ALMA MATER STUDIORUM
UNIVERSITY OF BOLOGNA

SCHOOL OF SCIENCE

MASTER'S DEGREE IN COMPUTER SCIENCE

Anomalous Activity Detection with Temporal Convolutional Networks in HPC Systems

---

Supervisor:
Prof. OZALP BABAOGLU

Author:
MATTEO BERTI

Co-Supervisors:
Prof. ANDREA BARTOLINI
Prof. ANDREA BORGHESI

II Session
Academic Year 2019/2020

Abstract

Detecting suspicious or unauthorized activities is an important concern for High-Performance Computing (HPC) systems administrators. Automatic classification of programs running on these systems could be a valuable aid towards this goal. This thesis proposes a machine learning model capable of classifying programs running on a HPC system into various types by monitoring metrics associated with different physical and architectural system components. As a specific case study, we consider the problem of detecting password-cracking programs that may have been introduced into the normal workload of a HPC system through clandestine means.

Our study is based on data collected from a HPC system called DAVIDE installed at Cineca. These data correspond to hundreds of physical and architectural metrics that are defined for this system. We rely on Principal Component Analysis (PCA) as well as our personal knowledge of the system to select a subset of metrics to be used for the analysis. A time series oversampling technique is also proposed in order to increase the available data related to password-cracking activities. Finally, a deep learning model based on Temporal Convolutional Networks (TCNs) is presented, with the goal of distinguishing between anomalous and normal activities.

Our results show that the proposed model has excellent performance in terms of classification accuracy both with balanced (95%) and imbalanced (98%) datasets. The proposed network achieves an F_1 score of 95.5% when training on a balanced dataset, and an AUC–ROC of 0.99 for both balanced and imbalanced data.

Sommario

Nell'ambito dei sistemi di calcolo ad elevate prestazioni (HPC) è utile poter determinare la natura dei programmi in esecuzione sul sistema, eventualmente segnalando attività sospette o non autorizzate dalle politiche del Data Center. Questa tesi propone un modello di apprendimento in grado di identificare una certa tipologia di programmi in esecuzione su un sistema HPC, analizzando i valori prodotti da alcuni componenti architetturali del sistema stesso. In questo contesto vengono identificati come anomali alcuni programmi che tentano di recuperare il valore originale dell'hash di una password.

I dati analizzati sono stati registrati da un sistema di monitoraggio molto avanzato installato sul cluster DAVIDE del Cineca. La mole di dati raccolta descrive l'andamento di centinaia di metriche fisiche e architetturali presenti nel sistema. La Principal Component Analysis (PCA) ci ha aiutato ad identificare quelli considerabili rilevanti in questo contesto. Viene inoltre proposta una tecnica di sovracampionamento di serie temporali che permette di aumentare i dati a disposizione relativi ai programmi di password-cracking. Infine, presentiamo un modello basato su Temporal Convolutional Networks (TCN) in grado di distinguere i programmi considerati anomali da quelli normali.

I risultati mostrano elevate prestazioni da parte del modello proposto, sia in presenza di un dataset bilanciato che in caso di sbilanciamento tra le classi. La rete mostra un'accuratezza del 95% e un F_1 score del 95.5% in caso di dataset bilanciato e un AUC–ROC di 0.99 sia in caso di dati bilanciati che non.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor, Prof. Ozalp Babaoglu, for offering me the opportunity to have this exciting experience, allowing me to work in such a stimulating research field as HPC systems.

My sincere thanks also go to my co-supervisors: Prof. Andrea Bartolini and Prof. Andrea Borghesi, for giving me access to the technical tools required for the development of the thesis, and for their invaluable support during the whole research process.

I would also like to thank Prof. Francesco Beneventi for his helpful contribution on data gathering on Cineca's HPC systems. And Prof. Antonio Libri for his useful insights during the early stages of research.

Finally, I would like to extend my gratitude to my family for encouraging and supporting me during my university years.

Contents

List of Tables

List of Figures

1 Introduction

High-Performance Computing (HPC) systems contain large numbers of processors operating in parallel and delivering vast computational power in the order of PetaFLOPs. HPC solutions are employed for different purposes across multiple domains including financial services, scientific research, computer-generated imaging and rendering, machine learning, artificial intelligence and many others.

HPC systems employ a variety of leading-edge hardware technologies that allow state-of-the-art computing power to be achieved. Yet, elevated performance levels cannot be obtained by relying only on hardware alone. Strong cooperation between hardware and software is required in order to obtain the highest performance levels.

In fact, system administrators usually rely on monitoring tools that collect, store and evaluate relevant operational, system and application data. The potential of these tools is vast, ranging from efficient energy management to retaining insights on the working conditions of the system. Over time, these tools started playing an increasingly relevant role in the system administration decision process and progressively expanded their monitoring scope.

An important use case for these system-wide monitoring frameworks is verifying that resource usage is kept within acceptable margins and that power consumption levels meet specific power band requirements. They are also crucial in supervising application behaviour. Since it has a direct impact on power consumption, the collected data are highly relevant to fine tune facility-wide power and energy management [1].

As part of this thesis we used a monitoring framework on a HPC system to create logs of system metrics during a 38-day period. One of the metrics recorded is the fine-grained per-node power consumption which is sampled every millisecond and comprises about 95% of the data. These data compose a large dataset that we make public with the intent to encourage research in the area of HPC systems.

Due to their enormous computational power, HPC systems are sometimes used also for illicit activities such as password cracking and cryptocurrency mining [2]. In order to detect and prevent these abuses, anomaly detection systems can be employed. However, oftentimes these techniques aim to detect system-specific anomalies, like performance drops, rather than user activities considered anomalous according to the data center policies. Traditionally, anomaly detection is based on analysis of system logs or log messages generated by special software tools [3]. More recently, new approaches based on analysis of sensor data and machine learning techniques have been proposed [4, 5, 6, 7].

The goal of this thesis is to analyse the physical and architectural metrics monitored by hardware sensors in a HPC system in order to detect anomalous jobs performing illicit tasks – like password cracking – employing supervised machine learning techniques.

We simulate password cracking activities based on brute force and dictionary attacks on the HPC system. The sensor data recorded by the monitoring infrastructure is collected and stored. We keep track of hundreds of architectural metrics which describe the state of the system during the execution of normal and anomalous programs. The Principal Component Analysis is applied to the data in order to obtain statistical insights on the relations between the metrics. Using this information and our knowledge of the system, a subset of metrics is selected for the anomalous activity detection. Since the password cracking programs injected in the system are significantly less than the non-anomalous programs, we propose a new time series oversampling technique, which we validate on our dataset. Finally, we employ Temporal Convolutional Networks in order to discriminate between production programs and anomalous programs carrying out illicit activities on the cluster. We test the model on both balanced and imbalanced datasets and the results obtained show an high accuracy of 95% and a F_1 score of 95.5% with balanced datasets. With imbalanced datasets the accuracy reaches 98% and the F_1 score 94%. In both cases the AUC–ROC is 0.99 which confirms the great performance of the model. All the

source code presented and used for this thesis are available on GitHub at: `www.github.com/methk/AAD-HPC`.

This thesis is organized as follows. The next Chapter describes the data collection process and the hardware infrastructure for monitoring the HPC system's architectural metrics. Moreover, the collected dataset that will be made available to the public in the near future is described. In Chapter 3 we detail the metrics recorded by the monitoring infrastructure and employed in anomalous activity detection. Chapter 4 describes the password cracking programs injected in the HPC system in order to simulate anomalous activity. In Chapter 5 we describe the feature selection process through which a subset of the metrics are chosen among all those recorded by the monitoring framework on the HPC system. Chapter 6 introduces *Temporal Convolutional Networks*, the machine learning technology that we choose to base our anomalous activity detector on. In Chapter 7 we propose a new technique to oversample multivariate time series of different lengths. Chapter 8 presents the results achieved by our classification model when discriminating between anomalous and normal activities. Chapter 9 concludes the thesis.

1.1 The DAVIDE HPC System

The system under study, DAVIDE (Development of an Added Value Infrastructure Designed in Europe), is a HPC system located at the Cineca computing center near Bologna. Cineca is a non-profit consortium consisting of 70 Italian universities, four national research centers and the Ministry of University and Research (MIUR). The High-Performance Computing department of Cineca — SCAI (SuperComputing Applications and Innovation) — is the largest computing center in Italy and one of the largest in Europe [8].

DAVIDE is an energy-aware PetaFLOPs Class High-Performance Cluster based on OpenPOWER architecture and coupled with NVIDIA Tesla Pascal GPUs with NVLink. It was designed by E4 Computer Engineering

Figure 1: Picture of the supercomputer DAVIDE.

for the PRACE (Partnership for Advanced Computing in Europe) project whose aim was to produce a leading edge HPC cluster optimizing performance and power consumption.

DAVIDE entered the TOP500 and GREEN500 list in June 2017 in its air-cooled version, while a liquid cooling system was adopted in the following years.

A key feature of DAVIDE is ExaMon, an innovative technology for measuring, monitoring and capping the power consumption of nodes and the entire system, through the collection of data from relevant components (processors, memory, GPUs, fans) to further improve energy efficiency.

DAVIDE mounts 45 compute-nodes, each of which consists of 2 POWER8 multiprocessors and 4 Tesla P100 GPUs for a total of 16 cores each, as well as a service node and a login node. For cooling SoCs (System-On-a-Chip) and GPUs, direct hot water is supplied. This allows the system to achieve a cooling capacity of roughly 40kW. As storage it mounts 1 SSD SATA. The peak performance of each node is 22 TeraFLOPs for double precision operations, and 44 TeraFLOPs for single precision operations. Giving a peak performance of ~1 PetaFLOP/s for the entire system.

DAVIDE was taken out of production in January 2020 [9].

1.2 Related Work

Very few studies use architectural metrics of HPC systems to detect particular behaviours, and at the time of this writing, we are not aware of any work aimed at classifying programs running on HPC systems based on measured metrics.

Tuncer et al. [4] aim to diagnose performance variations in HPC systems. This issue is critical. It can be caused by resource contention, as well as software or firmware-related problems, and it can lead to premature job termination, reduced performance and wasted computing platform resources. They collect several measurements through a monitoring infrastructure and convert the resulting data into a group of statistical features modeling the state of the supercomputer. The authors then train different machine learning algorithms in order to classify the behaviour of the supercomputer using the statistical features.

Baseman et al. [5] propose a similar technique for anomaly detection in HPC systems. They collect a large amount of sensor data and apply a general statistical technique called *classifier-adjusted density estimation* (CADE) to improve the training of a Random Forest classifier. The learning model classifies (as normal, anomalous, etc.) each data point (set of physical measurements).

Netti et al. [6] compare multiple online machine learning classifiers in order to detect system faults injected through FINJ, a fault injection tool. They identify Random Forests as the more suitable model for this type of task.

The methods described rely on supervised machine learning models. Borghesi et al. [7] propose instead a semi-supervised approach which makes use of autoencoders to learn the normal behavior of a HPC system and report any anomalous state.

The studies mentioned above detect system behavior anomalies which could lead to poor performance, wasted resources and lower reliability during job execution. The purpose of this thesis is instead to detect pro-

grams which are considered anomalous by the system administrators based on subjective criteria, and not on software or hardware anomalies. This makes the detection more difficult because the anomaly is not intrinsic in the program or system behavior, and the same job could be marked as anomalous or normal depending on context.

2 Data Collection

During the period from the end of November 2019 to the beginning of January 2020, we collected 884 GB of data containing physical and architectural metrics. These data were recorded by the *ExaMon* [10] infrastructure monitoring the DAVIDE HPC system at Cineca.

The exact dates on which the data were collected are: 29/11/2019, 30/11/2019, 01/12/2019, 02/12/2019, 03/12/2019, 04/12/2019, 05/12/2019, 06/12/2019, 07/12/2019, 08/12/2019, 09/12/2019, 10/12/2019, 11/12/2019, 12/12/2019, 13/12/2019, 14/12/2019, 15/12/2019, 16/12/2019, 17/12/2019, 18/12/2019, 19/12/2019, 20/12/2019, 21/12/2019, 22/12/2019, 23/12/2019, 24/12/2019, 25/12/2019, 26/12/2019, 27/12/2019, 28/12/2019, 29/12/2019, 30/12/2019, 31/12/2019, 08/01/2020, 09/01/2020, 10/01/2020, 11/01/2020 and 12/01/2020.

In order to access the sensor data, we created a project on DAVIDE through which we could interface with the monitoring infrastructure and run the scripts to collect the recorded data. The following sections give an overview of the monitoring infrastructure and the scripts which were used to collect data.

2.1 The ExaMon Monitoring Infrastructure

ExaMon is a highly scalable HPC monitoring infrastructure capable of handling the massive sensor data produced by the exascale HPC physical and architectural components. The monitoring framework can be divided into four logical layers, as illustrated in Figure 2.

Sensor collectors: these are the low-level components having the task of reading the data from the several sensors scattered across the system and delivering them, in a standardized format, to the upper layer of the stack. These software components are composed of two main objects: the MQTT API, which implements the MQTT messaging protocol, and the Sensor

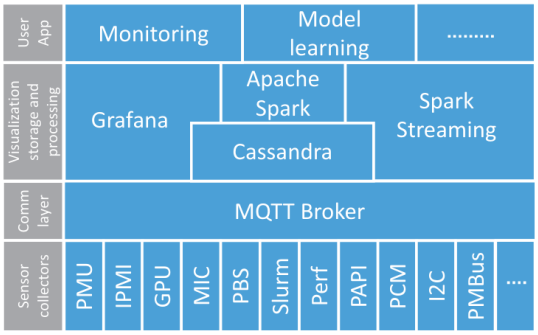


Figure 2: ExaMon architecture.

API, which implements the custom sensor functions related to the data sampling.

Communication layer: the framework is built around the MQTT protocol. MQTT implements the publish-subscribe messaging pattern and requires three different agents to work: the *publisher*, which has the role of sending data on a specific topic. The *subscriber*, which needs certain data and therefore subscribes to the appropriate topic. The *broker*, which has the functions of receiving data from publishers, making topics available and delivering data to subscribers. The basic MQTT communication mechanism works as follows: when a publisher agent sends data having a certain topic as a protocol parameter, the topic is created and made available at the broker. Any subscriber to that topic will receive the associated data as soon as it is available to the broker. In ExaMon, collector agents have the role of publishers.

Visualization, storage and processing: the data published by the collectors is used for three main purposes: real time visualization using web-based tools (Grafana), short-term storage in NoSQL databases (Cassandra), useful both for visualization and for batch processing (Apache Spark), and real time data processing (Spark Streaming). In this scenario, the adapters that interface Grafana and Cassandra to the MQTT broker, and thus to the data published by the collectors, are MQTT subscriber agents that establish a link between the communication layer and each specific tool.

User applications: finally, in the upper layer of the stack lie all the other applications that can be built on top of the layers below, such as infrastructure monitoring, model learning, process control, data analytics and so on [10].

2.2 Data Gathering

Generally, resource management in a parallel cluster is enforced by a scheduler. The scheduler manages user requests and provides the resources while trying to guarantee a fair sharing. On DAVIDE, the resource manager is SLURM (Simple Linux Utility for Resource Management) [11], a free and open-source job scheduler for Linux and Unix-like kernels. In this context, a *job* is a unit of execution that performs some work in the HPC system.

SLURM consists of a `slurmd` daemon running on each compute-node and a central `slurmctld` daemon running on the login node. The `slurmd` daemon provides fault-tolerant hierarchical communications. SLURM provides the following commands:

`sbatch` is used to submit a job script for later execution. The script will typically contain one or more `srun` commands to launch parallel tasks.

`scancel` is used to cancel a pending or running job. It can also be used to send an arbitrary signal to all processes associated with a running job.

`squeue` reports the state of the jobs. It has a wide variety of filtering, sorting, and formatting options. By default, it reports the running jobs and the pending jobs sorted by priority.

SLURM provides many other commands [11]. The following commands were used to run, check and stop the data collection script:
`sbatch back_up_jobs_metrics.sh`, runs the script as a job on the system.
`scontrol show job <jobID>`, reports the specified job's status.
`scancel <jobID>`, stops the specified running or pending job.
`squeue -u $USER`, reports the status of all the jobs launched by the user.

In order to retrieve and store the data collected by ExaMon, a simple

Python script was used. It establishes a connection to a database and executes a few queries. This program was run on DAVIDE and it backed up all the sensor data on the DRES storage space: a shared storage area of 6.5 PB mounted on all system login-nodes.

Algorithm 1 Backup job and metric data

```
selected_metrics ⇐ read list of metrics from user input
authenticate user to Cassandra
connect to Cassandra DB
result ⇐ execute("SELECT * FROM davide_jobs")
filtered_jobs ⇐ all result rows executed the day before
save filtered_jobs to parquet file
close connection to Cassandra DB

ExamonQL ⇐ authenticate user to KairosDB via ExaMon Client
for metric in selected_metrics do
  if metric = "power" then
    for node in [1...45] do
      result ⇐ ExamonQL.execute("SELECT * FROM {metric}
                WHERE node = {node}")
      save result to parquet file
  else
    result ⇐ ExamonQL.execute("SELECT * FROM {metric}")
    save result to parquet file
```

Algorithm 1 describes the pseudo-code for the data gathering script. The script can be divided into two main parts. First, job data collection. These data include a large amount of information about the jobs, like the start and end time, the nodes on which a job was running, the user who launched the execution and more.

Secondly, metric data collection. These data describe the HPC system activity by monitoring its physical and architectural components. The metrics have a sampling rate of 5 or 10 seconds, except for the per-node power consumption metric, which has a fine-grained sampling rate of 1 millisecond. Due to this high sampling rate, the power metric generates

a massive amount of data, which is stored separately for each compute-node.

In order to automate the data collection process, the software utility `cron` was adopted. The script launch was scheduled every day at 8:30 AM. Since the job execution time limit on DAVIDE is 8 hours, collecting data at 8:30 AM assured that even the jobs launched at 11:59 PM the previous day would have completed their execution.

2.3 The Dataset

The dataset that we have built is a valuable contribution to numerous research fields, both related to HPC systems analysis and machine learning in general. In fact, fine-grained power metrics with 1 millisecond sampling rate are quite rare for typical HPC datasets. Moreover, the massive amount of data (884 GB) gives an extensive overview of an actual in-production HPC system during more than one full month of operation. For these reasons we decided to make the dataset available to the public. Any sensitive data will be anonymized and the link to the dataset will be published in the main page of this thesis' GitHub repository.

The dataset is divided into two parts: the jobs data, which contain information about the jobs which run on the system. The metrics data, which contain the architectural components' measurements recorded by ExaMon. The data are contained in Parquet files [12], grouped by day of recording. Apache Parquet is a free and open-source column-oriented data storage format, it provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk. As mentioned, the files are grouped in folders by day of recording. These folders are named as follows:

`FROM_<dd>_<MM>_<YYYY>_<HH><mm><ss>_TO_<dd>_<MM>_<YYYY>_<HH><mm><ss>`

Where `<dd>` is the day of the month, `<MM>` the month as integer, `<YYYY>` the

year and <HH><mm><ss> the time, which is always from midnight to midnight ("000000"). These folders contain the Parquet files related to that specific day. Since the fine-grained power metric produces a massive amount of data, the recorded measurements are divided in 45 Parquet files, one for each node, and contained in the "power" folder inside the day's directory.

It is worth mentioning that during the data collection period only the following nodes were monitored by ExaMon: 17, 18, 19, 20, 21, 22, 23, 24, 34, 36, 37, 38, 39, 41, 42, 43, 44 and 45.

Attribute	**Description**
`index *`	Table integer index.
`timestamp`	Time when the measurement was taken (timestamp).
`value`	Measurement value.
`name`	Name of the metric.
`node`	Node where the measurement was taken.
`plugin`	The architectural component under analysis.
`unt`	Unit of measurement.
`cmp`	Type of measurement: per-processor, per-core, etc.
`id`	Core id (only if the type of measurement requires it).
`occ`	OCC id (only in OCC metrics).
`ts`	Measurement timestep: 1s, 1ms (only in power metric).

Table 1: Metrics table schema (the marked attribute is the table index).

Table 1 details and explains the attributes available for metrics data. Note that the attributes depend on the type of metric, for instance OCC metrics have per-core measurements (therefore cmp, occ and id attributes), while system cooling metrics have not. These differences are highlighted in Chapter 3.

Attribute	Description
job_id *	Integer id assigned to the job by the system.
part	String specifying the system's partition used.
user_id	Integer id assigned to the user by the data center.
job_name	Name of the job.
account_name	String id assigned to the project when created.
nodes	String with the nodes specified for the job.
exc_nodes	String with the nodes where the job executed.
req_nodes	String with the nodes requested for the job.
node_count	Number of nodes used for the job.
time_limit	Maximum duration of the job.
cpu_cnt	Number of CPUs used.
min_nodes	Minimum number of nodes used for the job.
min_cpus	Minimum number of CPUs used for the job.
exit_code	Exit code returned when the execution terminated.
elapsed_time	Time elapsed from the launch.
wait_time	Time the job was pending before the execution.
num_task	Total number of parallel tasks.
num_task_pernode	Number of parallel tasks per node.
cpus_pertask	Number of CPUs used for each parallel task.
pn_min_memory	Per node minimum memory.
submit_time	Time of submission to SLURM (timestamp).
begin_time	The earliest starting time (timestamp).
start_time	Time when execution started (timestamp).
end_time	Time of termination (timestamp).

Table 2: Jobs table schema (the marked attribute is the table index).

Table 2 shows the table template related to jobs information, detailing and explaining each attribute available for this type of data.

3 System Metrics

The ExaMon monitoring framework collects massive amounts of data from a variety of hardware components in the DAVIDE HPC system. For instance, for the Intelligent Platform Management Interface hardware component temperature, current, voltage and power consumption are monitored.

This Chapter briefly describes the architectural metrics that are monitored by ExaMon and made available to our anomaly detection system.

3.1 Cooling System

DAVIDE has three racks consisting of 15 compute-nodes each and an independent Asetek RackCDU liquid cooling system mounted on each rack. The metrics recorded for these components are detailed in Figures 3a and 3b [13]:

Metric name	Description	Unit
ASETEK-RACKCDU-SMI-V2-MIB-V17::temperatureFacilityOut_0	Temperature facility out	°C
ASETEK-RACKCDU-SMI-V2-MIB-V17::temperatureServerOut_0	Temperature server out	°C
ASETEK-RACKCDU-SMI-V2-MIB-V17::temperatureFacilityIn_0	Temperature facility in	°C
ASETEK-RACKCDU-SMI-V2-MIB-V17::temperatureServerIn_0	Temperature server in	°C
ASETEK-RACKCDU-SMI-V2-MIB-V17::temperatureAmbient_0	Temperature ambient	°C
ASETEK-RACKCDU-SMI-V2-MIB-V17::pressureFacility_0	Pressure facility	mbar
ASETEK-RACKCDU-SMI-V2-MIB-V17::pressureServer_0	Pressure server	mbar
ASETEK-RACKCDU-SMI-V2-MIB-V17::flowFacility_0	Flow facility	ml/s
ASETEK-RACKCDU-SMI-V2-MIB-V17::heatload_0	Heat load	W

Figure 3a: Cooling system metrics.

ASETEK-RACKCDU-SMI-V2-MIB-V17::controllerOut_0	Current control value	%
ASETEK-RACKCDU-SMI-V2-MIB-V17::controllerOutAlpha_0	Controller output average	%
ASETEK-RACKCDU-SMI-V2-MIB-V17::deltaOutMax_0	Controller output limit	Num
ASETEK-RACKCDU-SMI-V2-MIB-V17::gainDifferential_0	Depends on future errors	Num
ASETEK-RACKCDU-SMI-V2-MIB-V17::gainIntegral_0	Depends on past errors	Num
ASETEK-RACKCDU-SMI-V2-MIB-V17::gainProportional_0	Depends on present errors	Num
ASETEK-RACKCDU-SMI-V2-MIB-V17::limitPwmMax_0	Controls the RackCDU max flow	Num
ASETEK-RACKCDU-SMI-V2-MIB-V17::limitPwmMin_0	Controls the RackCDU min flow	Num
ASETEK-RACKCDU-SMI-V2-MIB-V17::serverLeak_0	Liquid leak detection	Num
ASETEK-RACKCDU-SMI-V2-MIB-V17::serverLevel_0	Liquid level sensor	Num

Figure 3b: Cooling system metrics.

3.2 Power Supply

ExaMon also keeps track of the LiteOn PSUs which supply power on each rack on the HPC system. Figure 4 lists the metrics recorded for these components:

Metric name	Description	Unit
LITEON-PSC-MIB::pscPSOutputLoad_0	Power Supply Output load	%
LITEON-PSC-MIB::pscPSOutputEfficiency_0	Power Supply Output efficiency	%
LITEON-PSC-MIB::pscBatteryCapacity_0	Battery capacity	%
LITEON-PSC-MIB::pscBatteryTemp_0	Battery temperature	°C
LITEON-PSC-MIB::pscPSPhaseRInputCurr_1	Power Supply phase R Input current	A
LITEON-PSC-MIB::pscPSPhaseSInputCurr_1	Power Supply phase S Input current	A
LITEON-PSC-MIB::pscPSPhaseTInputCurr_1	Power Supply phase T Input current	A
LITEON-PSC-MIB::pscPSOutputCurr_0	Power Supply Output current	A
LITEON-PSC-MIB::pscPSPhaseRInputFreq_1	Power Supply phase R Input frequency	Hz
LITEON-PSC-MIB::pscPSPhaseSInputFreq_1	Power Supply phase S Input frequency	Hz
LITEON-PSC-MIB::pscPSPhaseTInputFreq_1	Power Supply phase T Input frequency	Hz
LITEON-PSC-MIB::pscPSOutputVolt_0	Power Supply Output voltage	V
LITEON-PSC-MIB::pscPSPhaseRInputVolt_1	Power Supply phase R Input voltage	V
LITEON-PSC-MIB::pscPSPhaseSInputVolt_1	Power Supply phase S Input voltage	V
LITEON-PSC-MIB::pscPSPhaseTInputVolt_1	Power Supply phase T Input voltage	V
LITEON-PSC-MIB::pscBatteryVolt_0	Battery voltage	V
LITEON-PSC-MIB::pscPSOutputPower_0	Power Supply Output power	W
LITEON-PSC-MIB::pscPSPhaseRInputPower_1	Power Supply phase R Input power	W
LITEON-PSC-MIB::pscPSPhaseSInputPower_1	Power Supply phase S Input power	W
LITEON-PSC-MIB::pscPSPhaseTInputPower_1	Power Supply phase T Input power	W
LITEON-PSC-MIB::pscBatteryPower_0	Battery power	W

Figure 4: Power supply metrics.

3.3 Intelligent Platform Management Interface

Intelligent Platform Management Interface (IPMI) is a hardware-based solution used for securing, controlling, and managing servers. Many metrics related to the IPMI are monitored by ExaMon for each compute-node. The metrics recorded for these components are listed in Figure 5:

Metric name	Description	Unit
Ambient_Temp	Node ambient temperature	°C
CPU_Core_Temp_1, ..., CPU_Core_Temp_24	Core temperature	°C
CPU_Diode_1, CPU_Diode_2	Package temperature (Diode)	°C
CPU1_Temp, CPU2_Temp	Package temperature	°C
DIMM1_Temp, ..., DIMM32_Temp	DIMMs temperature	°C
GPU_Temp_1, ..., GPU_Temp_4	GPU temperature	°C
Mem_Buf_Temp_1, ..., Mem_Buf_Temp_8	Memory temperature (Centaur)	°C
CPU_VDD_Curr	CPU current	A
Fan_1, ..., Fan_4	Fan speed	RPM
CPU_VDD_Volt	CPU voltage	V
Fan_Power	Fan power	W
GPU_Power	GPU power	W
Mem_Cache_Power	Memory power (Centaur)	W
Mem_Proc0_Pwr, Mem_Proc1_Pwr	DIMMs power	W
PCIE_Proc0_Pwr, PCIE_Proc1_Power	PCIExpress power	W
Proc0_Power, Proc1_Power	CPU power	W
System_Power	Node total power	W

Figure 5: IPMI metrics.

3.4 On-Chip Controller

The On-Chip Controller (OCC) is a co-processor embedded directly on the main processor die. It can be used to control the processor frequency, power consumption, and temperature in order to maximize performance and minimize energy usage. Figures 6a, 6b and 7 detail the metrics recorded for these components:

Metric name	OCC	CMP	ID	Description	Unit
CUR_VDD0	1,2	PROC	0	Current consumption at processor's Vdd regulator output	A
TEMP_P0	1,2	PROC	0	Vector sensor that is the average of all core temperatures for this processor	°C
TEMP_P0	1,2	CORE	0-11	Average temperature of DTS sensors for processor's core <id>: 100% corresponds to nominal frequency	°C
TEMP_P0PEAK	1,2	PROC	0	Vector sensor that is the peak of all core temperatures for this processor	°C
NOTFIN_P0	1,2	CORE	0-11	Not finished (stall) cycles counter for core <id> on this processor	cyc
NOTBZE_P0	1,2	CORE	0-11	Not busy (stall) cycles counter for core <id> on this processor	cyc
M4RD_MEM	1,2	MEM	0-11	Memory cached (L4) read requests per sec for processor's MC <occ>, Centaur <id> (MBA01/MBA23)	GB/s
M4WR_MEM	1,2	MEM	0-11	Memory cached (L4) write requests per sec for processor's MC <occ>, Centaur <id> (MBA01/MBA23)	GB/s
CMBW_P0	1,2	CORE	0-11	Average memory bandwidth for core <id> on this processor	GB/s
MRD_P0	1,2	MEM	0,1,4,5	Memory read requests per sec for processor's memory controller <id>	GB/s
MWR_P0	1,2	MEM	0,1,4,5	Memory write requests per sec for processor's memory controller <id>	GB/s

Figure 6a: On-chip controller metrics.

FREQ_P0	1,2	PROC	0	Average of all core frequencies for processor	MHz
FREQ_P0	1,2	CORE	0-11	Requested frequency from OCC for core <id>	MHz
FREQA_P0	1,2	CORE	0-11	Average/actual frequency for this processor, core <id> based on OCA data	MHz
IPS_P0	1	PROC	0	Vector sensor that takes the average of all the cores in processor	MIP
IPS_P0	1,2	CORE	0-11	Instructions per second for core <id> on this processor	MIP
VOLT_V0	1,2	VRM	0	Voltage request for this processor's Vdd voltage rail	mV
VOLT_V1	1,2	VRM	0	Voltage request for this processor's Vcs voltage rail	mV
WINKCNT_P0	1,2	PROC	0	Count the number of cores that are winkled in this processor	Num
SLEEPCNT_P0	1,2	PROC	0	Count the number of cores that are sleep in this processor	Num
UTIL_P0	1,2	CORE	0-11	Utilization of this processor's core <id> (100% = fully utilized). NOTE: per thread HW counters are combined as appropriate to give this core level utilization sensor	%
UTIL_P0	1,2	PROC	0	Average of all core utilizations for this processor (100% = fully utilized).	%

Figure 6b: On-chip controller metrics.

The metrics listed in the figure below describe the per-component power consumption:

Metric name	OCC	CMP	ID	Description	Unit
PWR_VDD0	1,2	PROC	0	Power consumption for this processor's Vdd regulator input (12V)	W
PWRPX_P0	1,2	CORE	0-11	Power proxy sensor for core <id> on this proc.	W
PWR_null	1	SYS	0	Bulk power of the system - Master only sensor	W
PWR_P0	1,2	PROC	0	Power consumption for this processor	W
PWR_MEM	1,2	PROC	0	Power consumption for memory for this processor	W
PWR_STORE	1	SYS	0	Power consumption of the storage subsystem (storage 12V rail) - Master only sensor	W
PWR_FAN	1	SYS	0	Power consumption of the system fans - Master only sensor	W
PWR_IO	1	SYS	0	Power consumption of the IO subsystem (including storage digital 5V or less trail) - Master only sensor	W
PWR_GPU	1	SYS	0	Power consumption of the GPU	W
PWR_VCS0	1,2	PROC	0	Power consumption for this processor's Vcs regulator input (12V)	W
PWR_APSS	1	SYS	1-15	Power provided by APSS channel <id>	W

Figure 7: On-chip controller metrics.

3.5 Fine-grained Power

Power consumption at compute-node power plug is recorded with a sampling rate of 1ms. This metric generates a huge amount of data in comparison to the aforementioned metrics, which have a sampling rate of 5 or 10 seconds.

Metric name	Ts	Description	Unit
power	1s, 1ms	Power consumption at node power plug	W

Figure 8: Fine-grained power metric.

A total of 160 metrics are monitored, but the number of metric parameters is significantly higher. For instance, each of the metrics that contain per-core measurements can be divided into 12, one for each of the 12 cores in each On-Chip Controller monitored.

4 Password Cracking Activities

In order to simulate anomalous activity, multiple password cracking jobs were run simultaneously. Since the use of existing offline password cracking software such as *Hashcat* or *John the Ripper* would probably have alerted system administrators, leading to my account being blocked, a simpler alternative was adopted.

The goal was to collect the hardware metric data generated by password cracking activities in the HPC system. Therefore two scripts were created, one implementing a *brute force* attack (a technique employed in both password cracking and crypto-mining) and one simulating a *dictionary password* attack.

Both scripts were kept as simple as possible so as to avoid detection by the current malware detection system, while allowing me to collect the HPC system's hardware component data. If a machine learning model were to successfully identify a set of common instructions (such as nested for-loops) as anomalous, detecting more specific and unique behaviours would be straightforward.

Again, it must be emphasized that the task was to collect the data generated by the HPC system while performing a specific activity, in order to discriminate it from the others. The goal was not to identify state-of-the-art advanced password cracking algorithms running on the system, but rather to build a learning model capable of classifying a specific collection of jobs based on the data generated by the hardware components.

On the following days: 23/12/2019, 24/12/2019, 27/12/2019, 28/12/2019, 29/12/2019, 30/12/2019, 31/12/2019, 08/01/2020, 09/01/2020, 10/01/2020, 76 anomalous jobs were run on the system. Half were brute force attacks, half dictionary attacks.

Some of them were run on a single node, while others on two nodes. Some engaged all available resources in the node(s), while others shared the computational power with multiple jobs running on the same node(s). This allowed us to collect sensor data from the HPC system while running

under different conditions.

4.1 Brute force attack

Algorithm 2 Password cracking: brute force attack

```
password ⇐ read first argument
if password is empty then
  password = random_password(length=random_length([3...6]))
password = hash_password(password)

for len in [1...6] do
  for guess in len long combinations of chars and digits do
    if hash_password(guess) = password then
      return true
return false
```

The brute force attack script is simple and its pseudo-code is shown in Algorithm 2. The script can receive a generic string (i.e. a plain-text password) as argument. If no string is passed as input, then a random ASCII string composed of characters and digits is generated. The string is then hashed using the `bcrypt` library [14], an implementation of the Blowfish cipher, which is the standard hashing algorithm for many operating systems, such as OpenBSD and SUSE Linux.

Finally, we generate all possible character and digit combinations of a variety of lengths, ranging from 1 to 6. If a matching password is found, the execution ends.

4.2 Dictionary attack

Algorithm 3 Password cracking: dictionary attack

```
mpos ⇐ read first argument
password ⇐ read second argument

if password is empty then
  password = random_password(length=random_length([3...6]))
else if password = 'random' then
  password = get_random_word('passwords.txt', max_pos=mpos)
password = hash_password(password)

for index, psw in 'passwords.txt' do
  if index < mpos then
    if hash_password(psw) = password then
      return true
  else
    break
return false
```

Dictionary attacks usually rely on a list of common passwords in order to crack a given hash. The script works with the text file `'passwords.txt'` which contains the 1,000,000 most common passwords according to SecLists [15].

The script can either draw a password at random from `'passwords.txt'`, receive a plain-text password as input or generate a random sequence of characters and digits. In the latter case the password will probably not be found during the dictionary attack, however it is useful to collect system data regarding this scenario also. The plain-text password is then hashed with the same cipher used in the brute force script.

Finally, the script loops over all the passwords in the dictionary. The user can also specify a maximum number of attempts as argument. If a matching hash is found or the number of attempts reaches the specified value, the execution ends.

5 Feature Selection

As described in Chapter 3, ExaMon monitors a variety of physical and architectural components, which results in many hundreds of metrics being recorded. However, training a machine learning model with such an amount of features could put unnecessary computation burden on the model, reducing generalization and increasing overfitting.

Therefore, feature reduction by means of selecting few valuable metrics is desirable. As previously seen, the recorded metrics describe very heterogeneous hardware components (cooling system, power supply, on-chip controllers, and so on) and record data in many different units of measurement (Celsius, Volt, Ampere, Watt, percentages, and so on). In this context it is preferable to let the end user choose which architectural metrics to use to detect anomalous activities according to their knowledge of the HPC system.

In order to select the metrics to use for our study we relied both on our knowledge of DAVIDE and on the insights provided by a statistical analysis of the data. We computed the Principal Component Analysis [16] in order to determine which linear combinations of metrics contain the larger amount of "information", and we used these insights to support the metric selection process.

5.1 Principal Component Analysis

Principal Component Analysis (PCA) is a technique used to reduce the dimensionality of large datasets, increasing interpretability while at the same time minimizing information loss. This method achieves these results by creating new uncorrelated variables that successively maximize variance.

This chapter provides a brief overview on the intuition behind PCA and how we took advantage of it to select our metrics.

5.1.1 Standardization

Standardization is a preprocessing technique which aims to standardize the range of the initial continuous variables so that each one of them contributes equally to the analysis.

PCA is quite sensitive to the variances of the initial variables. In fact, variables with larger ranges dominate over those with smaller ranges (e.g. a variable that ranges between 0 and 100 will dominate over a variable that ranges between 0 and 1), which may lead to biased results. Therefore, scaling the data to comparable scales can prevent this problem. Oftentimes, the difference in data ranges is caused by the adoption of different units of measurement (Ampere, Volt, Celsius, percentages, etc.), as is the case with our data.

The Z-score is one of the most popular methods to standardize data, and it consists in subtracting the mean (μ) from each value (v) and dividing the result by the standard deviation (σ), as shown in the following equation:

$$z = \frac{v - \mu}{\sigma}.$$

Once the standardization is completed, all the features have a mean of zero, a standard deviation of one, and thus the same scale.

5.1.2 Covariance Matrix

A covariance matrix is computed in order to underline how the features of the input dataset vary from the mean with respect to each other. In other words, to determine the relationship that exists between them. As a matter of fact, features are sometimes highly correlated in such a way that they contain redundant information.

A covariance matrix is a symmetric matrix whose entries are the covariances associated with all the possible pairs of initial features. For instance, given the $n \times p$ sample matrix S, the variables $X_1, X_2, \ldots, X_p$ represent the columns of S, each of which represent the samplings of an initial feature.

The covariance matrix is a $p \times p$ matrix of the form

$$C = \begin{bmatrix} Cov(X_1, X_1) & \dots & Cov(X_1, X_p) \\ \vdots & \ddots & \vdots \\ Cov(X_p, X_1) & \dots & Cov(X_p, X_p) \end{bmatrix}.$$

In statistics, the *covariance* is a measure of the joint variability of two variables. The sample covariance between two variables A and B can be computed using the following formula:

$$Cov(A, B) = \frac{\sum_{i=1}^{n}(A_i - \bar{A})(B_i - \bar{B})}{n},$$

where A_i is the i-th element of the sample for variable A, $\bar{A}$ is the sample mean for A, B_i is the i-th element of the sample for variable B and $\bar{B}$ is the sample mean for B. The covariance matrix can also be represented using matrix operations:

$$C = S^T S N^{-1}.$$

Since the covariance of a variable with itself is its variance ($Cov(a, a) = Var(a)$), the main diagonal of the covariance matrix (top left to bottom right) contains the variances of each initial feature. Moreover, since the covariance is commutative ($Cov(a, b) = Cov(b, a)$), the entries of the covariance matrix are symmetric with respect to the main diagonal, which means that the upper and the lower triangular portions are equal.

If the covariance between two features is positive, then these features increase or decrease together (i.e. they are correlated). Otherwise if the covariance is negative, then the two features move in opposite directions (i.e. they are inversely correlated).

5.1.3 Principal Components

Principal Components (PC) are new uncorrelated features that are constructed as linear combinations of the initial features. They are constructed in such a way that most of the information contained within the initial features is "compressed" into the foremost components.

The principal components can be seen as the axes that provide the best angle to see and evaluate the data, such that the differences between the observations are best visible.

There are as many principal components as there are features in the data. Principal components are constructed so that the first PC accounts for the largest possible *variance* in the dataset. The second PC is computed in the same way, with the condition that it be uncorrelated with (i.e. perpendicular to) the first principal component and that it account for the next highest variance. This process continues until a total of p principal components have been computed, equal to the original number of initial features.

Figure 9 shows a visual example of principal components in a 2 dimensional dataset.

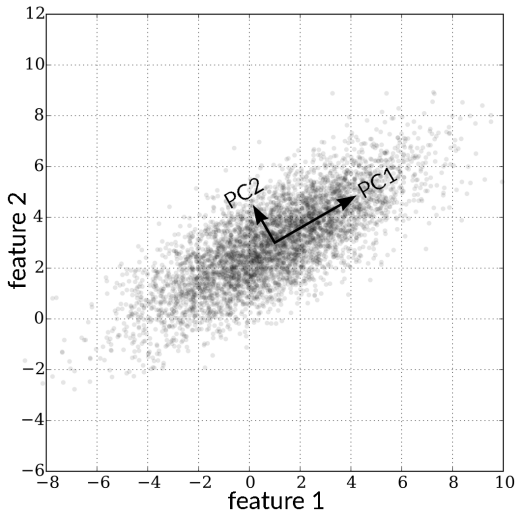


Figure 9: Example of PCA for 2-dimensional data.

Eigenvectors and eigenvalues of the covariance matrix C are computed in order to determine the principal components of the data.

The eigenvectors of the covariance matrix are the directions of the axes where there is the *most variance* (most information), called principal components. Eigenvalues are the coefficients attached to eigenvectors, which give the *amount of variance* carried in each principal component.

In order to find the eigenvalues of the covariance matrix C, the following equation must be solved:

$$det(C - \lambda I) = 0,$$

where I is the identity matrix. The solutions to the equation above are the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_p$ of the covariance matrix. The rank p of the matrix determines the equation degree, which is equal to the number of eigenvalues.

Finally, in order to compute the eigenvectors of the correlation matrix C, the following equation must be solved:

$$C\,\bar{x}_i = \lambda_i\,\bar{x}_i,$$

where λ_i is the i-th eigenvalue of matrix C and $\bar{x}_i = [x_1, x_2, \ldots, x_p]^T$ is a column vector which represents the i-th eigenvector of the correlation matrix.

The result is a $p \times p$ matrix P, where the rows are the eigenvectors of C and specify the orientation of the principal components relative to the original features. These vectors are usually sorted by their eigenvalues, which quantify the variance explained by each principal component.

5.1.4 Feature Importance

The components of each eigenvector are called *loadings* and they describe how much each feature contributes to a particular principal component. Large loadings (positive or negative) suggest that a particular feature has

a strong relationship with a certain principal component. The sign of a loading indicates whether a variable and a principal component are positively or negatively correlated. An example is shown below:

	feature 1	feature 2	feature 3	feature 4	expl. var.
$PC1$	2.12	1.37	-2.38	1.42	0.59
$PC2$	0.23	-1.05	0.16	0.02	0.33

In the example above, only the first two principal components are considered, because together they account for 92% of the total explained variance, whereas the two remaining PCs are not considered relevant enough to be taken into account. The first principal component is most related to features 3 and 1, while the second PC is most related to feature 2.

The loadings were taken into account during the metric selection process. We selected the metrics according both to our knowledge of the system and their statistical importance.

5.2 Data Processing

Performing PCA on the whole dataset would be unnecessarily resource-consuming. Therefore, the analysis was carried out on data from four days in December: 09/12/2019, 22/12/2019, 28/12/2019 and 30/12/2019, Which represent about 10% of the whole dataset.

The script `split_metrics_data.py` creates a CSV file for each available metric. Each file contains one or more columns depending on the type of recorded data. For instance, metric `"TEMP_P0"`, which records processor and core temperatures, logs data from 2 different On-Chip Controllers (`CMP=PROC` and `OCC=1,2`) and from 12 cores for each OCC (`CMP=CORE`, `OCC=1,2` and `ID=0-11`). Therefore, a total of 26 data columns are created: 12 columns for core ids 0 to 11 in OCC 1, 12 columns for core ids 0 to 11 in OCC 2 and two more columns for average processor temperature for OCC 1 and OCC 2.

The script `metric_columns_to_df.py` merges all of the columns for

each metric created with the script described above, in order to create a single CSV file. During the merging process, some data preprocessing was carried out. For instance, the metrics "`CPU_Core_Temp_*`", "`DIMM*_Temp`" and "`Mem_Buf_Temp_*`" were merged together by averaging the column values. The same was done for processor and core measurements (dividing by `CMP`: `CORE` and `PROC`). This preprocessing is useful in reducing the number of features that are related to the same architectural metrics.

Finally, the script `perform_pca.py` performs the Principal Component Analysis using the Python library scikit-learn [17]. The resulting CSV file contains the principal components sorted by explained variance.

5.3 Selection of Relevant Metrics

As mentioned before, the resulting file contains the principal components sorted by explained variance. For each component we list its loadings. That is, the degree of contribution of each metric to the principal component.

Expl.Var.	M1	M2	M3	M4	M5
46.78	CPU1_Temp	PWR_VDD0	PWR_P0	TEMP_P0PEAK	CPU_Core_Temp
13.83	Fan_3	Fan_2	Fan_1	Fan_4	PWR_FAN
8.41	MRD_P0	GPU_Temp_3	Mem_Proc0_Pwr	GPU_Temp_2	IPS_P0 CMP:CORE
6.27	Proc1_Power	UTIL_P0 CMP:CORE	IPS_P0 CMP:CORE	UTIL_P0 CMP:PROC	SLEEPCNT_P0
3.74	PWR_VCS0	PCIE_Proc1_Power	PCIE_Proc0_Pwr	GPU_Power	M4RD_MEM
2.87	Mem_Proc1_Pwr	PWR_MEM	PCIE_Proc1_Power	PWR_VCS0	Mem_Proc0_Pwr
2.52	PCIE_Proc1_Power	Mem_Proc1_Pwr	Mem_Proc0_Pwr	PWR_VCS0	M4RD_MEM
2.14	PWRPX_P0	VOLT_V0	VOLT_V1	Proc1_Power	UTIL_P0 CMP:CORE
1.85	Mem_Cache_Power	Mem_Buf_Temp	M4RD_MEM	DIMM_Temp	Ambient_Temp
1.71	FREQ_P0 CMP:CORE	M4RD_MEM	PCIE_Proc1_Power	IPS_P0 CMP:PROC	PWR_VCS0

Figure 10: First 10 principal components and their first 5 metrics, sorted by highest-to-lowest loading values.

Figure 10 shows the 5 metrics that are most related to each of the first 10 components. The metrics selected for the anomalous activity detection described in the next chapters are listed in Table 3:

Metric	**PC**	**Description**	**Unit**
CPU1_Temp	1	IPMI package temperature	°C
PWR_VDD0	1	Power consumption for Vdd regulator	W
PWR_P0	1	Processor power consumption	W
Fan_3	2	Fan 3 speed	RPM
Fan_2	2	Fan 2 speed	RPM
MRD_P0	3	Memory read requests per sec.	GB/s
Proc1_Power	4	CPU power consumption	W
PWR_VSC0	5	Power consumption for Vcs regulator	W

Table 3: Metrics selected for the anomalous activity detection.

These metrics provide a general overview of the system state by considering different measures like power consumption, IPMI temperature, fan speed, memory requests, and so on. We do not apply the PCA to the whole dataset and use the resulting variables in the analysis because we want the learning model to detect anomalous activities using initial features as they are delivered by the monitoring framework. Thus, no extra expensive computations are required in potential real-world applications.

6 Temporal Convolutional Networks

Temporal Convolutional Network is a relatively new deep learning architecture which has shown significant results in processing time series data. For this reason, it is employed in this thesis to discriminate between normal and anomalous activities.

The data collected by ExaMon model the state of the HPC system by inspecting many physical and architectural metrics. These temporal sequences of sensor data form multivariate time series, which are submitted to a Temporal Convolutional Network with the aim of identifying anomalous activities.

For time series analysis, the most commonly used technologies are Recurrent Neural Networks (RNN). Derived from feed-forward neural networks, RNNs can use their internal state, called memory, to process variable-length sequences of inputs. However, the basic RNN model is generally not directly suitable for computations requiring long-term memory; rather, a RNN variant known as Long Short-Term Memory (LSTM) is usually employed. LSTMs can process sequences with thousands or even millions of time points, and have good processing abilities even for long time series containing many high (and low) frequency components [18].

However, the latest research shows that the Temporal Convolutional Network (TCN) architecture, one of the members of the Convolutional Neural Network (CNN) family, shows better performance than the LSTM architecture in processing very long sequences of inputs [19]. A TCN model can take a sequence of any length as input and output a processed sequence of the same length, as is the case with an RNN. Moreover, the convolution is a *causal* convolution, which means that there is no information "leakage" from future to past.

TCNs use a one-dimensional, Fully Convolutional Network (1D FCN) architecture to produce an output of the same length as the input. Namely, each hidden layer is zero-padded to maintain the same length as the input layer. To avoid leakage from future to past, a causal convolution is

adopted, which means that an output at time t is the result of the convolution of exclusively elements from times t or earlier in the previous layer [20].

6.1 Sequence Modeling

Sequence modeling is the task of predicting what value comes next. Given the input sequence $x_0, x_1, \ldots, x_T$, the task is to predict some corresponding outputs $y_0, y_1, \ldots, y_T$ at each time. In this context, the key constraint is that to predict the output y_t for some time t, only the inputs that have been previously observed (that is, $x_0, x_1, \ldots, x_t$) can be used.

Formally, a sequence modeling network is any function $f : X^{T+1} \rightarrow Y^{T+1}$ that produces the mapping

$$f(x_0, \ldots, x_T) = \hat{y}_0, \ldots, \hat{y}_T,$$

while satisfying the causal constraint that y_t depend only on $x_0, x_1, \ldots, x_t$ and not on any "future" input $x_{t+1}, x_{t+2}, \ldots, x_T$.

The goal of learning in a sequence modeling setting is to find a network f that minimizes some expected loss between the actual outputs and the predictions, $L(y_0, \ldots, y_T, f(x_0, \ldots, x_T))$, where the sequences and outputs are drawn according to some distribution [21].

6.2 Fully Convolutional Networks

As mentioned above, a TCN makes use of a one-dimensional Fully Convolutional Network architecture in order to produce an output sequence of the same length as the input.

In an FCN, all the layers are convolutional layers, hence the name "fully convolutional". There are no fully-connected layers at the end, which are typically used for classification with a CNN. Instead, FCNs up-sample the class prediction layer and use a final convolutional layer to classify each value in a matrix (e.g. pixels in an image).

Therefore, the final output layer is the same height and width as the input matrix, but the dimension of each resulting cell is equal to the number of classes. Using a softmax probability function, the most likely class for each cell can be computed.

For instance, Figure 11 shows a 2D FCN used for image segmentation [22]. In the last layer each pixel is an N-dimensional vector with elements the probabilities of the image belonging to each class (which in the example are dog, cat, couch or background).

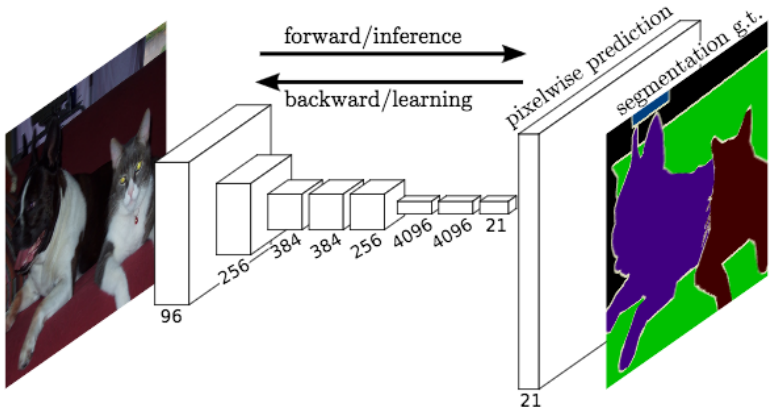


Figure 11: Example of image segmentation with an FCN.

Generally, TCNs make use of 1D FCNs which can process univariate time series data. However, if time series have multiple observations for each time step, the TCN can employ multi-dimensional FCNs.

6.3 Causal Convolutions

The second principle of TCNs is the absence of leakage from the future into the past. This is achieved by using causal convolutions.

Causal convolutions are a type of convolution used for temporal data which ensures that the model cannot violate the ordering in which the data is presented: the prediction y_t emitted by the model at timestep t cannot depend on any of the future timesteps $x_{t+1}, x_{t+2}, \ldots, x_T$.

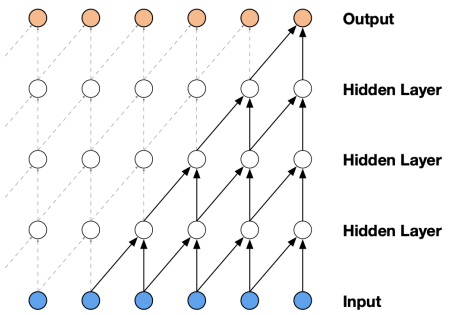


Figure 12: Example of causal convolutions in a network with 3 hidden layers.

Figure 12 illustrates which neurons each node of the output layer depends on.

6.4 Dilated Convolutions

A simple causal convolution is only capable of looking back at a history with size linear in the depth of the network. This makes it challenging to apply the aforementioned causal convolution on sequence tasks, especially those requiring longer history. A solution is to employ dilated convolutions that enable an exponentially large receptive field [21].

More formally, given a 1D sequence input $x \in \mathbb{R}^n$ and a filter $f : \{0, 1, \ldots, k-1\} \to \mathbb{R}$, the dilated convolution operation F on element s of the sequence is defined as

$$F(s) = \sum_{i=0}^{k-1} f(i) \cdot x_{s-d \cdot i},$$

where d is the dilation factor, k is the filter size, and $s - d \cdot i$ accounts for the direction of the past.

When $d = 1$, a dilated convolution is identical to a regular convolution. Using a larger dilation factor enables an output at the top level to depend on a wider range of inputs, thus effectively expanding the receptive field

of the Convolutional Network. There are, therefore, two ways to increase the receptive field of the TCN: choosing larger filter size k and increasing the dilation factor d [21].

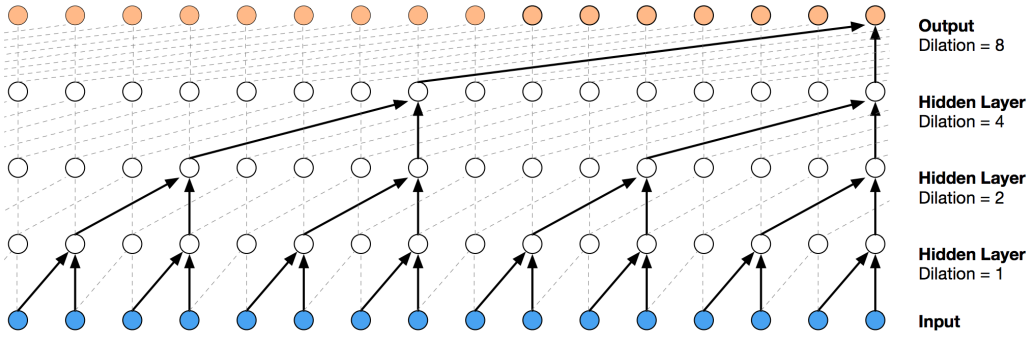


Figure 13: Example of dilated convolutions. Thanks to the dilations, the receptive field is larger than the one in Figure 12.

Figure 13 shows an example of dilated convolutions with filter size $k = 2$ and dilations $d = [1, 2, 4, 8]$.

6.5 Residual Connections

Since the receptive field of a TCN depends on the network depth n as well as the filter size k and the dilation factor d, stabilization of deeper and larger TCNs becomes important.

Very deep neural networks are difficult to train because of the vanishing and exploding gradient problems. In order to mitigate these issues, residual blocks and residual connections can be employed.

Residual blocks consist of a set of stacked layers, whose inputs are added back to their final output. This is accomplished by means of the so-called residual (or skip) connections [23].

A residual block contains a branch leading out to a series of transformations F, whose outputs are added to the input x of the block. Usually an *Activation* function is applied to the resulting value, which produces

the residual block output:

$$o = Activation(x + F(x)).$$

This effectively allows layers to learn modifications to the identity mapping rather than entire transformations, a choice which has repeatedly been shown to benefit very deep networks.

Figure 14 shows an example of the residual blocks used in TCNs. If input and output have different widths, an 1×1 convolution is added to ensure that the element-wise addition receives tensors of the same shape.

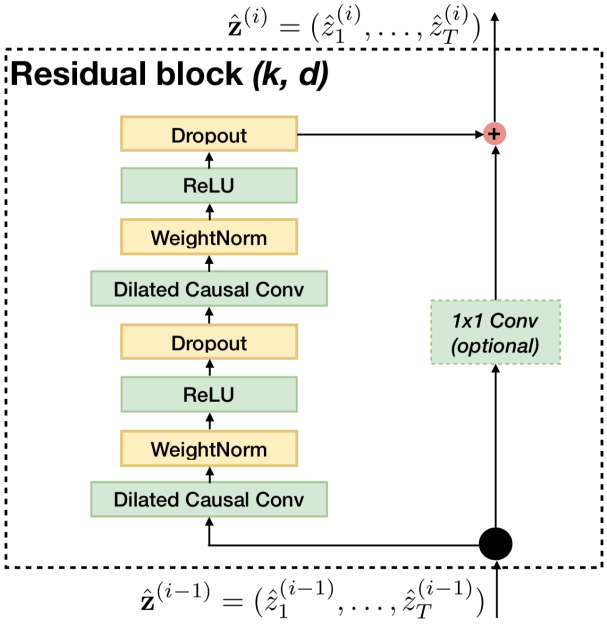


Figure 14: TCN residual block scheme.

7 Minority Class Oversampling

When the class distributions in a dataset are highly imbalanced, data are said to suffer from the *Class Imbalance Problem*. In this context, many classification learning algorithms have low predictive accuracy for the infrequent classes. Various approaches may be adopted in order to solve this issue.

Undersampling is a technique which aims to balance the class distribution of a dataset by removing observations from the majority class. This approach is preferable when the minority class contains a large amount of data, despite being outnumbered by the majority class.

Oversampling is a different approach which addresses the class imbalance problem by integrating the minority class with new observations. Some popular techniques are: Random Oversampling, which randomly duplicates the samples in the minority class. SMOTE (Synthetic Minority Over-sampling TEchnique), which generates synthetic samples based on nearest neighbours according to euclidean distance between data points in the feature space. ADASYN (Adaptive Synthetic), which generates new samples for the minority class privileging the data points that are harder to learn according to their position in the feature space.

During the period of time in which we stored the data monitored by ExaMon, a total of 2761 jobs were recorded: 86 of them were password-cracking jobs, which we consider anomalous in the context of this thesis, whereas 2675 were considered normal activities in this analysis.

This disproportion between the classes could adversely affect the accuracy of the model. In order to mitigate this issue we oversample the anomalous class. This chapter describes how the system sensor data were translated to time series and the technique we adopted to oversample the minority class.

7.1 Data Preprocessing

As detailed in Chapter 3, the monitoring framework records a large amount of sensor data. In addition, ExaMon stores job information such as start and end date, execution time, the nodes on which a job was running, the user who ran the job, the user's project account and more. Thanks to these data we are able to pair sensor values with the jobs that were running on the system.

Algorithm 4 shows the pseudo-code for the preprocessing script, which disaggregates sensor data into single metrics which are combined to create a multi-dimensional time series for each job run on the system.

Disaggregating sensor data into multi-dimensional time series associated with the recorded jobs is crucial in order to correctly identify the anomalous *jobs* executed on DAVIDE instead of just anomalous *states* of the system.

First, the script retrieves the information available for all the jobs which ran on the specified day. Then the jobs which did not run on monitored nodes and those which ran for more than 8 hours are filtered out. The reason is that during the data collection period, ExaMon was only configured on the following 18 compute-nodes: 17, 18, 19, 20, 21, 22, 23, 24, 34, 36, 37, 38, 39, 41, 42, 43, 44, 45. Moreover, the maximum job execution time for standard users on DAVIDE was 8 hours, while jobs lasting longer were run by system administrators. Filtering out sysadmin jobs, which could run up to 12 hours, decreased the size of the resulting dataset and as a consequence it reduced the computational cost required to train the learning model.

When a job runs over multiple days, the time series construction is delayed: the sensor data regarding the current day being processed is translated to the job time series, while the rest of the series is completed when the next day's sensor data is processed. This is done by saving the pending job id on a file in the `'old'` folder of the next day. Then, each time the script is run, it checks if any jobs were saved in the current day's `'old'`

folder. If so, these jobs are also included in the processing.

Algorithm 4 Time series generation

```
metrics ← read input {list of metrics to extract}
from_date ← read input {consider only jobs run in this date}
job_info ← retrieve data for jobs run in from_date

valid_jobs ← create dictionary
for job in job_info do
  if job.nodes in MONITORED_NODES
  and job.duration < 8H
  and job.nodes ≠ 'login' then
    for node in job.nodes do
      valid_jobs[node].append(job)

if from_date/old folder is not empty then
  old_jobs ← read jobs in from_date/old
  for job in old_jobs do
    for node in job.nodes do
      valid_jobs[node].append(job)

metric_data ← read metric data for selected metrics
for node in valid_jobs do
  for job in valid_jobs[node] do
    if job.end_date = from_date + 1 day then
      write job to next day's /old folder
    ts_files ← open CSV files for each combination of:
    job.id, node, CMP, OCC and ID available for metrics

  for data in metric_data[node] do
    for job in valid_jobs[node] do
      if job was running in data.timestamp then
        param_comb ← job.id+node+data.cmp+data.occ+data.id
        ts_files[param_comb].write(data.measurement)
  save and close all ts_files
```

Finally, the script reads the full metric data. As illustrated in Chapter

3, metrics have different parameters depending on their type (e.g. cooling system metrics can be grouped by rack id, on-chip controller metrics can be filtered by occ, cmp and core id, and so on). Therefore each metric parameter is a distinct dimension in the output job time series. For instance, the metric "UTIL_P0" adds 26 different dimensions to the job time series: 1 for average utilization of the first processor (OCC=1, CMP=PROC), 1 for the second processor (OCC=2, CMP=PROC), 1 for each of the 12 cores in the first processor (OCC=1, CMP=CORE, ID=0-11) and 1 for each of the 12 cores in the second processor (OCC=2, CMP=CORE, ID=0-11). With this decomposition, the learning model can take into account all the fine metric parameters, and not just the high-level metric overview (i.e. the model can learn from any of the 26 parameters of UTIL_P0 instead of just from their aggregation).

In Chapter 4 we said that 76 password-cracking jobs were executed, however from this Chapter on we will refer to 86 anomalous jobs. This increase is due to the preprocessing script, which splits a password-cracking job executed over multiple nodes into multiple anomalous time series. This distinction is necessary because from the point of view of the monitoring framework a job execution over different nodes is equivalent to multiple parallel and independent executions.

7.2 Existing Oversampling Techniques

The simplest technique employed for oversampling is Random Oversampling. This method requires to randomly select samples from the minority class and duplicate them in order to increase the number of samples available. The more advanced approaches can be divided into three main groups: interpolation techniques, probability distribution-based methods and structure preserving approaches.

A popular oversampling technique based on interpolation is SMOTE (Synthetic Minority Over-sampling TEchnique). The main idea is to randomly interpolate synthetic samples between the feature vectors of two

neighboring data points in the minority class [24]. Another famous technique is ADASYN (Adaptive Synthetic) which also generates new data by interpolating neighboring samples, but privileging those minority class data points which are surrounded by a large number of majority class samples. The reason behind this choice is that these data points are considered harder to learn [25]. However, since these techniques take into account only the local characteristics of the samples and not their correlation over time, many random data variations can be introduced, weakening the inherent interrelation of the original time series data.

RACOG and wRACOG are two probability distribution based techniques. These methods use the joint probability distribution of data attributes and Gibbs sampling in order to generate new minority class samples. While RACOG selects samples produced by the Gibbs sampler based on a predefined "lag", wRACOG selects those samples that have the highest probability of being misclassified by the existing learning model [26].

MDO (Mahalanobis Distance-based Oversampling) is an example of structure-preserving approach. This technique generates synthetic samples having the same Mahalanobis distance from the considered class mean as the other minority class samples [27]. Another example is SPO (Structure Preserving Oversampling), which generates synthetic minority samples based on multivariate Gaussian distribution by estimating the covariance structure of the minority class and regularizing the unreliable eigenspectrum [28].

A quite recent technique that should be mentioned is OHIT (Oversampling of High-dimensional Imbalanced Time-series), which generates the structure-preserving synthetic samples based on multivariate Gaussian distribution by using estimated covariance matrices [29].

All the aforementioned techniques have advantages and disadvantages. A common drawback is the lack of reliable and easily available libraries, or the unsuitability of such libraries for time series data. These approaches and the theory behind them are quite complex and their implementation from the ground up is beyond the scope of this thesis. For this reason

we developed an oversampling technique, inspired by Random Oversampling, which generates new samples that are similar, but not identical, to the minority class time series; this plays an important role in preventing overfitting. Standard oversampling techniques, including Random Oversampling, are almost always designed for cross-sectional data. This often makes them unsuitable for time series data. We intend to address this deficiency by proposing a computationally inexpensive technique which can oversample time series data by generating synthetic series close in feature space and with a similar trend to minority class samples.

7.3 Proposed Oversampling Method

The main idea is to generate synthetic samples by randomly selecting minority class time series that can act as guidance for generating new, similar series. This way the synthetic samples will be close to the original time series in feature space and they will consist of similar — yet not identical — sequences of values. The technique can be divided into two main parts: producing new time series lengths and generating sequences of synthetic values.

7.3.1 Generating Synthetic Time Series Lengths

Time series in our dataset represent the architectural metric data which were recorded while the jobs were running on the system, therefore their length can vary significantly. In order to oversample this type of data we first need to generate the lengths of the soon-to-be synthetic samples. A naive approach would be to randomly sample new uniformly distributed lengths. However this solution may not generate a faithful representation of the minority class, especially if this contains jobs of non-uniformly distributed lengths. To address this, we first compute the job length distribution of the minority class, and then we randomly oversample similar lengths according to the original distribution.

All time series have a sampling rate of 5 seconds (those of 10 seconds are interpolated to 5 seconds) and the maximum execution time is 8 hours, therefore there could be 5760 possible different lengths. Since it is very infrequent to find time series of identical lengths, we consider as similar time series whose lengths differ by a few minutes. Thus we split the set of possible lengths in multiple "windows" which contain time series with similar lengths. We define the size of the windows as W. As a consequence, the total number of windows is $N = \frac{5750}{W}$. The j-th time series of length l_j belongs to the i-th window if $w_i \leq l_j < w_i + W$, where w_i is the i-th window's shortest length, that is, $i \cdot W, \forall i \in [0, 1, \ldots, (N-1)]$.

After the aforementioned aggregation we can easily compute a vector v whose i-th element is the total number of time series belonging to the i-th window. Then we can compute the discrete distribution vector p which contains the percentage of time series for each window. Finally, we can randomly generate the lengths of the synthetic series according to the distribution p of the original time series lengths.

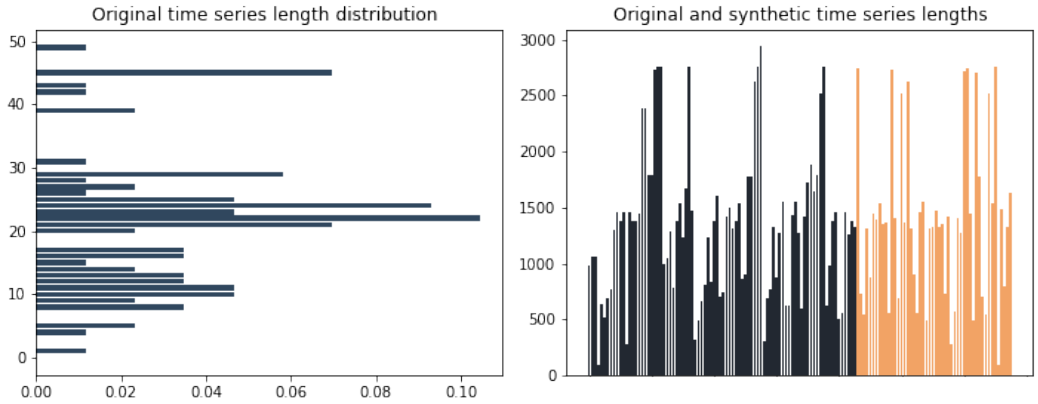


Figure 15: Original time series length distribution on the left, original (black) and synthetic (orange) time series lengths on the right.

Figure 15 shows on the left the distribution of time series lengths in the minority class, grouped by windows. The window size is $W = 60$, which means that jobs belonging to the same window differ by a maximum of 5 minutes. The plot shows that many jobs have run for about 2

hours. The chart on the right shows the original (black) and the synthetic (orange) time series lengths. It is clear the synthetic lengths have the same distribution as the original ones.

7.3.2 Sampling Synthetic Time Series Values

In order to generate the synthetic time series data points we do not sample random numbers or just duplicate existing values. Instead, we pair each synthetic time series with a time series from the original dataset and we generate values which follow the same trend. Since we only know the length of the new time series, we pair synthetic time series to random original series which belong to the same window.

The synthetic data points follow the trend of the paired time series values. The value at timestep t in a synthetic time series will be similar to a data point close or equal to t in the paired original time series. This process consists of two steps:

1. First, a random position close to t in the original time series is selected. The random sampling is not uniform, but rather it follows the standard normal probability density function:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2},$$

   where $\mu = t$ is the position of the synthetic data point and the standard deviation σ determines the amount of dispersion away from the mean. The lower the standard deviation, the higher the probability of choosing an original data point close to timestep t. Since the random sampling returns floating point values, the result is truncated in order to obtain the reference timestep.

   The probability of choosing a reference data point close to timestep t (e.g. $t-3, \ldots, t+3$) in the original time series is higher than selecting a point far in the "past" (e.g. $t-6, t-7, t-8, \ldots$) or far in the "future" (e.g. $t+6, t+7, t+8, \ldots$), as shown in Figure 16.

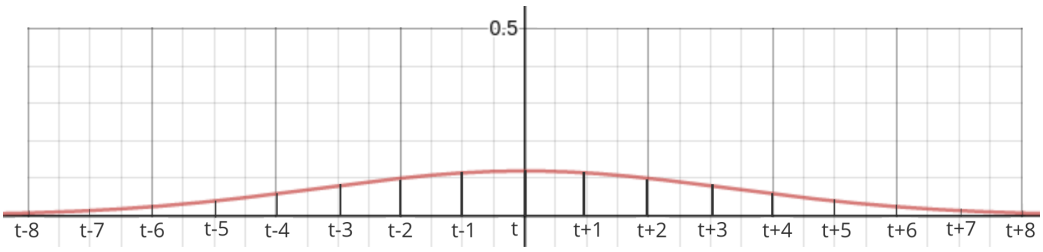


Figure 16: Random sampling probability density function.

2. In order to introduce slight randomness in the synthetic time series values and not merely duplicate the original data points, we randomly compute a new value close to the selected original one. The goal is to sample a random value inside a circle around the selected data point. Since the time series may have high dimensionality, a uniformly random point on a hypersphere is generated. A simple way to generate points uniformly at random on the interior of a d-sphere is by the generalization of the Muller algorithm [30]. The procedure consists in:

   (a) Generating d random variables $x_i, i = 1, 2, \ldots, d$ normally distributed in the range $[0, 1]$.

   (b) For each point $x = (x_1, x_2, \ldots, x_d)$, locating a point y on the unit d-sphere having the d-direction cosines

$$\frac{r \cdot x_i}{\sqrt{\Sigma_{i=1}^{d} x_i^2}}, \qquad i = 1, 2, \ldots, d,$$

   where $r = \sqrt[d]{k}$ and k is a normally distributed random variable which determines the distance from the center of the sphere.

Figure 17 broadly illustrates the oversampling process described above.

Time series (a) represents a job in the original dataset, the timesteps are placed on the x-axis and the values on the y-axis. For the purpose of this example the series has only one dimension.

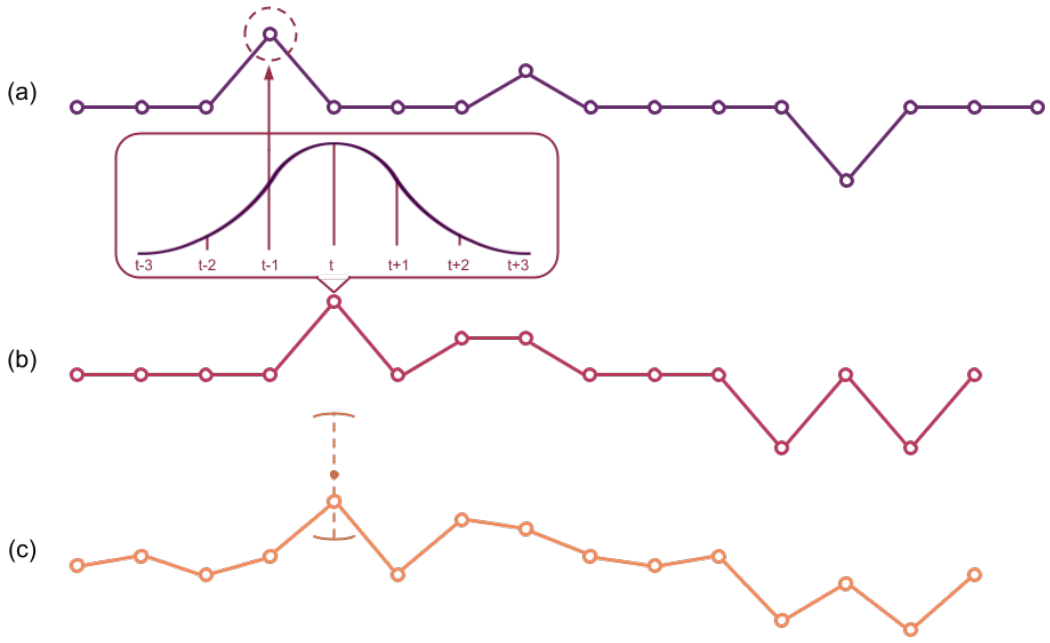


Figure 17: Oversampling process. (a) The original time series selected as reference to sample a new synthetic one. (b) Each point at time t in the synthetic time series corresponds to a data point in the paired series close or equal to time t. (c) Selected values in the previous step are slightly randomized.

Time series (b) shows the reference points, which are randomly sampled from (a), for each new synthetic data point. For instance, for the fifth data point in (b), the algorithm randomly chose the fourth data point (i.e. the $(t-1)$-th element) in the paired time series.

Time series (c) is equivalent to (b) but with a slight randomization of series values using the Muller algorithm. A moving average can also be applied to smooth the resulting values.

In conclusion, time series (c) has a similar (but not identical) trend to time series (a), which is the goal of this oversampling technique.

7.4 Validation

In the context of this thesis, we validate the technique described above on the data we use for the anomaly detection. In order to do so, we train a machine learning model on the original time series data and test it on both original and synthetic time series. The goal is to ensure that the model is not able to distinguish original time series from synthetic ones, and therefore that it has a similar classification accuracy on both types of time series.

For the validation we employ a TCN, which is the same model we adopt for the anomaly detection. We trained the TCN for 30 epochs on 128 original time series: half from the anomalous class and half from the normal class. Then we tested the model on two datasets. The first contains 88 synthetic time series generated with the aforementioned oversampling technique: 44 for the anomalous class and 44 for the normal class. The second contains 44 original time series, not present in the training set: half from the anomalous class and half from the normal class. This process is illustrated in Figure 18.

	Exp1	Exp2	Exp3	Exp4	Exp5	Exp6	Average
Original	90.91	84.10	81.82	88.64	88.64	79.55	**85.61**
Synthetic	93.18	88.64	87.50	90.91	92.05	86.36	**89.77**

Table 4: Testing accuracies of 6 different training and test sessions.

Table 4 shows the *accuracy* (i.e. the number of correctly classified time series out of all the series in the test set) during the classification of synthetic and original time series. The whole training and testing process was repeated 6 times with different training and test sets in order to compare the accuracy between multiple independent experiments. The data show that despite the small size of the datasets, the testing accuracy in the classification of synthetic and original time series is comparable. Moreover, we can notice a slightly better performance on the synthetic test set, which is probably due to the very small size of the original time series test set.

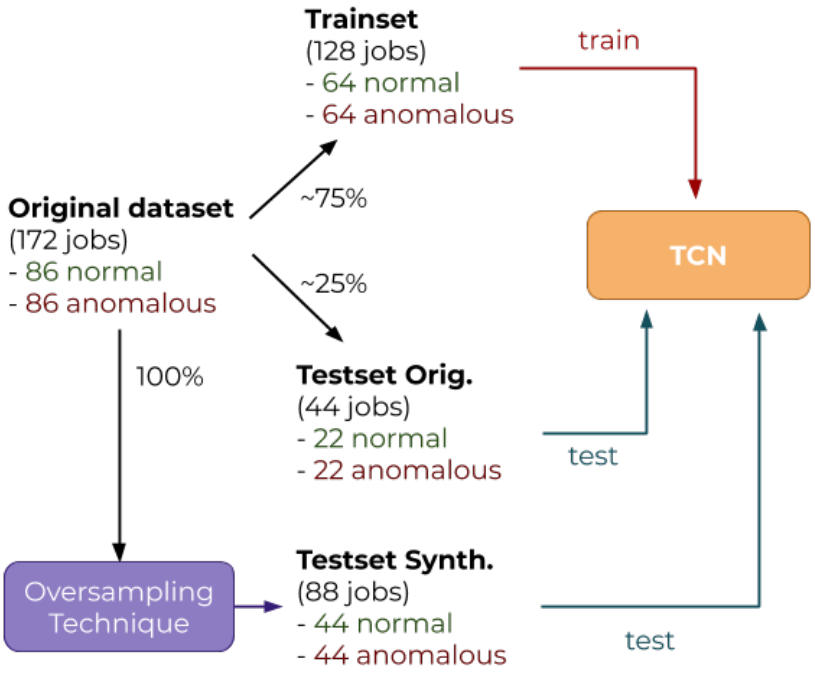


Figure 18: Validation process of the proposed oversampling technique.

In conclusion, we can affirm that the model cannot reasonably distinguish between original and synthetic time series, having a comparable classification accuracy on the two.

A few advantages of adopting this approach are that it is suitable for our dataset, it has a low computational cost, it is easy to understand and if a learning model correctly identifies anomalous activities when this basic technique is adopted, then it can certainly have better performance when a more complex oversampling approach is employed.

8 Anomaly Detection

This chapter gives an overview of the learning model employed to classify the anomalous jobs and the obtained results. We first acquaint ourselves with this type of neural network by modeling a different task: account classification. Then we use the model to distinguish anomalous jobs from normal jobs, which is the main objective of this thesis.

8.1 Account Classification

In order to become familiar with Temporal Convolutional Networks and explore the available data, we first tried to classify jobs according to the account id that launched the execution. In fact, a user must create or be added to a project account in order to execute jobs on DAVIDE. In this context, a project account gathers multiple users which work on the same project, it is identified by an id and it has a limited amount of hours to run jobs on the system.

There are 25 different accounts which ran jobs in the period from December 2019 to January 2020. Some of them executed very few jobs, so we removed these accounts from the classification task. In total, 10 accounts executed more than 40 jobs each in the observed period of time. The highest-to-lowest number of jobs per account is: 753, 498, 247, 196, 184, 164, 148, 147, 120, 62. Since a high class imbalance can lead to poorer performance, we opted to oversample each class to 300 jobs (and undersample the two most populated classes to the same size). We trained a Temporal Convolutional Network many times using different hyperparameters, a varying number of metrics and multiple epochs. Nonetheless, the loss did not fall below ~0.85 and the testing accuracy ranged between 67% and 76%.

Since these results showed that the network was struggling to determine the account associated with a job, we decreased the number of accounts taken into consideration. In fact, a possible reason for the poor

performance could be the similarity between the jobs executed by different accounts. Therefore, for classes we selected only 5 accounts, which we expected to run jobs unrelated to one another, in order to see if this could affect the model accuracy. Surprisingly the loss dropped to ~0.18 and the testing accuracy ranged between 83% and 89%.

As an additional proof, we trained and tested the same model with a dataset containing jobs from the other 5 accounts, which we expected to have run similar jobs. As anticipated, the training loss was stable around ~0.68 and the testing accuracy ranged between 71% and 78%.

In light of this, the most plausible reason for the poor performance on the account classification task was that the accounts tended to execute jobs which could be similar between different accounts. In fact, it is not uncommon for a user to take part in different projects and as a consequence to be associated with multiple accounts. In conclusion, we can assume that the account id is not a significant feature for classifying jobs. However, the main goal of this analysis was to become familiar with TCNs, as well as the available data, and to fine tune the network hyperparameters.

8.2 Learning Model Configuration

In section 8.1 we tested the potential of TCNs, studying and modeling the network to obtain the best-performing configuration. In Temporal Convolutional Networks, the kernel size and the dilation factors are critical parameters. The former determines the size of the convolutional filters used in the TCN. The latter influence the range of inputs encoded in an output, increasing or decreasing the receptive field of the network. Based on previous experiments, we know that the model can achieve excellent performance by employing 8 filters for each layer, with a kernel size $k = 4$, and the following dilation factors:

L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11
1	2	4	8	16	32	64	128	256	512	1024

Table 5: Dilations used between TCN's hidden layers for job classification.

Therefore, the network has a receptive field of 4×1024 data points, which means that each output value is produced as a function of 4096 input values.

Furthermore, we apply a Dropout layer with dropout factor 0.3 to each Residual Block, since we noticed that it helps reduce overfitting. Moreover, Layer Normalization in Residual Blocks is implemented in order to normalize the input layers by re-centering and re-scaling the values. This increases training speed and stability. The TCN is followed by a Dense layer with a sigmoid activation function to binary-classify the input.

Finally, the Adam optimizer, with a learning rate of 10^{-3}, and the binary cross-entropy loss function are employed in order to optimize the model. The binary cross-entropy is defined as follows:

$$H(P,Q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot \log Q(y=1|x_i) + (1-y_i) \cdot \log(1 - Q(y=1|x_i)),$$

where $y_i = P(y=1|x_i)$ is the real classification (1 for anomalous jobs and 0 for normal jobs) of the i-th job and $Q(y=1|x_i)$ is the predicted probability of the i-th job being anomalous. Therefore, when the label is 1, the log probability of the job being anomalous is added to the loss. Conversely, when the label is 0 the log probability of the job being normal is added. Using the logarithm of the probabilities (as opposed to the plain probabilities) is useful to significantly increase the loss value when a job is not correctly classified. The Adam optimizer, an extension to Stochastic Gradient Descent, makes use of this value in order to update the internal parameters of the model and reduce the error.

8.3 Evaluation Metrics

In this section we present the evaluation metrics employed to evaluate the TCN configuration described in the previous section, as well as the subsequent results. For this classification task we only have two classes: each job can be labeled as normal (label 0) or anomalous (label 1).

Given the following variables: TP, the number of anomalous jobs classified as anomalous; TN, the number of normal jobs classified as normal; FP, the number of normal jobs classified as anomalous and FN, the number of anomalous jobs classified as normal, we employ the following metrics:

- **Classification Accuracy**: this metric computes the proportion of correct predictions among the total number of cases examined.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}.$$

  This metric is less relevant when the dataset is imbalanced. In fact, if the model classifies all dataset samples as belonging to class 0 and the test set consist of 90% of 0-labeled elements, the accuracy is as high as 90%.

- **Log Loss**: this metric takes into account the uncertainty of a prediction based on how much it varies from the actual label. The formula is equivalent to the binary cross-entropy described above:

$$Loss = -y\ log(p) - (1 - y)\ log(1 - p),$$

  where y is the actual label and p is the probability of predicting 1.

- **Precision**: this metric computes what proportion of predicted positives is truly positive.

$$Precision = \frac{TP}{TP + FP}.$$

- **Recall**: this metric computes what proportion of actual positives is correctly classified.
$$Recall = \frac{TP}{TP + FN}.$$

- **F_1 Score**: this metric is the harmonic mean between *Precision* and *Recall* and it determines how precise the classifier is (i.e. how many instances it correctly classifies) as well as how robust it is (i.e. whether it misses a significant number of instances or not):
$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}.$$
This metric is helpful to better understand if the accuracy result depends on the imbalance of the dataset or if it is due to the actual predictive ability of the model.

- **AUC–ROC**: the ROC (Receiver Operating Characteristics) is a probability curve and AUC (Area Under the Curve) explains how capable the model is in distinguishing between two classes. The higher the AUC, the larger the number of samples the model can correctly classify. The ROC curve is plotted with *TPR* (on the y-axis) against the *FPR* (on the x-axis). The *TPR* (True Positive Rate) is equal to *Recall*, namely the proportion of anomalous jobs that were correctly classified. The *FPR* (False Positive Rate) computes the proportion of normal jobs that were incorrectly classified as anomalous:
$$TPR = \frac{TP}{TP + FN}, \qquad FPR = \frac{FP}{FP + TN}.$$
Figure 19 shows an example of an ROC curve. Each point on the curve is a different threshold value in the sigmoid classification function. If a sample is under the threshold, then it is classified as normal, otherwise it is classified as anomalous. The higher the area under the curve is, the better performance the model has.

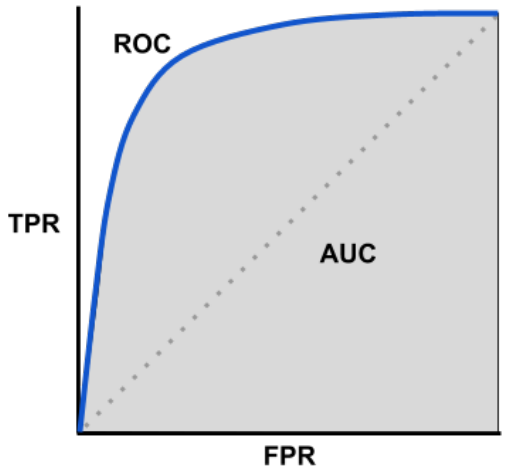


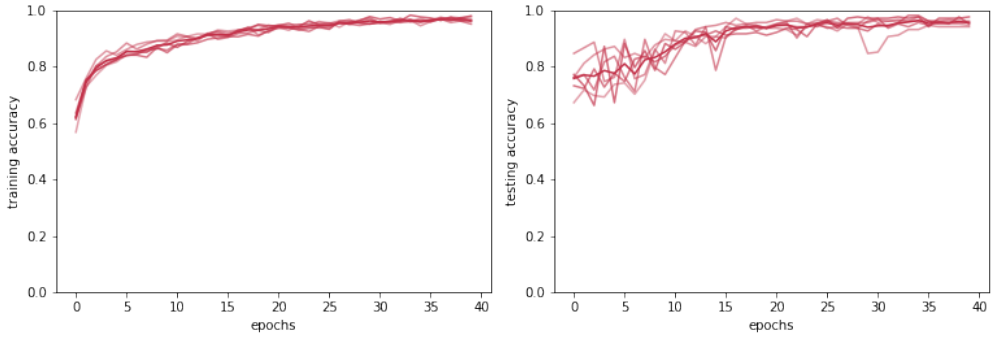
Figure 19: Example of AUC–ROC Curve. The higher the curve, the better the model.

8.4 Results on Detection of Anomalous Activities

We have 86 anomalous jobs, which represent password-cracking activities, and 2675 jobs executed by the system users, which we consider normal. The minority class is oversampled by applying the custom random oversampling technique described in Chapter 7. The metrics selected in order to train the learning model to detect if a job is anomalous or not are the following: `CPU1_Temp`, `PWR_VDD0`, `PWR_P0`, `Fan_3`, `Fan_2`, `MRD_P0`, `Proc1_Power` and `PWR_VSC0`, as described in Chapter 5.

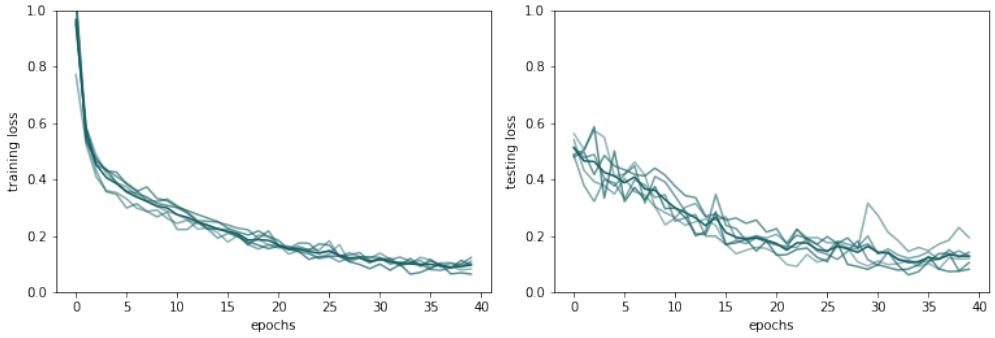
Using the configuration and the parameters described in Subsection 8.2, we implemented the learning model in Python using `Keras` with the `TensorFlow` backend. For the TCN layers we relied on the `Keras TCN` implementation by Philippe Rémi, available on GitHub [31]. First, we trained and tested the model for 40 epochs on a fully balanced dataset of 1000 jobs: 500 anomalous and 500 normal. 80% of the dataset is employed in the training process, while the remaining 20% is used for testing. Only the anomalous class is oversampled, while the normal class is randomly undersampled. Moreover, we preserve the same proportion of synthetic and

original jobs in the anomalous class in the training set and the test set.



(a) (b)

Figure 20: Training (a) and testing (b) accuracy with a balanced dataset containing 1000 jobs.



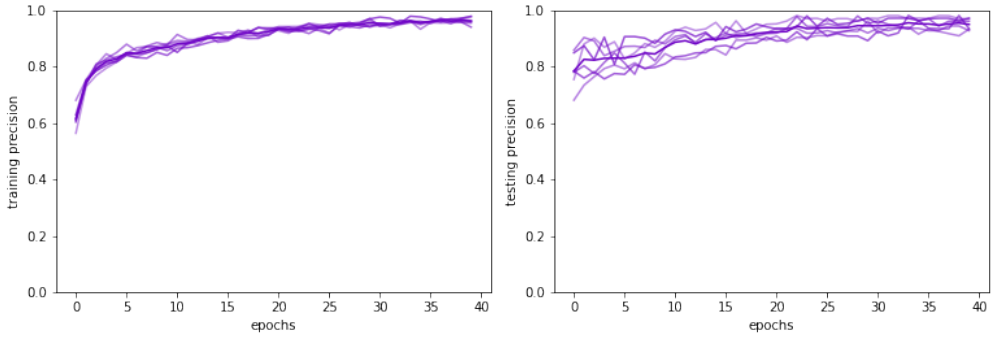
(a) (b)

Figure 21: Training (a) and testing (b) loss with a balanced dataset containing 1000 jobs.

Figure 20 shows the model accuracy progression during training and testing. Each line is a different training and test session, with different data but with the same proportions as described above. The darker line is the average of the values produced by each session. The model achieves an

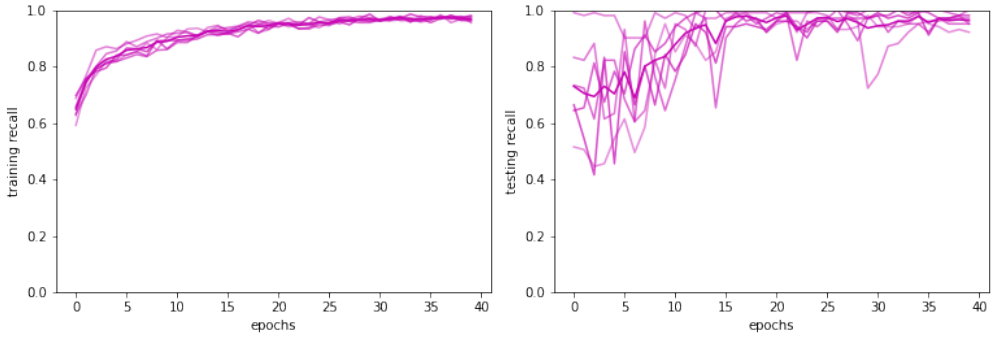
average testing accuracy of 95% after 40 epochs. Since the dataset is quite small, we consider this result excellent.

Figure 21 shows the loss of the model during training and testing. The average testing loss achieved after 40 epochs of training is 0.13.



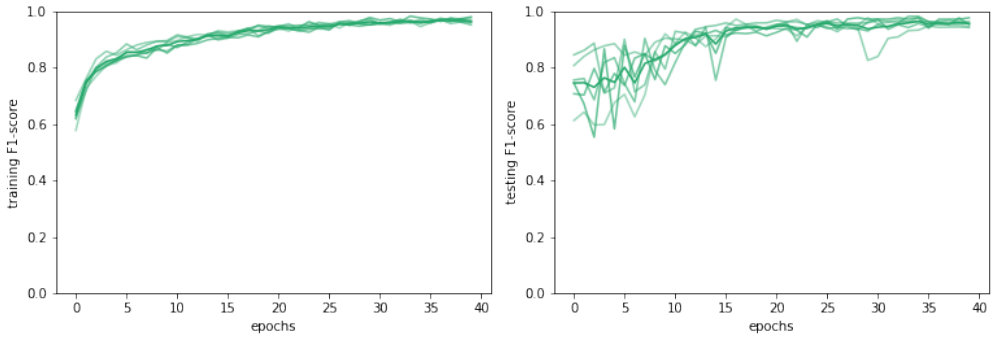
(a) (b)

Figure 22: Training (a) and testing (b) precision with a balanced dataset containing 1000 jobs.



(a) (b)

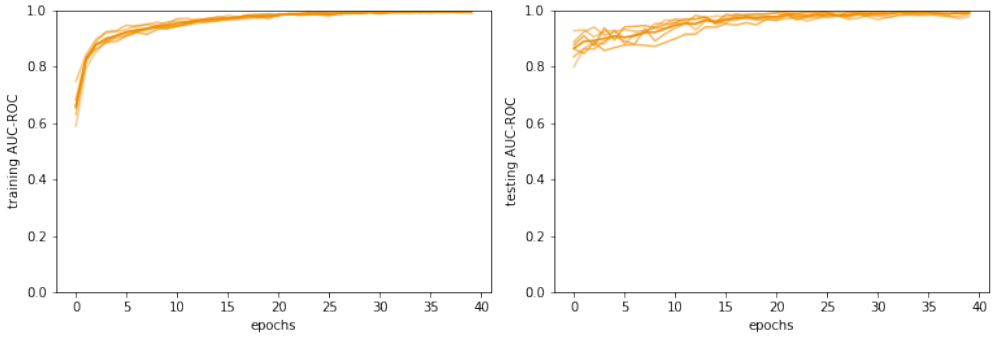
Figure 23: Training (a) and testing (b) recall with a balanced dataset containing 1000 jobs.



(a) (b)

Figure 24: Training (a) and testing (b) F_1 score with a balanced dataset containing 1000 jobs.

Figure 22 and Figure 23 show, respectively, precision and recall metrics during both training and testing. The average values after 40 epochs are 95% precision and 96% recall when validating the model. Figure 24 shows the progression of the F_1 score. This metric reaches a 95.5% average value during validation after 40 epochs of training, which is an excellent result.



(a) (b)

Figure 25: Training (a) and testing (b) AUC with a balanced dataset containing 1000 jobs.

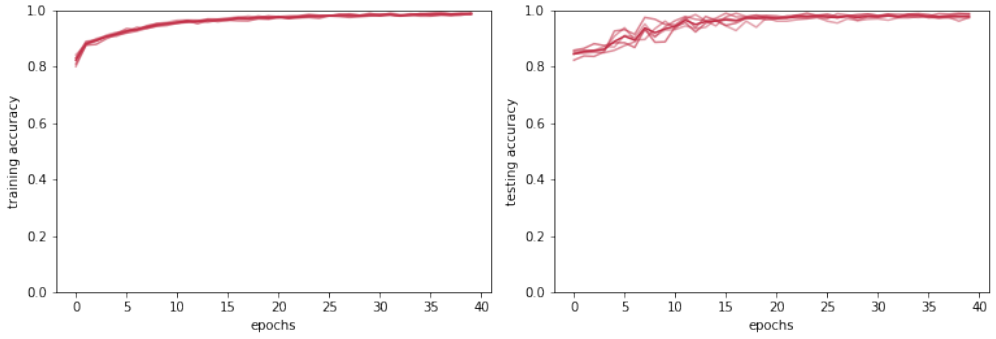
Finally, the AUC–ROC reaches an average of 0.99 during model vali-

dation, which confirms the great performance of the model.

	Accuracy	Loss	Precision	Recall	F_1 score	AUC
Training	96%	0.10	96%	97%	96.5%	0.99
Testing	95%	0.13	95%	96%	95.5%	0.99

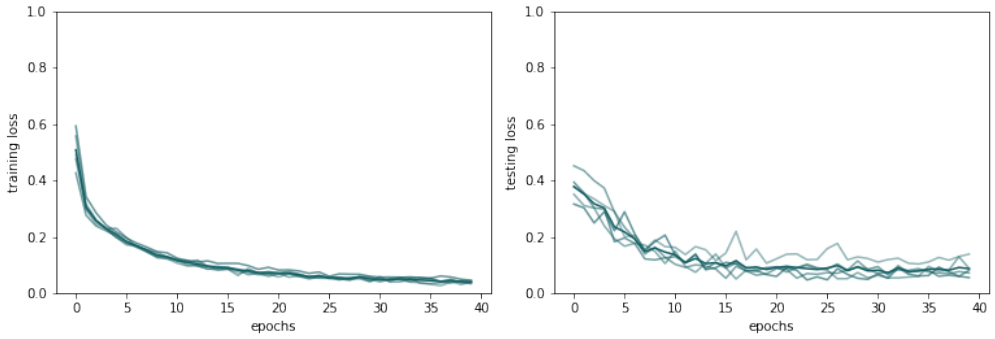
Table 6: Training and testing evaluation metric values after 40 epochs, averaged over multiple experiments. The dataset is balanced and consists of 1000 jobs.

Since the model can achieve valuable results on a balanced dataset, we also attempt to train it on a larger and imbalanced dataset. This condition is more in line with a real anomaly detection scenario, where the vast majority of the jobs is normal and few are anomalous. The full dataset contains 3000 jobs: 500 anomalous (17%) and 2500 normal (83%). We split the dataset in two parts: the training set, which contains 2400 jobs (80%), and the test set, which contains 600 jobs (20%). The two classes are proportionally split between the two datasets. Also, the distribution of synthetic and original jobs in the anomalous class is the same in the training set and the test set.



(a) (b)

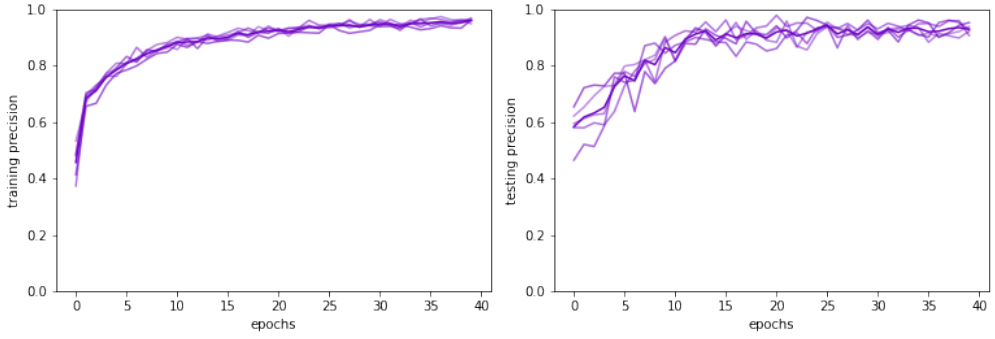
Figure 26: Training (a) and testing (b) accuracy with an imbalanced dataset containing 3000 jobs.



(a) (b)

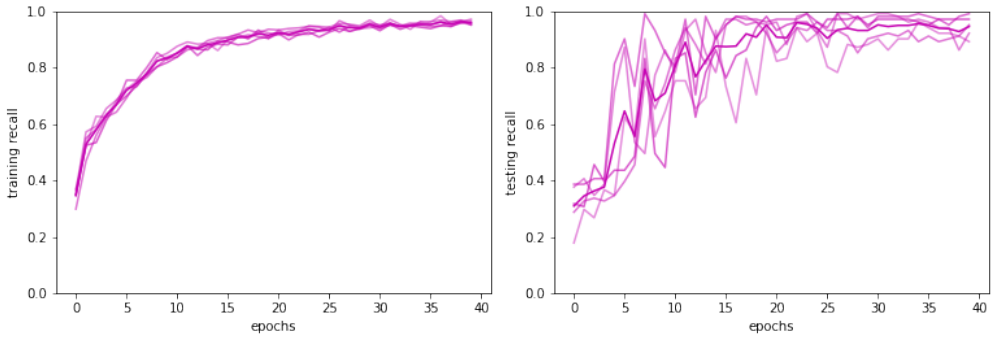
Figure 27: Training (a) and testing (b) loss with an imbalanced dataset containing 3000 jobs.

Increasing the size of the dataset seems to also increase the testing accuracy, which reaches 98% after 40 epochs, as shown in Figure 26. However, since the dataset is not balanced, this metric alone does not provide a reliable evaluation of the model. Figure 27 shows the loss metric progression, which after 40 epochs settles around 0.09, which is also better than the previous experiment.



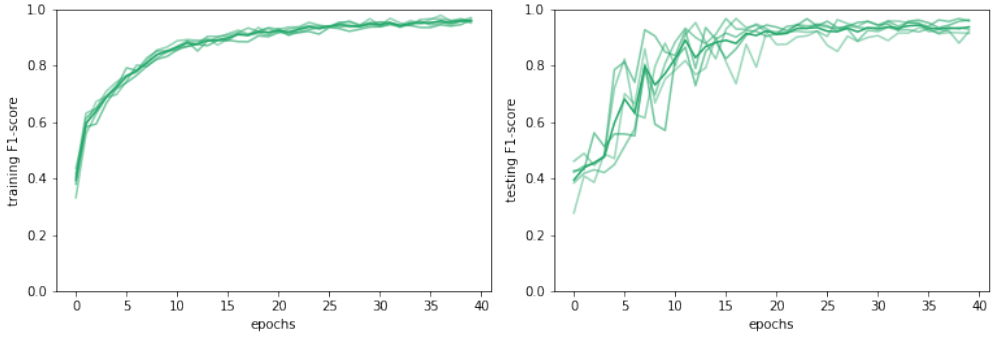
(a) (b)

Figure 28: Training (a) and testing (b) precision with an imbalanced dataset containing 3000 jobs.



(a) (b)

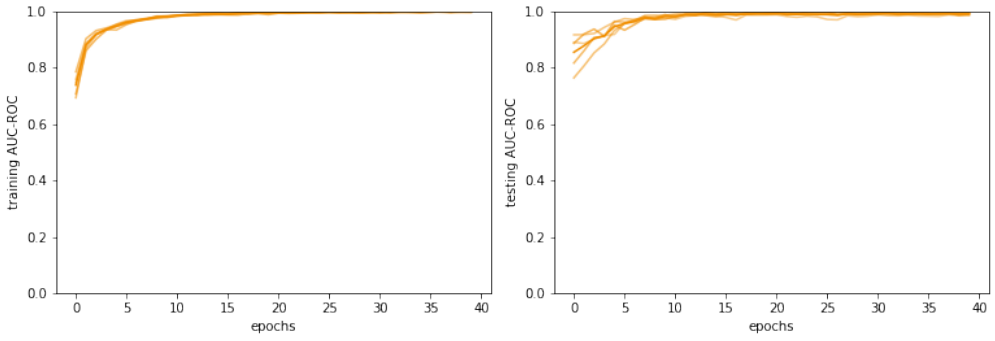
Figure 29: Training (a) and testing (b) recall with an imbalanced dataset containing 3000 jobs.



(a) (b)

Figure 30: Training (a) and testing (b) F_1 score with an imbalanced dataset containing 3000 jobs.

After 40 epochs of training, the average testing precision, recall and F_1 score are, respectively: 93%, 95% and 94%. These values are slightly lower than those measured in the previous experiment, which is probably due to the dataset imbalance.



(a) (b)

Figure 31: Training (a) and testing (b) AUC with an imbalanced dataset containing 3000 jobs.

The AOC - ROC metric gives similar results to the training sessions with 1000 jobs: 0.99. This confirms the high performance of the model.

	Accuracy	Loss	Precision	Recall	F_1 score	AUC
Training	99%	0.04	96%	96%	96%	0.99
Testing	98%	0.09	93%	95%	94%	0.99

Table 7: Training and testing evaluation metric values after 40 epochs, averaged over multiple experiments. The dataset used is imbalanced and consists of 3000 jobs.

In conclusion, the evaluation metrics observed confirm the model is able to detect password cracking jobs injected on the system with a very low error rate. Moreover, the model presented shows excellent performance also with an imbalanced dataset, which is quite similar to a real-world scenario.

9 Conclusion

This thesis aimed to illustrate the effectiveness of Temporal Convolutional Networks in detecting anomalous activities on HPC systems. Based on few architectural metrics, jobs considered anomalous can be correctly identified and reported to system administrators. The results show that the presented model classifies anomalous and normal jobs with a very high accuracy, comparable to related studies [4, 5, 6, 7], proving the effectiveness of the proposed approach.

Another relevant contribution is the creation of a public dataset containing the architectural metrics recorded in over a month of monitoring. Specifically, the fine-grained per-node power consumption is a very valuable metric which is uncommon to find in typical HPC system datasets.

First, we described the sensor data collection process, detailing the monitoring infrastructure and the HPC system under analysis. Then, we detailed the available system metrics, together with the password-cracking jobs that we injected into the system to serve as anomalous activities. After a statistical analysis, we selected a few relevant metrics out of the hundreds of system metrics available. Since the dataset had a class imbalance problem, we presented a new oversampling technique and we applied it to the minority class in order to rebalance the dataset. Finally, we illustrated the learning model based on Temporal Convolutional Network and the classification results.

We show our model reaches an accuracy of 95% and a F_1 score of 95.5% when applied to a balanced dataset. When the data is imbalanced the model reaches an accuracy of 98% and a F_1 score of 94%. In both cases the AUC–ROC is 0.99, confirming the great performance of the model.

The problem we solved with this work is not well examined in the literature. Some studies show relevant results in system anomaly detection, which is quite different from detecting a specific class of jobs considered anomalous based on subjective criteria.

9.1 Future Work

We presented a machine learning model based on batch data analysis. However, real-world applications usually need an online approach in order to detect anomalous activities at the earliest indicators. An interesting extension of the model could be to train the model on a stream of data instead of batches.

Furthermore, the number of metrics could be increased, or different metrics could be selected according to other statistical analyses or based on the sysadmin's knowledge of the system. In this context, the feature selection heavily depends on the type of metrics made available by the monitoring infrastructure.

Moreover, we used architectural metric data as recorded by the monitoring framework, while many forms of statistical analysis could be applied in order to transform them or extract valuable statistical features from them. On one hand, applying complex data transformations could improve the network ability to learn from the data. On the other hand, this could significantly slow down execution, which can be critical in stream data processing.

More advanced oversampling techniques could also be adopted in order to increase the number of elements in the anomalous class. This could possibly strengthen the model accuracy when training on imbalanced datasets.

References

[1] Netti, A., Muller, M., Auweter, A., Guillen, C., Ott, M., Tafani, D. & Schulz, M. (2019). From facility to application sensor data: modular, continuous and holistic monitoring with DCDB. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. https://dl.acm.org/doi/10.1145/3295500.3356191

[2] BBC News (2020, May 18). Europe's supercomputers hijacked by attackers for crypto mining. Retrieved from www.bbc.com/news/technology-52709660

[3] Oliner, A., & Stearley, J. (2017). What Supercomputers Say: A Study of Five System Logs. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. https://ieeexplore.ieee.org/document/4273008

[4] Tuncer, O., Ates, E., Zhang, Y., Turk, A., Brandt, J., Leung, V. J., Egele, M. & Coskun, A. K (2019). Online Diagnosis of Performance Variation in HPC Systems Using Machine Learning. IEEE Transactions on Parallel and Distributed Systems. https://ieeexplore.ieee.org/document/8466019

[5] Baseman, E., & Lissa (2016). Interpretable Anomaly Detection for Monitoring of High Performance Computing Systems. www.semanticscholar.org/paper/Interpretable-Anomaly-Detection-for-Monitoring-of-Baseman-Lissa/7c74655b13e4977a5c4f4076e3cc6abb87db2a50

[6] Netti, A., Kiziltan, Z., Babaoglu, O., Sirbu, A., Bartolini, A. & Borghesi, A. (2019). A Machine Learning Approach to Online Fault Classification in HPC Systems. Distributed, Parallel, and Cluster Computing. www.arxiv.org/abs/2007.14241

[7] Borghesi, A., Libri, A., Benini, L., & Bartolini, A. (2019). Online Anomaly Detection in HPC Systems. IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS). https://ieeexplore.ieee.org/document/8771527

[8] Cineca. SuperComputing Applications and Innovation. Retrieved from www.hpc.cineca.it/content/about-us

[9] Cineca. D.A.V.I.D.E (Development of an Added Value Infrastructure Designed in Europe). Retrieved from www.hpc.cineca.it/hardware/davide

[10] Beneventi, F., Bartolini, A., Cavazzoni, C., & Benini, L. (2017). Continuous learning of HPC infrastructure models using big data analytics and in-memory processing tools. Design, Automation & Test in Europe Conference & Exhibition (DATE). https://ieeexplore.ieee.org/document/7927143

[11] Slurm Workload Manager Website (www.slurm.schedmd.com) contains detailed information about the Slurm resource manager.

[12] Apache Parquet Website (https://parquet.apache.org) contains detailed information about the Apache Parquet format.

[13] RackCDU Monitoring System User Guide. Retrieved from https://jp.fujitsu.com/platform/server/primergy/manual/peripdf/r11a-0551-01en.pdf

[14] Bcrypt File Encryption Utility Website (www.bcrypt.sourceforge.net) contains more information about the bcrypt library.

[15] SecLists 1000000 Most Commond Password. Retrieved from www.github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-1000000.txt.

[16] Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components. Journal of Educational Psychology. www.psycnet.apa.org/record/1934-00645-001

[17] scikit-learn Machine Learning in Python (www.scikit-learn.org) contains more information about the scikit-learn library.

[18] Yan, J., Mu, L., Wang, L., Ranjan, R. & Zomaya, A. Y. (2020). Temporal Convolutional Networks for the Advance Prediction of ENSO. Nature Scientific Reports. www.nature.com/articles/s41598-020-65070-5

[19] Qiu, Q., Xie, Z., Wu, L. & Li, W. (2018). DGeoSegmenter: A dictionary-based Chinese word segmenter for the geoscience domain. Comput. & geosciences. www.sciencedirect.com/science/article/pii/S0098300418300852?via%3Dihub

[20] Lea, C., Flynn, M. D., Vidal, R., Reiter, A. & Hager, G. D. (2017). Temporal convolutional networks for action segmentation and detection. In proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. www.arxiv.org/abs/1611.05267.

[21] Bai, S., Kolter, Z. J. & Koltun, V. (2018). An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling. www.arxiv.org/abs/1803.01271.

[22] Long, J., Shelhamer, E. & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. IEEE Conference on Computer Vision and Pattern Recognition (CVPR). https://ieeexplore.ieee.org/document/7298965.

[23] He, K., Zhang, X., Ren, S. & Sun, J. (2016). Deep Residual Learning for Image Recognition. IEEE Conference on Computer Vision and Pattern Recognition (CVPR). https://ieeexplore.ieee.org/document/7780459

[24] Chawla, N. V., Bowyer, K. W., Hall, L. O. & Kegelmeyer, P. W. (2002). SMOTE: synthetic minority over-sampling technique. Journal of Artificial Intelligence Research. https://dl.acm.org/doi/10.5555/1622407.1622416

[25] He, H., Bai, Y., Garcia, E. A. & Li, S. (2008). ADASYN: Adaptive synthetic sampling approach for imbalanced learning. IEEE International Joint Conference on Neural Networks. https://ieeexplore.ieee.org/document/4633969

[26] Das, B., Krishnan, N. C. & Cook, D. J. (2015). RACOG and wRACOG: Two Probabilistic Oversampling Techniques. IEEE Transactions on Knowledge and Data Engineering. https://ieeexplore.ieee.org/document/6816044

[27] Abdi, L. & Hashemi, S. (2016). To Combat Multi-Class Imbalanced Problems by Means of Over-Sampling Techniques. IEEE Transactions on Knowledge and Data Engineering. https://ieeexplore.ieee.org/document/7163639

[28] Cao, H., Li, X., Woon, Y. & Ng, S. (2011). SPO: Structure Preserving Oversampling for Imbalanced Time Series Classification. IEEE 11th International Conference on Data Mining. https://ieeexplore.ieee.org/document/6137306

[29] Zhu, T., Lin, Y. & Liu, Y. (2020). Oversampling for Imbalanced Time Series Data. www.arxiv.org/abs/2004.06373

[30] Muller, M. E. (1959). A note on a method for generating points uniformly on n-dimensional spheres. https://dl.acm.org/doi/10.1145/377939.377946

[31] Keras TCN Python Implementation by Philippe Remy. Retrieved from www.github.com/philipperemy/keras-tcn