Alma Mater Studiorum · University of Bologna

School of Science
Department of Physics and Astronomy "Augusto Righi"
Master Degree in Physics

Machine Learning "as a Service" for High Energy Physics (MLaaS4HEP): evolution of a framework for ML-based physics analyses

Supervisor:
Prof. Daniele Bonacorsi

Co-Supervisor:
Dr. Luca Giommi

Submitted by:
Mattia Paladino

Academic Year 2020/2021

To you, in 2000 years.

Contents

List of Figures vii

List of Tables x

1 High Energy Physics at LHC 4
- 1.1 Overview on LHC . . . 4
- 1.2 The LHC machine . . . 6
- 1.3 LHC experiments . . . 9
- 1.4 The CMS experiment . . . 14
  - 1.4.1 Structure of the detector . . . 15
- 1.5 The CMS Computing Model . . . 21
  - 1.5.1 The Worldwide LHC Computing Grid . . . 21
  - 1.5.2 The WLCG Tiers architecture . . . 22
  - 1.5.3 CMS data formats . . . 24
  - 1.5.4 Tasks of the Computing Model . . . 26
- 1.6 Towards High Luminosity LHC . . . 27
  - 1.6.1 Upgrades for Run 3 . . . 27
  - 1.6.2 Upgrades for HL-LHC . . . 28
  - 1.6.3 CMS Computing Model towards HL-LHC . . . 29

2 The Machine Learning Landscape 33
- 2.1 What is Machine Learning? . . . 33
  - 2.1.1 Terminology and Keywords . . . 34
- 2.2 Machine Learning Approaches . . . 35
- 2.3 Supervised Learning . . . 36
  - 2.3.1 Training, validation and test . . . 38
  - 2.3.2 Performance Metrics . . . 40
  - 2.3.3 Supervised Learning Algorithms . . . 43
- 2.4 Artificial Neural Networks . . . 44
  - 2.4.1 Training of a Neural Network . . . 46
- 2.5 Machine Learning Frameworks . . . 47
- 2.6 Machine Learning in HEP . . . 47
  - 2.6.1 Machine Learning Algorithms and Applications in HEP . . . 48
  - 2.6.2 Collaborating with other communities . . . 50
  - 2.6.3 Machine Learning software and tools in HEP . . . 51

2.6.4 Computing and Hardware Resources 51
2.7 Machine Learning in CMS 52

3 Machine Learning as a Service for High Energy Physics 54
3.1 Machine Learning as a Service 54
3.2 Machine Learning as a Service for HEP 55
3.3 MLaaS4HEP Architecture 55
3.3.1 Data Streaming Layer 57
3.3.2 Data Training Layer 58
3.3.3 MLaaS4HEP Training Workflow 59
3.3.4 Data Inference Layer 60
3.4 Validation and Performance 61
3.4.1 The $t\bar{t}H$ analysis 61
3.4.2 MLaaS4HEP Validation 62
3.4.3 MLaaS4HEP and TFaaS Performance 62

4 New features in MLaaS4HEP and the Higgs Boson ML challenge use case 65
4.1 Towards uproot4 65
4.1.1 Preprocessing operations 67
4.2 Improvements on the training method 72
4.3 Generalization to other Frameworks 74
4.3.1 MLaaS4HEP with Keras 74
4.3.2 PyTorch implementation 75
4.3.3 Scikit-Learn implementation 76
4.3.4 Providing useful metrics score 79
4.4 Validation of new features 80
4.5 MLaaS4HEP application on the Higgs Boson ML challenge 84
4.5.1 Challenge description 84
4.5.2 Datasets 85
4.5.3 Strategy 86
4.5.4 Results 90
4.6 Future directions 91

A Models definition 95
A.1 Keras NN 95
A.2 PyTorch NN 96
A.3 Scikit-Learn 97
A.4 Challenge models 98

B ROOT vs uproot cuts 100

Bibliography 112

List of Figures

1.1 The LHC tunnel. 4
1.2 The CERN accelerator complex. 5
1.3 Schematic Layout of LHC. 6
1.4 LHC beam structure. 8
1.5 The four main LHC detectors. 10
1.6 The ATLAS detector. 11
1.7 The CMS detector. 11
1.8 Schematic view of the LHCb detector. 12
1.9 ALICE detector. 13
1.10 Coordinate system adopted by CMS. 15
1.11 Schematic view of the CMS section with interacting particles. 16
1.12 Paths of different particles crossing the CMS detector. 19
1.13 Barrel transverse view of the CMS detector. 20
1.14 Schematical view of the WLCG tiered architecture. 22
1.15 The CERN Data Centre. 23
1.16 Description of the CMS data tiers. 25
1.17 Evolution of data formats and their respective sizes. 26
1.18 Plan of HL-LHC Project. 28
1.19 Schematic description of CMS upgrades preparing for Run 3 [52]. 29
1.20 Schematic description of the CMS planned upgrades to be prepared for HL-LHC phase [52]. 30
1.21 Estimated evolution of CPU (top), disk (middle) and tape (bottom) resources needs in function of time through Run 4. 31

2.1 Representation of k-fold CV process with $k = 5$. 39
2.2 Content of a 2×2 Confusion Matrix. 41
2.3 ROC Curve obtained varying the classification thresholds. 43
2.4 Examples of different NN architectures. 45
2.5 Representation of a single perceptron. 45
2.6 Example of a MLP NN composed by... 46
2.7 Fraction of (mainly HEP) physics paper involving ML classification **(a)** and regression **(b)** techniques [85]. 48

3.1 MLaaS4HEP architecture diagram representing the Data Streaming, Data Training and Data Inference layers. 56

3.2 Output representation of the Python Generator. 57
3.3 Representation of a Jagged Array vector with padding values. 58
3.4 Vector representation of Jagged array with the corresponding mask array. 59
3.5 Schematic representation of the phases in the MLaaS4HEP Training Workflow. 60
3.6 Feynman diagram for the $t\bar{t}H(b\bar{b})$ decay. 62

4.1 Example of a `preproc.json` file provided a by the user to MLaaS4HEP to apply preprocessing operations on ROOT file branches. 68
4.2 AUC and loss metrics comparison between original and standard training approaches using an SGD optimizer with learning rate equal to 0.01. The plots in **(a)** and **(b)** were obtained setting the chunk size equal to 100, while in **(c)** and **(d)** the chunk size has been set to 100k. 73
4.3 Timing comparison between original and standard training approaches considering 100 and 100k as chunk size. **(a)** refers to the case where the chunk size has been set to 100 and **(b)** refers to a chunk size of 100k. . . 74
4.4 Comparison of the metrics score (loss **(a)**, Accuracy **(b)** and AUC **(c)**) for training, validation and test set using all the events in the files, read with a chunk size of 10k events. 75
4.5 Trend of AUC **(a)** and loss **(b)** scores as a function of the number of chunks considering the default PyTorch training function. 77
4.6 Example of Classification Report. In addition to precision, recall, F1 and support, it also provides reported averages, which include macro average (averaging the unweighted mean per label), weighted average (averaging the support-weighted mean per label), and, only in the multilabel classification case, sample average. 80
4.7 Comparison of the metrics score (loss **(a)**, Accuracy **(b)** and AUC **(c)**) obtained in the training, validation and test set. To test the correct transition of MLaaS4HEP to uproot4, two different approaches were considered: (1) using MLaaS4HEP to read and normalize events, and to train the ML model; (2) using a Jupyter Notebook to perform the entire pipeline without using MLaaS4HEP. 81
4.8 Comparison of the metrics score (loss **(a)**, Accuracy **(b)** and AUC **(c)**) obtained in the training, validation and test set. To test the correct implementation of the MLaaS4HEP preprocessing functions, two different approaches were considered: for two different approaches: (1) using MLaaS4HEP to read and normalize events applying cuts on branches, and to train the ML model; (2) using a Jupyter Notebook to perform the entire pipeline without using MLaaS4HEP. 83
4.9 Correlation Matrix of the features in the training dataset. 87
4.10 Feature importance obtained using XGBoost Classifier. 88
4.11 Histogram representation of three feature distributions. In **(a)**, **(b)**, **(c)** is shown the distribution of a feature selected for the model training, a feature to be log-transformed and a feature selected to be removed respectively. 89

List of Tables

1.1 LHC main technical parameters 9

3.1 Mean Values of event throughput and total time spent using all the events (28.5M) for the specs computing phase and for the chunks creation in the training phase. Values are reported as intervals, where the upper limit for the event throughput and the lower limit for the total time are reached using local files, while the opposite limits have been obtained using remote ROOT files. 63

4.1 Measured time and throughput values for the main steps of the MLaaS4HEP workflow considering the two uproot versions. The results were obtained by running MLaaS4HEP 5 times with a chunk size equal to 100k and a total number of events to be read equal to 500k and averaging the obtained times and throughputs. 67

4.2 MLaaS4HEP event throughput for five different cases considering the application of preprocessing operations. 70

4.3 MLaaS4HEP preprocessing timing values for five different cases, i.e. no cut application, application on a flat branch, application on a jagged branch, definition of a new branch and application of cuts both on flat, jagged and new branch. 71

4.4 AUC scores related to different Scikit-Learn models: SGDClassifier, MLPClassifier and AdaBoost. 78

4.5 Accuracy and AUC scores on training, validation and test set for the two approaches using AdaBoost. 82

Introduction

The scientific success of the LHC experiments at CERN highly depends on the availability of computing resources which efficiently store, process, and analyse the amount of data collected every year. This is ensured by the Worldwide LHC Computing Grid infrastructure that connect computing centres distributed all over the world with high performance network. LHC has an ambitious experimental program for the coming years, which includes large investments and improvements both for the hardware of the detectors and for the software and computing systems, in order to deal with the huge increase in the event rate expected from the High Luminosity LHC (HL-LHC) phase and consequently with the huge amount of data that will be produced.

Since few years the role of Artificial Intelligence has become relevant in the High Energy Physics (HEP) world. Machine Learning (ML) and Deep Learning algorithms have been successfully used in many areas of HEP, like online and offline reconstruction programs, detector simulation, object reconstruction, identification, Monte Carlo generation, and surely they will be crucial in the HL-LHC phase.

This thesis aims at contributing to a CMS R&D project, regarding a ML "as a Service" solution for HEP needs (MLaaS4HEP). It consists in a data-service able to perform an entire ML pipeline (in terms of reading data, processing data, training ML models, serving predictions) in a completely model-agnostic fashion, directly using ROOT files of arbitrary size from local or distributed data sources. This framework has been updated adding new features in the data preprocessing phase, allowing more flexibility to the user. Since the MLaaS4HEP framework is experiment agnostic, the ATLAS Higgs Boson ML challenge has been chosen as physics use case, with the aim to test MLaaS4HEP and the contribution done with this work.

Chapter 1 presents an overview on the LHC accelerator and its experiment experiments, focusing particularly on CMS and its computing model.

Chapter 2 describes ML concepts and terminology, with an in-depth study on the Supervised Learning. It also offers a description about ML applications in the HEP context and in the CMS experiment.

Chapter 3 introduces the MLaaS4HEP project and provides a general description of its architecture and performance.

Chapter 4 presents the original results of this project: throughout this last chap-

ter, the new features implemented within MLaaS4HEP will be described and analyzed, followed by a description of its application to a physics use case: the Higgs Boson ML challenge hosted by Kaggle in 2014.

Chapter 1

High Energy Physics at LHC

1.1 Overview on LHC

The Large Hadron Collider (LHC) [1] [2] [3] is a two-ring-superconducting-hadron accelerator and collider operating at CERN since 2010 (Fig. 1.1). It is located beneath the Franco-Swiss border near Geneva, placed in the existing tunnel that was constructed between 1984 and 1989 for the CERN Large Electron-Positron (LEP) machine. The purpose of the LHC project is to give scientists an experimental apparatus that would enable to advance the knowledge in the high energy physics (HEP) field, e.g. looking for theoretically predicted particles, test the validity of theories beyond the Standard Model (SM), search for particles that would indicate the existence of dark matter, etc.

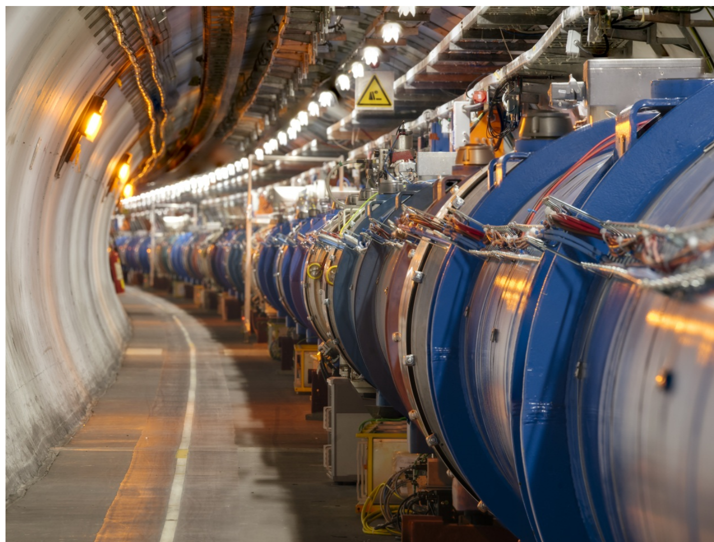

Figure 1.1: The LHC tunnel.

The LHC project [4] was approved by the CERN Council in December 1994. Because of financial reasons, the initial plan was to set up the project in two stages. However, after intense negotiations which ensured significant contributions from the non-member states, in 1996 the CERN Council approved the construction of the machine in a single

step. The decision to build LHC at CERN was strongly influenced by the cost saving to be made by re-using the LEP tunnel and its injection chain. The tunnel geometry, originally designed for the LEP machine, was composed by eight crossing points flanked by long straight sections for RF cavities that compensated the high synchrotron radiation losses.

LHC consists of a 27 km long circular ring, designed to accelerate protons and heavy ions at high energies. LHC is characterised by two accelerated beams that travel in opposite directions inside different parts of the same pipe at ultrahigh vacuum. In order to make the collision possible, bending is needed: it is achieved by using compact twin-bore superconducting magnets, cooled at 2.1 K with superfluid helium, yielding a magnetic field of 8.3 T.
The beams collide in four points of the two rings where the main LHC experiments are located.

The LHC is the last stage of a complex acceleration system, and a process of pre-acceleration is needed before the beams are injected in the machine: a complex chain of smaller particle accelerators is used to boost the particles to their final energies and provide beams to a whole set of smaller experiments. The CERN accelerator complex (shown in Fig. 1.2) comprises the Linear Accelerator (Linac), the Proton Synchrotron Booster (PSB) , the Proton Synchrotron (PS) and the Super Proton Synchrotron (SPS).

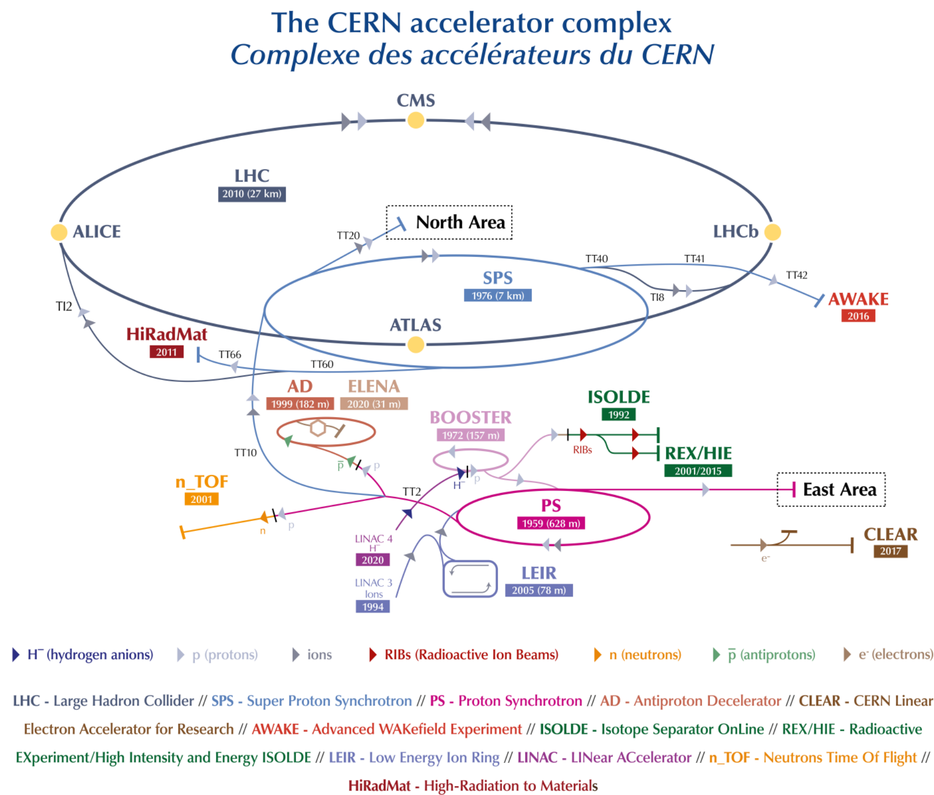

Figure 1.2: The CERN accelerator complex.

Initially, hydrogen atoms are ionized in order to produce protons and then injected in a linear accelerator called LINAC 2, which transfers the first amount of energy to protons, which reach an energy of 50 MeV. In the following step, protons enter another component, the PSB, where are accelerated up to an energy of 1.4 GeV. Then they enter the PS, where 277 electromagnets push the protons to 99,9 the speed of light with an energy of 25 GeV. After that, proton bunches are accelerated in the SPS, a circular particle accelerator with a circumference of 7 km, and reach an energy of 450 GeV. Finally, they are injected into LHC in two separate pipes in which they move in opposite directions: here, particles are accelerated up to their maximum designed energy i.e. 6-7 TeV during Run 1 and 13 TeV in Run 2. The two pipes intersect in four caverns, where four detectors are located to reveal the products of the collisions: ATLAS [5] [6], CMS [7] [8], LHCb [9] [10] and ALICE [11] [12].

1.2 The LHC machine

The shape of the LHC collider [13] is organized in eight 2.45 km long arcs, each one containing 154 dipole bending magnets, and eight straight sections approximately 528 m long with, at its end, a transition region called *insertion*, see Fig.1.3.

The layout of each straight sections depends on the specific use of the insertion, which can be oriented to the beam collisions within an experiment, the beam dumping, the injection or the beam cleaning. The part of the machine between two insertion points is defined as *sector*.

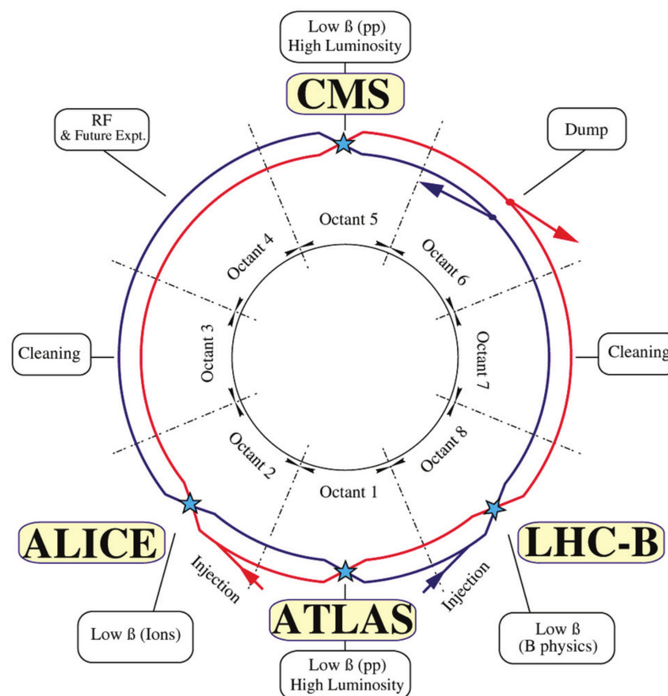

Figure 1.3: Schematic Layout of LHC.

LHC relies on three different elements: vaacum system, magnet system and cavities.

Vaacum System

To ensure that particles do not lose energy in the acceleration process, the vaacum system is necessary. The LHC vacuum system is composed by the beam vacuum, the insulation vacuum for cryomagnets and the insulation vacuum for helium distribution: the presence of three separate systems with different requirements represents a special feature for LHC.

The beam vaacum is an ultrahigh vacuum with a pressure of 10^{-7} Pa at a cryogenic temperature of 5 K and, close to the interaction points, the pressure is lower than 10^{-9} Pa. In this way, it avoids collisions with gas molecules allowing proper beam lifetime and low background to the experiments.

The insulation vacuum for the superconducting magnets is cooled down with liquid helium to 1.9 K: it ensures a good thermal insulation of the cooling system and so it allows to maintain the low temperatures.

Then, the insulation vacuum for the helium distribution line acts similarly to the previous one in order to reduce the amount of heat coming from the surrounding room-temperature environment into the cryogenic parts.

Magnet System

A strong magnetic field is required in order to maintain particles on a circular trajectory. In the LHC, there are different magnets and each type contributes to optimizing the particles trajectory.

- Dipoles are the most numerous (1232) and provide the 8.3 T field thanks to the high current which flows in them. Dipoles task is to maintain the beams in their circular orbit.
- Quadrupoles (392) are used to focus the beam down to the smallest possible size at the collision points in order to maximize the chance of two protons colliding head-on.
- High order multipole magnets contribute to correct for possible imperfections of the magnetic field in the main ring magnets.

LHC dipoles represent one of the most relevant technological challenge in the LHC design. In a proton accelerator like LHC, the maximum energy that can be achieved, given the acceleration circumference, is directly proportional to the strength of the dipole field. Here, the dipole magnets are superconducting electromagnets which use niobium-titanium (NbTi) cables with an operating temperature of 1.9 K and makes the LHC unique among superconducting synchrotrons.

Cavities

Along LHC length there are radiofrequency (RF) cavities, metallic chambers that contain an electromagnetic field and operate at 4.5 K arranged in groups of four in cryomodules with two cryomodules per beam. The 16 RF cavities on the LHC are housed in four cylindrical refrigerators called “cryomodules” which keep the RF cavities working in a superconducting state, without losing energy to electrical resistance. The RF fields are exploited to accelerate beams and to squeeze proton bunches as compact as possible in order to make head-on collisions easier and to guarantee high luminosity at the collision points. To enable an RF cavity to accelerate particles, an RF power generator supplies an electromagnetic field: charged particles passing through the cavity experience the overall force and direction of the resulting electromagnetic field, which transfers energy to push them forward along the accelerator. The field in an RF cavity is made to oscillate at a given frequency which, in the LHC case, is tuned at 400 MHz. So, the ideally timed proton, with exactly the right energy, will see zero accelerating voltage when the LHC is at full energy, while protons with slightly different energies, arriving earlier or later, will be accelerated or decelerated in order to get close to the energy of the ideal particle: particle beam is sorted into discrete packets called “bunches” (shown in Fig.1.4).

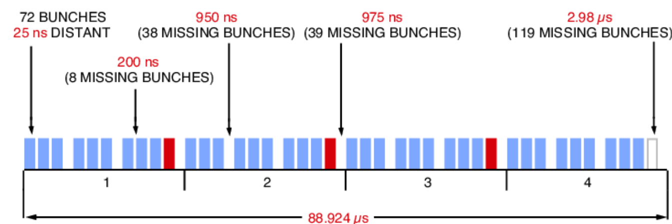

Figure 1.4: LHC beam structure.

In the LHC, under nominal operating conditions, each proton beam consists of 2808 bunches and each bunch contains as many as 1011 protons. Bunches of particles measure a few centimeters long and a millimeter wide when they are far from a collision point. However, the bunch size is not constant around the ring: circulating around the ring, each bunch gets squeezed and expanded. By squeezing its length as much as possible around the collision points, the bunches are compressed to about 16 nm, increasing the probability of a p-p collision.

Technical Parameters

LHC aims to produce and observe rare events coming from physics beyond the SM. In order to do that, the collision rate $R = \sigma \cdot \mathcal{L}$ has to be high, where σ is the cross section of the process and $\mathcal{L}$ is the instantaneous luminosity: a high instantaneous luminosity is crucial for the success of the experimental program. The instantaneous luminosity is a decisive parameter which depends on the properties of the beams and is defined as:

$$\mathcal{L} = \frac{f_{rev} n_b {N_b}^2}{4\pi\epsilon_n\beta^*}H. \tag{1.1}$$

where in the numerator n_b and N_b indicate, respectively, the number of bunches per beam and the number of particles in each bunch and f_{rev} is the revolution frequency, while, in the denominator, η_n is the normalized transverse beam emittance and β^* represents the focal length at the collision point, also called β function. The last term H is the hourglass factor, which take into account the geometric reduction of the instantaneous luminosity.

ATLAS and CMS experiments have a high designed luminosity $\mathcal{L} = 10^{34}$ cm^{-2} s^{-1}, while LHCb and ALICE are designed for a lower luminosity of $\mathcal{L} = 10^{27}$ cm^{-2} s^{-1} and $\mathcal{L} = 10^{32}$ cm^{-2} s^{-1} respectively. In Table 1.1, some of the most relevant LHC parameters are listed.

Quantity (measurement unit)	Value
Circumference (m)	26659
Magnets working temperature (K)	1.9
Number of magnets	9593
Number of dipoles	1232
Number of quadrupoles	392
Number of radiofrequency cavities per beam	16
Protons nominal energy (TeV)	6.5
Ions nominal energy (TeV/Nucleon)	2.76
Magnetic field maximum intensity (T)	8.33
Project luminosity (cm$^-2$ s^1)	$2.06 \cdot 10^{34}$
Number of proton bunches per beam	2808
Number of protons per bunch	$1.1 \cdot 10^{11}$
Minimum distance between bunches (m)	~ 7
Number of revolutions per second	11245
Number of collisions per second (millions)	600

Table 1.1: LHC main technical parameters

1.3 LHC experiments

After the acceleration phase, the beams along the ring collide at four interaction points, where four main detectors are installed: ATLAS, CMS, LHCb and ALICE (Fig. 1.5). These experiments, characterized by different detectors and subdetectors, are run by large collaborations of scientists from institutes all over the world.

ATLAS

A Toroidal LHC ApparatuS (ATLAS) is the other general-purpose detector at LHC, shown in Fig. 1.6. Its main purpose is to investigate new physics exploiting the extremely high energy reached by LHC machine. With its $7 \cdot 10^4$ tonnes, ATLAS has the dimension of a $46 \times 25 \times 25$ m^3 cylinder that make it the largest volume particle detector ever

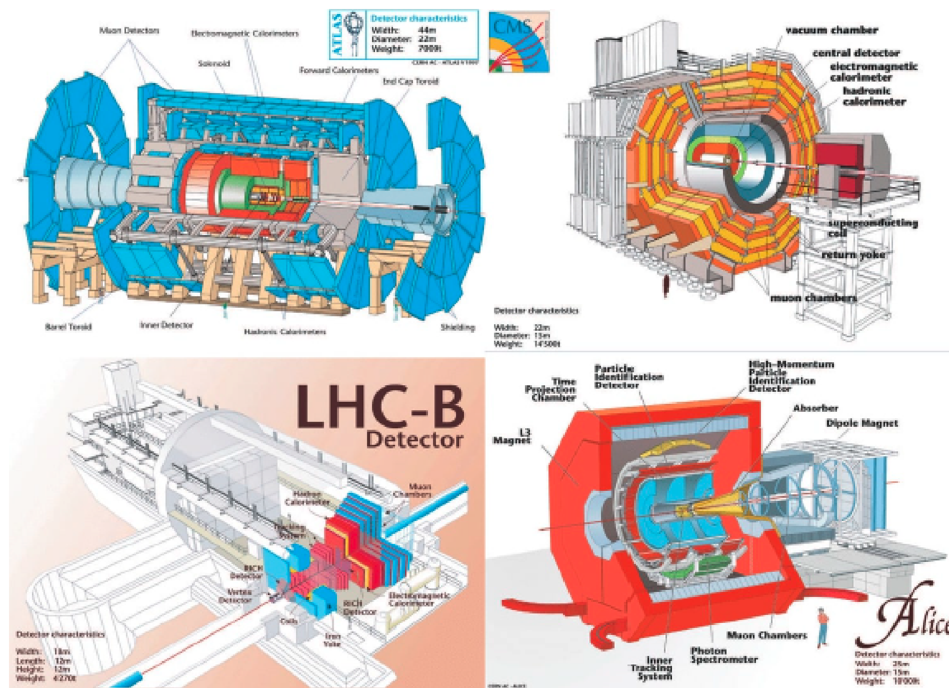

Figure 1.5: The four main LHC detectors.

constructed. The detector is forward-backward symmetric with respect to the interaction point and is composed by four main layers:

- magnet system, which bends the charged particles.
- inner Detector, which measures the path done by charged particles.
- calorimeters, which identify particles.
- muon spectrometer, which recognized muons presence.

The magnet configuration comprises a thin superconducting solenoid surrounding the inner-detector cavity which provides a 2 T magnetic field, and three large superconducting toroids which produce a toroidal magnetic field of approximately 0.5 T and 1 T for the muon detectors in the central and end-cap regions, respectively. Outside the solenoid, the hadronic calorimeter is installed: it is a sampling calorimeter, in which steel is used as an absorber and scintillating tiles have been chosen as active material. Although it has the same scientific goals as the CMS experiment, it uses different technical solutions in some subsystems and a different magnet-system design.

CMS

The Compact Muon Solenoid (CMS) detector, shown in Fig. 1.7, is a multi-purpose apparatus designed to observe any new physics phenomena that the LHC might reveal. CMS physics program is quite extensive and ranges from studying the SM to the research of dark matter. The CMS detector is built around a cylindrical solenoid magnet which

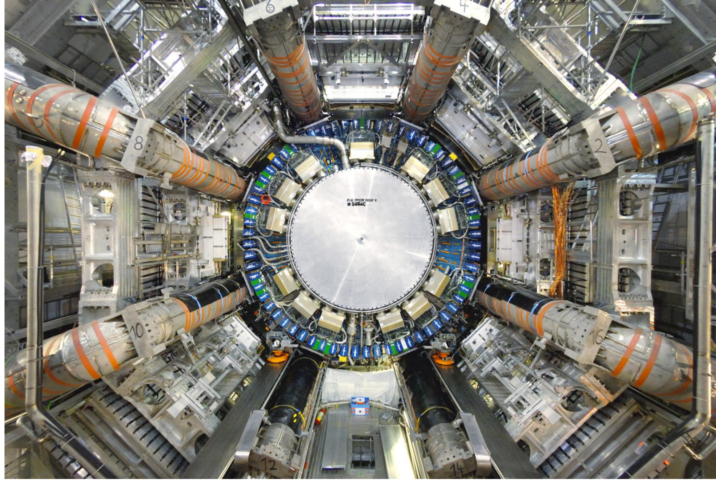

Figure 1.6: The ATLAS detector.

produces a 3.8 T magnetic field, about 10^5 times the magnetic field of the Earth. The field is confined by a steel "yoke" that forms the bulk of the detector's $1.4 \cdot 10^4$ tonne weight. The description of the apparatus in further details will be covered in the next section (Section 1.4).

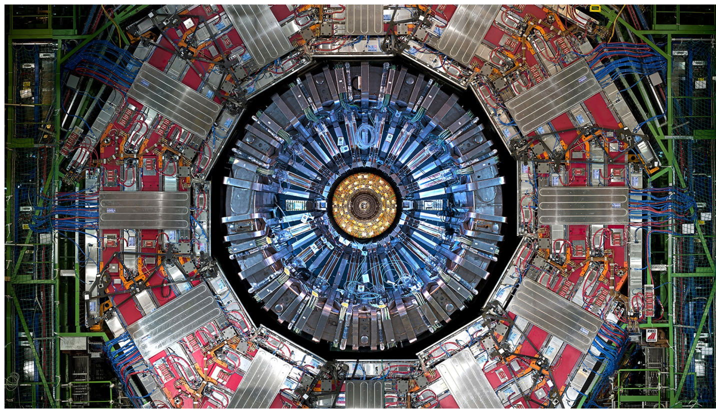

Figure 1.7: The CMS detector.

LHCb

The Large Hadron Collider beauty (LHCb) is an experiment specialized in investigating differences between matter and antimatter by studying the "b" quark and its primary goal is to look for indirect evidence of new physics in CP violation and rare decays of beauty and charm hadrons like B_d, B_s and D mesons. Unlike the other experiments, LHCb does not surround the collision point with an enclosed detector but uses a series of sub-detectors to mainly detect forward particles. The $5.6{\cdot}10^3$ tons LHCb detector is made up of a forward spectrometer and planar detectors, with total dimensions $21 \times 10 \times 13$ m^3 and placed 100 m below the ground. The choice of the detector geometry is justified by the fact that both the b- and $\overline{\text{b}}$-hadrons are predominantly produced in the same

forward or backward cone at high energies. Starting from the interaction point and moving outwards, LHCb presents, as shown in Fig. 1.8, the following subdetectors: the vertex locator (called VELO), an aerogel Ring Imaging Cherenkov, a silicon Trigger Tracker, the magnet, three Tracking stations, another aerogel Ring Imaging Cherenkov, a scintillator Pad Detector and Preshower, two calorimeters (first an electromagnetic and then an hadronic one) and, in the end, a series of Muon chambers.

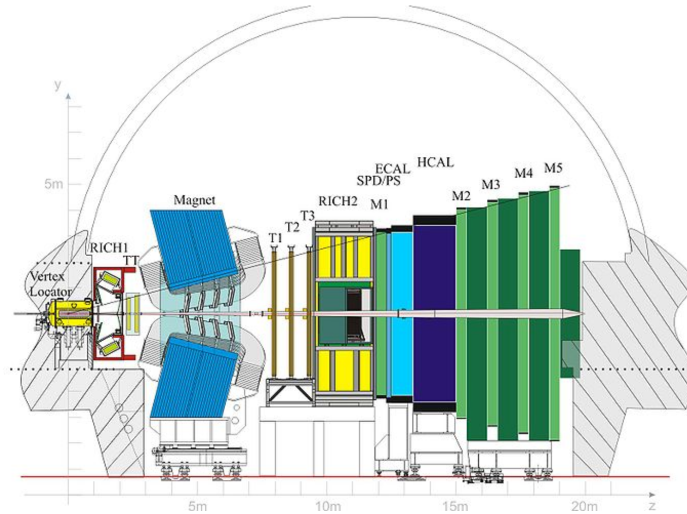

Figure 1.8: Schematic view of the LHCb detector.

ALICE

ALICE (A Large Ion Collider Experiment), shown in Fig. 1.9, is a detector dedicated to heavy-ion physics, designed to study the physics of strongly interacting matter at extreme energy densities, where a phase of matter called quark-gluon plasma forms. LHC is able to recreate conditions similar to those just after the Big Bang through collisions between lead ions, which generate temperatures more than 10^5 times hotter than the centre of the Sun. Under these extreme conditions, protons and neutrons "melt", freeing the quarks from their bonds with the gluons: this state is the so-called quark-gluon plasma. The existence of such a phase and its properties are key issues for understanding the phenomenon of confinement in the theory of quantum chromodynamics (QCD) and for the so-called chiral-symmetry restoration problem. ALICE studies the quark-gluon plasma state as it expands and cools, observing how it progressively gives rise to the particles that constitute the matter of our universe today. In order to do that, ALICE collaborations uses a detector with dimensions are $16 \times 16 \times 26$ m^3 with a total weight of approximately 10^4 tons. The experiment consists of a central barrel part, used to perform measurements on photons, electrons and hadrons, and a forward muon spectrometer. ALICE is constituted of several subdetectors: moving from the beam pipe outwards, the barrel contains an Inner Tracking System, composed of six planes of silicon pixel, drift and strip detectors, a cylindrical Time Projection Chamber, three particle identification arrays of Time-of-Flight, a Ring Imaging Cherenkov and Transition Radiation detectors

and two electromagnetic calorimeters. The barrel is embedded in a large solenoid magnet capable of a magnetic field up to 0.5 T.

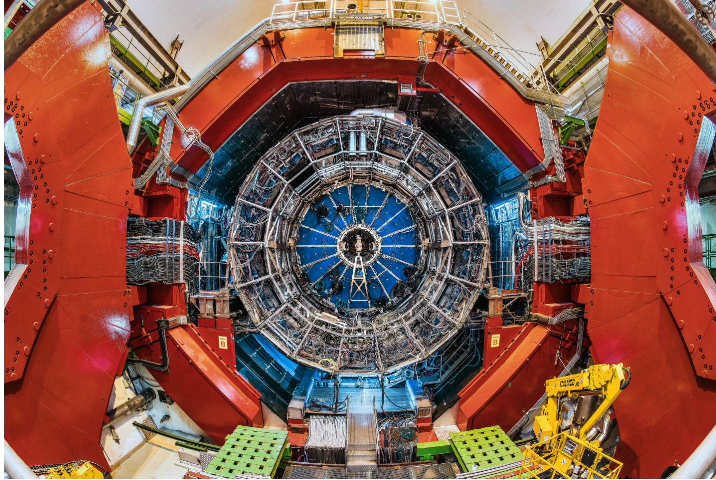

Figure 1.9: ALICE detector.

Other Experiments

The four experiment previously described are the main ones at LHC, but there are also other smaller experiments that can be listed. One of them is TOTEM [14], designed to measure p-p total elastic and diffractive cross section with the luminosity independent method and study elastic and diffractive scattering at the LHC by measuring protons emerging at small angles with respect to the beam line. TOTEM detectors are spread across half a kilometre around the CMS interaction point in order to track emerging particles and perform measurements on scattered protons.

Another smaller experiment is Large Hadron Collider forward (LHCf) [15], which uses particles thrown forward by collisions in LHC as a source in order to simulate cosmic rays in laboratory conditions. LHCf is made up of two detectors which sit along the LHC beamline, at 140 m from either side of the ATLAS collision point. The location allows the observation of particles at nearly zero degrees to the proton beam direction. Each of the two detector has small weight and dimensions (40 kg and $30 \times 80 \times 10$ cm^3 wide)

Then, there is also the Monopole and Exotics Detector at the LHC (MOEDAL) experiment [16], which main purpose is to search directly for the magnetic monopole. This detector aims to search for this hypothetical particle with magnetic charge and the detector is an array of 400 modules that cover a total area of 250 m^2. MOEDAL is also involved in the investigation for highly ionizing Stable Massive Particles (SMPs), predicted by theories beyond the SM.

For the next data-taking, two new experiments have been installed: FASER [17] and SND [18].

FASER (Forward Search Experiment) is designed to search for light and extremely weakly interacting particles which existence is predicted by many models beyond the

SM that attempt to explain the nature of dark matter, the origin of neutrino masses, and the imbalance between matter and antimatter in the present-day universe. FASER is being installed during the Long Shutdown 2 of CERN's accelerator complex and will start taking data during Run 3 of the LHC.

The SND experimet (Scatterin and Neutrino Detector) has been approved in march 2021 by the CERN Research Board and it is designed to detect and study neutrinos, the most enigmatic fundamental particles in the universe. SND will perform measurements on neutrinos produced at LHC and could open new frontier in the neutrino physics field.

1.4 The CMS experiment

In this section, a more detailed description of the CMS experiment and its subdetectors will be provided.

As mentioned before, CMS is a general purpose detector [8], composed by a cylindrical barrel and two endcaps, built around a huge superconducting solenoid and designed to investigate a wide range of physics. It is placed about 100 m underground, near the French village of Cessy: CMS is a 21.6×14.6 m^2 cylinder made of concentric layers, each layer corresponding to a different subdetector or piece of equipment that is designed to stop, track, measure or identify the particles coming from the p-p and heavy-ion collisions. According to the experiment's goals, the main design considerations were chosen in order to identify and measure muons, photons, electrons and hadrons with high precision. For this, CMS needs a highly performant muon system, a high quality electromagnetic calorimeter, a robust and powerful central tracking system to achieve excellent charged particle reconstruction and detailed vertex reconstruction, a hadron calorimeter and a strong magnetic field. Thanks to these features, the CMS experiment provided the most precise Higgs boson (H) mass measurement up to now. Therefore, the detector is composed by six concentric layers: starting from the collision point outwards there is the silicon Tracker, the Electromagnetic Calorimeter, the Hadronic Calorimeter, the Magnet, and, lastly, the Muon detector.

In order to represent the events recorded by the experiment and their location, it is necessary to define a coordinate system. The cylindrical coordinate system adopted by CMS has the origin centered at the nominal collision point inside the experiment: the x-axis points to the center of the accelerator ring, the y-axis points upwards and the z-axis is parallel to the beam pipe and the solenoid magnetic field. Besides these coordinates, the polar angle θ (angle between direction of flight and the beam pipe), measured from the z-axis, and the azimuthal angle ϕ (angle around beam axis), measured from the x-axis and the radial coordinate r, can be defined. The visual representation of the coordinate system is reported in Fig. 1.10

From the definition of the previous coordinate system, it is possible to introduce a series of quantities. Instead of the polar angle, it is convenient to construct a quantity with better transformation property under Lorentz boost: this quantity is called rapidity and is defined as:

$$y = \frac{1}{2} \log \frac{E + p_z}{E - p_z}. \tag{1.2}$$

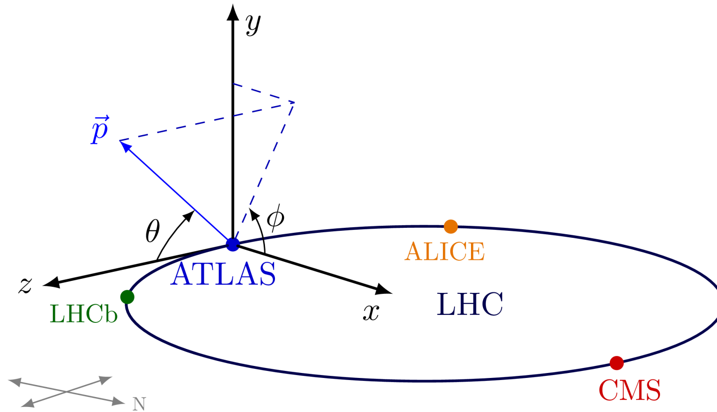

Figure 1.10: Coordinate system adopted by CMS.

However, this quantity does not provide an intuitive interpretation, so the pseudo rapidity η is introduced:

$$\eta = -\ln\left(\tan\frac{\theta}{2}\right). \tag{1.3}$$

Other useful quantities to describe events generated in collisions are the energy and momentum measured in the transverse plane (x-y plane) with respect to the beam direction, denoted as E_T and p_T. The transverse energy and momentum are defined in the following way:

$$E_T = E\sin\theta, \tag{1.4}$$

$$p_T = \sqrt{p_x^2 + p_y^2} \tag{1.5}$$

In the end, to describe particles in terms of pseudorapidity η and azimuthal angle ϕ, another quantity, ΔR, is defined:

$$\Delta R = \sqrt{\Delta\eta^2 + \Delta\phi^2} \tag{1.6}$$

This quantity is given by $\Delta\eta$ and $\Delta\phi$, which represent the spatial separation of two object.

1.4.1 Structure of the detector

The CMS apparatus is instrumented with different components which will be described starting from the inner layers (schematic view of CMS in Fig. 1.11).

Silicon Tracker

The CMS tracking system [19] is designed to efficiently and precisely measure the trajectories, and so the momenta, of charged particles coming from the LHC collision, and thus precisely reconstruct primary and secondary vertices. The Tracker is the innermost

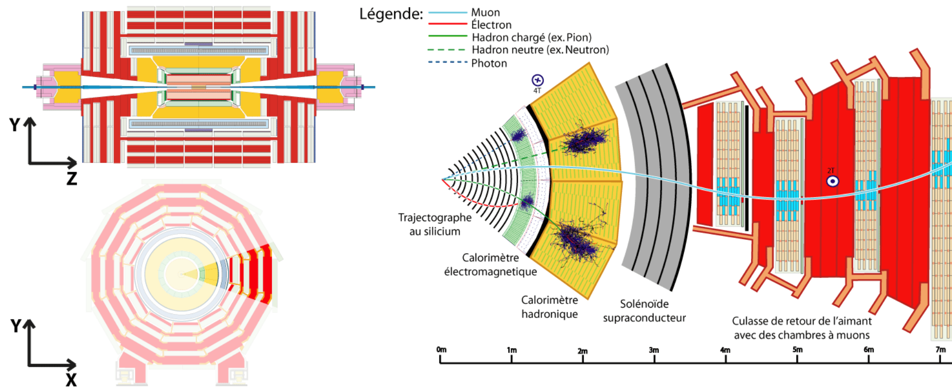

Figure 1.11: Schematic view of the CMS section with interacting particles.

detector part of the experiment, it surrounds the interaction point with a length of 5.8 m and a diameter of 2.5 m, and so it receives the highest flux of particles. For this reason the construction materials are carefully chosen in order to be radiation resistant and with a low degree of interference with particles. On the other hand, the detector has high granularity and fast response in order to keep up with the rate of particles crossing the Tracker. The previous requirements lead to a design entirely based on silicon detector technology.

The inner tracking system, entirely made of silicon (Si) is based on pixel and fine-grained microstrip detectors. At the end of 2016, the readout chips of the CMS tracker were found to be severely damaged by the radiation; thus, during the shutdown, in 2017 the system has been fully replaced. The tracker is currently composed of a pixel detector with four barrel layers at radii between 4.4 cm and 10.2 cm and a silicon strip tracker with 10 barrel detection layers extending outwards to a radius of 1.1m. Each system is completed by endcaps which consist of 3 disks in the pixel detector and 3 plus 9 disks in the strip tracker on each side of the barrel. Pixel detectors are placed in the inner region below 20 cm, close to the interaction point, to allow the measurement of the impact parameter of charged particle tracks, as well as the position of secondary vertices. Furthermore, the pixel detector is able to distinguish and to reconstruct the tracks: each one contains 65 million pixels which allow to perform measurements on particles path with very high precision. When a charged particle passes through, it gives enough energy to electrons to be ejected from the silicon atoms, creating electron-hole pairs. Each pixel uses electric current to collect these charges on the surface resulting in a small electric signal. Exceeded the pixels, particles pass through ten layers of silicon strip detectors; this part contains 15200 highly sensitive modules with a total of 10 million detector strips read by 80000 microelectronic chips.

In order to evaluate particles momentum, a magnetic field that bends charged particles is needed: the CMS solenoid provides a homogenous magnetic field of 4 T over

the full tracker volume. Therefore, by using the signal coming from the tracker when particles pass, is possible to reconstruct the bent path and so the momentum of the particle. The Tracker is able to reconstruct the path of hadrons, electrons and muons as well as tracks produced by short-lived particles decay with high efficiency.

Electronmagnetic calorimeter

The Electromagnetic Calorimeter (ECAL) [20] of CMS is a homogenous calorimeter which is used for the identification of photons and electrons and for the precise measurement of the energy of incident particles. It is made of 61200 lead tungstate crystal ($PbWO_4$) mounted in the central barrel part, closed by 7324 crystals in each of the two endcaps. The ECAL also contains preshower detectors which allow to distinguish between pairs of low-energy photons and a single high-energy photon.

The high density ($8.28g/cm^3$), the short radiation length (0.89cm), the small Moliere radius (2.2 cm) and fast light emission are some of the properties that characterise the $PbWO_4$ crystals, making them the most appropriate choice since they are able to contain the electromagnetic showers in an optimal way, ensuring an excellent energy resolution and a compact calorimeter. Also, they are highly transparent and scintillate when photon and electron pass through them: . The energy measurement is based on the conversion of the incident particles to an electromagnetic shower, that interacts with the ECAL material producing scintillation light: the crystals produce a light signal which is proportional to the energy of the particle. The main drawback of $PbWO_4$ is the low light output emission and, for this reason, the ECAL needs very large photodiodes. The photodetectors, designed to work in the strong magnetic field, are glued onto the back of each crystal to detect the scintillation light and convert it to an electrical signal that is amplified and sent out for further processing and analysis.

The energy resolution of an electromagnetic calorimeter is defined by the following formula:

$$\left(\frac{\sigma}{E}\right)^2 = \left(\frac{a}{\sqrt{E}}\right)^2 \oplus \left(\frac{n}{E}\right)^2 \oplus c^2 \tag{1.7}$$

where:

- E is the particle energy in GeV;
- a is a stochastic term which considers a photostatistics contribution and fuctuations on the lateral containment of the shower or on deposits of energy;
- n is related to the noise, which include contributions from electric noise (dominant at low-energy)and energy pile-up (dominant at high-luminosity)
- c is a constant term reflecting energy leakage of the crystals, non-uniformity of the longitudinal light collection and calibration errors.

The constants for the energy resolution having electrons in beam tests are: $a = 2.8\%$, $n = 12\%$ and $c = 0.3\%$.

Hadronic calorimeter

The energy measurement for hadrons, particles made of quarks and gluons, is carried out by the Hadron Calorimeter (HCAL) [21], which is specifically built to measure particles interacting strongly. Hadrons detection and identification is fundamental for the experiment; in fact, from their decay, the production of new particles may occur and, in order to detect them, the HCAL must be able to capture, as far as possible, every particle produced from collisions. The HCAL also provides tools for an indirect measurement of neutrinos, produced in electroweak interactions. Since HCAL is built to encapsulate as hermetically as possible the interaction point, it allows to detect "invisible" particles, such as neutrinos, by looking for missing energy and momentum from the reconstruction of the collision event.

The HCAL is a sampling calorimeter and so it is composed by a series of alternating passive and active material layers. The role of the passive layer, also called "absorber" is to make the particles loose energy, while the active material (fluorescent scintillator) records snapshots of the shower development caused by the hadrons traversing the calorimeter. The light emitted by the scintillators is then absorbed by special optic fibres with a diameter of 1 mm and feed it into readout boxes where the signal is amplified and converted into digital data. Then, by summing up the amount of light over many layers in a given region, the total amount of light yields a measure of the particle energy. HCAL has two more components with respect to the ECAL: these are called "forward sections" and are designed to detect particles moving with low scattering angles, extending the total coverage of the HCAL system. Like in the ECAL case, the achievable energy resolutions are described by

$$\left(\frac{\sigma}{E}\right) = \left(\frac{a}{\sqrt{E}}\right) \oplus c^2 \tag{1.8}$$

where E (deposited energy) is measured in GeV; c is equal to 5%; a is equal to 65% in the barrel, 85% in the endcap and 100% in the forward HCAL: the resolutions vary across the various region of the calorimeter.

Magnet

A strong magnetic field is fundamental for a detector used to bend the paths of charged particles from collisions in the LHC and measure their momentum. The CMS superconducting magnet [22] is a solenoid designed to provide a 4 T magnetic field. The solenoid has a diameter of 6 m and is 12.5 m long, making it the biggest superconductive magnet ever built, and stores an energy of 2.6 GJ at full current. In order to achieve such a current, the magnet must operate at a temperature of 4.6 K. Also, this structure allows to place the tracker and the calorimeters inside the magnet coil: in this way we get a much more compact detector with respect to others of similar weight.

Muon System

An efficient detection of muons [23] is a powerful tool to recognize signatures of interesting processes. Muons are charged particles like e^+ and e^- but with a mass 200 times heavier

and we expect them to be produced in decay of potential new particles: a decay into four muons represents the golden decay channel for the Higgs boson and also other exotic theories, such as SUSY, predict final states containing muons. Muons detection is one of the main aspects within the CMS experiment and a great attention has been payed in the design of the muon system, which is devoted to muon identification, momentum measurement and triggering. Since muons are very penetrating particles, they are not stopped inside any calorimeters (see Fig. 1.12). In order to detect them, the muon system exploit three different kinds of gaseous detector placed at the edge of the detector, where muons are the only particles able to produce signal.

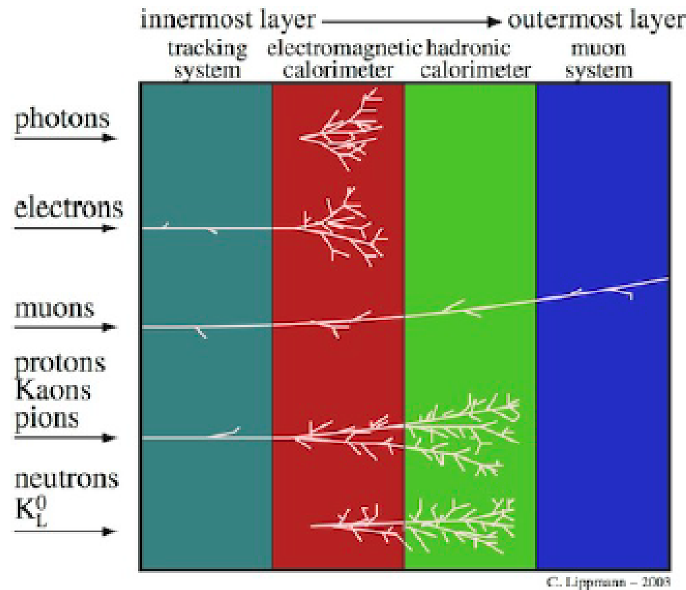

Figure 1.12: Paths of different particles crossing the CMS detector.

The muon detector, see Fig. 1.13, has a cylindrical, barrel section and two planar endcap regions. In the barrel region, where the muon rate is low and the magnetic field is uniform and mostly contained in the return yoke, drift chambers with rectangular drift cells are used. The barrel Drift Tube (DT) chambers are organized into 4 concentric stations interspersed among the layers of the flux return plates. In the two endcap regions, where the muons rate is larger and the magnetic field is not uniform, the Cathode Strip Chambers (CSC) are used. CSC is characterized by a fast time response, radiation resistance and fine segmentation and is used to track the particle position and contribute to the trigger system. There are 4 stations of CSCs in each endcap, the chambers are positioned perpendicular to the beam and interspersed between the flux return plates. The third type of detector, deployed throughout the central and forward regions, is the Resistive Plate Chamber (RPC): this kind of chamber offer a fast response and an excellent time resolution, thus are mainly dedicated for triggering. Six layers of RPCs are placed in the barrel region, two in each of the first two stations, and one in each of the last two stations.

By fitting the hits from the four muon stations, it is possible to measure the muon path. The muon stations, which are intervealed with iron return yoke plates, trace the

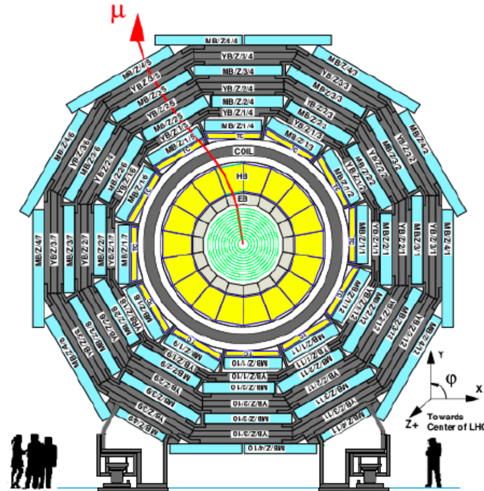

Figure 1.13: Barrel transverse view of the CMS detector.

particle path by tracking its position through the multiple layers placed in each station. Combining the signal with the tracker measurements, informations on the momentum are obtained: the higher is the momentum, the smaller is the bend due to the magnetic field.

Trigger System

During LHC operation, about one billion p-p collisions takes place every second. Among the huge number of collisions, produced with a frequency of 40 MHz, the potentially interesting events are selected through the so-called "trigger" system. Since no storage capacity and computing power are available to save and analyze such a big amount of data, any LHC experiment need a trigger system, which is able to reduce the huge amount of events to a smaller set with the interesting events.

CMS reduces the events rate by using a two tiered trigger system: the Level-1 (L1) [24] trigger and the High Level Trigger (HLT) [25]. The first one is hardware-based and performs data selection in a fast and automatic way by selecting data according to the physics of interest. The L1 trigger uses coarsely segmented data from the muon system and calorimeters while holding the high-resolution data in pipelined memories in the front-end electronics. This system performs the first event selection in CMS, by reducing the acquired data to some hundreds of event per second: the rate of collisions is reduced from the nominal 40 MHz to 100 kHz after the L1 selection. Then, the events are passed to the next trigger system. The HLT is a software-based system implemented in a filter farm of thousands of commercial processors and reduce the L1 output rate to the sustainable level for storage and physics analysis. The HLT algorithms are numerous, quickly evolve in time and are related to the needs of specific fields of research. This trigger system has access to the complete readout data and it can perform complex calculations, if required, for interesting events. The HLT is subdivided in different paths, each one corresponding to a dedicated trigger. Each path is a sequence of reconstruction and filtering modules and each module performs a well-defined task such as reconstruc-

tion of physics objects (electrons, muons, jets, etc.). The HLT is able to reduce the rate of collisions from the 100 kHz obtained in L1 trigger to 100 Hz, resulting in an overall rate reduction of $\sim 10^6$.

1.5 The CMS Computing Model

The CMS experiment presents challenges not only in terms of physics discover, detector building and its operation, but also in terms of data volume and computing resource. During LHC operation the frequency of collisions in each detector is $\sim 10^9$ Hz: such a high frequency results in a huge amount of data (∼1 PetaBytes per second (PB/s) in each detector) which has to be collected. As seen in section 1.4.1, this amount of data is selected by the trigger system, which reduces the data size to hundreds of MegaBytes per second (MB/s). In order to handle this amount of data, the Worldwide LHC Computing Grid has been designed and deployed.

1.5.1 The Worldwide LHC Computing Grid

The Worldwide LHC Computing Grid (WLCG) [26], [27], [28], [29] is a computing infrastructure arranged in Tiers characterised by computing centres distributed worldwide that interact with each other as a Grid service. The aim of WLCG is to provide computing resource and services to store and manage the $\sim 50-70$ PB of data expected every year of operations from LHC. Furthermore, each LHC computing model has to guarantee the availability of data on some computing resource to all partners, regardless of their physical location, and also the possibility to perform analysis processes on such data. In order to store, distribute and analyse data, the WLCG has a structure [30] [31] composed by four main layers: networking, physics software, middleware and hardware.

The networking layer manage the exchange of information between the WLCG centres. The file-transfer service has been adapted to support the special need of grid computing, i.e. authentication, confidentiality feature and others, and it allows to distribute data to hundreds of collaborating institutions around the world.

The WLCG centres need analysis software tools to satisfy the changing demands of the HEP community. The physics software layer includes programs such as ROOT [32], which is a set of object-oriented libraries used by LHC, and other software for modelling elementary particles behaviours.

Middleware is the software infrastructure which allows users to access distributed computing resources and is able to support powerful and complicated data analysis: it is called in this way because it sits between the operating systems of the computers and the physics-applications software.

For what concerns the hardware layer, large-scale management systems have the purpose of automating the necessary services, such as installation and updating of the required software, to manage the large collection of computers and storage systems in each grid. They ensure that the correct software, from the operating system to the experiment-specific libraries, is installed and make this information available to the overall Grid scheduling system, which decides which centres are available to run a particular

job.

1.5.2 The WLCG Tiers architecture

The computing environment of CMS, and also of the other LHC experiments, relies on the tiered architecture of the WLCG. The structure made by Tiers was formalised by the MONARC working group [33] and in the First Review of LHC Computing [34], that foresaw a rigid hierarchy. The centres composing the WLGC belong to few level (Tiers) which are labelled with 0, 1, 2 and 3: the number associated to each Tier indicate the level and the quantity of services and functionalities offered to the community. Therefore, each Tier is made up of computing resources centres which provide a different set of services depending on its level, e.g. the lower the label number is, the higher is the the provided quantity of services. Furthermore, a centre of the WLCG can be used by one or more LHC experiments and could cover different roles for each experiment, i.e. a Tier could have the function of Tier-1 for the LHCb experiment and acts as Tier-2 for CMS. By following this approach, CMS gains access to valuable resources and also improve robustness and data security through redundancy among multiple centres. The Tiers hierarchy is shown in Fig. 1.14 and now a brief description of each Tier will follow.

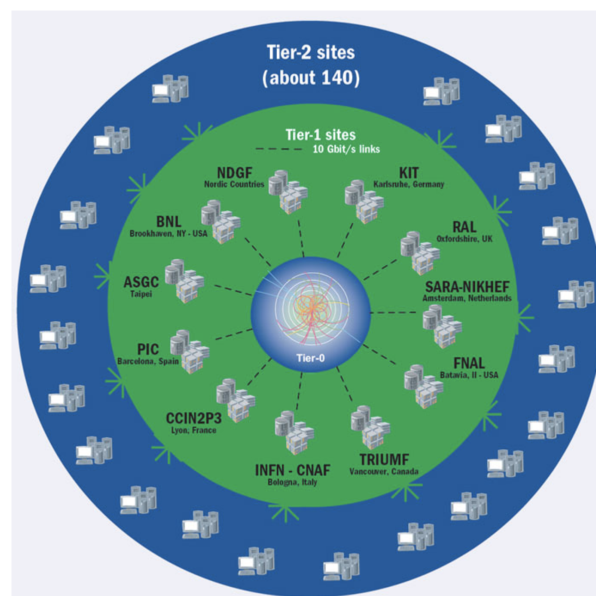

Figure 1.14: Schematical view of the WLCG tiered architecture.

Tier-0

Tier-0 is the CERN data centre (see Fig. 1.15) and all the LHC data pass through this central hub. This tier is devoted entirely to the storage and sequential reconstruction of RAW data: it receives and accepts data from each detector, stores them in tapes and performs prompt first pass reconstruction. The Tier-0 then distributes the RAW data and the reconstructed output to Tier-1s. The Tier-0 classifies promptly reconstructed

data, received from the Data Acquisition System, into primary datasets based on trigger information and archives such datasets into tapes, then the raw datasets are distributed among the next tier stage (Tier-1s): in this way, data are stored in two copies, one at CERN (the so-called "hot" copy) and one in Tier-1s (the so-called "cold" copy), thus offering an efficient and available mechanism for the data storage and distribution. Furthermore, in order to transfer data from Tier0 to Tier1 safely and efficiently, the LHC Optical Private Network (LHCOPN) [35] has been designed: this is an high-performance network infrasctructure, based on fiber optic working at 10 gigabits per second, which allows a performant connection between the two Tier level.

Figure 1.15: The CERN Data Centre.

Tier-1

There is a set of 13 Tier-1 sites, large enough to store LHC data, which are situated in different part of the world . The tasks carried out at Tier-1 centres principally relate to data storage, sequential processing of data and extraction of datasets to be further analysed at Tier-2 centres. Each Tier-1 centre receives a subset of data from the Tier-0 and provides a secure second copy by storing it on both disk caches and tapes. Furthermore, Tier-1 provides substantial CPU power that allow a wide range of computing services and operations such as event selection, data reprocessing, skimming and calibration. The results of such operations will be subsequently distributed to the next Tier level (Tier-2).

Tier-2 and Tier-3

There are ∼155 Tier-2 centres around the world, usually located at laboratories or universities, that can store sufficient data and provide an adequate computing power for specific tasks of analysis. The flexible resources and the large processing power provided by each Tier-2 allow to perform fast and detailed Monte Carlo event generation, data processing for physics analysis, data processing for calibration and alignment tasks, and also detector studies; on the other hand, Tier-2 sites have a looser storage, availability

and connectivity requirements if compared with a Tier-1 site. Then, there also exists the Tier-3 level, the lower one, which can consists of local clusters in a university department or even an individual PC, and provide direct access to users. Differently from the Tier-2, there is no formal engagement between a Tier-3 site and WLCG. This level is a good resource for the local community of physics end-users and analysts.

1.5.3 CMS data formats

In order to perform efficient HEP analysis, a physicist has to combine different information such as the reconstructed information from the recorded detector data, simulations of the physics signal of the process of interest under investigation and its relative background. All these information are stored into *event collections* and *datasets*. An "event collection" is defined as a data event which pass a specific trigger selection in a given "run" (a single bunch crossing), while a "dataset" is defined as a set of "event collection" that would be grouped and analysed together. Basically, an event collection is the smallest unit within a dataset that a user can select. HEP data are store in ROOT files. The smallest unit in computing space is called "file block", which corresponds to a group of ROOT files likely to be accessed together. This requires a mapping from the physics abstraction of the event (event collection or dataset) to the file location.

After being produced by the online system, the raw data undergoes successive degrees of processing that allow to refine such data, apply calibrations and create higher level physical objects. During each step of these operations chain, each bit of data must be written in a supported data format, i.e. a C++ class which defines a data structure (a data type with data members). CMS uses several data formats with varying degrees of detail, size and refinement to group the information from each step in the simulation and reconstruction chains: such formats, each one containing different level of information about the same event, are grouped into the so-called Data-Tier [36]. Each physics event is stored into each Tier level with different data formats, each one with different level of information about the event. The three main data formats are explained below.

- RAW: the full event information from the Tier-0, which contain the 'raw' detector information, i.e. detector element hits. This format is not used directly in analysis due to its size and complexity.

- RECO: which is the output from the first processing by Tier-0. The RECO format, which stays for RECOnstructed data, contains reconstructed physics objects which still contains many details about the event, resulting in a large size data and so generally is not used in analysis.

- AOD (Analysis Data Object): a "skimmed" version of the RECO event information which is expected to be used for most analysis. This format provides a good compromise between the event size and the complexity of its available information in order to optimise the analysis efficiency.

RECO data contains objects from all stages of reconstruction, it is derived from RAW data and provides access to reconstructed physics objects for physics analysis in

a convenient format. AOD are derived from the RECO information to provide data in a convenient, compact format which is directly usable for physics analysis. The AOD contains enough information about the event to support all the typical usage patterns of a physics analysis: it will contain a copy of all the high-level physics objects (such as muons, electrons, taus, etc.), and also a summary of the RECO information sufficient to support typical analysis actions. In Fig. 1.16 the description of other CMS data tiers is shown.

Event Format	Contents	Purpose	Data Type Ref	Event Size (MB)
DAQ-RAW	Detector data from front end electronics + L1 trigger result.	Primary record of physics event. Input to online HLT		1-1.5
RAW	Detector data after online formatting, the L1 trigger result, the result of the HLT selections (HLT trigger bits), potentially some of the higher level quantities calculated during HLT processing.	Input to Tier-0 reconstruction. Primary archive of events at CERN.		0.70-0.75
RECO	Reconstructed objects (tracks, vertices, jets, electrons, muons, etc.) and reconstructed hits/clusters	Output of Tier-0 reconstruction and subsequent rereconstruction passes. Supports re-finding of tracks, etc.	RECO & AOD	1.3-1.4
AOD	Subset of RECO. Reconstructed objects (tracks, vertices, jets, electrons, muons, etc.). Possible small quantities of very localised hit information.	Physics analysis, limited refitting of tracks and clusters	RECO & AOD	0.05
TAG	Run/event number, high-level physics objects, e.g. used to index events.	Rapid identification of events for further study (event directory).		0.01
FEVT	Full Event: Term used to refer to RAW+RECO together (not a distinct format).	multiple		1.75
GEN	Generated Monte Carlo event	-		-
SIM	Energy depositions of MC particles in detector (sim hits).	-		-
DIGI	Sim hits converted into detector response. Basically the same as the RAW output of the detector.	-		1.5

Figure 1.16: Description of the CMS data tiers.

Over time, the CMS computing model has evolved and new formats, that are not shown in the previous figure, have been introduced: MiniAOD [37] and NanoAOD [38].

At the end of 2013, the total size of AOD stored by CMS for Run 1 was about 20 petabytes. For the Run-2, in which the center-of-mass energy has reached the 13 TeV, the amount of the total size would have far exceeded the available storage resources. For this reason, CMS has developed a new, condensed data format called MiniAOD. The MiniAOD format was designed to reduce the event size and to increase the flexibility in handling available computing resource with small impact on physics analyses. MiniAOD reduces the event size by one order of magnitude with respect to the AOD format and is now the standard format for CMS analysis in Run-2.

With LHC entering a regime where the collision energy will not be dramatically increased, much larger datasets will need to be processed in physics analysis to achieve

their ultimate sensitivity. In order to do that, a further data reduction is needed and it cannot be achieved by increasing trigger threshold or imposing additional skimming requirements because the energy scale of interest will be driven by the particles masses to be studied. Therefore, in order to reduce more the event size, a new and more compact data tier has been researched and developed: the NanoAOD format. It consists of an Ntuple like format which contains the per-event information that is needed in most generic analysis. Therefore, the tiered structure of the data model employed by CMS is defined through different data formats. As shown in Fig. 1.17, each subsequent format contains a more compact summary of the event data than its predecessor, starting from RAW data, with a size per event of ∼1 MB, up to NanoAOD, with a size per event of the order of 1 Kb.

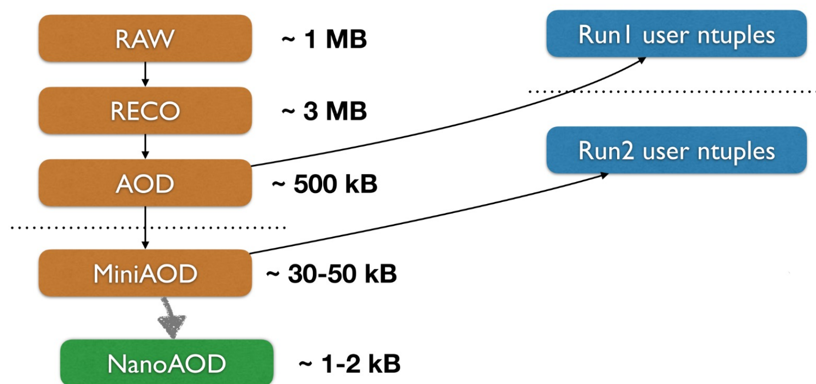

Figure 1.17: Evolution of data formats and their respective sizes.

1.5.4 Tasks of the Computing Model

CMS executes a variety of tasks on its distributed computing infrastructure [39] which allow to reconstruct or simulate collision data and analyze both. CMS uses a mainly object-oriented event data model to store event information in ROOT files. Standardized groups of objects are stored in different files which event content is described by different data tiers. Various tasks executed by the Computing Model are listed below:

- Generation task, this is the first step in collision data simulation, in which software packages are used to generate events from theoretical principles. CMS relies on a rich set of event generators, such as Pythia [40], MadGraph [41] and others.

- Simulation task, which takes the output of the generator and models the energy deposition of particles interacting with the CMS detector material. To perform the simulation, there are two separate but complementary frameworks: a Geant4-based [42] and a parametrised one, also called as "fastsim" [43]. These two frameworks are complementary: Geant4-based approach is used for the so-called "full simulations", which is very detailed but CPU and time consuming, while the parametrised one (fastsim) is used for the "fast simulation", where a reduction of details results in the gain of speed. Therefore, depending on the aim of the simulation, one of the two approaches can be used.

- Digitalisation task, which has to model the electronic processing of the signal produced by the energy deposition. It also simulates the effect of additional p-p interactions within the same (in-time) or neighboring (out-of-time) bunch crossings, the so-called pileup.

- Reconstruction task, which executes all the algorithms needed to interpret signals as being due to the interaction of identifiable particle with the detector. The reconstruction output comes in several different formats, such as RECO, AOD, NanoAOD, designed for the different needs of the analysis tasks.

- Analysis task, which reduces the size of the output of the reconstruction through the filtering of events and the elimination of information not required for a particular physics measurement. In this task specific event properties are calculated, and information get reduced to histograms or graphs. A statistical interpretation of the results is built.

1.6 Towards High Luminosity LHC

From the beginning of the project conception, LHC had a detailed plan of upgrades based on specific research objectives and improvements of the technology, that gradually led the machine to be more and more powerful. Anyway, during the years, there were some adjustments and delays on the original expected upgrades due to various obstacles encountered, e.g. in the supply of the materials, in the installation of the detectors, during the operations of data taking. The LHC plan is divided in three different data taking period, called Runs, with two Long Shutdown (LS) that link the different Runs (see Fig. 1.18). During a LS all the planned upgrades on LHC and detectors are integrated. Currently we are in LS2 and in the late spring / early summer Run 3 will start, which will last for almost three years. After that, another LS will bring to a new phase called High Luminosity LHC (HL-LHC) [44] [45] [46].

1.6.1 Upgrades for Run 3

Regarding the main upgrades of LHC for the upcoming Run 3, the center-of-mass energy will be pushed up to 14 TeV, the completion of the LHC Injectors Upgrade (LIU) will offer to the LHC the opportunity and the challenge to operate with up to two times higher beam brightness and the integrated luminosity is expected to reach 350 $\text{fb}^{-}1$ with a high pile-up from interactions in a single bunch crossing [47]. In addition a consolidation and/or replacement of all the components were performed. All LHC experiments also faced with consolidations and replacements and in some cases the sub-detectors have completely changed the type of technology. Considering the CMS experiment there are many upgrades introduced during LS2 [48] [49] [50] [51], e.g.:

- installation of a new beam pipe;

- installation of the new Gas Electron Multiplier (GEM) sub-detector to complement the muon system;

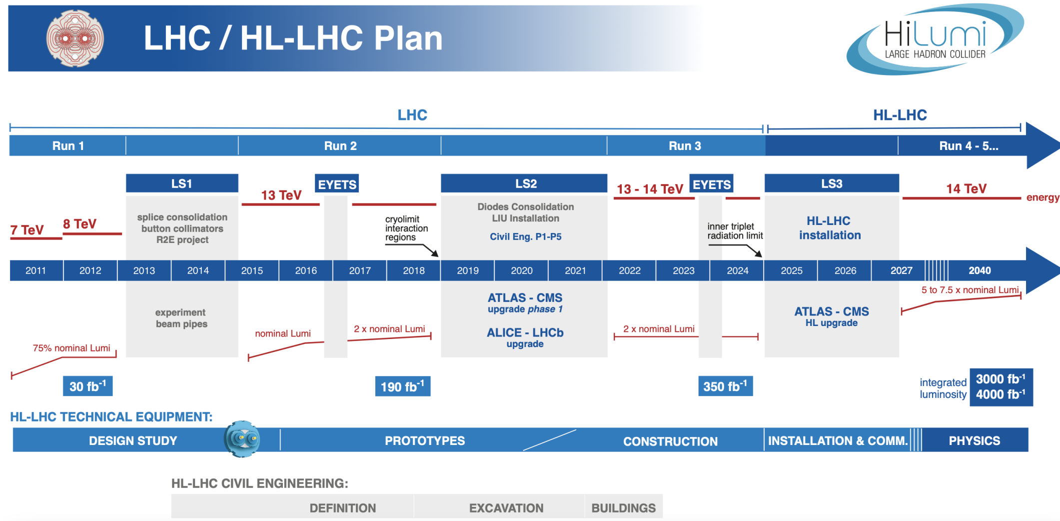

Figure 1.18: Plan of HL-LHC Project.

- installation of a new protection system for the central muon chambers, aimed to keep the outer DT chambers shielded from background radiation present in the experimental cavern during collisions;
- upgrade of the CSC electronic to be capable of withstanding the higher particle rate and radiation conditions of HL-LHC (since the chambers will not be accessible during LS3);
- usage of Machine Learning (ML) techniques in HCAL to identify subtle anomalies in the on-detector electronics quickly;
- completion of the new Layer-1 barrel pixel detector;
- modifications on trigger and Data Acquisition (DAQ) System.

Other main upgrades in CMS are shown in Fig 1.19. For CMS, the expected increase in integrated luminosity, of a factor 2 for p-p collisions and of a factor 3-5 for Pb-Pb collisions, together with the increase in centre of mass energy for p-p collisions will allow to substantially improve all the results obtained until now and to largely increase the sensitivity to new physics. Also, further improvements in searches for unknown phenomena are expected thanks to improved triggers and improved online reconstruction. The latter may also profit from the usage of GPUs in the high-level trigger, considered to be already deployed in production already in 2022.

1.6.2 Upgrades for HL-LHC

The High Luminosity Large Hadron Collider (HL-LHC) is an upgrade of the LHC which aims to achieve instantaneous luminosities at least a factor of five larger than the LHC nominal value, thereby enabling the experiments to enlarge their data sample by one

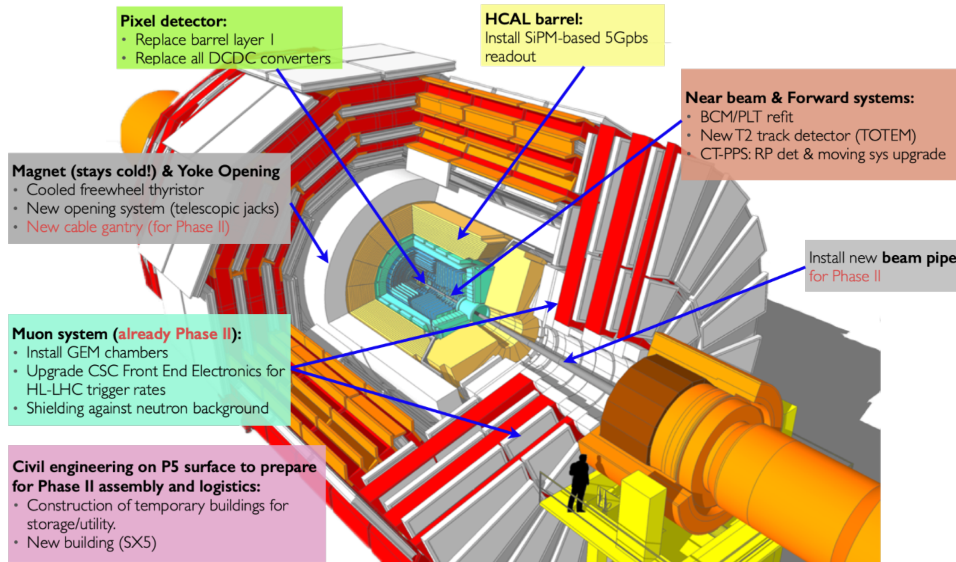

Figure 1.19: Schematic description of CMS upgrades preparing for Run 3 [52].

order of magnitude compared with the LHC baseline programme. The higher the luminosity, the more data the experiments can gather to allow them to observe rare processes, e.g. HL-LHC will produce at least 15 million Higgs bosons per year, compared to around three million from the LHC in 2017. Following five years of design study and R&D, this challenging project will require about ten years of developments, prototyping, testing and implementation; hence operation is expected to start in the second half of this decade. The timeline of the project is connected to the fact that, in the coming years, many critical components of the accelerator will reach the end of their lifetime due to radiation damage and will thus need to be replaced. The upgrade phase is therefore crucial to enable operation of the collider beyond 2027. The HL-LHC will rely on a number of key innovative technologies, including cutting-edge 11-12 T superconducting magnets, compact superconducting crab cavities with ultra-precise phase control for beam rotation, new technology for beam collimation, high-power, loss-less superconducting links, among others. For a detailed description of the project and its technological and operational challenges see [53]. To be prepared for the HL-LHC phase, also the CMS experiment planned many upgrades, the main ones are reported in Fig. 1.20.

1.6.3 CMS Computing Model towards HL-LHC

The luminosity of HL-LHC and the upgraded detector promise scientific results of great impact but, at the same time, pose new challenges on the CMS computing model, which will have to manage an amount of data that will be ∼30 times [54] greater than the one currently collected by LHC. Therefore, in order to exploit the full potential of HL-LHC, an evolution of all the area in the CMS software and computing infrastructure is needed.

In particular, the nature of the computing hardware, i.e. CPU processors, storage and networks, is evolving with radically new paradigms and by looking at market trends, one

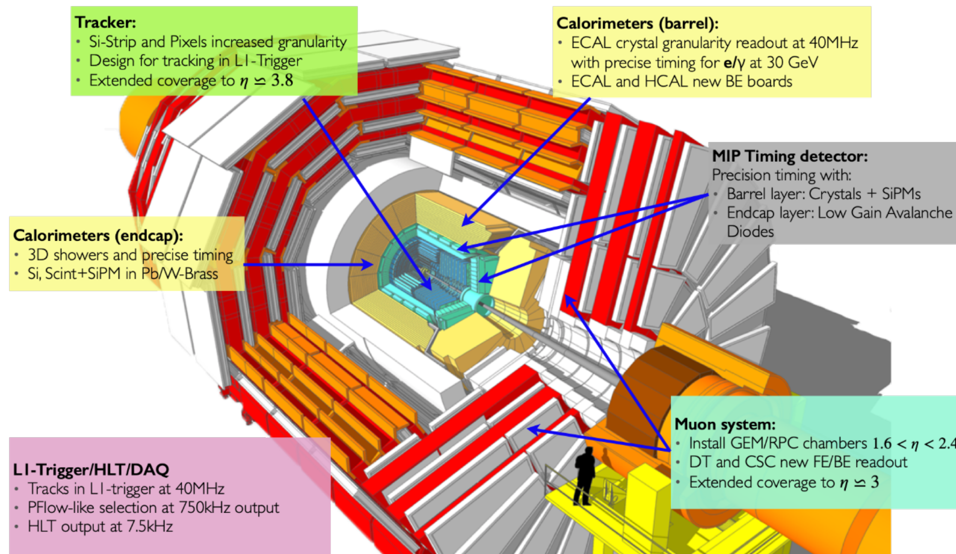

Figure 1.20: Schematic description of the CMS planned upgrades to be prepared for HL-LHC phase [52].

might anticipate that the hardware on which the HEP software will run could be different with respect to the present, a heterogeneous pool of computing resources composed of different CPU architectures accompanied by external accelerators. Also the concept of a HEP data center will evolve; indeed, specialized farms, such as data storage centers or analysis facilities, are likely to be built.

Then, as a consequence of the increase in the quantity and complexity of the data to be processed, more sophisticated analysis techniques will also be required to maximize the scope of physics and allow physicists to access the data efficiently and competently.

Furthermore, the duration of the HL-LHC program poses challenges to the CMS collaboration not only in terms of maintenance, but also in term of lifetime of its code base and related infrastructure, such as version control, continuous integration, building and testing. Assuming a software life cycle of about a decade, code developed today may be near its end-of-life in the early years of the HL-LHC. In such circumstances, a solid evolution of the software sector is therefore required: the development and implementation of sustainable code for future and updated experiments will be a technical and social challenge in which the retention of expertise, its renewal, and continued person power engagement are fundamental.

Another aspect that must be taken into account is the growing need for computing resources, which has been projected for CPU, disk and tapes in Fig 1.21 and shows a significant increase in resource demand over the next decade.

In conclusion, a simultaneous update on all areas that characterize the software and computing system is therefore necessary: a consistent approach to all the update plans needed towards HL-LHC will allow to cope with the huge increase in data and to collect them by fully exploiting all hardware upgrades.

The evolution of the computing model is also based on the use of ML: more details about its usage in HEP and in CMS will be discussed in Sec. 2.6.1.

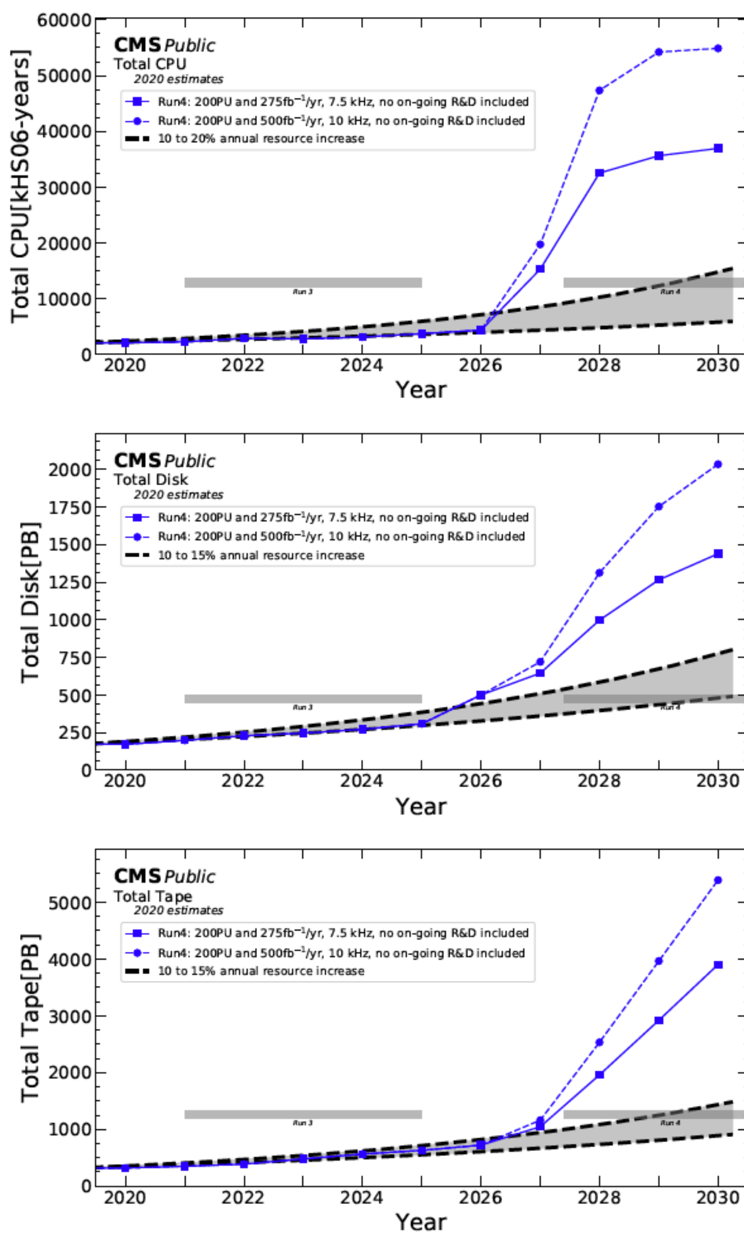

Figure 1.21: Estimated evolution of CPU (top), disk (middle) and tape (bottom) resources needs in function of time through Run 4.

Chapter 2

The Machine Learning Landscape

2.1 What is Machine Learning?

ML [55] [56] is the science and art of giving computers the ability to learn to make decisions from data without being explicitly programmed. The first definition of ML was given by Arthur Samuel in 1959 [57]:

"[*ML is the*] *field of study that gives computers the ability to learn without being explicitly programmed.*"

Another definition, a more engineering-oriented one, was given by Tom Mitchell in 1997 [58]:

"*A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.*"

When most people hear ML or Artificial Intelligence (AI), the first thing that comes to mind are robots or something related to a futuristic fantasy, but ML has been around for decades: the first ML application that became mainstream and improved the lives of millions people took over back in the '90s: the spam filter. The e-mail spam filter is a ML program that, given examples of emails labeled as spam (marked by the users) and example of regular emails, can learn to label spam emails. ML has been introduced in a lot of aspects of everyday's life: services like Netflix, Facebook, Amazon, Google work so efficiently thanks to learning algorithms which record all the preferences and choices of each users in order to build a customised environment. ML algorithms are used over a wide range of field (such as medicine, engineering, physics, etc.) and became one of the main pillars of computer and data science.

2.1.1 Terminology and Keywords

Before we continue with the description of the ML landscape, in this paragraph, a list of general and widely used ML terms is presented with the intention to be a guide for the reader throughout this thesis.

- **AI**: it is the theory and development of computer systems able to perform tasks normally requiring human intelligence.
- **Classification**: Supervised Learning method in which the output variable is a category (or “label”), such as “Male” or “Female”.
- **Deep Learning** (DL): is a ML subset based on Artificial Neural Network. Neural Networks (NN) attempt to simulate the behavior of the human brain, albeit far from matching its ability, allowing it to “learn” from large amounts of data.
- **Hyperparameter**: is a parameter that is set before the learning process begins. These parameters are tunable by users and can directly affect how well a model trains. Examples of model hyperparameters are the Learning Rate or the number of Epochs.
- **Inference**: term which refers to the process of making predictions by applying the trained model to unlabeled examples.
- **Loss Function**: is a function that maps an event or values of one or more variables onto a real number intuitively representing some “cost” associated with the event. An optimization problem seeks to minimize a loss function.
- **ML Algorithms**: are procedures implemented in code and are run on data to create a ML model.
- **ML Model**: is what is saved after running a ML algorithm on training data and represents the rules, numbers, and any other algorithm-specific data structures required to make predictions. Models provide a type of automatic programming where machine learning models represent the program.
- **Model Parameter**: is a configuration variable that is internal to the model and which value can be estimated from data. They are the part of the model that is learned from historical training data.
- **Model Training**: is the process in which it is tried to adapt the best combination of weights and bias to an ML algorithm in order to minimize a loss function. This process is performed by feeding the algorithm with a training dataset and is intended to build the best mathematical representation of the relationship between data. Throughout this elaborate, “fitting” will be used synonymously with “training”.

- **Performance Metric**: is defined as a logical and mathematical construct designed to measure how close are the actual results from what has been expected or predicted.
- **Regression**: is a Supervised Learning method where the output variable is a continuous value, such as "amount" or "weight".
- **Test Set**: data subset used to evaluate the trained model.
- **Training Set**: data subset used to train a ML model.
- **Validation Set**: data subset used to provide an unbiased evaluation of a trained model to adjust hyperparameters.

2.2 Machine Learning Approaches

In ML, the efficiency of an algorithm improves by repeatedly performing tasks by using data instead of programs; such improvement can be achieved through several ML approaches which are divided into four major categories, that can be classified according to the amount and type of supervision. These main categories are Supervised Learning, Unsupervised Learning, Semi-Supervised Learning and Reinforcement Learning.

Supervised Learning

Supervised Learning is a ML category which uses algorithms that determine a predictive model using labeled datasets with known outcomes. The algorithm is trained through the training set, which includes inputs and correct outputs, and this allow the model to learn over time. Furthermore, the aim of a Supervised Learning algorithm is to find a mathematical function able to predict the correct output on unseen and unlabeled data. Since the work of this thesis deals mainly with the Supervised Learning category, it will be covered in more detail in Sec. 2.3.

Unsupervised Learning

Unsupervised learning uses ML algorithms to analyze and cluster datasets, containing unlabeled data point (or inputs). These algorithms discover hidden patterns or data groupings in the data without having any external guidance. Therefore, by looking at these unlabeled inputs, the algorithms look for commonalities in data, and their reaction will depend on the presence or the absence of such commonalities. Unsupervised learning is used for different applications and analysis. In the following the main ones are listed.

- **Clustering analysis**, which aim is to group similar samples together into "clusters": samples within the same cluster present similar features according to one or some predesignated criteria, while samples belonging to different clusters are dissimilar.

- **Anomaly detection**, where the unsupervised algorithm learn what "normal" data looks like. Then, using this knowledge, it is able to recognise and detect anomalies from the input data.
- **Density estimation**, used to estimate the Probability Density Function (PDF) of the random process that generates the input data (very useful for data analysis and visualisation).

Semi-Supervised Learning

Semi-Supervised learning sits between unsupervised learning (no labeled input data) and Supervised Learning (labeled input data). Since labelling data is usually time-consuming, input file may be composed by plenty of unlabeled samples and few labeled ones: to deal with partially labeled data, Semi-Supervised learning algorithms are used. This kind of algorithm is mostly implemented as a combination of supervised and unsupervised algorithms.

Reinforcement Learning

This category is quite different from the others. Before describing how the reinforcement learning works, it is necessary to define some keywords.

- **Agent**: is the learning system, a computer program that acts for a user.
- **Action**: is the move/choice that an Agent can perform. The action performed by the agent is predefined into a set, where all the possible choices that the Agent can make are stored.
- **Environment**: is the "world" in which the agent can move and perform action.
- **Reward/Penalties**: is the feedback that measure the success (Reward) or failure (Penalties) of the action performed by the agent.

In Reinforcement Learning, the Agent can observe the surrounding environment and select and perform actions. Reinforcement Agents are able to automatically figure out how to optimize their behavior given a system of rewards and punishments: if the Agent select the right action, it gets a reward in return, otherwise it get a penalty. Therefore, the learning system learn by itself what is the best strategy, also called "policy", to get the most reward over time. The policy defines what action the agent should choose to get its reward. Being a very generic field, Reinforcement Learning is studied in different disciplines and has been applied successfully in many fields, such as economics, genetics, as well as game playing.

2.3 Supervised Learning

Supervised Learning [59] [60] [61] [62] is the ML task which looks for a function or a mathematical model that maps a (or some) certain input value(s), also called feature, to

an output learning from provided input-output pairs. A Supervised Learning algorithm is trained using a set of data containing features that, unlike the unsupervised case, are associated to a label (or target). Such dataset is composed by a set of "events", each one containing one or more feature and its relative output, and can be represented as a vector. The aim of Supervised Learning is to build a model that is able to predict the target variable given the data features thus automating time-consuming or expensive manual tasks. Through iterative optimisation of the Loss Function, Supervised Learning develop a function that can be used to classify correctly the unseen (or "new") inputs. There are two main types of Supervised Learning tasks: classification and regression.

The classification task uses algorithms to accurately assign events into specific categories, i.e. it is used to predict a discrete value. By discrete values we mean a count that involve integers in which only a limited number of values is possible: for example, in a binary classification problem the discrete value to be predicted can be either 0 or 1. So, in classification task, the algorithm recognises specific entities within the dataset and attempts to draw some conclusions on how those entities should be labeled or defined.

The regression one, instead, is used to find and understand relationship between dependent and independent variable. In regression tasks, the target value is an uncountable variable that varies continuously in a specific range.

In order to make classification and regression tasks more clear, one example each is reported.

Consider a medical researcher who wants to define a ML model capable of recognizing lung cancer. To do this, first of all it is necessary to take a set of x-ray lung images, which represents the inputs, and from these determines whether or not the tumor is present associating a label (such as "yes" or "no", "cancer" or "not cancer") to each image. Then, the idea is to build a model such that, if it recognizes the presence of a tumor on a new set of images, the returned output is "yes", otherwise the output is "no". Moreover, with such a delicate task, it is necessary that the model reports an answer that is as much correct as possible. To do this, the ML model has to be trained. Let's assume that the training set consists of the many x-ray images, some containing tumors and labeled as "yes" and some others without tumors labeled as "no". After many training iterations, the algorithm learns how to identify tumors with high accuracy. Since the researcher tries to predict a discrete value as output, i.e. "yes" or "no", this represents an example of classification problem.

Now let's consider the case in which an economist wants to predict an employee's salary based on his/her age and, as there seems to be a correlation between salary and age, the starting assumption is that, between the two quantities, there is a linear dependence described by $y = a + bx$: y represents the salary, a and b represent the coefficients of the equation and x is the age of the employee. At this point, in order to predict y given x, it is necessary to know the values of the coefficients. In order to do that, the algorithm must be trained with a dataset containing the age of the employees of a given company and their respective salary. During this phase, the coefficients a and b are trained and fitted to training data: the aim of the training is to find the fit line that best matches with the training set, thus also estimating the values of the a and b coefficients. Since in this example the task is to predict a continuous value output, i.e.

the salary, this represents an example of regression problem.

2.3.1 Training, validation and test

In ML, a common task is the construction of algorithms that can learn from data and make predictions on them. In order to do that, the ML algorithms make use of the input data to build a mathematical model. To build an efficient model, the input data are divided into three datasets, which are used in different stages: training, validation and test set [63] [64]. The motivation to split data into these sets is to avoid overfitting and memorization: without splitting, a model can over-memorize the data and so its performance will seem to improve even if it will generalize worse to new data. Therefore, dividing the dataset will allow to measure the model performances on "unseen" data and to ensure that the measured success gained from such data is not due to overfitting or data memorization: this will allow to get an idea of the expected performance on the new data that will be used.

The Supervised Learning algorithm is initially trained through the training set and produce a result that gets compared with the label of each input vector in the datasets. Then, depending on the result of the comparison and the used algorithm, the model parameters, such as weights of connections between neurons in a NN (see Sec 2.4), are adjusted. Basically, through the training set, the model tries to learn what the expected output looks like. During the training stage, the model analyzes the dataset repeatedly to deeply understand its characteristics and adjust itself for better performance.

After the "training" stage, the "validation" one takes over: the fitted model is used to make prediction on a second dataset called validation set. Such dataset is used to provide an unbiased evaluation on the trained model while tuning the model's hyperparameters: changing the value of hyperparameters alters the produced model and this will allow to select the best one. Hyperparameters are special values, which characterize the model, used to configure how the algorithm learns relationships in data. They can not be inferred while training the model and need to be specified before fitting as part of the model definition. In order to build a good model, the choice of the proper hyperparameters is fundamental because it will affect the goodness of the model and the speed of the training process. This phase is called hyperparameters tuning: users tries a bunch of different values for each hyperparameter and choose the ones that brings the model to perform better.

Lastly, the test set is used to provide an evaluation of the final model on unseen data. According to the performance obtained by different models on the test set, the one that best fits the new data is selected.

The goal of a good ML model is to generalize well from the training data to any data of the problem domain that the model has never seen.

A trained model, however, may run into the problems of overfitting and underfitting, which affect its performance. Overfitting refers to a trained model that has trained "too well" and fit too closely to the training dataset. This usually happens when a model learns the detail and noise in the training data, which means that random fluctuations (or noise) in the training set are picked up and learned by the model. The problem is that these concepts negatively impact on the model performance on new data and

model ability to generalize. The overfitting problem usually occurs when the model is too complex, e.g. too many adjustable parameters, so it will be very accurate in fitting known data (training set) but less accurate in predicting new data.

The underfitting instead is the inverse of overfitting. When a model is underfitted, it means it is too simplistic to represent the data accurately and therefore misses the trends in data, e.g. it could happen in a regression model when a linear model is fitted on non-linear data. Basically, the underfitting problem refers to a model which can neither model the training data nor generalize to new data: the model is too simple to learn the underlying structure of the data.

In both cases, a not generalised model is obtained, which means that it will make bad predictions on new data since it has a poor predictive ability.

To remedy the underfitting issue, a common solution is to choose a model with more complex structure, e.g. switch from a linear to a non-linear model or by adding hidden layers to a NN.

To prevent overfitting instead, some possible solutions could be to select a simpler model with fewer parameters, gather more training data and reduce the noise in the training set. Another solution is given by the K-fold Cross-Validation (CV), used as a preventive measure against overfitting.

K-fold Cross-Validation

K-fold CV is an useful technique which allows to split the dataset into training and validation set and to estimate the performance (see next section) of the ML algorithm. In k-fold CV, the original dataset is randomly divided into k equal sized subsets called “folds”. The k-th subset is used as validation set, while the other k-1 subsets are used as training set. The CV process is then repeated k times and each of the k subset is used once as validation set. The process is shown in Fig. 2.1. Then a single estimation of the reference metric is calculated as the average of the values obtained by all k folds.

Figure 2.1: Representation of k-fold CV process with $k = 5$.

Therefore, following this approach, all the subsets are used both for training and validation and the variance of the resulting estimate is reduced as k increase. There is, however, a trade-off as using more folds is more computationally expensive. This is

because the operation described is performed k times and, as a result, the training phase will take longer to complete, especially if you are dealing with a large datasets.

2.3.2 Performance Metrics

After the implementation of a ML model, the next step is to evaluate how effective the model is. In order to perform such evaluation, the so-called Performance Metrics [65] [66] are used: they allow to quantify the goodness of predictions obtained by the trained model. As introduced before, the main tasks in Supervised Learning are classification and regression and both of them have their respective metrics. The work of this thesis focuses on a classification problem, consequently the regression metrics will only be listed.

Since regression models have continuous output, the regression metric should be based on the calculation of a sort of "distance" between the value predicted by the model and the true one. For the regression model evaluation, some of the main metrics are:

- Mean Absolute Error (MAE);
- Mean Squared Error (MSE);
- Root Mean Squared Error (RMSE);
- R^2 (R-Squared).

Classification metrics, instead, evaluate the performance of the model by quantifying the correctness of the classification. Some of the main metrics for the evaluation of a Classification model are:

- Confusion Matrix;
- Precision;
- Recall;
- F1;
- Classification Accuracy;
- AUC (Area Under the ROC Curve).

Confusion Matrix

The Confusion Matrix is the simplest way to evaluate the performance of a Classification model. A confusion matrix is a summary of prediction results for a classification problem where the number of correct and incorrect predictions are summarized and divided for each label. It is a tabular visualization of the "Actual" vs "Predicted" label, where each raw of the matrix represents the predicted labels and each column represents the true ones.

Suppose, for example, to deal with a signal vs background classification problem: in this case, a 2×2 Confusion Matrix like the one in Fig. 2.2 is obtained and each cell of the matrix represent an evaluation factor which is described below.

- **True Positive** (TP), which represents the number of events correctly predicted as Signal,
- **True Negative** (TN), which represents the number of events correctly predicted as Background,
- **False Positive** (FP), which represents the number of Background events predicted by the model as Signal,
- **False Negative** (FN), which represents the number of Signal events predicted by the model as Background

Given any model, the confusion matrix can be filled according to its predictions. The correctly labeled signal events are referred to as true positives and correctly labeled background events as true negatives. While incorrectly labeled signals will be referred to as false negatives and incorrectly labeled backgrounds as false positives.

Figure 2.2: Content of a 2×2 Confusion Matrix.

Actually, the Confusion Matrix itself is not a performance metric, but it provides useful information and almost all the performance metrics are based on it and on its evaluation factors.

Precision

Precision metric [67] is defined as the ratio of correctly predicted positive observations to the total predicted positive observations. In terms of Confusion Matrix elements:

$$Precision = \frac{TP}{TP + FP} \tag{2.1}$$

Precision basically represents the ability of the classifier to not mislabel a sample that is negative.

Recall

The Recall metric [68], also called Sensitivity, is defined as the ratio of correctly predicted positive observations to the all observations in actual class. Mathematically:

$$Recall = \frac{TP}{TP + FN} \tag{2.2}$$

The recall metric basically represent the ability of the classifier to find all the positive samples.

F1

The F1 metric [69] is the weighted harmonic mean of Precision and Recall and it is defined as follow:

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.3}$$

This score takes both FP and FN into account and assumes values between 0 and 1: the expected performance of the model will be better for F1 values approaching 1.0.

Classification Accuracy

The Classification Accuracy is the most common performance metric for classification problems. The Accuracy is defined as the ratio between the number of correct predictions and the number of all predictions made. In terms of confusion matrix component, the Accuracy is defined as shown in the following formula:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{2.4}$$

Even if it is pretty common, Accuracy is not always a useful performance metric. To clarify, consider a classification problem in which 95% of events are background and only the 5% are signal: by building a model which predict all events as background, this would have an Accuracy of 95% even if it never predicts signal events. However, such classifier will never predict signal and so it fails completely at its original purpose. When dealing with datasets in which one class is more frequent than the other, Accuracy is not suitable for establishing model performance and a more efficient one must be used.

Before introducing the next metric, it is necessary to define the concept of "classification threshold". In probability, the threshold is the specified cut off, on the probability predicted by the ML model, for an observation to be classified, for example as either 0 or 1. In general, for binary classification problems, the threshold value is setted by default at 0.5, so all the values equal or greater than the threshold are mapped to 1 and all the other are mapped to 0. For some classification problems, such default threshold value is not optimal and result in poor model performance: to avoid this problem, it is necessary to adjust this threshold value and this can be done directly by using the ROC Curve.

AUC

Recalling the components of the confusion matrix, let's define the True Positive Rate (TPR) and False Positive Rate (FPR) as follows:

$$TPR = \frac{TP}{TP + FN} \qquad FPR = \frac{FP}{FP + TN} \tag{2.5}$$

ROC Curve plots TPR vs FPR by changing the classification thresholds as show in Fig. 2.3, allowing also to find the optimal threshold value for the given task.

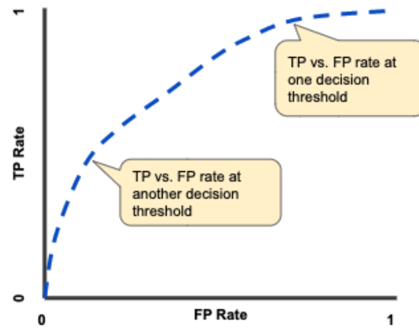

Figure 2.3: ROC Curve obtained varying the classification thresholds.

The AUC metric provides a measure of the model performance across all the possible thresholds by measuring the entire two-dimensional area under the ROC Curve: the larger the area under the ROC curve, the better the model is. For a binary classifier that make predictions randomly, it would be correct approximately 50% of the time and the resulting ROC curve would be a diagonal line in which the True Positive Rate and False Positive Rate are always equal and the Area under this ROC curve would be 0.5. Then, if the AUC is greater than 0.5, the model is better than random guessing.

2.3.3 Supervised Learning Algorithms

Supervised Learning has a wide range of algorithms, here are some of the most important ones.

- **k-Nearest Neighbour** (kNN): classifies data by clustering the train set in classes. The probability of belonging to a certain class is evaluated considering user-defined distance in the feature space.
- **Linear Regression**: this technique finds out the linear relationship between input and output.
- **Support Vector Machines** (SVM): has the objective to find a hyperplane in an N-dimensional space (where N is the number of features) that classifies the inputs.

- **Decision Trees** (DTs): uses a flowchart like a tree structure where data are subjected to a series of feature-based splits. A DT algorithm divides a dataset into smaller subsets, while at the same time an associated DT is developed incrementally. As end result, the user obtains a tree with decision nodes and leaf nodes, which contain respectively the splitting condition and the outcomes, that can successfully discriminate the target feature.

- **Artificial NNs**: uses a structure comprised of node layers to recognize underlying relationships in data through a process that mimics the way the human brain operates. Artificial NNs will be treated more clearly in the next section.

2.4 Artificial Neural Networks

NN [70] [71] [72] [73] is a ML algorithm inspired by the network of the biological neuron of the human brain which enables to learn from observational data. From this inspiration, in 1943 there was the first experiment to try to simulate intelligent behavior through electrical circuits and, in the '50s, researchers began to seek a way to implement this "intelligence" in computational systems: the first network was successfully implemented at MIT in 1954 and in 1959 the first NN successfully applied to a real-world problem was developed. These early successes gave rise to a growing hype about the capabilities and potential of NN, but also led to several problems that caused a temporary halt in research: one of the main problems concerned the extremely long execution times required to run NN. In the '70s, research in the field of NN was completely stopped and began the age now famously referred to as 'the AI winter'. Fortunately, with the rediscovery of the backpropagation concept, i.e. an AI algorithm used to fine-tune mathematical weight functions, that had already already introduced in the '60s, the 1990s saw the great return of NNs with potential that, over the years, overcame projected expectations.

NNs are now the core of DL, they are versatile, powerful and scalable tools, ideal for tackling large and highly complex ML tasks such as powering speech recognition services, such as Siri (Apple) and Alexa (Amazon), or recommending the best TV shows to hundreds of millions of users, such as Netflix.

A NN is based on a collection of nodes, called "artificial neurons", where each node, which has an associated weight and threshold (or bias), is connected to another one: through such connection, each node receives a signal, process it and transmit it to the other neurons. Depending on how the nodes are connected, different NN architectures [74] have been developed during the years (some of them are reported in Fig. 2.4). In the following we will focus the attention to one of the simplest architecture, the Multilayer Perceptron [75] (MLP).

A MLP is a class of feedforward NN, which means that data flows in the forward direction from input to output layer. As the name suggests, MPLs are composed of neurons called "perceptrons". Each perceptron (see Fig. 2.5) receives as input a set of n feature, such as $x = (x_1, x_2, ..., x_n)$, and each of this feature is associated to a weight w_i. The input features are then passed to an input function u, which computes the weighted sum of the input features, as shown in the following equation:

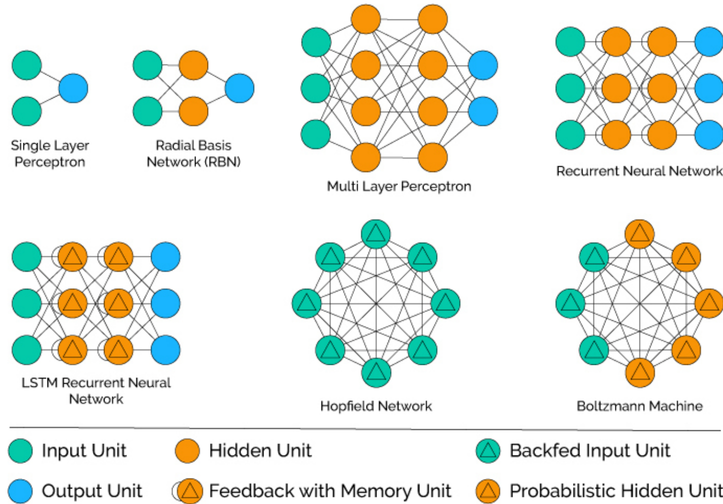

Figure 2.4: Examples of different NN architectures.

$$u(x) = \sum_{i=1}^{n} w_i x_i \tag{2.6}$$

Then, the result of the input function is passed to the activation function f, which will produce the perceptron output adding a non-linearity feature.

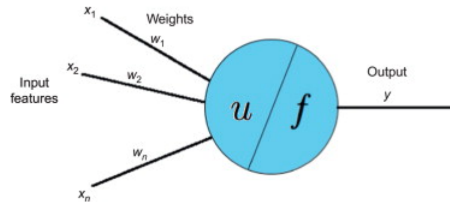

Figure 2.5: Representation of a single perceptron.

By stacking multiple perceptrons together, the MLP NN is obtained (see Fig. 2.6). The nodes in the MLP are organised in at least three layers:

- input layer, which receives external data and distributes them to the first hidden layer.
- hidden layers of perceptrons, the first hidden layer receives as input the features of the input layer, while the others receive as inputs the output of the previous layer.
- One output layer of perceptrons, which receive the output of each perceptron of the last hidden layer.

The MLP NN is able to approximate any continuous function thanks to the role covered by the activation function. The activation function is applied to the weighted sum of each perceptron and add non-linearities to the model. There are different kinds of activation functions, two of the most used are presented below.

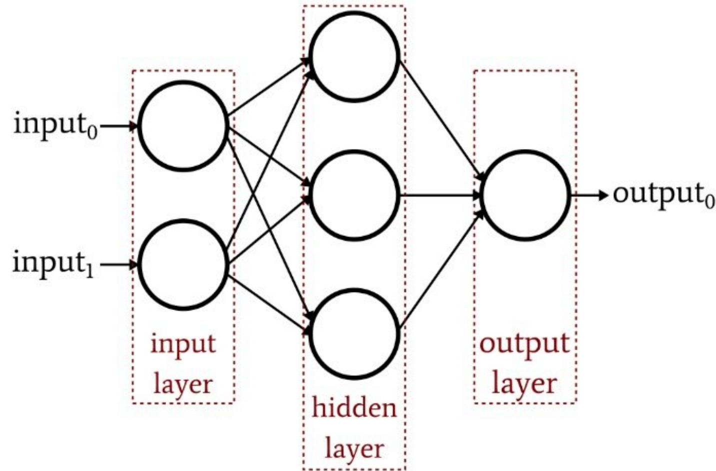

Figure 2.6: Example of a MLP NN composed by...

- **Sigmoid**: converts the weighted sum of the input features to a value between 0 and 1 according to the following definition (where x is the weighted sum):

$$f(x) = \frac{1}{1 + \exp^{-x}} \tag{2.7}$$

- **ReLU** (Rectified Linear Unit): is a piecewise linear function defined as:

$$f(x) = \max(0, x) \tag{2.8}$$

where x represents the input received by the hidden layer.

2.4.1 Training of a Neural Network

When a NN is defined, initially its weights are initialized randomly and, as a consequence, it will not provide good results. However, through the training process, the NN is able to adjust the weights of each node to fit the set objective. In Supervised Learning, training of a NN is usually conducted by determining the discrepancy between the processed output of the network and a target output: this difference defines the loss function of a neural network. A loss function is then a method of evaluating how well your algorithm models your data set: the lower the loss function is, more reliable are the results obtained from the NN. Basically, the training process of a NN involves finding a set of weights that minimize the loss function, in order to be good enough at solving the specific problem. This is an iterative process, which means that it proceeds step by step with small updates on the NN weights, thus leading to a change in model performance for each iteration.

In order to execute properly the training of a NN, it is necessary to define an *optimizer*. An optimizer is an algorithm that modifies the values of the NN weights with the purpose of reducing the loss function and improving accuracy of the model. Over time, several optimizers have been defined: among them, one of the most widely used is Stochastic Gradient Descent (SGD) optimizer, in which weights are updated using the backpropagation of error algorithm.

2.5 Machine Learning Frameworks

Nowadays, ML techniques are used in almost every industry, including finance, insurance, healthcare, and marketing. The growth in success of ML in the last few years has led to development of several ML frameworks. A ML Framework is a tool or library that allow to build ML model without getting into the underlying algorithms. It provides a clear, concise way for defining ML models using a collection of pre-built, optimized components.

In the following, some of the most popular frameworks in use today are listed.

- **TensorFlow** [76]: is a library created by Google and released as an open-source project. It is a versatile and powerful ML tool with a comprehensive, flexible ecosystem of tools, libraries and community resources.
- **PyTorch** [77] [78] [79]: is an open-source ML framework for NN design. It is based on Torch [80] and Caffe2 [81] and has been developed by the Facebook's AI Researcher lab (FAIR).
- **Scikit-Learn** [82]: is an open source ML library built on NumPy, SciPy and matplotlib that supports supervised and unsupervised learning algorithms.
- **Theano** [83]: is a Python library that let the user define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently and have them compiled in a highly optimized fashion either on CPUs or GPUs, attaining high speeds that gives a tough competition to C implementation.

2.6 Machine Learning in HEP

ML is applied to many problems in particle physics research since several years [84]. In the 1990s and 2000s ML was initially applied to high-level physics analysis only, but then, in the 2010s, an important phase of expansion for ML techniques and applications followed. So, the HEP community began to apply ML on different problems, such as event and particles identification and reconstruction, obtaining excellent results. ML is a rapidly evolving approach which is able to describe and characterise data in a way that may change how data are collected, reduced and analysed. Also, ML spans a wide range of possible applications in the HEP context; it can be used to allow more efficient use of processing and storage resource or to qualitatively improve the access to datasets. Consequently, to being able to use a powerful tool such as ML, the HEP community needs to build domain-specific applications on top of existing toolkits and ML algorithms, written by scientific software developers both from inside and outside the HEP "world".

Since the HL-LHC will reach an integrated luminosity 20 times larger than the actual one, new challenges in terms of event size and data volume will be faced with the risk to be limited by the performance of algorithms and computational resources.

Implementing ML in HEP workflows will require significant research and development, but it will give an important help in this scenario.

2.6.1 Machine Learning Algorithms and Applications in HEP

The most frequently used ML algorithms in HEP are the Boosted Decision Trees (BDTs) and NNs, typically used to classify events and particles but also for regression problems. The main types of NNs used in HEP are Convolutional (CNN), Fully-Connected (FNC) and Recurrent (RNN). Furthermore, NNs are also used in the Generative Models context, where a NN is trained to reproduce a multidimensional distribution of the instances of the training set.

There are many areas in HEP where ML is successfully used. A general overview about the fraction of physics papers (mainly HEP) for different areas and analyses where ML techniques have been used is shown in Fig 2.7. In the following we will describe some of the main HEP areas where ML is used.

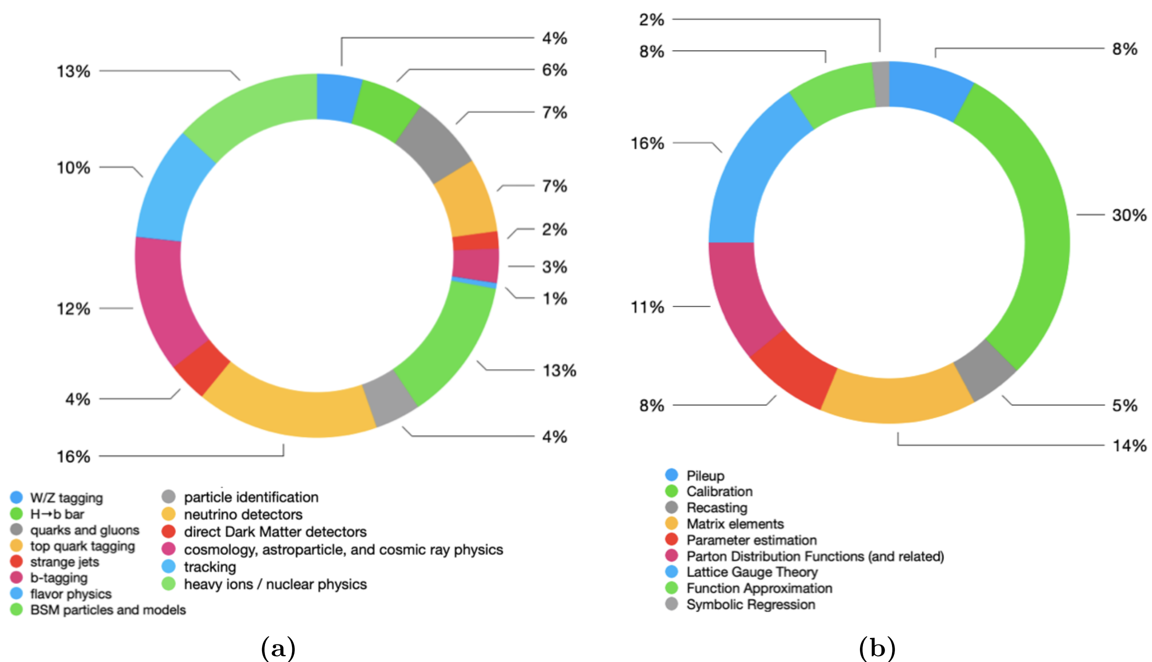

Figure 2.7: Fraction of (mainly HEP) physics paper involving ML classification **(a)** and regression **(b)** techniques [85].

Simulation

Discovery of new particles relies on the ability to accurately compare the expectations base on the physics model with the observed detector response data. In order to simulate the particles behaviour inside the detector, Monte Carlo simulation tools, like Geant4, have been developed.

For the HL-LHC, in order to achieve the right statistical accuracy to perform precision hypothesis testing, it is necessary to simulate trillions of collisions and simulations are highly computational expensive. Recently, there has been plenty of progress in high fidelity fast generative models, such as GANs (Generative Adversarial Networks) and

VAEs (Variational Autoencoders), which, by learning from existing data samples, are able to sample high dimensional feature distributions. Such techniques offer a promising alternative to the existing fast simulation techniques, in particular in terms of simulation speed.

Real Time Triggering and Analysis

The traditional approach to data analysis assumes that the interesting events recorded by a detector can be selected in real-time through the process of *triggering* and, once selected, the events can be stored and distributed for further selection and analysis. However, with the new HL-LHC, because of the luminosity gain, these previous assumptions break down: the events will be so abundant that cannot be completely stored for successive analysis. To deal with the increasing event complexity, the use of sophisticated algorithms, i.e. better suited to handle such an increase in complexity, will be explored at all trigger levels.

Monitoring of Detectors, Hardware Anomalies and Preemptive Maintenance

Data-taking is continuously monitored by physicists taking shifts to check the quality of incoming data from collisions in each subdetector. This is done using hundreds of reference histograms defined by experts and, when an unexpected deviation with respect to the reference occurs, shifters should be prompt to notice it. In such situation, a class of unsupervised learning algorithms making "anomaly detection" might be useful for automating this task. Such algorithms learn from data and, when a deviation is seen, they produce an alert. By monitoring several variables together, such algorithms are sensitive to forewarning sign of imminent failure, so that preemptive maintenance can be scheduled. The next step in this task would be to associate an anomaly detection algorithm to an appropriate action, such as restart an online computer or contact an expert.

Identification and Tracking

Physical processes of interest in HEP experiments occur on too short time scales to be directly observed and so the decay products of the initial particles are used to infer their properties. A better knowledge of the decay products properties allows a more accurate reconstruction of the initial process. In the past decade, experiments have trained ML algorithms (BDTs was one of the most popular) were trained on features from combined reconstruction algorithms to perform particle identification. More recently, researchers focused on extracting better information by using DNNs e.g. convolutional, recurrent and adversarial NNs. For tracking detectors instead, the pattern recognition is the most computationally challenging step that becomes intractable for the HL-LHC; therefore, the hope is to find adequate ML techniques providing a solution that scales linearly with the collision density. Several efforts in the HEP community have already started to investigate different ML algorithms for track pattern recognition on many-core processors.

Computing Resource Optimization and Control of Networks and Production Workflows

Data operations is one of the significant challenges for the upcoming HL-LHC. Currently, LHC experiments rely on in-house solutions for managing the data. While these approaches work reasonably well today, ML can give its contribution to automate and improve the overall system throughput and reduce operational costs.

ML can be applied in many areas of computing infrastructure, workflow and data management. For instance, dataset placement optimization and reduction of transfer latency can lead to a better usage of site resources and an increased throughput of analysis jobs. One of the current examples is predicting the "popularity" of a dataset from dataset usage, which helps to reduce disk resource utilization and improve the time to perform a physics analysis [86]. Data volume in data transfers is one of the challenges facing the current computing systems as thousand of users need to access thousands of datasets across the Grid. There is an enormous amount of metadata collected by application components, like information about failures, file accesses, etc. Resource utilization optimization based on these data can improve the overall operations. Understanding the data transfer latencies and network congestion may improve operational costs of hardware resources.

2.6.2 Collaborating with other communities

It is important to search a collaboration between the data science and HEP communities by finding a common language and working together to further science. Both communities can benefit from such collaboration. The HEP community can explore new research directions and applications of ML, novel algorithms, and direct collaboration on HEP challenges. The ML community can benefit from a diverse set particle physics problems with unique challenges in scale and complexity. Anyway having a productive connection between these two worlds is not so trivial. The HEP community needs to define its challenges in a language that the ML community can understand. ML likewise has a significant amount of domain knowledge. Thus, ideas and solutions provided by both communities should be presented in an understandable way for scientists without in-depth knowledge.

It is important for HEP physicists to engage and collaborate with the academic community focused on ML algorithms and applications. Conferences and workshops are a core aspect of the academic ML community, and organizing or contributing to key conferences is a means of gaining interest.

Also the industry engagement plays an import role in this scenario. One of the promising areas is the adoption of dedicated specialized hardware and high performance co-processors. GPUs, FPGAs (Field Programmable Gate Arrays) [87], and high core count co-processors all have the potential to dramatically increase performance of ML applications relevant to the HEP community. One of the challenges is gaining the human expertise for development and implementation. There are specific areas of development where industry has expressed interest in collaborating with HEP. Automated resource provisioning, data placement, and scheduling are similar to industrial applications to

improve efficiency. Applications such as data quality monitoring, detector health monitoring and preventative maintenance can be automated using techniques developed for other industrial quality control applications. There are two more forward looking areas that coincide with HEP physics drivers, namely computer vision techniques for object identification and real-time event classification. These present a challenge to industry due to its complexity and benefit outside of HEP. For all these reasons, CERN decided to start the OpenLab program to provide an interface between CERN and industry.

2.6.3 Machine Learning software and tools in HEP

Currently there are two ML software methodologies in HEP. The first approach is to implement abstract ML algorithms in HEP-developed toolkits, such as the Toolkit for Multivariate Analysis (TMVA) in ROOT. This provides on site support to physicists and dedicated development for HEP data formats and applications. The second approach is to rely on externally developed software, of which there are many examples. This often requires tedious and repetitive work to adapt HEP data formats to external software, breaks analysis workflows and introduces difficulties in the analysis software development. Interfaces between HEP and external tools have also been developed during the years. Currently, many published HEP analyses with machine learning have made use of TMVA. At the same time, the ML landscape has evolved and many different ML tools have emerged and gained popularity. There are a growing number of published results based on externally developed tools. Advantages of using external tools from the HEP world are the size of the community that uses and supports them, being able to easily keep up with progress in the industry and profit from the forefront of the ML research. Advantages of using internal tools are that decisions about long-term support remain in the community, and the tools can be adapted to the specific needs of HEP.

2.6.4 Computing and Hardware Resources

In order to progress to evaluation of complex ML algorithms, high computing power is needed in both the training and evaluation stages, as larger amounts of data are needed to feed models with tens or hundreds of thousands of parameters. This implies the expansion of the current computing model to include architectures that are well suited to ML tasks, such as many integrated core (MIC), graphical processing units (GPUs) and Tensor Processing Units (TPUs). These architectures provide a significant computational speed improvement for both training and evaluation of ML algorithms, but require dedicated hardware, drivers, and software configuration. Similarly, the locality and bandwidth of large data stores will need to be optimized in order to avoid bottlenecks in training and evaluation for analysis. Data placement and the need to use dedicated hardware indicate that a transition to HPC, or HPC-like, architectures may be needed to achieve the desired performance. Due to significant synergy with the direction of industry in this respect, use of commercially available resources should be considered for future high-energy physics computing models.

2.7 Machine Learning in CMS

Following the success of ML and DL, the CMS collaboration [88] [89] has relied on an extensive and diverse set of supervised and unsupervised ML methods at every stage of the experiment, from the online selection of events to the offline analysis of the recorded data. This has resulted both in more efficient detector operations, as well as a significantly improvement on the quality of the physics results delivered. Such new methods have been developed in order to increase the performance of the CMS experiment to better face the challenges imposed by the Run-3 and HL-LHC.

Managing the amount of data from the LHC collisions is a major challenge for CMS physicists. The detector produces a huge amount of data and, even after sophisticated real-time events processing and filtering, tens of PetaBytes of data are saved offline for further analysis each year. Data storage will become even more challenging with the HL-LHC program. In order to be prepared, CMS physicists are using ML algorithms at every stage of data processing in order to improve experiment performance, accelerate computations, improve the quality of the data and optimize searches for new physics signatures. The main goal of the processing task is to select only the most interesting events during collisions in order to reduce the storage space needed. In order to achieve this real-time data reduction, the algorithms are implemented in custom electronics using FPGA.

ML is also used to improve the data analysis and physics results: it is used for the particle identification such as tau leptons decaying hadronically or jet tagging, and to increase the sensitivity of analyses thanks to the usage of NN driven signal extraction.

Furthermore, CMS scientific computing group is using ML algorithms to predict dataset usage to optimize storage and retrieval of data. CMS also pioneered integration of deep learning frameworks into its core software stack, allowing the use of trained models for analysis and reconstruction. Other innovative ideas have been studied for real-time anomaly detection in CMS data to ensure data quality, detector monitoring, and even search for new physics.

Chapter 3

Machine Learning as a Service for High Energy Physics

As described in 2.6, ML has achieved great popularity in the HEP context and its role will be crucial for the upcoming HL-LHC program, when it is expected, in the next decade, an amount of data of the order of ExaByte.

Even if several ML models have been successfully applied in many areas, such as detector simulation, particle identification and so on, there is a constant seek for scalable and performant of ML workflows. Furthermore, as stated in "ML in HEP Community White Paper" [84], the gap between ML experts and HEP physicists and the consequent difficulty to formulate HEP problems in a way that match the Computer Science (CS) skills, results in the lack of engagement from CS experts to address HEP ML challenges. Such gap is due to the specificity of some parts of the HEP typical workflow and solutions. One of these HEP context specificity is the use of the ROOT data format, which allow to store HEP events in a tree-based data structure. On the other hand, existing ML frameworks rely on fixed-size data representation of individual events: the most common formats in which such data are stored are NumPy [90], CSV [91], HDF5 [92] and others. The ROOT data format is mostly unknown outside the HEP community and presents a structure that can not be directly fed into the existing ML framework. This and other reasons led to an artificial gap between the two communities.

In order to reduce this gap and to simplify the use of ML techniques for not-ML-pratictioner physicists, a "ML as a Service for HEP" (MLaaS4HEP) [93] [94] [95] [96] solution has been proposed in 2018 as a product of R&D activities within the CMS experiment. In this Chapter, information and details on the structure of MLaaS4HEP and its features will be provided, while the personal contribution within this project will be presented in Chapter 4.

3.1 Machine Learning as a Service

ML as a Service (MLaaS) [97] refers to a set of ML tools offered by cloud service providers. The main benefit of this solution is that it allow to get started with ML quickly and without installing any specific software. MLaaS is a well know concept in the industry

and the major IT companies, such as Amazon, Google, Microsoft, offer such service to their costumers: such service consists of a ML platform with a wide range of services, with some ready-to-use solutions for the majority of popular ML applications and which allow to use different ML frameworks, such as TensorFlow or PyTorch. MLaaS platforms are provided on a cloud environment where all stages of the ML process, i.e. data management, model development, training, inference and performance monitoring, are handled by one provider which ensures the maximum efficiency of the learning process. Moreover, MLaaS providers offer services (high-level APIs) with trained models that the user can feed with his own data to make inference.

3.2 Machine Learning as a Service for HEP

A solution like MLaaS in the context of HEP would be very beneficial. However there are some difficulties in using this service directly on HEP data and one of these is related to the aforementioned ROOT data format: since such format is not suited for ML, a conversion step is needed. Furthermore, the HEP physicists usually perform some pre-processing operations on data that may be more complex than the one offered by the MLaaS providers. On the other hand, different examples of R&D solutions have been proposed, but they are oriented towards a specific phase of the ML pipeline and, consequently, do not cover it in its entirety. Furthermore, there are some existing custom solutions for specific analyses in CMS, but they do not represent a real "as a Service" solution since can not be easily generalized. It was therefore concluded that there was no final product that can be used as ML as a Service for distributed HEP data, and so the "MLaaS4HEP" solution were proposed.

The MLaaS4HEP proposed solution has the following aims to:

- provide a transparent access to HEP datasets stored in the ROOT data format of arbitrary size, both from local and remote sites;
- use heterogeneous resource both for training and inference, such as local CPU, GPUs, Cloud Resources, etc;
- allow the user to be able to select different ML libraries and frameworks, such as TensorFlow, Keras [98], Pytorch, etc;
- serve pre-trained HEP models and a way to access them easily from any place, code and framework.

3.3 MLaaS4HEP Architecture

A typical ML workflow consists of three steps: *i*) acquire data necessary for training, *ii*) use a ML framework to train the model, *iii*) use the trained model to make prediction on new data. In MLaaS4HEP, this workflow can be summarized in three phases: data streaming, data training and data inference. The three layers that compose MLaaS4HEP, shown in Fig. 3.1, are defined as:

- **Data Streaming Layer**: it deals with reading of local and remote ROOT files and streaming data into the Data Training Layer.
- **Data Training Layer**: it receives data from the previous layer and transforms them to a format suitable for the used ML framework. Then, these transformed data are used for model training.
- **Data Inference Layer**: it has been implemented as "TensorFlow as a Service" (TFaaS) and it deals with the inference part of the service. It allows to upload pre-trained TF models and use them for inference via HTTP protocol.

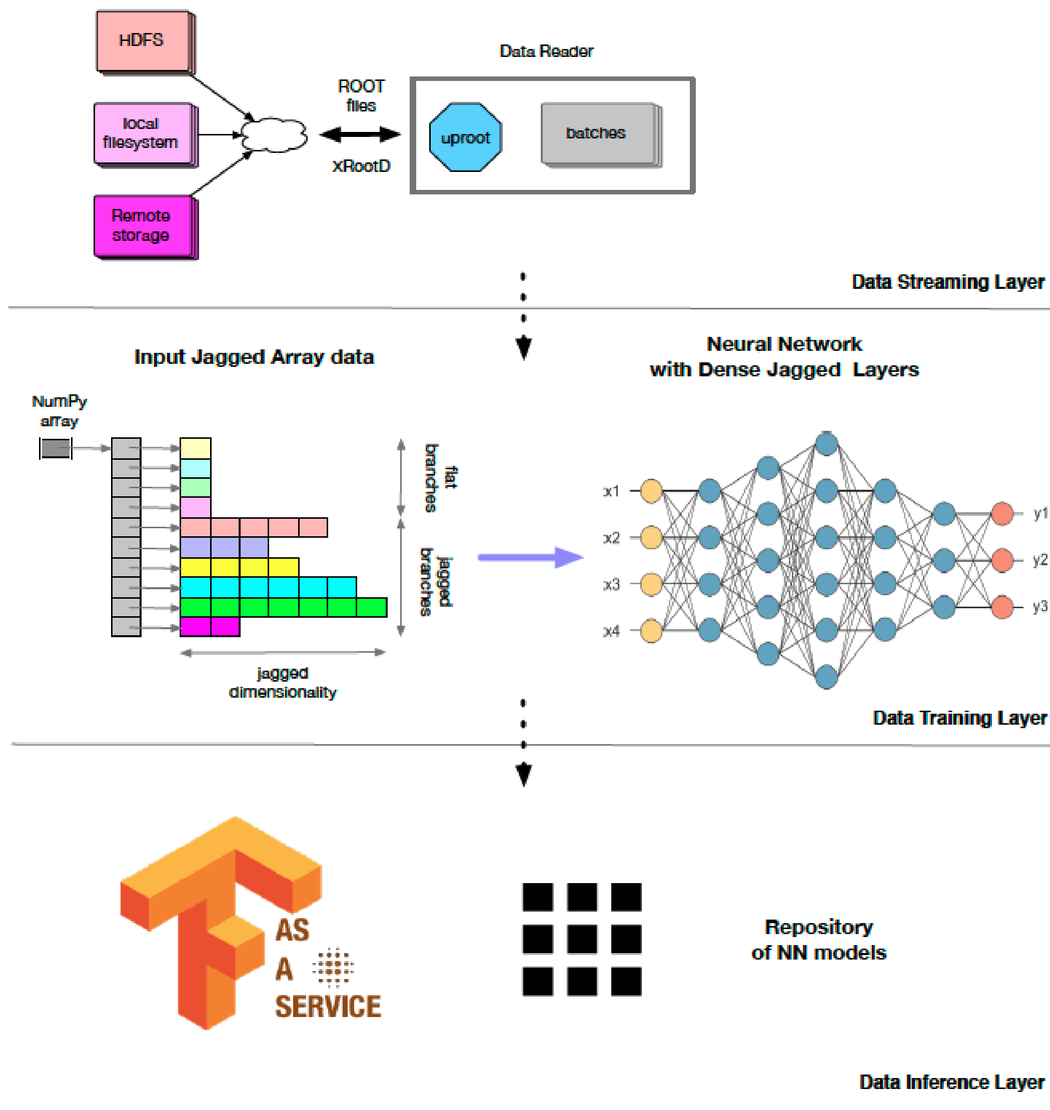

Figure 3.1: MLaaS4HEP architecture diagram representing the Data Streaming, Data Training and Data Inference layers.

In the following subsections, some details about MLaaS4HEP architecture and its performance will be provided.

3.3.1 Data Streaming Layer

Initially, the reading of ROOT file was possible only through the use of C++ [99] or PyROOT [100] frameworks, but the subsequent development of ROOT I/O allowed access to ROOT data directly from Python. The Data Streaming Layer takes care of streaming data from local and remote data storage and it has been developed using the uproot framework [101] supported by the DIANA-HEP initiative [102]. Uproot is an I/O library which allows to read and write ROOT files in pure Python and stream such data into ML libraries. It is independent from the ROOT toolkit and uses NumPy to cast blocks of data from the ROOT file as NumPy arrays. Also, uproot allows to read ROOT files remotely via XrootD protocol [103]. In order to correctly feed the chosen ML framework, the Streaming Layer is composed as a Python Generator, i.e. a function that behaves like an iterator and so can be used in loop, which is able to read ROOT data and deliver them as "chunks", i.e. a "set of events", to the Data Training Layer and which size is defined by the user. Separating the data into chunks provides efficient access to large dataset and and prevents the entire dataset from being loaded into the RAM of the training node.

The output of such Generator is a set of NumPy arrays with flat and Jagged Array attributes, shown in Fig. 3.2. In order to understand what a "Jagged Array" is, just consider a generic ROOT file: it has a tree structure composed by flat and non-flat branches. In flat branches, simple values are stored, such as integer or floating number, while in the non-flat ones vectors with variable dimensions are stored. Uproot converts flat branches into classical NumPy arrays, while the non-flat ones are converted to Jagged Arrays.

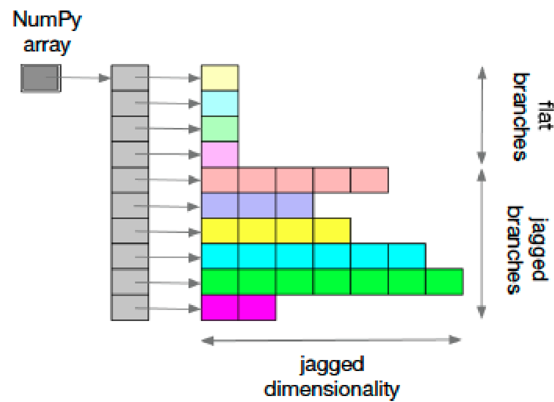

Figure 3.2: Output representation of the Python Generator.

3.3.2 Data Training Layer

The role of Data Training Layer is to process the output of the Python Generator and use it to train the ML model chosen by the user. The HEP tree-based data representation is optimized for data storage but, since it is not directly suitable for ML frameworks, a data transformation is required to feed such data structure into the ML framework. Therefore, it is necessary to convert the tree-data structure passed by the Data Streaming Layer into a flat one and, in order to do that, Jagged Arrays attributes need to be properly handled. As introduced before, Jagged Arrays are compact representations of event data, with variable size, produced in HEP experiments. Such representation needs to be converted into a flat representation and this has been done by following a two-step procedure.

The first step consists in scanning all the events in the ROOT file, determining, for each jagged branch, the respective minimum and maximum. The second step, instead, consists in updating the dimension of the Jagged Branch to the maximum one computed in the previous step, filling additional dimensions with padding values, see Fig. 3.3. These padding values are assigned as NaNs (Not a Number) to not affect the attribute spectrum and their position is saved in a separate masking vector, see Fig. 3.4, in this way it is possible to distinguish assigned padded values from real attribute values. Additionally, in this step, a proper normalization of each attribute is provided and it can be defined directly by the user. After this procedure, data are ready to be used in the training of the ML model.

In the next subsection, some details about the MLaaS4HEP workflow that combines the Data Streaming and Data Training layers will be covered.

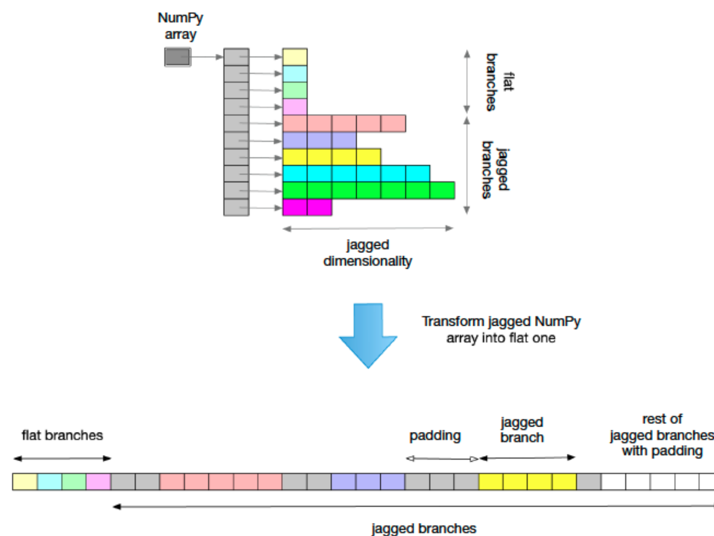

Figure 3.3: Representation of a Jagged Array vector with padding values.

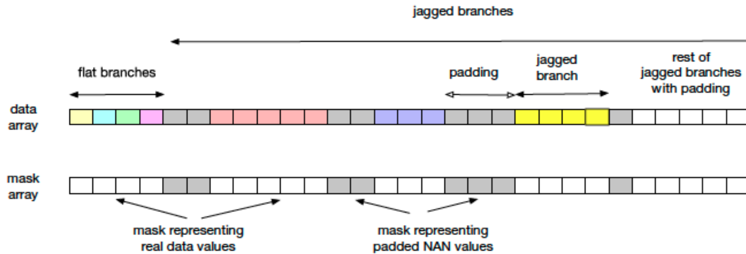

Figure 3.4: Vector representation of Jagged array with the corresponding mask array.

3.3.3 MLaaS4HEP Training Workflow

Both the Data Streaming and Data Training layers have been implemented by using the Python programming language and, together, contribute to the MLaaS4HEP Training Workflow, which consists in two different phases represented in Fig. 3.5. The first phase, denoted as ① in Fig. 3.5, represents the reading part of the MLaaS4HEP pipeline and consists of reading the input ROOT file chunk by chunk: each chunk, which size is defined by the user through the N_{chunk} parameter, is read by the Data Streaming Layer and, after the reading all chunks, the computation of the "specs file" is performed: this file contains all the useful information about the events in the ROOT files, such as minimum and maximum values for each branch and the maximum dimension of Jagged Branches. Since the reading of the input files in performed in chunks, the information in the specs file are continuously updated until the reading phase ends.

The second phase instead, denoted as ②, represents the training part and consists in a loop. The loop starts by reading N_{chunk} events from the i-th file f_i, which are stored into the i-th chuck c_i. Then, defining n_i as the number of events from file f_i and N_{tot} as the total number of events from all files, $N_{chunk} \cdot n_i/N_{tot}$ events are taken from c_i and, with the necessary precautions for the management of Jagged arrays they are converted in NumPy arrays. The reading of the events and their conversion is performed for all the files. Then, a chunk composed by N_{chunk} events from different files is created and used to train the ML model. The training part is performed by using "batches" of data taken from the created chunk and run for a certain number of epochs: also in this case, the size of the batch (N_{batch}) and the number of epochs (N_{epochs}) is fixed by the user. Then the loop starts all over again and continue, creating new chunks of events to use for the training, until all the ROOT files are completely read. When all the event from all files have been read and the training have been completed for all the epochs, at the end of the process a trained model is produced.

The discussed training procedure can be applied to a variety of use cases and its independent from the ML model and framework: the users can choose the desired framework and provide the definition of the model that wants to be trained.

The MLaaS4HEP Training Workflow is performed by running the *workflow.py* Python script which takes several argument as input.

```
./workflow.py --files=files.txt --labels=labels.txt --model=model.py
```

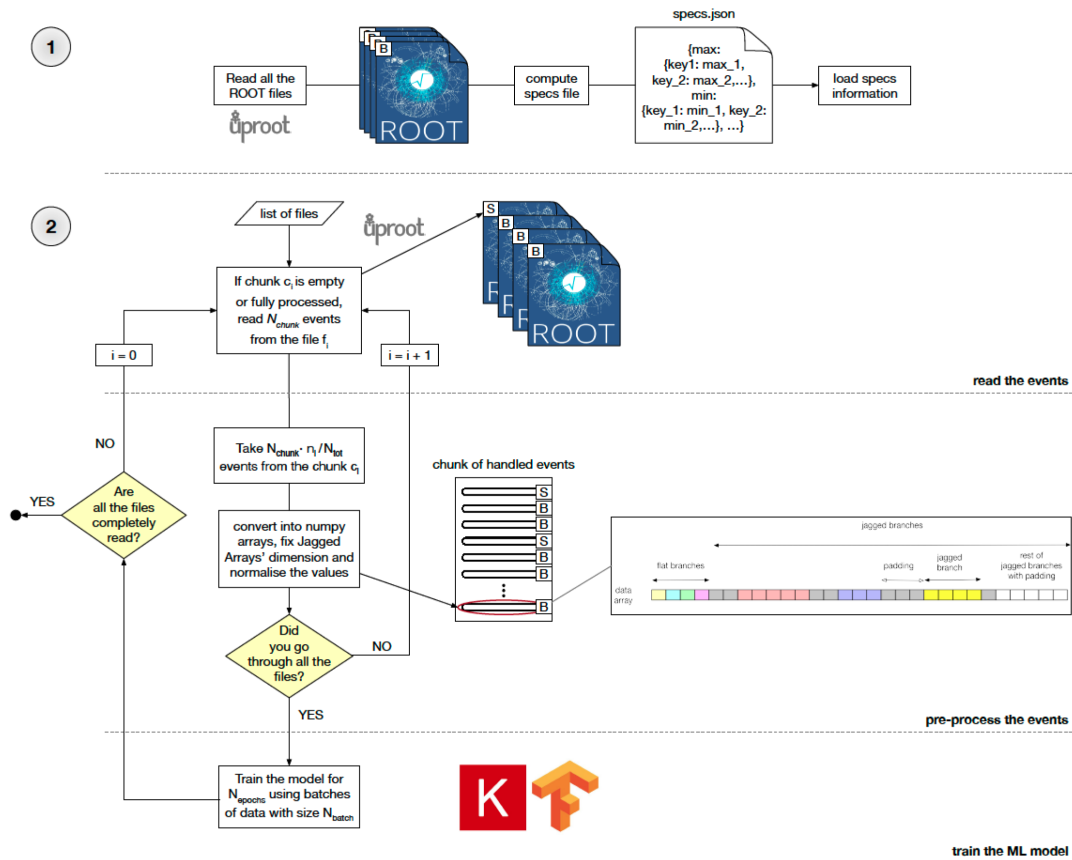

Figure 3.5: Schematic representation of the phases in the MLaaS4HEP Training Workflow.

```
--params=params.json
```

where:

- **files.txt** stores paths and names of the input ROOT files.
- **labels.txt** contains the labels of the respective ROOT files, it is used for classification problems.
- **model.py** contains the definition of the ML model chosen by user.
- **params.json** stores parameters on which MLaaS4HEP relies, such as chunk size, batch size, number of epochs and so on.

3.3.4 Data Inference Layer

The Data Inference Layer has been implemented as TensorFlow "as a Service" (TFaaS) [104]. Such solution can be easily integrated into cloud platforms and can be used as a repository of pre-trained TF model that can be served to the users for making predictions.

TFaaS is implemented using HTTP protocol that guarantees easy adaptation After the evaluation of different ML frameworks for the inference phase, it was decided to use TF. TF library provides APIs which allow to read TF models in different programming languages: between them, the Go programming language was chosen to implement the inference part of MLaaS4HEP. Go language was choose because it is very well integrated with the TF library and provides a final static executable which significantly simplifies its deployment on-premises and to various (cloud) service providers. The TFaaS framework can be used also outside of HEP to serve any kind of TF models uploaded to TFaaS service via HTTP protocol. With TFaaS the clients can test multiple TF models at the same time allowing a rapid development or continuous training of TF models, and their validation.

3.4 Validation and Performance

In order to test MLaaS4HEP performance and to validate its results from the physics point of view, it was decided to test the infrastructure on a real physics use case: the $t\bar{t}$Higgs analysis ($t\bar{t}H(b\bar{b})$) in the boosted, all hadronic final state [105].

3.4.1 The $t\bar{t}H$ analysis

The Higgs boson (H), discovered by ATLAS and CMS in 2012 at CERN, is considered the most relevant discovery of the last few years in HEP and, in order to measure its properties, different analyses have been performed. In the SM, H is predicted to couple with fermions via Yukawa-like interaction, where its coupling is proportional to the mass of the fermion. The top quark (t), the heaviest one, is responsible for coupling to H. The $t\bar{t}H$ production plays an important role in the study of the top-Higgs Yukawa coupling and its highest branching ratio, $\sim 25\%$, is represented by the all-hadronic decay channel. The W bosons produced by $t\bar{t}$ decay into a pair of light quark, while H decays into a $b\bar{b}$ pair (see Fig. 3.6). In the final state, at least eight partons are present and four of them are bottom (b) quarks. Despite the higher branching ratio (BR), the all hadronic final state is very challenging: it is dominated by large QCD multi-jet production and, due to the presence of many jets, there are large uncertainties in channel; on the other hand, since all decay products are observable, it represent the only way to fully reconstruct $t\bar{t}H$. With a 13 TeV center-of-mass energy, t quarks with very high p_T can be produced via $t\bar{t}H$.

The events of interest in this analysis are those with an all-jet final state in which the H decays into a pair of well resolved jets, identified as a result of the b quarks hadronization, and, additionally, at least one of the jets in the final state is a *boosted* jet, i.e a single and wide jet in which the decay products of top quarks are highly collimated, this happens if their Lorentz boost is high enough.

Therefore, for the identification of events containing a resolved-H decay, a ML model based on Boosted Decision Tree (BDT) was used by CMS in the analysis and the training was done within the TMVA framework.

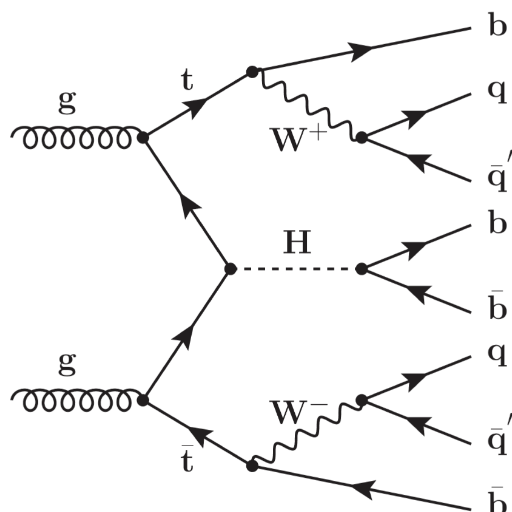

Figure 3.6: Feynman diagram for the $t\bar{t}H(b\bar{b})$ decay.

3.4.2 MLaaS4HEP Validation

In order to validate MLaaS4HEP functionalities and compare them with the "traditional" procedure based on ROOT and TMVA, a set of ROOT files from the resolved-Higgs analysis have been used. The idea was to demonstrate that the model trained within MLaaS4HEP framework obtains comparable results with the ones produced by the model from TMVA. In order to do that, it was decided to use a generic ML model consisting in a Keras Sequential NN composed by two hidden layers made by 128 and 64 neurons and with a 0.5 dropout regularization between layers. For the training part, the number of epochs and the batch size have been set to 5 and 100 respectively.

The analysis has been performed by using a set of ROOT file subjected to physical cuts in order to deal only with events that passed the selection criteria previously mentioned. Such set of files is composed by 8 ROOT files containing background events, and 1 file containing signal events. Each file has 27 flat branches, with a total number of events equal to 350k and a total size of about 28 MB. The ratio between the number of signal events and background events is approximately 10.8%. The analysis dataset have been splitted into training, validation and test set: 64% for training, 16% for validation and 20% for test purposes.

As described in the previously mentioned "MLaaS4HEP: Machine Learning as a Service for HEP" article, through this validation test it was possible to validate MLaaS4HEP, which brought consistent results with the TMVA-based approach.

3.4.3 MLaaS4HEP and TFaaS Performance

The MLaaS4HEP performance was tested using ROOT files without any physics cut, resulting in a dataset whit a total size of about 10.1 GB, containing about 28.5M events with 74 branches (22 flat and 52 jagged). The performance test was done by using two

different platform: a macOS, 2.2GHz Intel Core i7 dual-core laptop with 8 GB of RAM and a CentOS 7 Linux, 4 VCPU Intel Core Processor Haswell 2.4 GHz CERN Virtual Machine with 7.3GB of RAM. The ROOT files have been read from local file-systems (SSD storages) and from three remote WLCG data-centers, located in Bologna, Pisa and Bari. By fixing the chunk size to 100k events, the performance results shown in Table 3.1 have been obtained. Since the time needed to train the ML model depends on the ML model complexity and the available hardware resources - and so, it is independent from the MLaaS4HEP framework - it has not been included in the performance.

	event throughput	total time using all events
specs computing phase	8.4k - 13.7k evts/s	35 - 57 min
chunks creation in the training phase	1.1k - 1.2k evts/s	6.5 - 7.5 hrs

Table 3.1: Mean Values of event throughput and total time spent using all the events (28.5M) for the specs computing phase and for the chunks creation in the training phase. Values are reported as intervals, where the upper limit for the event throughput and the lower limit for the total time are reached using local files, while the opposite limits have been obtained using remote ROOT files.

Also, by projecting these results for much larger datasets (at the TB scale), it has been estimated that the specs computing phase and the training phase would take about 100 and 1k hours respectively. Consequently, further optimisation in the MLaaS4HEP pipeline are required to be able to process datasets of this size: such optimisation may include parallelization of I/O, distributed ML training and others.

Concerning TFaaS performance, it has been tested using a variety of ML model. In particular, several benchmarks have been performed, using a TFaaS server, running on CentOS 7 Linux, 16 cores, 30 GB of RAM: firstly 1k calls with 100 concurrent clients, and then 5k calls with 200 concurrent clients were tested. In both cases, the achieved throughput was $\sim$ 500 requests/sec. Such values have been obtained by serving a mid-size pretrained NN model written in Keras with 27 features and 1024×1024 hidden layers and a similar performance was found for the MNIST [106] classification problem. Actually, TFaaS performance depends, as in the MLaaS4HEP case, on the available hardware resources and on the complexity of the served ML model, and, in order to provide the desired throughput for concurrent clients, it can be easily horizontally scale by using Kubernetes [107] or other cluster orchestrated solution.

Chapter 4

New features in MLaaS4HEP and the Higgs Boson ML challenge use case

All operations performed on MLaaS4HEP during this thesis project, that will be covered in the following sections, were made using a local copy of the original MLaaS4HEP code [108], which is located on a GitHub [109] [110] repository. GitHub is a popular programming cloud-based resource used for code sharing, facilitating project management and collaboration. It allows multiple developers to work on a single project simultaneously and reduces the risk of conflicting or duplicative work. With GitHub, developers can build code, track changes, and innovate solutions to problems that might arise during the site development process simultaneously. In order to give my contribution to the project, the MLaaS4HEP repository was forked, i.e. a copy of the repository was created, on my personal GitHub page [111]. The forked repository was then subsequently cloned and saved locally, ready to be subjected to changes that do not affect the original repository. At this point, once the changes have been made and verified, it is possible to submit a pull request, i.e. demand to retrieve and merge the changes, to the code in the original repository. The changes and new features that will be described in the following will be uploaded to the official repository of MLaaS4HEP.

After describing the main changes made with the addition of new features in the code, we will describe the use of MLaaS4HEP in a particular use case: the Higgs Boson ML challenge proposed by ATLAS in 2014. The steps performed and the results obtained will be reported in Sec. 4.5 and, in the final part of this chapter, the future directions of this project will be described.

4.1 Towards uproot4

The main issue in manipulating HEP data using Python is related to the presence of "not rectangular" data, i.e. jagged arrays. In order to manipulate arrays with complex data structures used by uproot, the Awkward Array library (`"awkward0"`) has been developed by the Scikit-HEP project [112]. As described in Sec. 3.3.2, MLaaS4HEP

has its own way of handling jagged arrays and, in order to do that, it makes use of the Awkward Array library: if the array structure is variable, it will be recognized as jagged and subsequently flattened through the two-step procedure. In order to understand if an array is flat or jagged, MLaaS4HEP make use of method provided by the `awkward0` library.

The first version of awkward, which was written in pure Python, has been subsequently rebuilt with a more complex architecture: such operation gave rise to a new version of Awkward Array called `awkward1`. The new version has significantly changed the interface of the library, allowing to deal with a larger variety of nested and variable-sized data and to perform, with respect to the previous version, more complex manipulation operations on them. Because of the strict relation between the two libraries, such update led to a new version of uproot, i.e. uproot4. So, in order to take advantage of the new Awkward Array features and to maintain the code updated, it was decided to update the MLaaS4HEP code to the new uproot version.

With the update to uproot4, new methods and functionalities have been defined and the code was changed to use the new syntax adopted by uproot4. To show a change in syntax between the two versions, the following example is given: in order to make distinction between flat and jagged branches, uproot3 use the `JaggedArray` type from `awkward0` but, since this method has been removed in the new version, uproot4 uses the type `AsJagged` to understand the nature of the branch. Within the code, the distinction between branches is performed using:

```
isinstance(branch, awkward0.JaggedArray) #uproot3
isinstance(branch, uproot4.AsJagged) #uproot4
```

Since some functions have been subject to changes, the execution times and the event throughput of the main components of the MLaaS4HEP workflow were measured in order to verify if there is any improvement in time by adopting uproot4. As already mentioned in Sec. 3.3.3, two main phases can be identified in the MLaaS4HEP workflow: reading with specs computation and training phase, where the latter contains the phase of handling the chunks and the ML model training. Since the ML model training part is totally independent from MLaaS4HEP itself, only the performance of the reading with specs computing phase and the handling one were measured. To do it, the same ROOT files used to test the MLaaS4HEP performance discussed in Sec 3.4.3, with a total size of 10 GB, were used. The performance test was done by using a macOS, 2.9 GHz Dual-Core Intel Core i5 laptop with 8 GB of RAM. By setting the chunk size to 100k events and the number of total events to be read equal to 500k, MLaaS4HEP has been executed for 5 runs and the average of the timing and throughput values related to the two phases and the total running time has been calculated. The values obtained from this measurement are reported in the Table 4.3.

Phases	u3 time	u4 time	u3 throughput	u4 throughput
Reading + Specs	199 ± 9 s	233 ± 7 s	15109 ± 300 evt/s	12895 ± 124 evt/s
Creating a chunk	93 ± 11 s	73 ± 17 s	1092 ± 123 evt/s	1424 ± 290 evt/s

Table 4.1: Measured time and throughput values for the main steps of the MLaaS4HEP workflow considering the two uproot versions. The results were obtained by running MLaaS4HEP 5 times with a chunk size equal to 100k and a total number of events to be read equal to 500k and averaging the obtained times and throughputs.

By measuring the time spent on the executing the line of code shown previously for the two versions of uproot, it was found that, in the uproot3 case, it took on average 1.334×10^{-6} seconds for each event, while in the uproot4 it took case 1.360×10^{-5} seconds. Therefore, there is a difference of about one order of magnitude between the two versions that leads to a faster reading with specs computing phase in the old version of MLaaS4HEP rather than in the new one.

Instead, for what concerns the handling phase, the table shows that MLaaS4HEP with uproot4 runs faster. The time difference between the two versions is due to a greater speed by uproot4 in reading the single events: for each event, the new version takes about 10^{-5} seconds, while the old one takes about 10^{-4} seconds, resulting in a difference of an order of magnitude. In the end turns out that the total runtime of MLaaS4HEP using uproot4, with an average time of 640 ± 17 s, is lower than the uproot3 case, with an average time of 700 ± 25 s.

4.1.1 Preprocessing operations

After updating the code and checking that data were handled correctly, adopting the new versions of the libraries it was possible to implement new functions within the MLaaS4HEP code that allow to perform preprocessing operations in case the user needs it. In particular, when we talk about preprocessing operations, we refer to operations that allow the users to manipulate data, i.e.:

- new branches definition,
- application of cuts on branches, both new and existing ones,
- removal of branches that may not be useful for model training.

To allow the users to exploit such functionalities, it was decided to introduce a `"preproc.json"` file where the user specifies which operations must be used. This file must be passed as argument to `workflow.py` script as shown below:

```
./workflow.py --files=files.txt --labels=labels.txt --model=model.py
--params=params.json --preproc=preproc.json
```

```
{
 "new_branch": {

    "log_partonE": {
       "def": "log(partonE)",
       "type": "jagged",
       "cut_1": ["log_partonE<6.31", "any"],
       "cut_2": ["log_partonE>5.85", "all"],
       "remove": "False",
       "keys_to_remove": ["partonE"]},

    "nJets_square": {
       "def": "nJets**2",
       "type": "flat",
       "cut": "1<=nJets_square<=16",
       "remove": "False",
       "keys_to_remove": ["nJets"]}},

 "flat_cut": {
    "nLeptons": {
    "cut": "0<=nLeptons<=2",
    "remove": "False"}},

 "jagged_cut": {
    "partonPt": {
    "cut": ["partonPt>200", "all"],
    "remove": "False"}}
 }
```

Figure 4.1: Example of a `preproc.json` file provided a by the user to MLaaS4HEP to apply preprocessing operations on ROOT file branches.

In the following, an example of `preproc.json` is provided in Fig. 4.1.

In order to correctly perform the operations previously described, the `preproc.json` [113] file has been grouped in three main categories:

- `"new_branch"`,
- `"flat_cut"`,
- `"jagged_cut"`.

The `"new_branch"` category allows the users to define new branches and, if required, it allows also to apply cuts on them. Instead, concerning the application of cuts on existing branches, the "flat" and "jagged" categories are used depending on the nature of the branch: it was necessary to divide the flat branches from the jagged ones because the way in which the cuts are applied changes depending on the type of branch. Moreover, if the user does not need one of these components for his/her purposes, unused components can be removed from the `preproc.json`. The three categories that compose the preprocessing file will be described in detail below.

`"new_branch"`

A new branch can be defined through mathematical operations involving existing branches. To create new branches the user should provide under the `new_branch` category a series of information in the following keys.

- `"new_branch_name"`: name of the new branch that the user wants to define;
- `"def"`: the mathematical definition of the new branch, which support basic operation, such as addition (+), product (*) or exponentiation (**), and also more complicated ones which could be performed using NumPy functions [114];
- `"type"`: type of the new branch, must be either `"flat"` or `"jagged"`;
- `"cut"` (optional): cut condition to be applied to the new branch;
- `"remove"`: if setted as `true`, allows to remove the new branch before the training phase begins;
- `"keys_to_remove"` (optional): list of string containing the name of existing branches to be removed.

Cuts are defined slightly differently depending on the nature of the new branch: if the new branch is flat, the cut condition is enclosed within a string, while, if jagged, it is necessary to specify the type of cut you want to apply. To clarify, because of its structure, jagged branches support two types of cuts: `"all"` [115] and `"any"` [116]. The `"all"` type is used when the cut condition must be satisfied by all values of a given jagged branch, while the `"any"` type when at least one element in a given jagged branch satisfies the cut condition.

Since for jagged branches it is necessary to specify the type of cut to be used, the cut condition is enclosed in a list in which the first element is given by the string defining the cut, while the second is given by the string containing the cut type, i.e. `"all"` or `"any"`. The cut condition has to be written in the following order:

```
new_branch_name + operator + numerical_value
```

where `operator` includes `==`, `!=`, `>=`, `<=`, `>` and `<`. The only exception is if the user is dealing with a flat branch and he/she wants it to be included between two values `a` and `b`. In this case it is possible to define the condition as

```
a </<= branch_name </<= b
```

with `a<b`.

In case the user wants to apply more than one cut, it is possible to do so by numbering the cuts using `cut_i` as key, where `i` indicates the number of the cut.

If the user does not want to apply cuts on the new branch or does not want to remove any branch, then the `cut` and `keys_to_remove` keys should be removed respectively from the `preproc.json` file.

`"flat_cut"` and `"jagged_cut"`

The structure of the two categories `"flat_cut"` and `"jagged_cut"` is very similar to the one described above. To apply cuts on flat and jagged branches, the following keys must be provided.

- `"existing_branch_name"` : name of the existing branch;
- `"cut"` (optional): cut condition to be applied;
- `"remove"` : allows to remove the existing branch before the training phase begins.

In contrast to the previous case, `"flat_cut"` and `"jagged_cut"` structure presents only two keys, `"cut"` and `"remove"`. Both keys have the same role described in the previous section, so the only difference between the two categories lies in the cut condition: `"flat_cut"` takes a string as value of the `"cut"` key, while `"jagged_cut"` takes a list where the user need to specify the condition and type of cut. Also for these two category, is possible to apply more than one cut within the same branch just numbering the cut key using `cut_i`.

The proper functioning of the preprocessing operations was tested using the same ROOT files used in Sec. 4.1 and performing the same operations both on MLaaS4HEP and on the ROOT software, which both led to the same results. The comparison on how to apply cuts over events between the uproot and ROOT approaches is shown in the Appendix B.

As far as the performance of these preprocessing operations is concerned, the results are not optimistic compared to what was achieved when switching to uproot4. In the case of preprocessing operations there is a strong performance degradation. In particular, MLaaS4HEP performance drops when new branches or cuts on jagged branches are defined. Because of this, only a part of the ROOT files described in the previous performance evaluation were used and, also, the chunk size, total number of events to be read and epochs were changed to 5k, 25k and 3 respectively.

For this test, 5 different cases were considered: in the first one, the preprocessing functions were not executed, in the second, third and fourth case a cut on flat branch, a cut on jagged branch and the definition of a new jagged branch were performed respectively. In the last case, denoted as “mixed”, we considered the case with all three components present, also applying a cut to the new feature. The `preproc.json` file provided for the last case is shown in Listing 4.1.

The averaged values of the timing and event throughput for the two phases of the MLaaS4HEP workflow are shown in the table.

Phases	No Cut	Flat Cut	Jagged Cut	New Branch	Mixed
Reading + Specs	9861 ± 837 evt/s	6121 ± 833 evt/s	556 ± 33 evt/s	546 ± 41 evt/s	439 ± 41 evt/s
Creating a chunk	1459 ± 209 evt/s	1330 ± 139 evt/s	108 ± 14 evt/s	99 ± 11 evt/s	98 ± 10 evt/s

Table 4.2: MLaaS4HEP event throughput for five different cases considering the application of preprocessing operations.

Phases	No Cut	Flat Cut	Jagged Cut	New Branch	Mixed
Reading + Specs	5.1 ± 0.4s	8.3 ± 1.2 s	90 ± 5 s	92 ± 6 s	115 ± 11 s
Creating a chunk	3.5 ± 0.8 s	3.8 ± 0.4 s	47 ± 6 s	51 ± 5 s	52 ± 6 s

Table 4.3: MLaaS4HEP preprocessing timing values for five different cases, i.e. no cut application, application on a flat branch, application on a jagged branch, definition of a new branch and application of cuts both on flat, jagged and new branch.

```
{
  "new_branch": {
    "log_partonE": {
      "def": "log(partonE)",
      "type": "jagged",
      "cut_1": [
        "log_partonE<6.31",
        "any"
      ],
      "cut_2": [
        "log_partonE>5.85",
        "all"
      ],
      "remove": "False"
    }
  },
  "flat_cut": {
    "nLeptons": {
      "cut": "0<=nLeptons<=2",
      "remove": "False"
    }
  },
  "jagged_cut": {
    "partonPt": {
      "cut": ["partonPt>200", "any"],
      "remove": "False"
    }
  }
}
```

Listing 4.1: `preproc.json` provided for the application of cuts on flat, jagged and new branch while testing MLaaS4HEP preprocessing performance.

The reason for this behavior is essentially related to a conversion issue: by applying cuts to flat branches, the same data type, i.e. Numpy ndarray [117], is maintained for the whole duration of the MLaaS4HEP workflow. However this is no longer true when defining new branches or applying cuts to jagged branches, because data are converted to a type, i.e. AwkwardArray, which is not supported by the ML algorithms and therefore a further conversion, passing through each event, is required and causes this slowdown. As can be inferred from the values reported in these tables, if the user needs to perform

complex operations, such as creating new features or cutting jagged branches and looking for good time performance, it is preferable to make this outside of MLaaS4HEP, especially when dealing with very large data. However, if simple operations such as flat branch cuts are required, the performance is very similar to the case without preprocessing.

4.2 Improvements on the training method

The current MLaaS4HEP training procedure, when a NN model is chosen, is performed chunk by chunk where each chunk is used to train the model for n epochs, with n defined by the user. It has been introduced an additional training procedure, the standard one, where each epoch is performed using all the chunks. Then the training continues for n times. The original MLaaS4HEP approach to train a ML model in chunks is useful when the dataset used is large and exceed the amount of RAM of the training node. This allows to train the ML model each time using a different chunk, until the entire dataset is completely read. In this case the user should pay close attention to the ML model convergence, and validate it after each chunk [118] [119] [120]. Using a different training approach has pros and cons. For instance, training on entire dataset can guarantee the ML model convergence, but the dataset should fits into RAM of the training node. The first approach allows to split the dataset to fit in the hardware resources, but it requires proper model evaluation after each chunk training. In terms of training speed, this choice should be faster than training on the entire dataset, since after having used a chunk for training, that chunk is no longer read and used subsequently (this effect is prominent when remote ROOT files are used). Thus the user can decide which approach to use and decide, depending on the use case, what is better. To make a proper comparison in the use case shown before, in Fig 4.2 the loss and AUC obtained at the end of the MLaaS4HEP training procedure are plotted, for the training, validation and test set and for both procedures (the original MLaaS4HEP approach and the standard one), using SGD as optimisers, respectively.

The plots in Fig. 4.2a and 4.2b are obtained using a chunk size equal to 100 events, while the plots in Fig. 4.2c and 4.2d are obtained using a chunk size equal to 100k events. It is possible to see how the two MLaaS4HEP approaches produce very similar results when the chunk size is 100k, while with 100 events as chunk size the standard approach shows evident improvements in the performance, with a smooth trend and a better learning. This means that in the original MLaaS4HEP training approach large chunks are enough for the model to learn from data with a training in n epochs for each chunk. But when the chunk size is decreased, a training for n epochs using a chunk and then move to another one results in a loosing of performance while on the contrary the model benefits from using a procedure that foresee to see all chunks for each epoch. Thus using chunks with a smaller size results in a more difficult convergence for the original MLaaS4HEP training apporach. This behaviour of course depends on the use case, but when two classes are difficult to be distinguished it is always preferred to use small chunks in order to guarantee a proper learning of the model.

At the same time the user should consider the timing to perform all the MLaaS4HEP workflow and choose a trade off between the time spent and the performance. So it

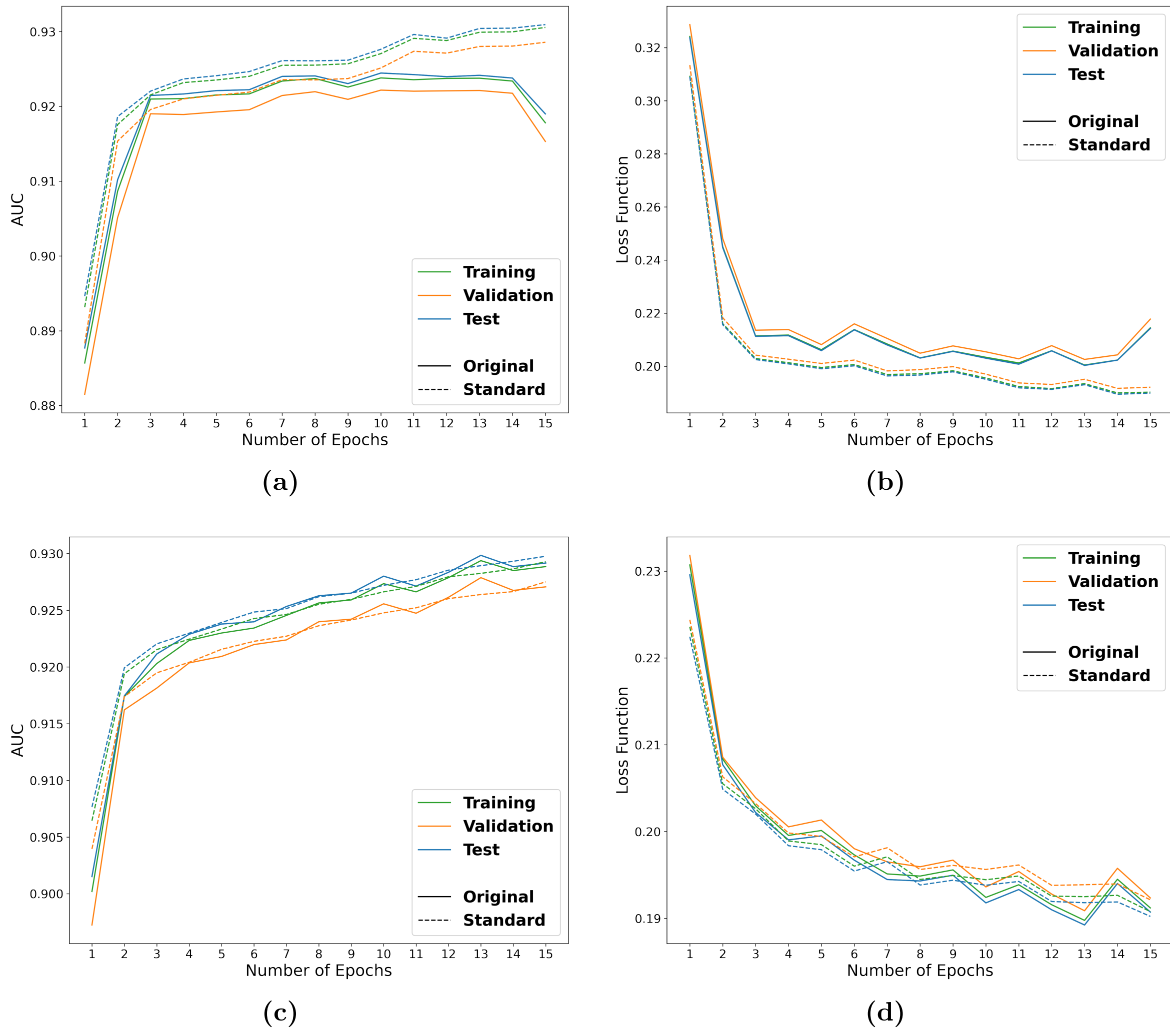

Figure 4.2: AUC and loss metrics comparison between original and standard training approaches using an SGD optimizer with learning rate equal to 0.01. The plots in **(a)** and **(b)** were obtained setting the chunk size equal to 100, while in **(c)** and **(d)** the chunk size has been set to 100k.

would be useful to plot the time spent by MLaaS4HEP in function of the number of epochs used for the training, as shown in Fig 4.3. In 4.3a the values are obtained fixing the chunk size to 100 events, while in Fig 4.3b the chunk size was 100k. In both cases the original MLaaS4HEP approach takes much less time than the standard one since each chunk is read only one time and then discarded. Then choosing 100 events as chunk size shows that the time is an order of magnitude greater than choosing 100k events as chunk size for both approaches. Thus the user should take into consideration all these elements before choosing which approach to adopt.

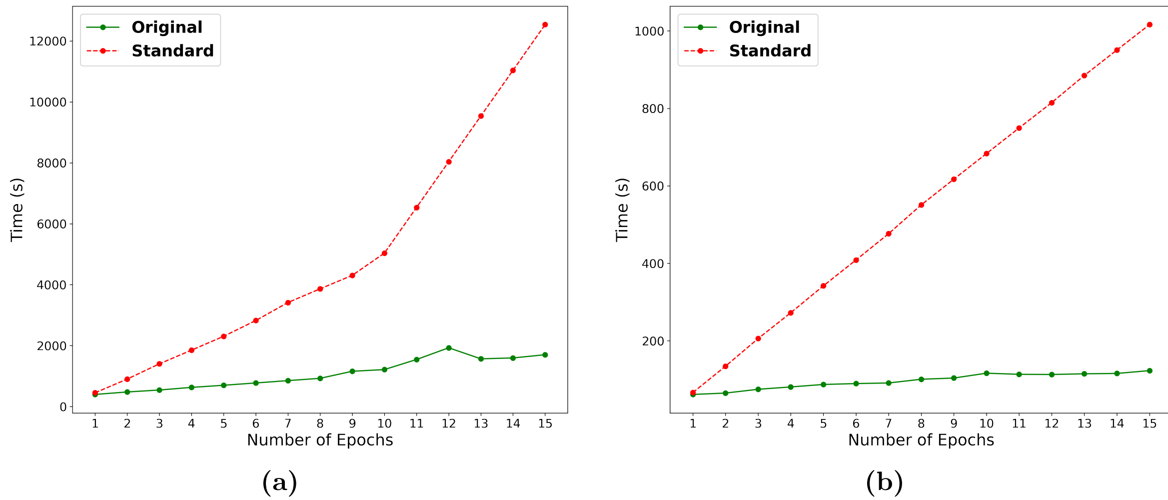

Figure 4.3: Timing comparison between original and standard training approaches considering 100 and 100k as chunk size. **(a)** refers to the case where the chunk size has been set to 100 and **(b)** refers to a chunk size of 100k.

4.3 Generalization to other Frameworks

As described in Chapter 3, MLaaS4HEP has already been tested using a Sequential NN written in Keras. However, among its main objectives, MLaaS4HEP aims to be ML framework agnostic, which means the user can decide to use any ML framework and algorithm, and, in order to get closer to this goal, some changes have been made in order to support two other frameworks: PyTorch and Scikit-Learn. In order to correctly integrate these two frameworks, it has been necessary to understand how the training of the model is performed and subsequently apply and integrate the appropriate changes in the MLaaS4HEP code. In the following we describe the integration of these two frameworks and the results obtained by testing MLaaS4HEP with new models. But before a brief overview of how the model training is performed within MLaaS4HEP using the Keras framework is given.

4.3.1 MLaaS4HEP with Keras

In order to properly train any ML model, the ability to read the data in chunks and mix it accordingly in each training batch is needed, especially if you are dealing with very large datasets. MLaaS4HEP, which supports this capability, allows the user to define the chunk size, ensuring that for each chunk the correct proportion of signal and background events presented in the ROOT files is maintained. Then, by passing one chunk after another to the model, the training phase is incrementally and efficiently performed. As example, considering the aforementioned Sequential NN, defined in Appendix A.1, by setting the chunk size to 10k events and the number of epochs to 5 and using the ROOT file mentioned in 3.4.2, the loss, Accuracy and AUC metrics have been computed.

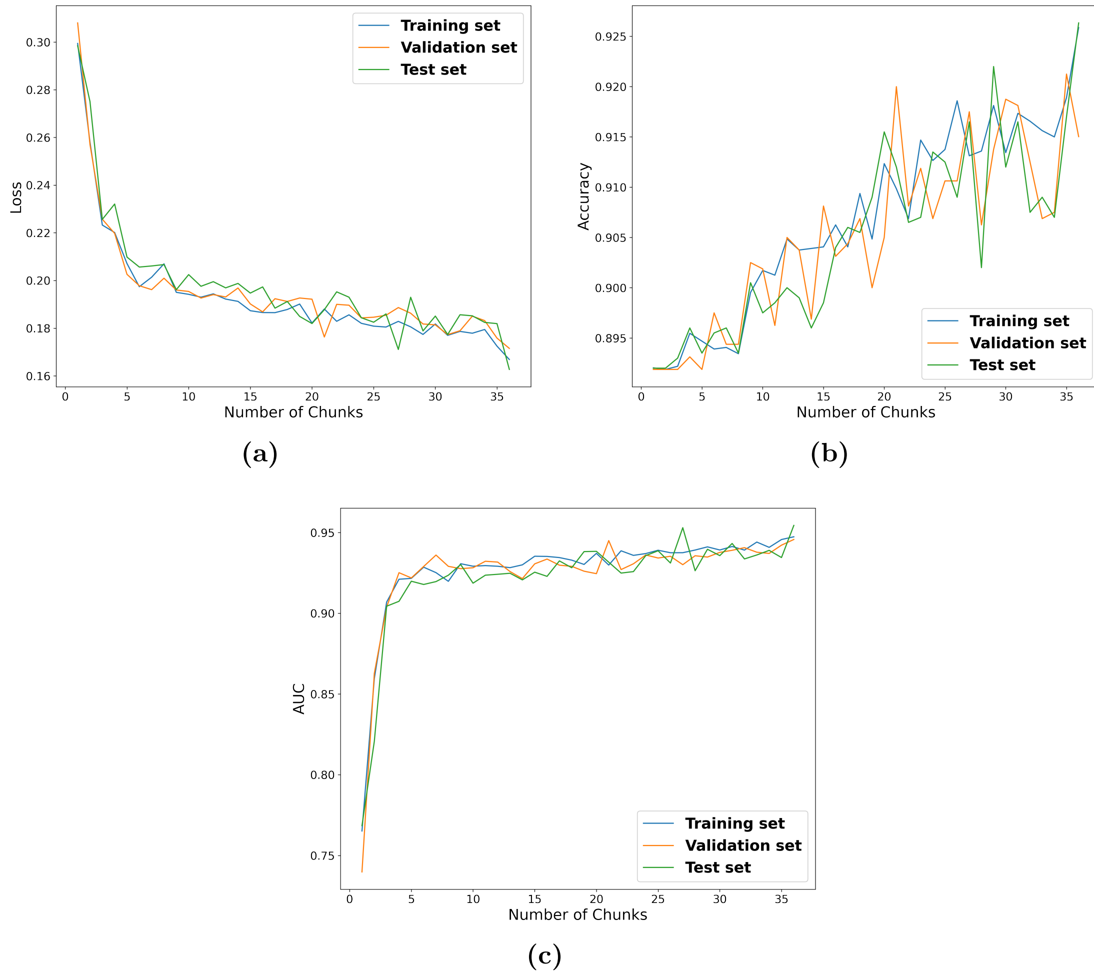

Figure 4.4: Comparison of the metrics score (loss **(a)**, Accuracy **(b)** and AUC **(c)**) for training, validation and test set using all the events in the files, read with a chunk size of 10k events.

As shown in Sec. 3.3.3, MLaaS4HEP allows to perform the training procedure in chunks. To check how the training is going chunk by chunk, we considered the aforementioned Sequential NN in Keras using the ROOT file mentioned in 3.4.2, and we plotted in Fig 4.4 the loss, Accuracy and AUC metrics: the Accuracy and AUC curves go up, while the losses decrease with the number of chunk and this indicates that the ML is actually learning from the data.

4.3.2 PyTorch implementation

As introduced in Sec. 2.5, PyTorch is an open-source library used in ML, developed using the Torch library. PyTorch popularity is due to several aspects, including its flexibility, computation power, its structure, which is very similar to that of traditional programming, and its data parallelism feature, which allows to do parallel processing with a strong GPU support. Pytorch data representation has many similarities with

NumPy: PyTorch equivalent of NumPy ndarray is called Torch tensor [121]. Tensors is a specialized data structure very similar to array and matrix, they are used to encode the input and output of models.

PyTorch has many differences compared to Keras, both in the preparation of the data and in the definition, training and evaluation of the ML model. The most common way to use a PyTorch model is to provide a class definition of it, containing the structure of the model and how it must be trained: this can be done easily by using PyTorch `nn.Module` class [122] which provide an ensemble of essential method to build a NN. So the user must be provide information at low level, differently compared to Keras where the information provided are at high level.

The MLaaS4HEP code has been updated in order to properly convert data into Tensors and to accept a PyTorch model, defined in model.py, passed to the workflow.py script. In this file, the user can define a custom method for training the model. If no training method is provided, a default one that we have already implemented is selected.

The correct integration of PyTorch was verified by building a fully connected Feed Forward NN, which description is presented in Appendix 4.3.2, consisting of two hidden layers made of 16 nodes each. As activation function for the hidden layers, ReLU was chosen, while, for the output layer, the Sigmoid function was chosen.

The data used to train the model are the same ROOT files used in the MLaaS4HEP validation described in Sec. 3.4.2. For what concerns the training, the function that regulates this process can be defined within the class by the user, otherwise the default training function will be executed.

To verify that the training process worked correctly, a tests were performed by setting the chunk size, the number of epochs and the batch size respectively to 25k, 5 and 100, and using the default training function. The AUC and loss values were plotted as a function of the number of chunks both for training and validation set. As shown in Fig. 4.5, after each chunk the AUC score improves while the loss decreases: such behaviour represents a model that is learning correctly.

4.3.3 Scikit-Learn implementation

Scikit-Learn (or `sklearn`) is one of the most used library for ML, it is mostly written in Python and is built upon NumPy, SciPy and Matplotlib. Such framework provides tools for ML and statistical modeling which can be used in a vast range of use cases e.g. facial recognition, image classification, anomaly detection and so on. While frameworks like Keras or PyTorch are mainly used in DL and for building NN, Scikit-Learn is a more general approach to ML and, since it is not intended to be used as a DL framework, it does not provide any GPU support. Scikit-Learn is a higher-level library that includes implementations of several ML algorithms, so it allows to define a model object in a single or a few lines of code. Given its power and simplicity, it was decided to also integrate this framework within MLaaS4HEP.

In this case it was not necessary to provide a training function in the model definition, unlike the PyTorch case, since Scikit-Learn provides a proper method, i.e. `fit`, to perform training on its algorithm.

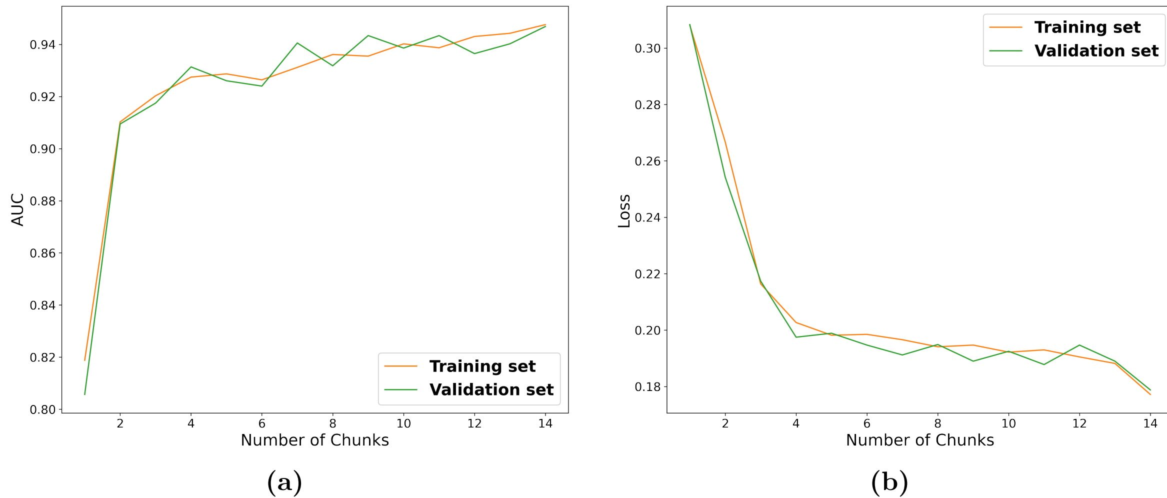

Figure 4.5: Trend of AUC **(a)** and loss **(b)** scores as a function of the number of chunks considering the default PyTorch training function.

However, there are some aspects of Scikit-Learn that require a different management of the training phase. Recalling what has been said previously, in MLaaS4HEP the training phase takes place by passing to the model one chunk of events at a time, allowing to access efficiently large datasets and preventing loading them entirely into the RAM of the training node. Such approach, called *incremental learning*, is not applicable to the vast majority of Scikit-Learn algorithms, that need to receive all the data at once. The only exceptions among Scikit-Learn algorithms are the ones that support the `partial_fit` API [123]. Consequently, in classification problem, if the model used is not among those in the above list then it is necessary to read all the data at once and pass them to the model for the training phase. Because of this, a new option has been added to MLaaS4HEP that allows the user to read all the events in a single chunk by setting the chunk size equal to -1 inside the `params.json` input file.

Once the model has been defined by the user, MLaaS4HEP will check if the model supports the `partial_fit` API: if so, the model will be trained incrementally chunk by chunk, otherwise the "classic" fit will be used.

To verify the correct integration of Scikit-Learn, three different algorithms from this framework, which definition is reported in Appendix A.3, were tested, two of which support the `partial_fit` method:

- SGDClassifier [124]: a linear classifier with SGD learning;
- MLPClassifier [125]: is the MLP supported by Scikit-Learn,
- AdaBoost [126]: an ensemble boosting algorithm which fit a sequence of weak learners on repeatedly modified versions of the data.

The tests were performed with the same data used previously and also in this case the AUC was used as a metric to judge models performance. Since SGDClassifier and

MLPClassifier support the `partial_fit` method, the training of these models was done incrementally by setting the chunk size to 25k, while for AdaBoost the chunk size was set to -1, thus passing to the model all the events contained in the ROOT files through a single chunk. The AUC values obtained by each model after being trained with each chunk have been calculated both for the training and validation sets and are reported in Table 4.4.

Model	**Number of Chunk**	**AUC on Training Set**	**AUC on Validation Set**
SGDClassifier	1	0.751	0.744
	2	0.772	0.782
	3	0.778	0.764
	4	0.771	0.764
	5	0.778	0.779
	6	0.770	0.789
	7	0.778	0.781
	8	0.775	0.773
	9	0.776	0.778
	10	0.781	0.774
	11	0.783	0.780
	12	0.779	0.765
	13	0.785	0.782
	14	0.796	0.773
MLPClassifier	1	0.553	0.542
	2	0.691	0.720
	3	0.745	0.731
	4	0.750	0.742
	5	0.763	0.764
	6	0.770	0.790
	7	0.777	0.780
	8	0.790	0.786
	9	0.799	0.806
	10	0.824	0.821
	11	0.841	0.843
	12	0.860	0.854
	13	0.876	0.879
	14	0.898	0.882
AdaBoost	1	0.934	0.932

Table 4.4: AUC scores related to different Scikit-Learn models: SGDClassifier, MLPClassifier and AdaBoost.

As shown in the table, incremental training leads to a higher AUC value as the number of chunks increases. However, between the two algorithms, it can be noted that the performance of MLPClassifier seems to be better than the one of SGDClassifier. The aim of this test was to be able to complete the MLaaS4HEP workflow using different algorithms belonging to the Scikit-Learn framework regardless of their performance; for this reason, we used default parameters and no hyperparameter tuning has been performed which, as previously introduced, can significantly change and improve the learning of a model. On the other hand, it is possible that SGDClassifier may not be the ideal algorithm for the type of problem we are facing. Instead, as regards AdaBoost, despite the absence of incremental learning, it achieved the best performance among the three models, thus making it the most suitable model for this classification task.

4.3.4 Providing useful metrics score

Since metrics are used to quantify the learning abilities of the models, it may be useful using MLaaS4HEP to have a general overview of the most common metrics. To provide such information, the new method called `performance_metric` has been defined in the code, regardless of the used ML framework. In particular, it was decided to provide the following metrics:

- AUC,
- Confusion Matrix,
- Classification Report.

AUC and Confusion Matrix metrics have already been covered in Sec. 2.3.2. The classification report [127] [128], as shown in Fig. 4.6, build a text report showing the main classification metrics provided by Scikit-Learn, including Precision, Recall and F1. It also provides the Support tool which specifies the number of actual occurrences of the label in the dataset.

The `performance_metrics` function prints the values related to the previous metrics both on the training and on the validation set for each chunk and has been implemented in such a way that it is the user who decides whether to execute this function or not: if the user is interested in obtaining the scores of these metrics, it is necessary to specify this request in the `params.json` file, by inserting the key "metrics". Moreover, an additional parameter was included within this key to obtain these scores: the threshold. If we consider, for example, a NN, the activation function connected to the output layer will map any variable in a range of values between 0 and 1. So, for binary classification problems, it is necessary to set a threshold which divides the values into two categories: values below the threshold are assigned to the 0 category otherwise to the 1 category. This threshold is usually set to 0.5 by default, but it may not always be the best solution and, for this reason, the user is given the opportunity to define it.

```
Classification Report:

              precision    recall  f1-score   support

         0.0       0.89      1.00      0.94     42817
         1.0       0.39      0.01      0.02      5183

    accuracy                           0.89     48000
   macro avg       0.64      0.50      0.48     48000
weighted avg       0.84      0.89      0.84     48000
```

Figure 4.6: Example of Classification Report. In addition to precision, recall, F1 and support, it also provides reported averages, which include macro average (averaging the unweighted mean per label), weighted average (averaging the support-weighted mean per label), and, only in the multilabel classification case, sample average.

4.4 Validation of new features

As shown in the previous sections, several changes have been made within the MLaaS4HEP code with the aim of making this service generalized and with new features. In this section, we will discuss the validation of the transition to uproot4 and the addition of the preprocessing operations using the same set of ROOT files and the same ML model seen in Sec. 3.4.2. Moreover, the AdaBoost model of Scikit-Learn was used to validate the results obtained from a framework different from Keras. The entire dataset, made up of about 350k events, was divided into training (64%), validation (16%) and test set (20%). To verify the consistency of the results obtained by the new version of MLaaS4HEP, two approaches were compared:

- use MLaaS4HEP to read and normalize events, perform preprocessing operations (if required) and for model training,
- perform the entire pipeline using a Jupyter Notebook, outside the MLaaS4HEP context. Here, the `MinMaxScaler` [129] method from Scikit-Learn was used for the normalization phase.

The NN was trained by receiving a single chunk containing all the events of the training set and the loss, Accuracy and AUC metrics scores for 10 epochs has been computed and subsequently plotted for both approaches, thus testing the correct transition to uproot4. The values obtained using uproot4 without preprocessing operations are shown in Fig 4.7.

To validate the code with preprocessing operations, cuts were applied to the data via the preproc.json file. Then, the same operations were then performed in the Jupyter Notebook using the pandas [130] library. The results for the three metrics are shown in Fig 4.8, while the performed cuts are reported Listing 4.2.

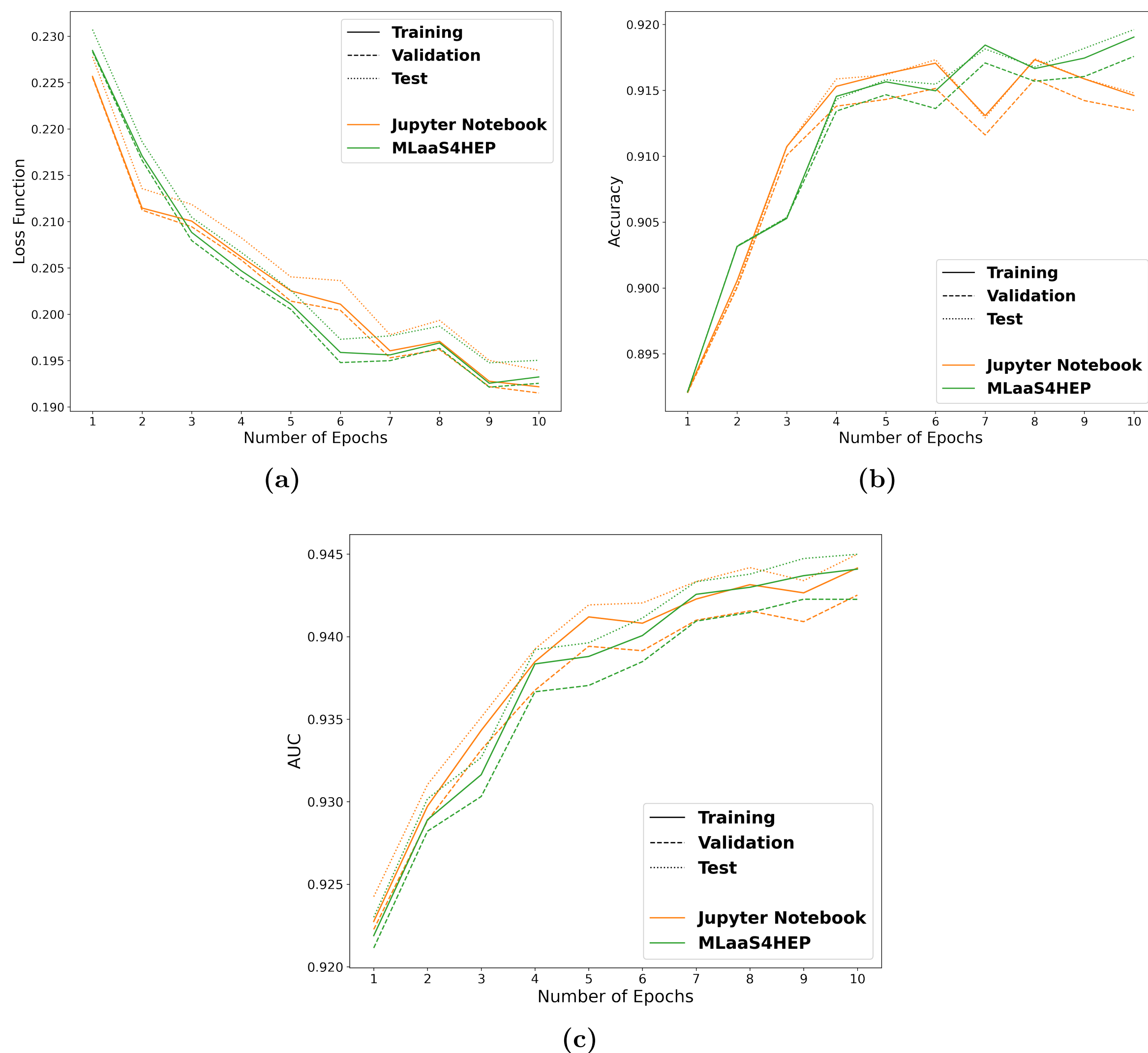

Figure 4.7: Comparison of the metrics score (loss **(a)**, Accuracy **(b)** and AUC **(c)**) obtained in the training, validation and test set. To test the correct transition of MLaaS4HEP to uproot4, two different approaches were considered: (1) using MLaaS4HEP to read and normalize events, and to train the ML model; (2) using a Jupyter Notebook to perform the entire pipeline without using MLaaS4HEP.

```
{
  "flat_cut": {
    "nJets": {
      "cut": "nJets<=3",
      "remove": "False"
    },
    "tau1": {
      "cut": "0.2<=tau1<=0.5",
      "remove": "False"
    },
    "nBJets": {
      "cut": "nBJets!=2",
      "remove": "False"
    }
  }
}
```

Listing 4.2: Example of preproc.json file passed to MLaaS4HEP, with the definition of the cuts applied on the events, to validate the use of preprocessing operations.

In order to validate the proper functioning of new frameworks, it was decided to test also Scikit-Learn AdaBoost model with the two approaches. In testing the AdaBoost model, since it does not have the number of epochs among its hyperparameters only one value of Accuracy and AUC was obtained and reported in Table 4.5.

Metric	Set	MLaaS4HEP	Jupyter Notebook
Accuracy	Training Set	0.901	0.901
	Validation Set	0.901	0.901
	Test Set	0.900	0.900
AUC	Training Set	0.928	0.928
	Validation Set	0.928	0.928
	Test Set	0.925	0.925

Table 4.5: Accuracy and AUC scores on training, validation and test set for the two approaches using AdaBoost.

The entire pipeline was also run including the application of the same cut defined in the previous section, and again, the Accuracy and AUC metrics returned the same values in both approaches.

In conclusion the results obtained by MLaaS4HEP for the validation of the switch to uproot4, the use of preprocessing functions and the implementation and use of a new framework (with and without preprocessing operations) are consistent with those obtained by an external approach, thus validating the correct functioning of the new implementations.

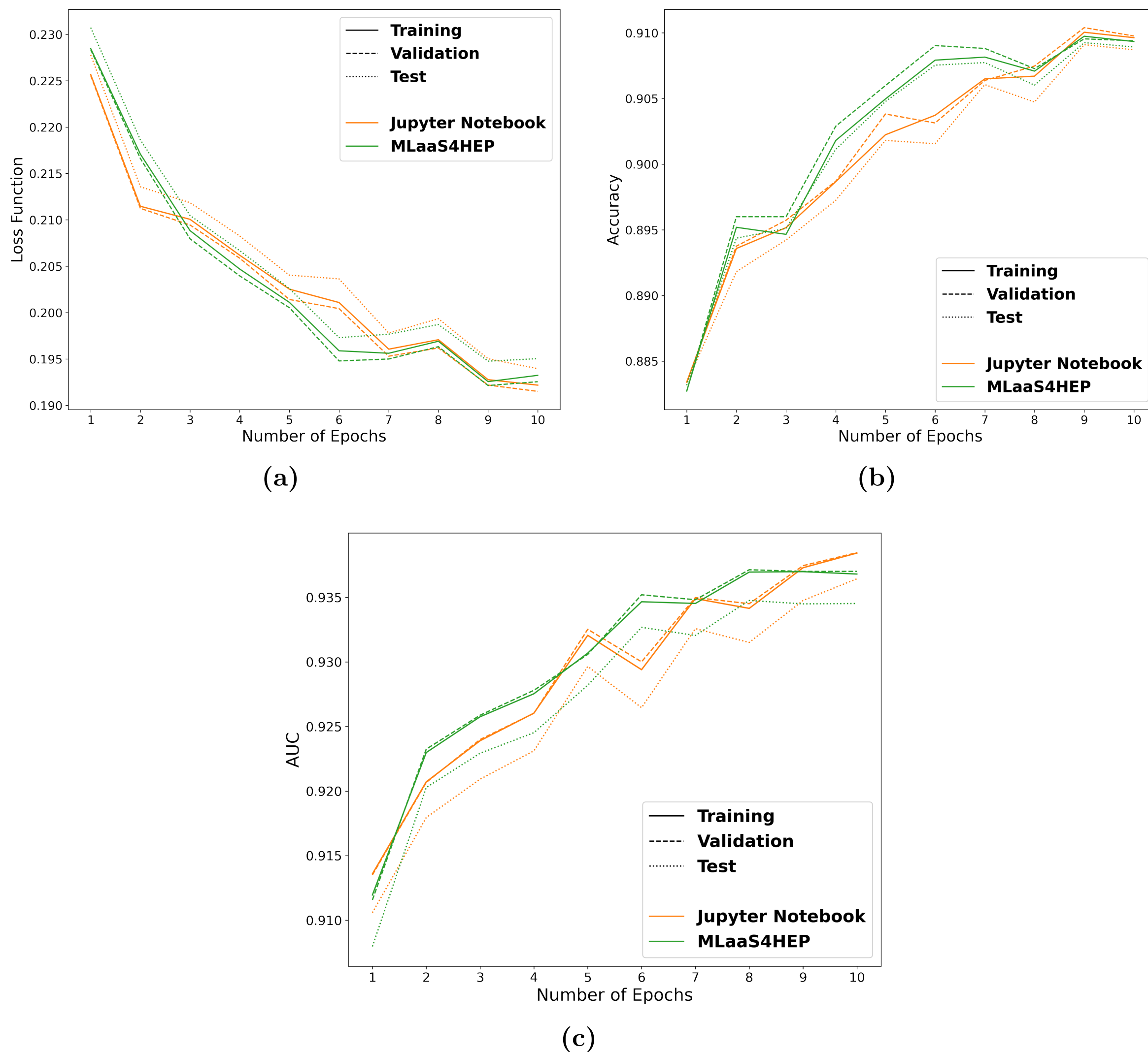

Figure 4.8: Comparison of the metrics score (loss **(a)**, Accuracy **(b)** and AUC **(c)**) obtained in the training, validation and test set. To test the correct implementation of the MLaaS4HEP preprocessing functions, two different approaches were considered: for two different approaches: (1) using MLaaS4HEP to read and normalize events applying cuts on branches, and to train the ML model; (2) using a Jupyter Notebook to perform the entire pipeline without using MLaaS4HEP.

4.5 MLaaS4HEP application on the Higgs Boson ML challenge

In this Section, the results of the MLaaS4HEP application to a physics use case will be presented and discussed. The considered use case is the Higgs Boson ML challenge [131] [132], a competition held in 2014 organized by a group of ATLAS physicists and data scientists and hosted by Kaggle [133] [134] [135]. Kaggle is a web platform for Data Science and ML competitions. It allows to solve Data Science challenges and compete with other participants in building the best predictive models. In addition, Kaggle provides a space where users can share their datasets and examine datasets shared by others. Users can also share pieces of code using these datasets and talk about them with other users in the discussion section. However, Kaggle is not limited to just competitions: it started in 2010 hosting ML competitions and now also offers a public data platform, a cloud-based workbench with powerful resources and AI training courses.

In the challenge context, typically the overall competition process consists of three main steps. First, Kaggle gives us a problem definition and the dataset to solve it. Then follows the development of the ML model that best fits the problem data and finally there is the preparation of the submission file that is uploaded to Kaggle, which internally measures the quality of the predictions and shows the results achieved by the model. Each Kaggle competition specifies a single metric that is used to rank participants. The better the metric the model achieves, the better the ranking.

4.5.1 Challenge description

The Higgs Boson ML challenge has been set up to promote collaboration between HEP physicists and data scientists. The simulated data for this challenge were provided by the ATLAS experiment and refers to data taken in 2012. The dataset has been made permanently available on the CERN Open Data portal [136]. H was theorized almost 60 years ago with the role of giving mass to other elementary particles. It is the final ingredient of the Standard Model of particle physics, ruling subatomic particles and forces. The H has many different processes through which it can decay. It has been seen first in three distinct decay channels which are all boson pairs, i.e. $H \to ZZ^*$, $H \to \gamma\gamma$ and $H \to WW^*$. To establish the mass generation mechanism for fermions as implemented in the SM, it is of prime importance to demonstrate the direct coupling of the H to fermions and the proportionality of its strength to mass. Then, the next important topics is to seek evidence on the decay into fermion pairs and the most promising candidate decay modes are the decays into tau leptons, i.e. $H \to \tau\tau$, and and b quarks, i.e. $H \to b\bar{b}$ and to precisely measure their characteristics. Due to the high background, the search for decays to $b\bar{b}$ is restricted to H produced in modes which have a more distinct signature but a lower cross-section, such as H production with an associated vector boson. The smaller rate of these processes in the presence of still large background makes their detection challenging. Then, more favourable signal-to-background conditions are expected for $H \to \tau\tau$ decays. From a theoretical point of view, this is an interesting channel since the coupling of the H to electrons and muons is

beyond the reach of the ATLAS experiment due to the small masses of these two leptons. Because we have little direct information on the coupling coupling of the H to leptons, this channel may be decisive. On the other hand, it is experimentally challenging because neutrinos are not detected and so their presence in the final state makes it difficult to evaluate the mass of the H candidate. Second, the Z boson can also decay in two τ; this happens much more frequently than the H decay and the two decays produce similar events which are difficult to separate.

In the challenge context, the different types of particles or pseudo particles of interest are electrons, muons, hadronic tau, jets, and missing transverse energy. Electrons and muons live long enough to reach the detector, so their properties (energy and direction) can be measured directly. τ, on the other hand, decay almost immediately after their creation into either an electron and two neutrinos, a muon and two neutrinos, or a bunch of hadrons and a neutrino. Such bunch can be identified as a pseudo particle called the hadronic τ. Jets are pseudo particles rather than real particles; they originate from a high energy quark or gluon, and they appear in the detector as a collimated energy deposit associated with charged tracks. Missing transverse energy is a pseudo-particle that refers to the leakage in particles detection. Neutrinos produced in the decay of a τ completely escape detection; however, it is possible to infer their properties using the law of conservation of momentum. Another difficulty is that many particles are lost in the beam tube along the z-axis, so momentum balance information is lost in the z-direction. Therefore, we can only do the summation in the transverse plane, from which we get the missing transverse energy.

The Higgs Boson ML challenge focuses on one particular decay topology among the many possible ones: events where one τ decays into an electron or a muon and two neutrinos, and the other τ decays in hadrons and a neutrino.

The objective of the participants is to define a classifier, trained on simulated events, able to select a signal-rich region in the feature space.

In the challenge, the Approximate Median Significance (AMS) metric was used to evaluate the quality of the classifier. The AMS metric is defined in Eq. 4.1, where s and b represent the estimated number of signal and background events respectively, while b_{reg} is a regularization term setted to 10 for the challenge.

$$AMS = \sqrt{2\left((s+b+b_{reg})\ln\left(1+\frac{s}{b+b_{reg}}\right)-s\right)} \tag{4.1}$$

4.5.2 Datasets

For the challenge, simulated samples were produced centrally by the ATLAS collaboration, using their official MC production workflow.

The signal events represent the production of the H, while the background events were generated simulating other known processes able to produce events with at least one electron or muon and a hadronic τ, mimicking the signal. For the Higgs challenge, only the following background processes were considered.

1. Z decay into τ pairs ($Z(\tau\tau)$), which produces events with a topology very similar

to the H decay one.

2. Events with pair of t quark, which can decay into lepton and hadronic τ.

3. W decay, where an electron or muon and a hadronic τ can appear simultaneously through imperfections of the particle identification procedure.

The simulated events were divided into two datasets: training set, which contains 250000 events and test set, which contains 550000 events. For all the events, 30 features were provided: for a full physics description see [132].

Additionally, in the training set, weights were provided in the training set so the AMS could be properly evaluated. Finally, for some entries some variables are meaningless or cannot be computed; in this case, their value was set to -999, which is outside the normal range of all variables.

4.5.3 Strategy

When approaching a ML problem, first of all it is necessary to understand the type: in this case we are dealing with a supervised binary classification problem, which purpose is to distinguish signal events from background events. Next, in order to see and understand the data we are working with, it is necessary to make some Exploratory Data Analysis (EDA) [137] on the training set. EDA refers to the critical process of performing initial investigations on data so as to discover patterns, to spot anomalies, to test hypothesis and to check assumptions with the help of summary statistics and graphical representations. To perform this operation we used the libraries pandas and pandas-profiling, through which it was possible to understand the shape of the dataset, the distributions of each feature and how much they are correlated. From this preliminary data exploration, the histogram representation of the distributions for each feature in the dataset and the correlation matrix, shown in Fig. 4.9, were obtained.

From the features distribution, the presence of values out of the range previously mentioned was found in some branches. These values were converted to NaN and then replaced with the median value of the respective feature. In addition, there are some branches characterized by a tailed distribution, to which a log-tranform can be applied, thus obtaining a less skewed distribution.

From the correlation matrix instead, it is possible to identify the highly correlated branches. In general, it is not recommended to have correlated features in the dataset, as a group of highly correlated features will not bring additional information, but will increase the complexity of the algorithm, thus increasing the risk of errors on unseen data. For this reason, it was decided to drop features with a high correlation index.

Finally, feature importance was performed to understand which variables are the most crucial. Feature importance is a technique that scores input features based on how useful they are in predicting a target variable. The degree of importance of each feature was extrapolated from this technique using the XGBoost model (which description will follow next) and is shown in Fig. 4.10 in descending order.

As an example for what has been described so far, the distributions of three features are shown in Fig. 4.11: the first shows the distribution of a feature selected for

Figure 4.9: Correlation Matrix of the features in the training dataset.

model training, the second shows the distribution of a removed feature, and finally the distribution of a feature that has been log-transformed.

The strategy defined so far can be performed within MLaaS4HEP using the newly implemented functionality. The last step is to define the model to be trained. To accomplish the challenge, a new library has been integrated within MLaaS4HEP: XGBoost [138] [139] [140]. XGBoost (eXtreme Gradient Boosting) is a popular and efficient open-source implementation of the gradient boosted trees algorithm and is well known for its performance which consistently outperforms other ensemble ML algorithms. XGBoost provides a parallel tree boosting that solve many data science problems in a fast and accurate way. Furthermore, it is well integrated with the Scikit-Learn framework and its API and, due to this feature, it was possible to easily integrate it within MLaaS4HEP.

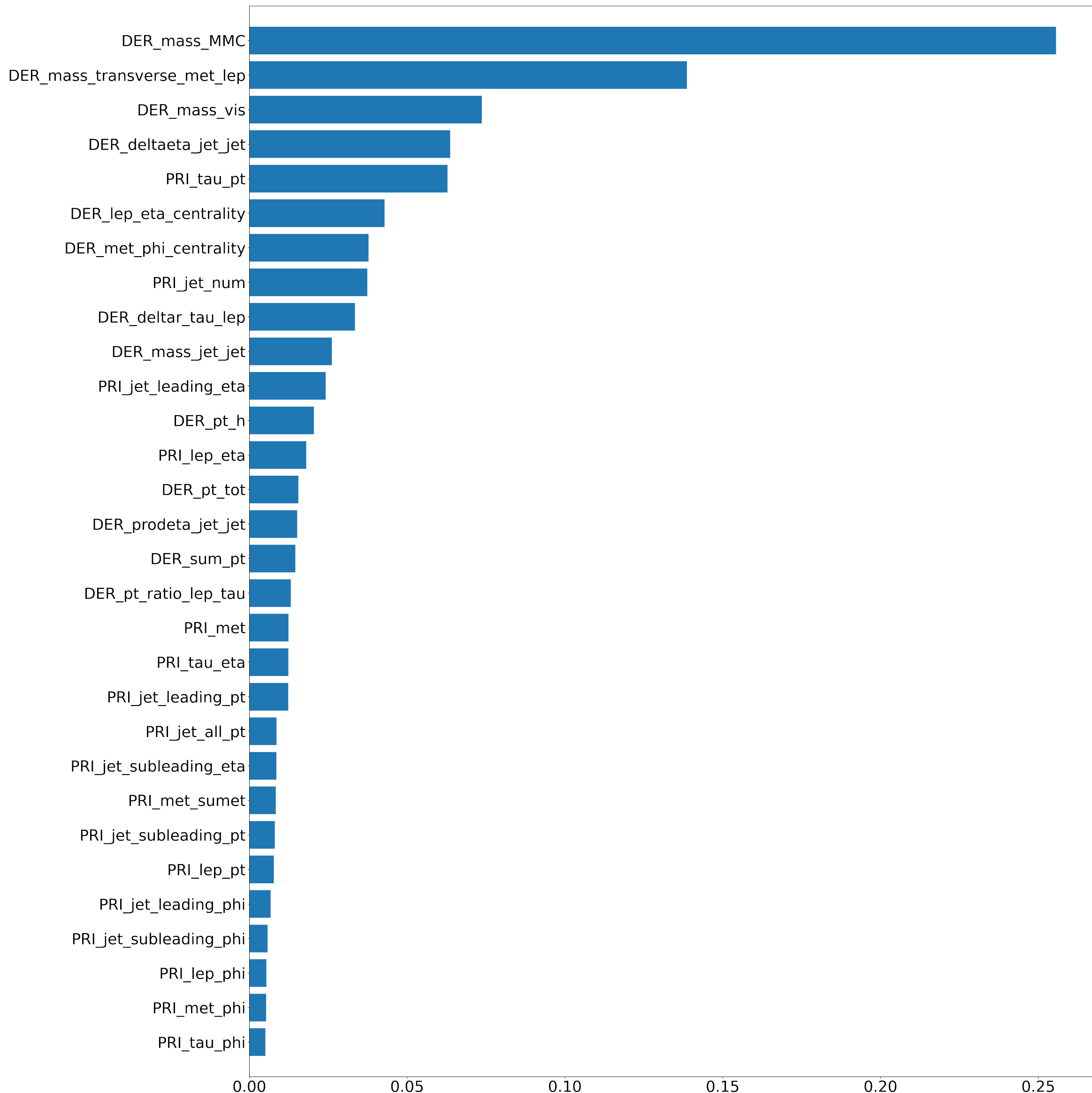

Figure 4.10: Feature importance obtained using XGBoost Classifier.

The challenge has been addressed using three different models: an XGBoost Classifier (XGBC), a Gradient Boosting Classifier (GBC) [141], which is another ensemble algorithm defined by Scikit-Learn, and finally a Sequential NN in Keras. The definition of these models is reported in Appendix A.4. Both the ensemble models are trained according to the syntax of Scikit-Learn and, since neither of them supports the `partial_fit` method, they have been trained receiving all the events in a single chunk. In contrast, the NN was trained incrementally with a chunk size of 25k.

The training dataset provided by Kaggle was released in CSV format. This datasets was split into two files, one for signal events and one for background events, which were subsequently converted to ROOT format using RDataFrame [142]. The training was performed within MLaaS4HEP: according to the strategy and feature importance

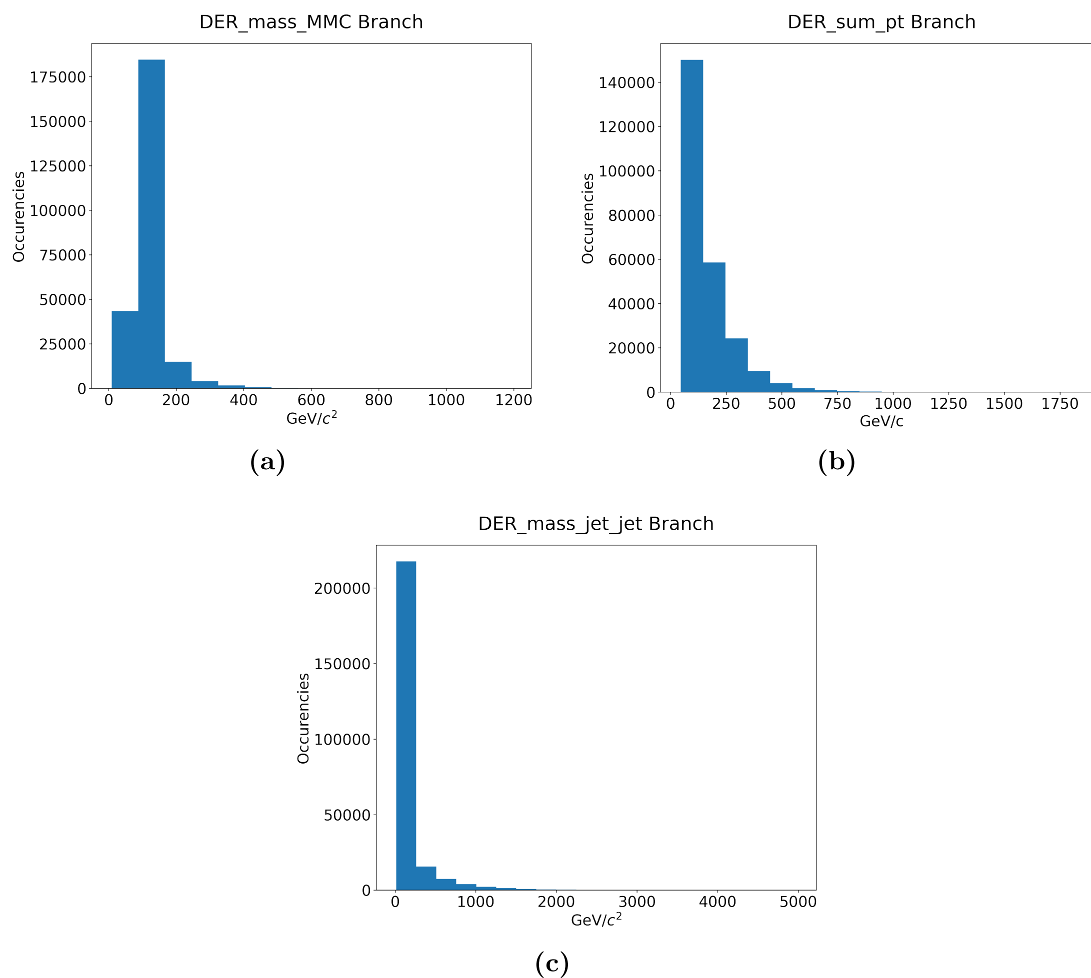

Figure 4.11: Histogram representation of three feature distributions. In **(a)**, **(b)**, **(c)** is shown the distribution of a feature selected for the model training, a feature to be log-transformed and a feature selected to be removed respectively.

previously computed, the MLaaS4HEP input files were processed in the following way:

- the highly correlated branches mentioned above were excluded from the training phase,
- the *-phi branches ere excluded too because of the results obtained from the feature importance,
- the branches characterized by a tailed distribution were log-transformed in order to reduce their skewness.

The training phase was performed within MLaaS4HEP two times for each model. During the first training, a data splitting was performed, in which 80% was used to train the model, while the remaining 20% was set aside to evaluate its performance. In addition, through this hold-out set, it was possible to extract an optimal threshold value for each model that allows us to associate the probabilities of class membership of each event with the correct label during the real testing phase. The optimal threshold value to be used in event classification was calculated for all three algorithms and was obtained from the ROC curve. The ROC curve uses a number of different thresholds to obtain the rate of TP and FP predictions. These values can be used to calculate the Geometric Mean (or GMean) metric. Such metric, defined in Eq. 4.2, is calculated as the geometric mean of the sensitivity (namely TPR) and specificity, defined as $1 - FPR$. Then, the optimal threshold correspond to the TPR and FPR values that return the largest G-Mean value.

$$GMean = \sqrt{sensitivity \cdot specificity} = \sqrt{TPR \cdot (1 - FPR)} \tag{4.2}$$

Once the optimal threshold values were found, a second training was performed: in this case, the model was trained using all events within training dataset and was then stored in the correct format.

Once the MLaaS4HEP workflow is finished, it is necessary to verify the predictions of the model on new data. For this purpose a Jupyter Notebook has been created in which the model is loaded and subsequently tested on the Kaggle test dataset (also called "leaderboad"). The predictions made by the model are then saved within a CSV file that needs to be uploaded to the challenge site. After completing the submission, Kaggle will communicate the score obtained by the model.

4.5.4 Results

To achieve the best performance, hyperparameter tuning of the three models was performed, thus finding the best combination. The trained models, i.e. XGBC, GBC and the Sequential NN, were then subsequently tested on the leaderboard. The predictions obtained from the three models were then submitted to the Kaggle challenge site to evaluate their performance in terms of the AMS metric. The scores obtained were good, placing the three submitted models in the top 150 of the best positioned models that participated in the challenge in 2014. Thus, although the winner's score was not achieved,

it was possible to verify the potential and elasticity of MLaaS4HEP to deal with a HEP problem with different ML frameworks and models. This challenge also provided insights for the integration of possible new features, such as data visualisation, and for other possible fields of application for MLaaS4HEP.

4.6 Future directions

Throughout Chapters 3 and 4 we presented a description of MLaaS4HEP, the implementation of new features, and discussed their use in the context of a HEP physical analysis. MLaaS4HEP presents is agnostic to HEP experiments and has the potential to be integrated with any Python based ML framework. It also provides the ability to read data in chunks, both local and remote, thus enabling ML training on very datasets. However, MLaaS4HEP performance strictly depends on the available hardware resources and its usage with TB or PB-size datasets needs additional improvements, both at code level integrating new solutions and exploring cloud solutions that can guarantee the usage of more performing resources. The creation of a cloud service for MLaaS4HEP has two advantages: it opens up to potentially better performing resources and also provides a true “as a Service” solution to the user. For these reasons, one of the next steps of this project is to move towards MLaaS4HEP cloudification, thus creating a solution where the user is provided on-demand with an infrastructure hosting a service able to run the MLaaS4HEP workflow. At the same time, since the current landscape of ML is evolving, another working area concerns the integration in MLaaS4HEP of other ML frameworks and solution. As example, further improvements on MLaaS4HEP can be achieved via the implementation of a multi-threaded I/O layer and distributed training [143] for the Data Streaming and Data Training Layers. Other approaches can also be used to improve TFaaS: the throughput can be further improved by switching from HTTP to a gRPC-based [144] solution, that can provide a fast inference layer based on FPGA and GPU-based infrastructure. There is also effort in the creation of a generic inference service, which allows to use not only TF but also other ML frameworks, e.g. PyTorch.

Conclusions

This thesis aims at contributing to the evolution of the ML "as a service" for HEP framework, MLaaS4HEP, which is an end-to-end data-service that trains and serves ML models for HEP use cases. The work presented in the thesis contributed to improve and evolve the MLaaS4HEP project with the implementation of new features, which allow to expand its capabilities, and providing a real application to a physics use case: the Higgs Boson ML challenge. The aspects of the code that have been subject to change are many. MLaaS4HEP uses the uproot library to convert ROOT files in a more suitable format for ML, and since this library was recently updated, one of the personal contributions was to update MLaaS4HEP to support the new version of uproot. Then, the performance of MLaaS4HEP with both versions of uproot was measured, which showed slightly better performance using the new version of uproot. Subsequently, thanks to this update, it was possible to implement new preprocessing operations on ROOT data, which allow to apply cuts and create new branches enabling the users to handle data before using them in the training phase. The performance of different case studies was reported, showing a decrease of the performance in relation to the increase of complexity of preprocessing operations.

Subsequently, part of the thesis work was devoted to generalize MLaaS4HEP to other ML frameworks, since originally it was only tested with Keras models. The properly integrated frameworks were PyTorch and Scikit-Learn, two libraries that are particularly popular and used in the ML world. To verify the proper functioning of the new features added during the thesis work, it was necessary to validate them by making a comparison of the results with those obtained reproducing the steps of the pipeline performed by MLaaS4HEP in an external environment, i.e. a Jupyter Notebook. The results obtained from these tests shown a good consistency between the two approaches.

Finally, MLaaS4HEP was applied to the Higgs Boson ML challenge use case. Three ML models were used: an XGBoost Classifier, a Gradient Boosting Classifier, and a Sequential NN. The three models reported a score that ranked us in the top 150 best participants. However, the main goal behind this challenge was not to achieve a higher score, but to test how MLaaS4HEP could approach to other use cases, not only CMS ones. MLaaS4HEP started as a CMS project but it can tackle physics use cases from other experiments. In this challenge data are public, stored also in the CERN Open Data Portal, and this shows the versatility of MLaaS4HEP.

The project is ongoing and part of the work described in this thesis have been presented at the International Symposium on Grids & Clouds (ISGC) in March 2022.

Appendix A

Models definition

In the following, the definitions of the models used in Chapters 3 and 4 will be presented and briefly described.

A.1 Keras NN

Listing A.1 shows the definition of the Keras Sequential NN mentioned in Sec. 3.4.2.

```
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense

model = keras.Sequential([keras.layers.Dense(128, activation='relu',
                         input_shape=(idim,)),
                         keras.layers.Dropout(0.5),
                         keras.layers.Dense(64, activation='relu'),
                         keras.layers.Dropout(0.5),
                         keras.layers.Dense(1, activation='sigmoid')])

model.compile(optimizer=keras.optimizers.Adam(learning_rate=1e-3), loss
   =keras.losses.BinaryCrossentropy(),
              metrics=[keras.metrics.BinaryAccuracy(name='accuracy'),
   keras.metrics.AUC(name='auc')])
```

Listing A.1: Definition of a Sequential NN written in Keras, used for validation purposes.

The structure of the NN has been defined by passing a list of layers to the Sequential constructor: here, `idim` represents the size of the input layer and `Dense` defines a regular densely-connected NN layer. Then, through the `compile()` Keras API, the model has been configured by selecting the optimizer (`Adam`), loss function (`BinaryCrossentropy()`), and metrics to be used (`BinaryAccuracy` and `AUC`).

A.2 PyTorch NN

The idiom for defining a model in PyTorch involves the definition of a class that makes use of PyTorch's `nn.Module` [122], which provides an ensemble of essential methods to build a NN. To create a model, the class constructor has to be defined in the `__init__` method, which defines the layers of the NN, and the `forward` method, that defines how to forward propagate input data through the layers of the model. In Listing A.2, the definition of the model used to verify the correct integration of PyTorch within MLaaS4HEP (see Sec. 4.3.2) is reported.

```
class ClassifierNN(nn.Module):

    def __init__(self, idim,
                 activation=fun.relu):
        super().__init__()

        self.layout = (idim, 16, 16, 1)
        self.inference_mode = True
        self.activation = activation
        self.layers = nn.ModuleList()
        for num_nodes, num_nodes_next in zip(self.layout[:-1], self.
    layout[1:]):
            self.layers.append(nn.Linear(num_nodes, num_nodes_next))

    def forward(self, x):
        for layer in self.layers[:-1]:
            x = self.activation(layer(x))

        x = torch.sigmoid(self.layers[-1](x))

        return x
```

Listing A.2: Model used to test the integration of PyTorch framework within MLaaS4HEP. Here `idim` represents the size of the input layer.

In Listing A.3, instead, the definition of the default training function adopted by MLaaS4HEP for the PyTorch framework is given. The training process requires that user defines a loss function and an optimization algorithm for the model. Then it is necessary to set up a first loop for the number of training epochs and an inner one for the batches which compose the training set. Inside the training loop, the trained model performs prediction and the selected loss function gets calculated, then, the weight of the model gets updated in order to improve the performance of the NN.

```
def torch_train(self, model, train_loader, val_loader, **kwds):
        """Default train function for PyTorch models"""
        epochs = kwds['epochs']
        loss_func = nn.BCELoss()
        optim = torch.optim.Adam(model.parameters(), lr=1e-3)
        self.model.train()
        for epoch in range(1, epochs+1):
            mean_train_loss = 0.0
```

```
            for i, (xs, ys) in enumerate(train_loader):
                optim.zero_grad()
                outputs = model(xs)
                train_loss = loss_func(outputs, ys)
                train_loss.backward()
                optim.step()
                mean_train_loss = (mean_train_loss * i + float(
    train_loss)) / (i + 1)

            mean_val_loss = 0.0
            for i, (xs, ys) in enumerate(val_loader):
                outputs = model(xs)
                val_loss = loss_func(outputs, ys)
                mean_val_loss = (mean_val_loss * i + float(val_loss)) /
     (i + 1)
            print('Epoch {}\nMean train/validation loss: {:.4f}/{:.4f}'
    .format(epoch, mean_train_loss, mean_val_loss))
```

Listing A.3: Default training function for PyTorch models.

The epochs are directly extracted from the `params.json` file (via `kwds`), then the loss function and optimizer are defined. Then, the model is set to the training mode (`self.model.train()`) and enters the second cycle, within which the average loss is calculated for both the train set (`train_loader`) and the validation set (`val_loader`) for each epoch.

A.3 Scikit-Learn

Defining a model using Scikit-Learn turns out to be much simpler than using PyTorch: all that is needed is to import the model you intend to use and possibly to make a hyperparameter setting. In Listing A.4, A.5, A.6, the definitions of the the models used to verify proper Scikit-Learn integration are shown. The selected parameters are not the result of tuning the hyperparameters and the definitions are included only for visual purposes.

```
from sklearn.linear_model import SGDClassifier

model = SGDClassifier(loss='log', tol=1e-3)
```

Listing A.4: SGDClassifier model definition.

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

model = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),
    n_estimators=100, algorithm="SAMME.R", learning_rate=0.5)
```

Listing A.5: AdaBoost model definition.

```
from sklearn.neural_network import MLPClassifier

model = MLPClassifier(solver="adam",
    learning_rate="adaptive", max_iter=350)
```

Listing A.6: MLPClassifier model definition.

A.4 Challenge models

Below, example definition of the models used in the challenge are reported, i.e. an XGBoost model (Listing A.7), a GBClassifier model (Listing A.8) and a Sequential NN written using Keras (Listing A.9). It can be seen from these definitions that the XGBoost model was defined with almost the same syntax as Scikit-Learn. The following examples show a simplified version of the models used for the challenge.

```
import xgboost as xgb

params = {
        'objective':'binary:logistic',
        'max_depth': 7,
        'alpha': 10,
        'n_estimators': 100
    }
model = xgb.XGBClassifier(**params)
```

Listing A.7: XGBoost model definition.

```
from sklearn.ensemble import GradientBoostingClassifier

model = GradientBoostingClassifier(n_estimators=120,
    max_depth=5,min_samples_leaf=50)
```

Listing A.8: Gradient Boosting Classifier model definition.

```
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense

model = keras.Sequential([keras.layers.Dense(1024, activation='relu',
   input_shape=(idim,)),
                    keras.layers.Dropout(0.6),
                    keras.layers.Dense(512, activation='relu'),
                    keras.layers.Dropout(0.6),
                    keras.layers.Dense(1, activation='sigmoid')])

    ml_model.compile(optimizer=keras.optimizers.Adam(learning_rate=1e
   -3),
    loss=keras.losses.BinaryCrossentropy(),
```

```
metrics=[keras.metrics.BinaryAccuracy(name='accuracy'),
keras.metrics.AUC(name='auc')])
```

Listing A.9: Sequential NN definition.

Appendix B

ROOT vs uproot cuts

In the following, we report the codes that allowed to verify the validity of the cuts performed with uproot. The cutting conditions were reproduced in a generic manner. As an example, the application of a cut on a flat branch (`branch_flat`), which must be greater than or equal to a value of `cut_cond1` , and a cut of the `all` type on a jagged branch (`branch_jagged`), which must be smaller than `cut_cond2` , is reported.

In Listing B.1, we report code written in C++ that applies a cut on the two branches, allows counting how many events on the entire file have fulfilled the cut condition, and allows creating an output ROOT file containing those events. As a reminder, an `"all"` cut type implies that the cut condition must be satisfied by all values in the jagged array. With this approach we verify that the two cut conditions are verified by checking each event in a for loop. Both `branch_flat` and `branch_jagged` are associated with a boolean variable that is set to False if the cut condition is not satisfied. We then compare the Boolean variables for each cut and decide whether to keep or remove the event.

In case the event was found to be consistent with the applied cut, it is inserted into the output file and the variable `events_after_cut` is incremented by 1, otherwise it is discarded. This operation is performed for each event until the end of the ROOT file.

```
void script () {

    //setting the output file
    TFile * outputfile  = new TFile( "output_file.root", "RECREATE" );
    TTree * tree_name = new TTree( "tree_name", "tree_name" );

    Float_t branch_flat, branch_jagged

    tree_name->Branch("branch_flat", &branch_flat, "branch_flat/F");
    std::vector<int> branch_jagged;

    //setting the input file
    TFile *inputfile  = new TFile( ''input_file.root'', "READ" );
    TTree *evt = (TTree*)inputfile->Get( "input_tree_name" );

    //variables setting
    float mybranch_flat = 0;
    evt->SetBranchAddress( "branch_flat", &mybranch_flat);
```

```
    std::vector<int> *mybranch_jagged = 0;
    evt->SetBranchAddress("branch_jagged", &mybranch_jagged);

    int events_after_cut = 0;

    for ( Long64_t ievent = 0; ievent < 10000; ++ievent ){
        //get i-th entry in tree
        evt->GetEntry( ievent );
        bool keep_mybranch_flat = false;
        bool keep_mybranch_jagged = true;

        //cut on flat branch
        if ( mybranch_flat >= cut_cond1 ){
            keep_mybranch_flat = true;
        }

        //a second for loop to access each element of the i-th event
   for a jagged branch
        for(int idx=0; idx < mybranch_jagged->size(); idx++) {
            if (mybranch_jagged->at(idx) >= cut_cond2){
                keep_mybranch_jagged = false; //mimick the "all" cut
                }
            }

        //here we check the condition of both cuts on the i-th event

        if ((keep_mybranch_flat & keep_mybranch_jagged) == true){
            //if the condition is satisfied, fill the output file with
   the i-th event
            branch_flat = mybranch_flat;

            for(int idx=0; idx < mybranch_jagged->size(); idx++) {
                branch_jagged.push_back(mybranch_jagged->at(idx);
            }
            events_after_cut += 1;
            tree_name->Fill();
            branch_jagged.clear();
        }

    }//end loop over events

    cout<<"Events after cut: "<< events_after_cut <<endl;
    inputfile->Close();
    outputfile->Write();
    outputfile->Close();
    delete outputfile;
}
```

Listing B.1: Performing cuts on a flat and a jagged branch using ROOT.

In Listing B.2 we report the same cuts made previously but now using uproot and Awkward Array. In this case the cuts are performed within an iterator that reads all the ROOT file one chunk at a time. The chunk size for this example was set to 10000. The cut on the flat branch is specified directly within the definition of the iterator, while the

jagged branch is specified within the for loop.

```
import uproot
import awkward as ak
tree=uproot.open("/path_to/file.root")["input_tree_name"]
chunk_size = 10000
 #here we set the cut on flat branch
gen = tree.iterate(step_size=chunk_size, cut='branch_flat>=cut_cond1',
    library="ak")
for original_chunk in gen:
     #here we set the cut on the jagged branch
     chunk = original_chunk[ak.all(original_chunk['branch_jagged']<
    cut_cond2, axis = 1)]
     x = len(chunk)
     print("events after cut: {}".format(x))
```

Listing B.2: Performing cuts on a flat and a jagged branch using uproot4.

Bibliography

[1] Oliver Sim Brüning et al. *LHC Design Report*. CERN Yellow Reports: Monographs. Geneva: CERN, 2004. DOI: 10.5170/CERN-2004-003-V-1. URL: https://cds.cern.ch/record/782076.

[2] Oliver Sim Brüning et al. *LHC Design Report*. CERN Yellow Reports: Monographs. Geneva: CERN, 2004. DOI: 10.5170/CERN-2004-003-V-2. URL: https://cds.cern.ch/record/815187.

[3] Michael Benedikt et al. *LHC Design Report*. CERN Yellow Reports: Monographs. Geneva: CERN, 2004. DOI: 10.5170/CERN-2004-003-V-3. URL: https://cds.cern.ch/record/823808.

[4] Lyndon Evans and Philip Bryant. "LHC Machine". In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08001–S08001. DOI: 10.1088/1748-0221/3/08/s08001. URL: https://doi.org/10.1088/1748-0221/3/08/s08001.

[5] *The ATLAS Collaboration*. URL: https://atlas.cern/.

[6] The ATLAS Collaboration. "The ATLAS Experiment at the CERN Large Hadron Collider". In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08003–S08003. DOI: 10.1088/1748-0221/3/08/s08003. URL: https://doi.org/10.1088/1748-0221/3/08/s08003.

[7] *The CMS Collaboration*. URL: https://cms.cern/detector.

[8] The CMS Collaboration. "The CMS experiment at the CERN LHC". In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08004–S08004. DOI: 10.1088/1748-0221/3/08/s08004. URL: https://doi.org/10.1088/1748-0221/3/08/s08004.

[9] *The LHCb Collaboration*. URL: https://lhcb.web.cern.ch/.

[10] The LHCb Collaboration. "The LHCb Detector at the LHC". In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08005–S08005. DOI: 10.1088/1748-0221/3/08/s08005. URL: https://doi.org/10.1088/1748-0221/3/08/s08005.

[11] *The ALICE Experiment*. URL: https://home.web.cern.ch/science/experiments/alice.

[12] The ALICE Collaboration. "The ALICE experiment at the CERN LHC". In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08002–S08002. DOI: 10.1088/1748-0221/3/08/s08002. URL: https://doi.org/10.1088/1748-0221/3/08/s08002.

[13] Thomas Sven Pettersson and P Lefèvre. *The Large Hadron Collider: conceptual design*. Tech. rep. Oct. 1995. URL: http://cds.cern.ch/record/291782.

[14] *The TOTEM Collaboration*. URL: https://home.cern/science/experiments/totem.

[15] *The LHCf Collaboration*. URL: https://home.cern/science/experiments/lhcf.

[16] *The MOEDAL Collaboration*. URL: https://home.cern/science/experiments/moedal.

[17] *The FASER experiment*. URL: https://home.cern/science/experiments/faser.

[18] *CERN approves new LHC experiment: SND*. URL: https://home.cern/news/news/experiments/cern-approves-new-lhc-experiment.

[19] V Karimäki et al. *The CMS tracker system project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: https://cds.cern.ch/record/368412.

[20] *The CMS electromagnetic calorimeter project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: http://cds.cern.ch/record/349375.

[21] *The CMS hadron calorimeter project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: https://cds.cern.ch/record/357153.

[22] *The CMS magnet project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: https://cds.cern.ch/record/331056.

[23] J. G. Layter. *The CMS muon project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: http://cds.cern.ch/record/343814.

[24] C. Foudas. *The CMS Level-1 Trigger at LHC and Super-LHC*. 2008. arXiv: 0810.4133 [physics.ins-det].

[25] Andrea Perrotta. "Performance of the CMS High Level Trigger". In: *Journal of Physics: Conference Series* 664.8 (Dec. 2015), p. 082044. DOI: 10.1088/1742-6596/664/8/082044. URL: https://doi.org/10.1088/1742-6596/664/8/082044.

[26] CMS Offline Software and Computing. *Evolution of the CMS Computing Model towards Phase-2*. Tech. rep. Geneva: CERN, Jan. 2021. URL: https://cds.cern.ch/record/2751565.

[27] *Computing Grid at CMS*. URL: https://cms.cern/detector/computing-grid.

[28] Achille Petrilli and Alain Hervé. "CMS Computing Model: The "CMS Computing Model RTAG"". In: (Dec. 2004). Ed. by Claudio Grandi, David Stickland, and Lucas Taylor.

[29] *Worldwide LHC Computing Grid*. URL: https://wlcg-public.web.cern.ch/.

[30] *Worldwide LHC Computing Grid Structure*. URL: https://wlcg-public.web.cern.ch/structure.

[31] *The Grid: Software, middleware, hardware*. URL: https://home.cern/science/computing/grid-software-middleware-hardware.

[32] *ROOT Data Analysis Framework*. URL: https://root.cern//.

[33] Aderholz et al. *Models of Networked Analysis at Regional Centres for LHC Experiments (MONARC), Phase 2 Report, 24th March 2000*. Tech. rep. Geneva: CERN, Apr. 2000. URL: https://cds.cern.ch/record/510694.

[34] Bethke et al. *Report of the steering group of the LHC computing review*. Tech. rep. Geneva: CERN/LHCC, 2001.

[35] *LHCOPN - Large Hadron Collider Optical Private Network*. URL: https://twiki.cern.ch/twiki/bin/view/LHCOPN/WebHome.

[36] *WorkBookDataFormats*. URL: https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBookDataFormats#AboutTiers.

[37] G Petrucciani, A Rizzi, and C Vuosalo. "Mini-AOD: A New Analysis Data Format for CMS". In: *Journal of Physics: Conference Series* 664.7 (Dec. 2015), p. 072052. DOI: 10.1088/1742-6596/664/7/072052. URL: https://doi.org/10.1088/1742-6596/664/7/072052.

[38] Rizzi, Andrea, Petrucciani, Giovanni, and Peruzzi, Marco. "A further reduction in CMS event data for analysis: the NANOAOD format". In: *EPJ Web Conf.* 214 (2019), p. 06021. DOI: 10.1051/epjconf/201921406021. URL: https://doi.org/10.1051/epjconf/201921406021.

[39] CMS Offline Software and Computing. *Evolution of the CMS Computing Model towards Phase-2*. Tech. rep. Geneva: CERN, Jan. 2021. URL: https://cds.cern.ch/record/2751565.

[40] *Welcome to PYTHIA*. URL: https://pythia.org/.

[41] *The MadGraph5$_a$MC@NLOhomepage*. URL: http://madgraph.phys.ucl.ac.be/.

[42] *Geant4: a Simulation Toolkit*. URL: https://geant4.web.cern.ch/node/1.

[43] Andrea Giammanco. "The Fast Simulation of the CMS Experiment". In: *Journal of Physics: Conference Series* 513.2 (June 2014), p. 022012. DOI: 10.1088/1742-6596/513/2/022012. URL: https://doi.org/10.1088/1742-6596/513/2/022012.

[44] Apollinari G. et al. *High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1*. CERN Yellow Reports: Monographs. Geneva: CERN, 2017. DOI: 10.23731/CYRM-2017-004. URL: https://cds.cern.ch/record/2284929.

[45] *High-Luminosity LHC*. URL: https://home.cern/science/accelerators/high-luminosity-lhc.

[46] *High-Luminosity LHC Project*. URL: https://hilumilhc.web.cern.ch/content/hl-lhc-project.

[47] Karastathis et all. "LHC Run 3 Configuration Working Group Report". In: (2019), 273–284. 12 p. URL: https://cds.cern.ch/record/2750302.

[48] *CMS - End of the year overview, 2020*. URL: https://cms.cern/news/end-year-overview-2020.

[49] *CMS prepares for Run 3*. URL: https://ep-news.web.cern.ch/content/cms-experiment-prepares-run-3.

[50] *Highlights of LHC, ATLAS, CMS and LHCb upgrades*. URL: https://indico.in2p3.fr/event/19802/contributions/79162/attachments/57949/77571/ProspectiveIN2P3_1203GT01F_2020.pdf.

[51] *Detector upgrade at Run3 and HL-LHC*. URL: https://indico.cern.ch/event/783977/contributions/3467002/attachments/1893181/3123312/Malik_DMatLHC_DetectorUpgradeAtRun3andHL-LHC1.pdf.

[52] Tommaso Diotalevi. "PhD Report". PhD thesis. 2019.

[53] O. et all Aberle. *High-Luminosity Large Hadron Collider (HL-LHC): Technical design report*. CERN Yellow Reports: Monographs. Geneva: CERN, 2020. DOI: 10.23731/CYRM-2020-0010. URL: https://cds.cern.ch/record/2749422.

[54] Albrecht et al. "A Roadmap for HEP Software and Computing RD for the 2020s". In: *Computing and Software for Big Science* 3.1 (Mar. 2019). ISSN: 2510-2044. DOI: 10.1007/s41781-018-0018-8. URL: http://dx.doi.org/10.1007/s41781-018-0018-8.

[55] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media, 2017. ISBN: 978-1491962299.

[56] *Wikiwand - Machine Learning*. URL: https://www.wikiwand.com/en/Machine_learning.

[57] Arthur L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM J. Res. Dev.* 3.3 (1959), pp. 210–229. DOI: 10.1147/rd.33.0210. URL: https://doi.org/10.1147/rd.33.0210.

[58] Tom M. Mitchell. *Machine learning, International Edition*. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997. ISBN: 978-0-07-042807-2. URL: https://www.worldcat.org/oclc/61321007.

[59] *Wikiwand - Supervised Learning*. URL: https://www.wikiwand.com/en/Supervised_learning.

[60] *Deep Learning*. URL: https://www.deeplearningbook.org/.

[61] *IBM Cloud Education - Supervised Learning*. URL: https://www.ibm.com/cloud/learn/supervised-learning.

[62] *DataCamp - Machine Learning Career with Python*. URL: https://app.datacamp.com/learn/career-tracks/machine-learning-scientist-with-python?version=1.

[63] *Wikiwand - Training, Validation and Test*. URL: https://www.wikiwand.com/en/Training,_validation,_and_test_sets.

[64] *Machine Learning Mastery - Difference Between Test and Validation Datasets*. URL: https://machinelearningmastery.com/difference-test-validation-datasets/.

[65] *Machine Learning Mastery*. URL: https://machinelearningmastery.com/.

[66] A. Botchkarev. "A New Typology Design of Performance Metrics to Measure Errors in Machine Learning Regression Algorithms". In: (). DOI: https://doi.org/10.28945/4184. URL: https://www.informingscience.org/Publications/4184.

[67] *Scikit Learn - sklearn.metrics.precision$_s$core*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html.

[68] *Scikit Learn - sklearn.metrics.recall$_s$core*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html.

[69] *Scikit Learn - sklearn.metrics.f1$_s$core*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html.

[70] *Wikiwand: Artificial Neural Network*. URL: https://www.wikiwand.com/en/Artificial_neural_network#/Network_design.

[71] *IBM Cloud Learn Hub / Neural Networks*. URL: https://www.ibm.com/cloud/learn/neural-networks.

[72] *Neural Network and Deep Learning*. URL: http://neuralnetworksanddeeplearning.com/.

[73] *A Concise History of Neural Networks*. URL: https://towardsdatascience.com/a-concise-history-of-neural-networks-2070655d3fec.

[74] *The Neural Network Zoo*. URL: https://www.asimovinstitute.org/neural-network-zoo/.

[75] Tim Menzies et al. In: *Sharing Data and Models in Software Engineering*. Ed. by Tim Menzies et al. Boston: Morgan Kaufmann, 2015, pp. 1–14. ISBN: 978-0-12-417295-1. DOI: https://doi.org/10.1016/B978-0-12-417295-1.00001-1. URL: https://www.sciencedirect.com/science/article/pii/B9780124172951000011.

[76] *TensorFlow*. URL: https://www.tensorflow.org/.

[77] *PyTorch*. URL: https://pytorch.org/.

[78] *EDUCBA - What is PyTorch?* URL: https://www.educba.com/what-is-pytorch/.

[79] *Machine Learning Mastery: PyTorch Tutorial*. URL: https://machinelearningmastery.com/pytorch-tutorial-develop-deep-learning-models/.

[80] *Torch.AI*. URL: https://www.torch.ai/.

[81] *Caffe2.AI*. URL: https://caffe2.ai/.

[82] *Scikit-Learn*. URL: https://scikit-learn.org/stable/.

[83] *Theano*. URL: https://theano-pymc.readthedocs.io/en/latest/.

[84] Kim Albertsson et al. *Machine Learning in High Energy Physics Community White Paper*. 2019. arXiv: 1807.02876 [physics.comp-ph].

[85] D. Bonacorsi. *65esimo Anniversario della Sezione INFN-Bologna,* 2021.

[86] V. Kuznetsov et al. "Predicting dataset popularity for the CMS experiment". In: 762 (Oct. 2016), p. 012048. DOI: 10.1088/1742-6596/762/1/012048. URL: https://doi.org/10.1088/1742-6596/762/1/012048.

[87] E. et al. Bartz. "FPGA-based tracking for the CMS Level-1 trigger using the tracklet algorithm". In: *Journal of Instrumentation* 15.06 (June 2020), P06024–P06024. ISSN: 1748-0221. DOI: 10.1088/1748-0221/15/06/p06024. URL: http://dx.doi.org/10.1088/1748-0221/15/06/P06024.

[88] *CMS and the Machine Learning at the forefront of Data Management*. URL: https://cms.cern/news/cms-and-machine-learning-forefront-data-management.

[89] *Applications of Machine Learning Methods by CMS Experiment at the CERN Large Hadron Collider*. URL: https://cse.umn.edu/dtc/events/applications-machine-learning-methods-cms-experiment-cern-large-hadron-collider.

[90] *NumPy Python Libary*. URL: https://numpy.org/.

[91] *Wikiwand - Command Separated Values*. URL: https://www.wikiwand.com/en/Comma-separated_values.

[92] *Wikiwand - Hierarchical Data Format*. URL: https://www.wikiwand.com/en/Hierarchical_Data_Format.

[93] Valentin Kuznetsov, Luca Giommi, and Daniele Bonacorsi. *MLaaS4HEP: Machine Learning as a Service for HEP*. 2020. arXiv: 2007.14781 [hep-ex].

[94] Luca Giommi et al. "Machine Learning as a Service for High Energy Physics on heterogeneous computing resources". In: *PoS* ISGC2021 (2021), p. 019. DOI: 10.22323/1.378.0019.

[95] Luca Giommi, Daniele Bonacorsi, and Valentin Kuznetsov. "Prototype of Machine Learning "as a Service" for CMS Physics in Signal vs Background discrimination". In: Oct. 2018, p. 093. DOI: 10.22323/1.321.0093.

[96] *CMS Machine Learning Documentation - MLaaS4HEP*. URL: https://cms-ml.github.io/documentation/training/MLaaS4HEP.html.

[97] *What is Machine Learning as a Service?* URL: https://www.topbots.com/comprehensive-guide-to-mlaas/.

[98] *Keras: the Python deep learning API*. URL: https://keras.io/.

[99] *The home of Standard C++ on the web*. URL: https://isocpp.org/.

[100] *Python interface: PyROOT*. URL: https://root.cern/manual/python/.

[101] *DIANA-HEP Scikit-hep uproot library*. URL: https://uproot.readthedocs.io/en/latest/#.

[102] *DIANA-HEP: an umbrella organization for bringing state-of-the art for HEP experiments*. URL: https://diana-hep.org/.

[103] *XrootD - A high performance, scalable fault tolerant access to data repositories of many kinds*. URL: https://xrootd.slac.stanford.edu/.

[104] *TensorFlow as a Service*. URL: http://github.com/vkuznet/TFaaS.

[105] F Iemmi. "$t\bar{t}H$ associated production in the all-jets final state with the CMS experiment". In: *Nuovo Cimento C* 43 (2020), 77. 3 p. DOI: 10.1393/ncc/i2020-20077-4. URL: https://cds.cern.ch/record/2765555.

[106] *The MNIST dataset of handritten digits*. URL: http://yann.lecun.com/exdb/mnist/.

[107] *Kubernetes: Production-Grade Container Orchestration*. URL: https://kubernetes.io/.

[108] *MLaaS4HEP Official GitHub Repository*. URL: https://github.com/vkuznet/MLaaS4HEP.

[109] *GitHub*. URL: https://github.com/.

[110] *An introduction to GitHub*. URL: https://digital.gov/resources/an-introduction-github/.

[111] *Personal MLaaS4HEP GitHub Repository*. URL: https://github.com/palamatt95/MLaaS4HEP.

[112] *Scikit-HEP project*. URL: https://scikit-hep.org.

[113] *MLaaS4HEP - Performing preprocessing operations using a preproc.json file*. URL: https://github.com/lgiommi/MLaaS4HEP/blob/uproot4_preproc/doc/preproc.md.

[114] *Github - scikit-hep/uproot4*. URL: https://github.com/scikit-hep/uproot4/blob/916085ae24c382404254756c86afe760acdece56/src/uproot/language/python.py#L237.

[115] *Awkward Array - ak.all*. URL: https://awkward-array.readthedocs.io/en/latest/_auto/ak.all.html.

[116] *Awkward Array - ak.any*. URL: https://awkward-array.readthedocs.io/en/latest/_auto/ak.any.html.

[117] *numpy.ndarray*. URL: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html.

[118] *Machine Learning Glossary*. URL: https://developers.google.com/machine-learning/glossary#convergence.

[119] *Machine Learning Mastery - A Gente Introduction to Premature Convergence*. URL: https://machinelearningmastery.com/premature-convergence/.

[120] *Machine Learning Mastery - How to use Learning Curves to Diagnose Machine Learning Model Performance*. URL: https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/.

[121] *PyTorch - Introduction To PyTorch Tensor*. URL: https://pytorch.org/tutorials/beginner/introyt/tensors_deeper_tutorial.html.

[122] *PyTorch - Torch.nn*. URL: https://pytorch.org/docs/stable/nn.html#module.

[123] *Scikit-Learn - Strategies to scale computationally: bigger data*. URL: https://scikit-learn.org/0.15/modules/scaling_strategies.html.

[124] *Scikit-Learn - sklearn.linear*$_m$*odel.SGDClassifier*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html.

[125] *Scikit-Learn - sklearn.neural*$_n$*etwork.MLPClassifier*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html.

[126] *Scikit-Learn - sklearn.ensemble.AdaBoostClassifier*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html.

[127] *Scikit-Learn - sklearn.metrics.classification*$_r$*eport*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html.

[128] *Classification Report in Machine Learning*. URL: https://thecleverprogrammer.com/2021/07/07/classification-report-in-machine-learning/.

[129] *Scikit-Learn - sklearn.preprocessing.MinMaxScaler*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html.

[130] *pandas - Python Data Analysis Library*. URL: https://pandas.pydata.org/.

[131] *Kaggle - Higgs Boson Machine Learning Challenge*. URL: https://www.kaggle.com/competitions/higgs-boson/data.

[132] Adam-Bourdarios et al. “The Higgs Boson Machine Learning Challenge”. In: *Proceedings of the 2014 International Conference on High-Energy Physics and Machine Learning - Volume 42*. HEPML’14. JMLR.org, 2014, pp. 19–55.

[133] *Kaggle: Your Home for Data Science*. URL: https://www.kaggle.com/.

[134] *DataCamp Interactive Course - Winning a Kaggle Competition in Python*. URL: https://app.datacamp.com/learn/courses/winning-a-kaggle-competition-in-python.

[135] *Kaggle - Getting Started*. URL: https://www.kaggle.com/getting-started/44916.

[136] *Higgs Boson ML Challenge Cern opendata*. URL: http://opendata.cern.ch/search?page=1&size=20&collections=ATLAS-Higgs-Challenge-2014.

[137] *What is Exploratory Data Analysis?* URL: https://towardsdatascience.com/exploratory-data-analysis-8fc1cb20fd15.

[138] *XGBoost Documentation*. URL: https://xgboost.readthedocs.io/en/latest/index.html.

[139] *DataCamp - Using XGBoost in Python Tutorial*. URL: https://www.datacamp.com/tutorial/xgboost-in-python.

[140] *Machine Learning Mastery - XGBoost Model*. URL: https://machinelearningmastery.com/develop-first-xgboost-model-python-scikit-learn/.

[141] *Scikit-Learn - Gradient Boosting Classifier*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html.

[142] *ROOT::RDATAFRAME*. URL: https://root.cern/doc/master/classROOT_1_1RDataFrame.html.

[143] Tal Ben-Nun and Torsten Hoefler. *Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis*. 2018. arXiv: 1802.09941 [cs.LG].

[144] *gRPC: A high performance, open source universal RPC framework*. URL: https://grpc.io/.

[145] Dustin Tran et al. *Deep Probabilistic Programming*. 2017. arXiv: 1701.03757 [stat.ML].

[146] $root_numpy - TheinterfacebetweenROOTandNumPy$. URL: http://scikit-hep.org/root_numpy/.

[147] *Awkward Array Library*. URL: https://awkward-array.readthedocs.io/en/latest/.

[148] *Awkward Array: Manipulating JSON like Data with NumPy like Idioms*. URL: https://www.youtube.com/watch?v=WlnUF3LRBj4.

[149] *awkward-1.0*. URL: https://github.com/scikit-hep/awkward-1.0#readme.

[150] *AI Geek Programmer - Data preparation with Dataset and DataLoader in Pytorch*. URL: https://aigeekprogrammer.com/data-preparation-with-dataset-and-dataloader-in-pytorch/.

[151] *Scikit-Learn - sklearn.model*$_s$*election.train*$_t$*est*$_s$*plit*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html.