on the X-axis the time flow in seconds, while on the Y-axis the nodes ID. In our application, we have 980 nodes, and we decided to assign an ID which goes from 1 to 980. In this sense, the first 20 nodes of the first rack have IDs which simply go from 1 to 20.
Each dot represents a node state: red for OFF, green for ON/IDLE and blue for ON/BUSY. Following this scheme, it is possible to graphically understand which nodes are executing jobs, which are powered on waiting for allocation, and which are turned off.

In this first test the aim is to understand if the constraints and objective function described in Chapter 5 work. To verify this, we assume that:

- One new job is submitted every 250 seconds.
- Jobs in execution never terminate. We only want to test the allocation phase, not the deallocation which will be tested later.
- We do not compute temperature predictions. We assume that the nodes with the lower ID number are the coldest ones.
- Simulation starts with all nodes turned OFF, except for the first five.
- We arbitrarily impose a simulation time of 2000 seconds.
- To make the graphic representation clearer, we consider only the first 20 nodes.

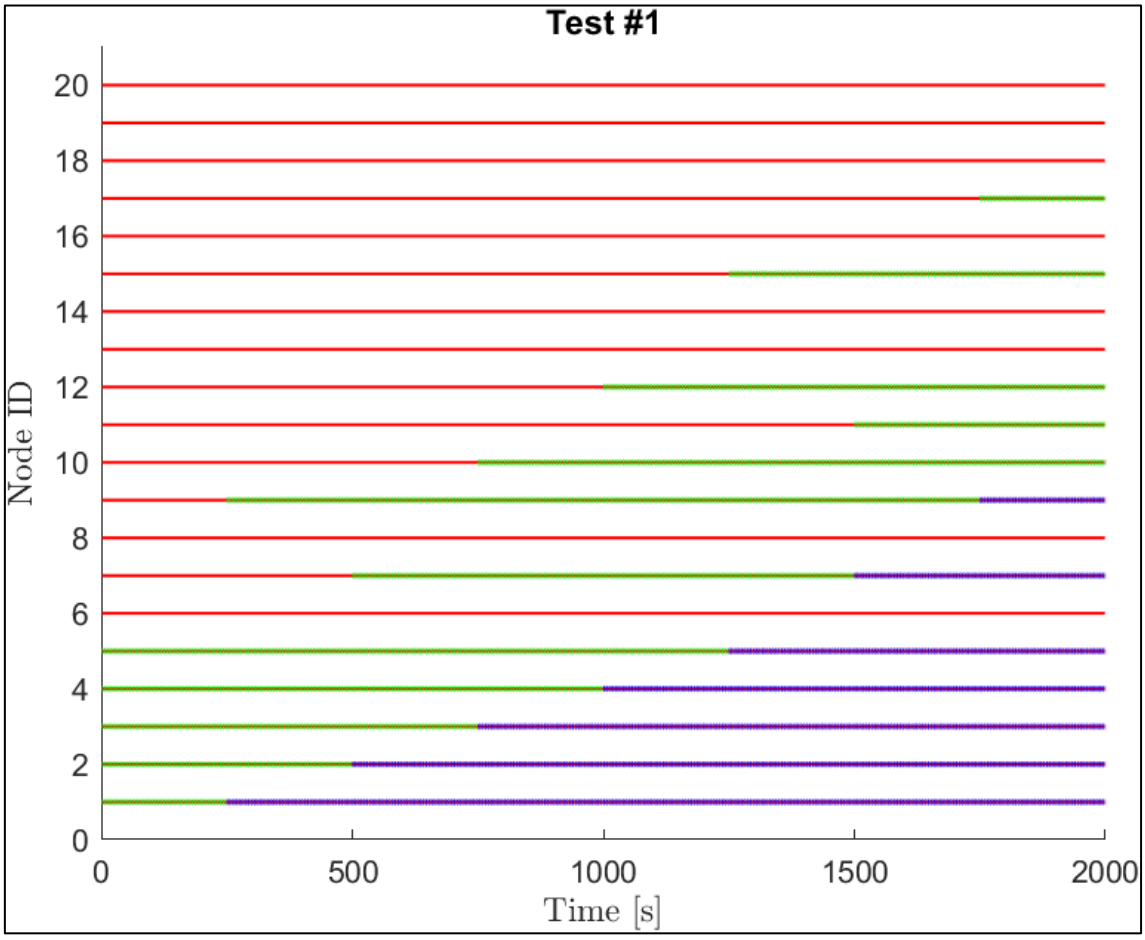


Figure 6.1: Test #1 Results

6.2 Second Test: Allocation & Deallocation

At this moment we want to verify if allocation is performed correctly, but we shall also understand if after jobs are terminated, the corresponding nodes come available again (de-allocation phase). The test is based on the following assumptions:

- One new job is submitted every 300 seconds.
- Each new job has an execution time equal to 1000 seconds.
- We do not compute temperature predictions. We assume that the nodes with the lower ID number are the coldest ones.
- Simulation starts with all nodes turned OFF, except for the first five.
- We arbitrarily impose a simulation time of 2500 seconds.
- To make the graphic representation clearer, we consider only the first 15 nodes.

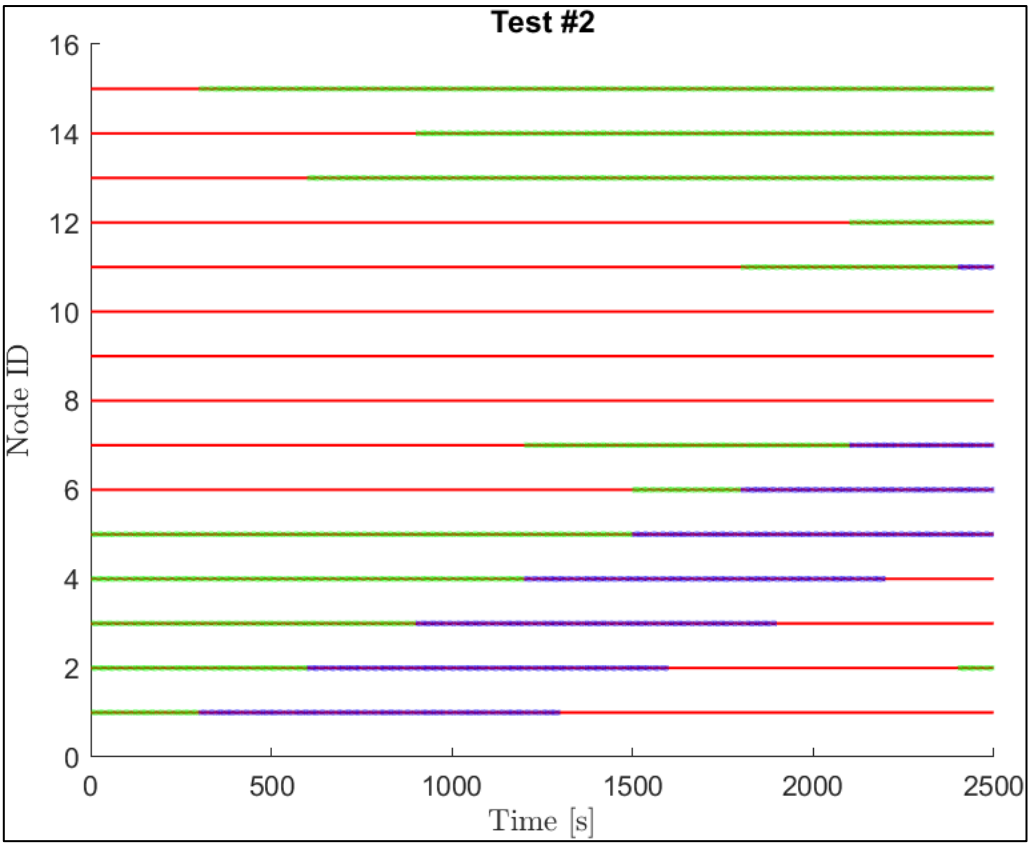


Figure 6.2: Test #2 Results

6.3 Third Test: Allocation & Deallocation - Entire Datacenter

With this third test we want to ensure that the population size of the Genetic Algorithm is appropriate for handling the number of optimization variables that we will have with the real datacenter. In fact, while in the previous two tests we limited the number of nodes to

20, we now consider all the 980 computation units. The assumptions will be the following:

- One new job is submitted every 50 seconds.
- Each new job has an execution time equal to 500 seconds.
- We do not compute temperature predictions. We assume that the nodes with the lower ID number are the coldest ones.
- Simulation starts with all nodes turned OFF, except for the first five.
- We arbitrarily impose a simulation time of 1000 seconds.

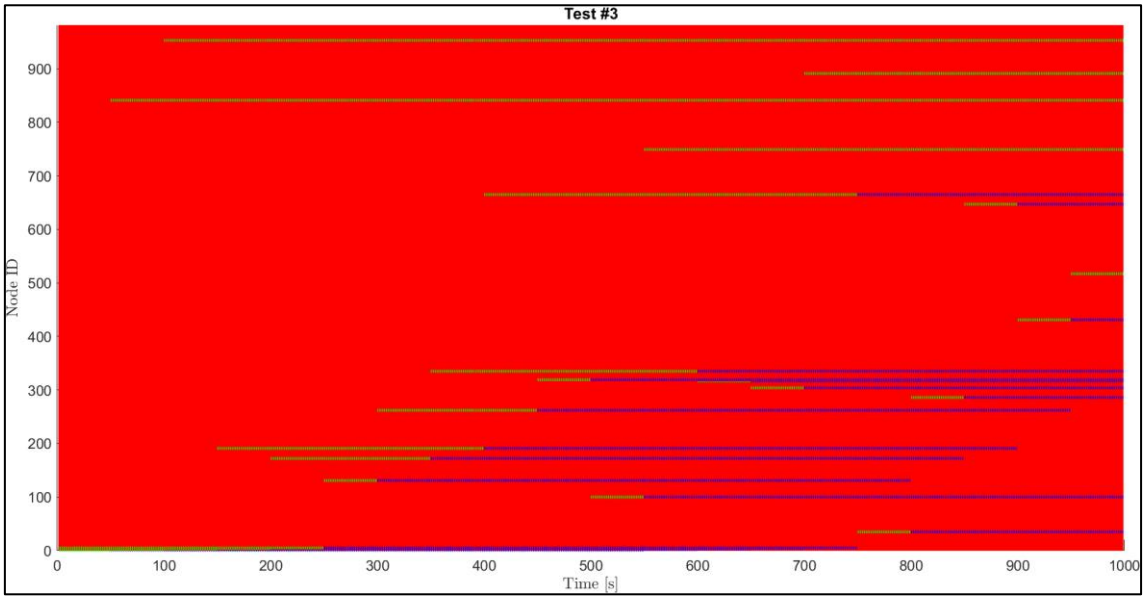
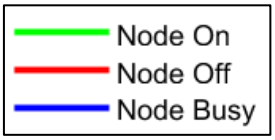


Figure 6.3: Test #3 Results (980 nodes)



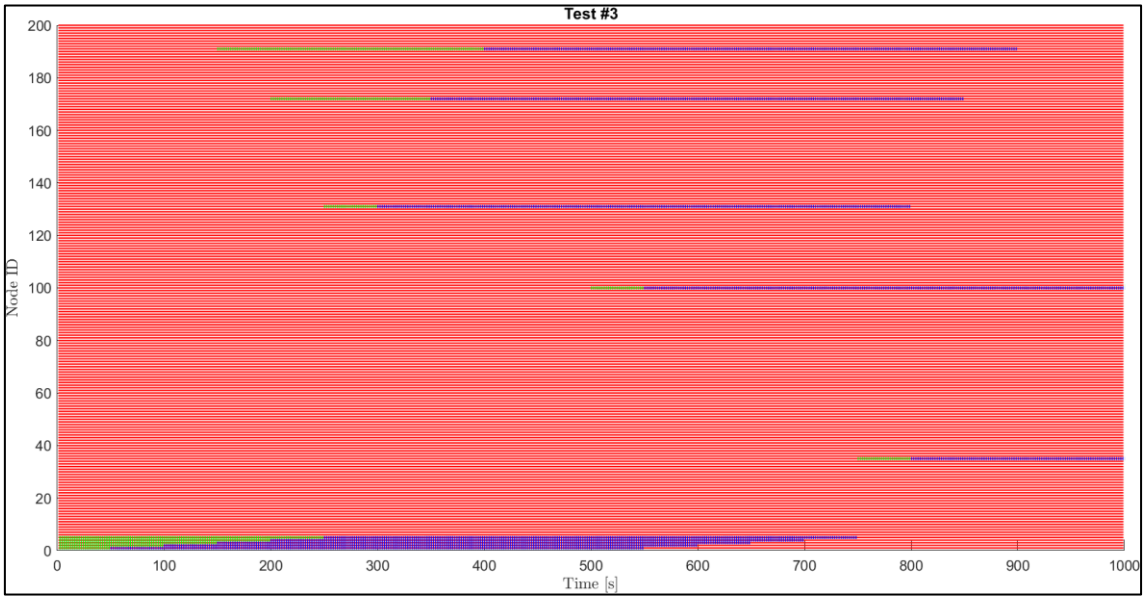


Figure 6.4: Test #3 Results (200 nodes)

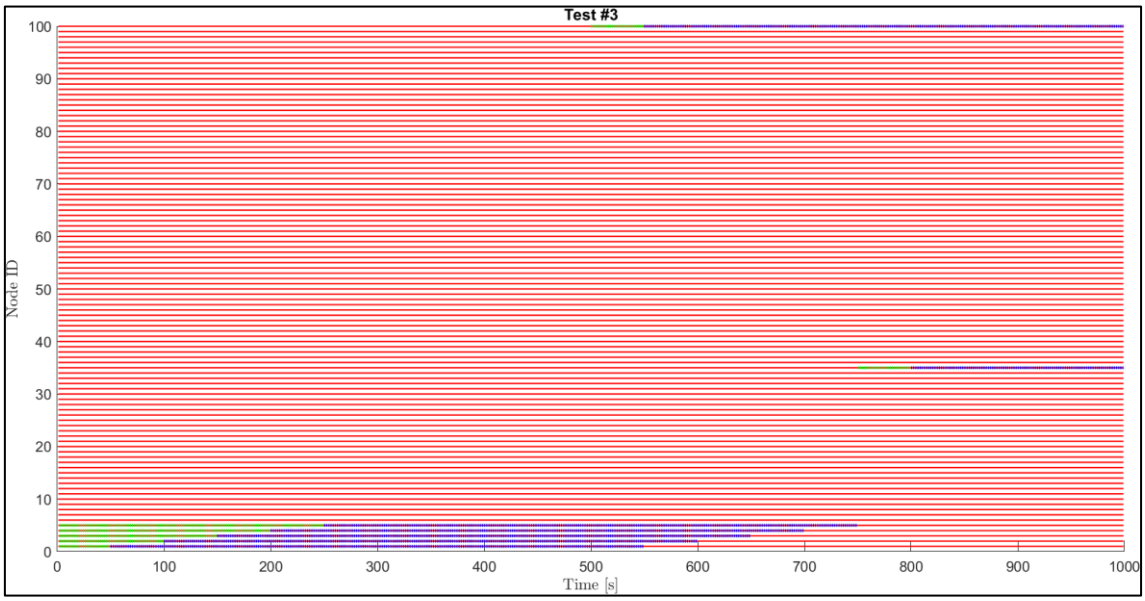


Figure 6.5: Test #3 Results (100 nodes)

6.4 Final test: One day simulation

In this section we are going to simulate the behavior of the optimizer in handling the entire datacenter and exploiting one day of job submissions retrieved from the Job Table.

In this case we are going to use both the "Job-Power" and the "Power-Temperature" models as described in Section [8.3].

- Referring to the Job Table, we submit jobs following the same time scheduling of the 30th of January, exploiting the "start_time" feature.
- As discussed in Section [8.3] we suppose that the execution time is equal to the one written in the Job Table.
- 3-steps ahead temperature predictions are performed to estimate which nodes are most suitable for allocation.
- Simulation starts with all nodes turned OFF, except for the first five.
- We impose a simulation time of 24 hours (86400 seconds).

In this way we are simulating how our job scheduler performs in allocating new jobs on a typical datacenter day.

For each node we record CPU and GPU temperatures behavior and the total power consumption. To make an example, in figures [6.6] and [6.7] we represent the CPU and GPU temperatures behavior relative to the node number 128. We notice that at around $3.99 * 10^4$ seconds a job is allocated, therefore the temperature increases and reaches a steady state value. At approximately $4.05 * 10^4$ seconds, when the job execution is terminated, the node starts cooling down.

From the power point of view, in figure [6.8] we notice that the power consumption increases and decreases in line with the job's allocation and deallocation.

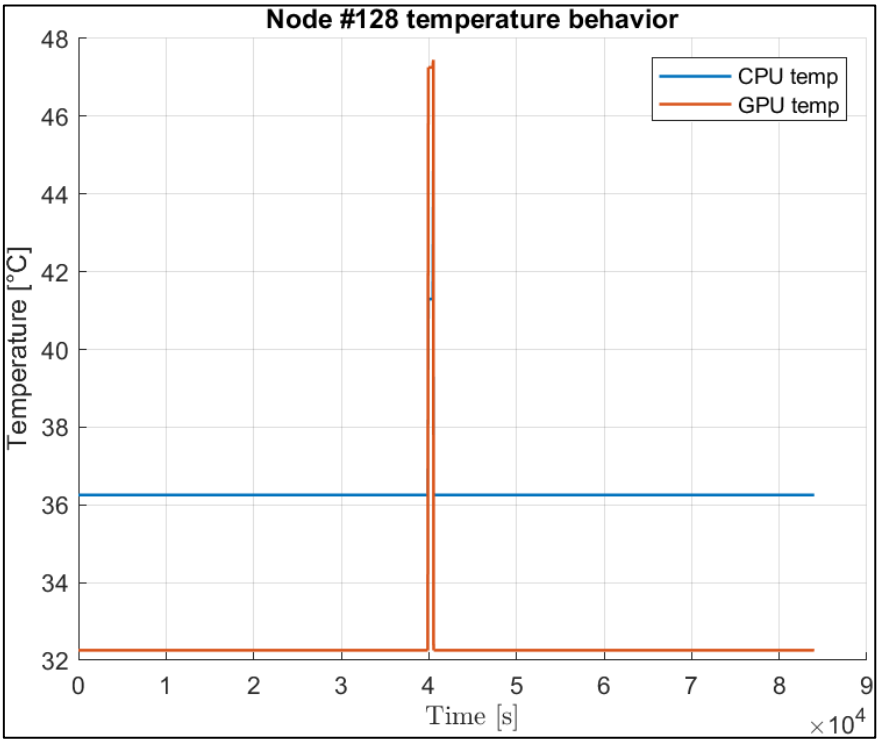


Figure 6.6: Node #128 temperature behavior

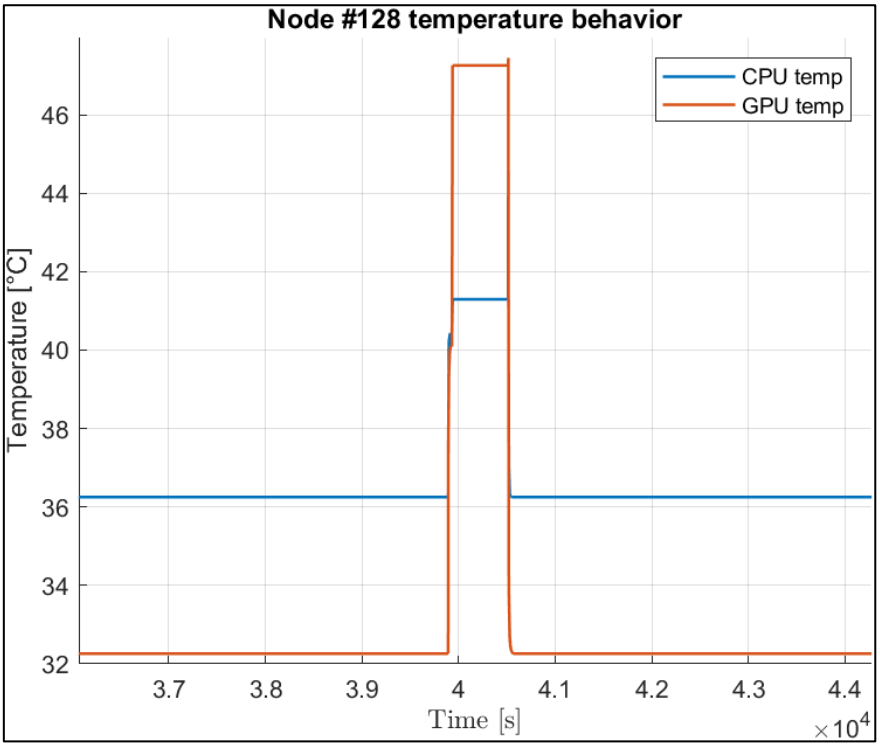


Figure 6.7: Node #128 temperature behavior (zoom)

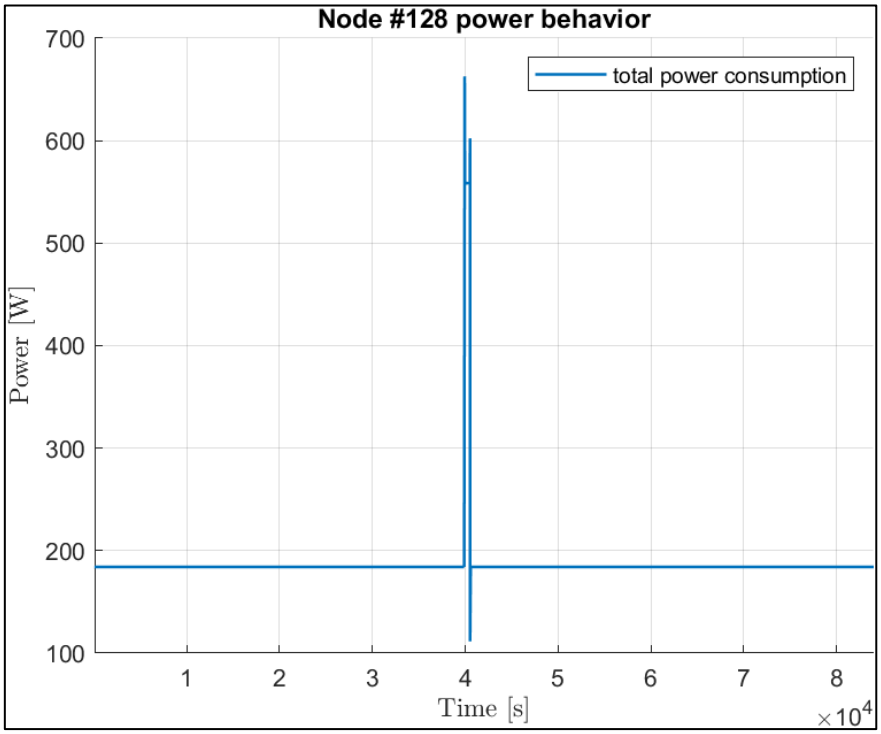


Figure 6.8: Node #128 power behavior

An interesting aspect could be to compare the total energy consumption due to the real job scheduler, and the one resulting from our thermal-aware optimizer. In this sense, we could approximately understand how good the logic behind our scheduler is.

As a result, we were able to save around 311 KWh of energy. In fact, considering the 30th of June, while the real datacenter consumed about 808 KWh keeping all nodes turned on, our algorithm, handling nodes powering on and off, succeeded in consuming only 497 KWh. This numbers can be translated into 38% of energy consumption savings.

Chapter VII

Conclusions and Future Work

7.1 Conclusions

In this thesis we aimed to achieve two primary objectives: to develop linear, nonlinear and neural network models for the prediction of the nodes' temperatures and to formulate a predictive optimization problem capable of acting as a thermal-aware job scheduler.

We successfully fulfilled both aims. Firstly, a Random Forest Regressor was developed to predict CPU and GPU mean power consumptions for the submitted jobs. Then, we developed several models capable of predicting nodes' temperature behaviors and we progressively made comparisons. Among them, LSTM neural networks performed better.

Two approaches for the optimization problem formulation were given. They are both based on a Genetic Algorithm, but with two different implementations for the optimization variables. The second one behaved better in terms of computational resources, for this reason it was selected for testing.

After the fulfillment of the testing phase, we submitted to the control algorithm an entire day of real jobs following the same time scheduling of the 30th of January, exploiting the Job Table.

As a result, we were able to save 311 KWh of energy. In fact, while the real datacenter consumed about 808 KWh keeping all nodes turned on, our thermal-aware algorithm, respecting jobs deadlines and avoiding the continuous turning on and off nodes, succeeded in consuming only 497 KWh with the great result of 38% of energy consumption savings.

7.2 Future work

In this thesis we made several assumptions:

- Each node is supposed to execute maximum one job at a time.
- Each job can be allocated to maximum one node.
- We cannot know the real execution time of the jobs on the assigned nodes.
- We cannot measure the runtime ambient temperature.
- We cannot measure the runtime total power consumption of each node.

Therefore, improvements in the definition of the optimization problem can be performed. First of all, constraints can be reformulated to consider the possibility to allocate more nodes to one single job, or more jobs to a single node.

Moreover, in this thesis, not having at disposal the real plant to simulate the thermal-aware job scheduler, we projected an open loop predictive control. An interesting future improvement would be to develop a simulator able to reproduce datacenter outputs.

Bibliography

[1] Stephanie Susnjara and Ian Smalley. ≪ *What is a datacenter?* ≫, 2024.
https://www.ibm.com/topics/data-centers
(cit. on pp. 4-5)

[2] Internation Energy Agency: IEA. ≪ *Tracking Data Centres and Data Transmission Newtorks* ≫, 2023.
https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks
(cit. on pp. 5-7)

[3] Daniele Cesarini, Andrea Bartolini, Andrea Borghesi, Carlo Cavazzoni, Mathieu Luisier, Luca Benini. ≪ *Countdown Slack: A Run-Time Library to Reduce Energy Footprint in Large-Scale MPI Applications?* ≫, 2020.
https://ieeexplore.ieee.org/document/9109637
(cit. on pp. 7-8)

[4] CINECA Technical Portal. ≪ *UG3.1: Marconi UserGuide* ≫, 2024.
https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.1%3A+MARCONI+UserGuide
(cit. on pp. 10-12)

[5] Industria Italiana, Direttore: Filippo Astone. ≪ *La superpotenza di calcolo del Marconi 100 di Ibm al servizio dell'industria* ≫, 2020.
https://www.industriaitaliana.it/ibm-marconi-100-cineca-nvidia-supercomputer-intelligenza-artificiale/
(cit. on pp. 10-12)

[6] Andrea Borghesi, Alessio Burrello, Andrea Bartolini. ≪ *Examon-X: A Predictive Framework for Automatic Monitoring in Industrial IoT Systems* ≫, 2021.
https://ieeexplore.ieee.org/document/9606215
(cit. on pp. 14-15)

[7] Michele Taragna, Dipartimento Di Elettronica e Telecomunicazioni, Politecnico di Torino. ≪ *Estimation, Filtering and System Identification course*≫, 2023.
(cit. on pp. 25-30)

[8] Diego Regruto Tomalino, Dipartimento Di Automatica e Informatica, Politecnico di Torino. ≪ *Laboratory of Robust Identification and Control course* ≫, 2023.
(cit. on pp. 25-30)

[9] MATLAB documentation, MathWorks. ≪ *Nonlinear ARX Model* ≫, 2024.
https://it.mathworks.com/help/ident/ref/nlarx.html
(cit. on pp. 30-32)

[10] Christopher Olah, Colah's blog on GitHub. ≪ *Understanding LSTM Networks* ≫, 2015.
https://colah.github.io/posts/2015-08-Understanding-LSTMs/
(cit. on pp. 33-35)

[11] IBM. ≪ *What is Random Forest?* ≫.
https://www.ibm.com/it-it/topics/random-forest
(cit. on p. 36)

[12] Atul Kumar, GeeksforGeeks. ≪ *Genetic Algorithms* ≫, 2024.
https://www.geeksforgeeks.org/genetic-algorithms/
(cit. on pp. 54-56)