ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN

FISICA

Ciclo 37

Settore Concorsuale: 02/A1 - FISICA SPERIMENTALE DELLE INTERAZIONI FONDAMENTALI

Settore Scientifico Disciplinare: FIS/01 - FISICA SPERIMENTALE

OPTIMIZATION OF ML-BASED BSM TRIGGERING WITH KNOWLEDGE DISTILLATION FOR FPGA IMPLEMENTATION IN THE CMS LEVEL-1 TRIGGER

Presentata da: Marco Lorusso

Coordinatore Dottorato

Alessandro Gabrielli

Supervisore

Daniele Bonacorsi

Esame finale anno 2025

Abstract

To enhance the discovery potential of the Large Hadron Collider (LHC) at CERN in Geneva and improve the precision of Standard Model measurements, the High Luminosity LHC (HL-LHC) Project was initiated in 2010 to extend its operation by another decade and increase its luminosity by approximately tenfold beyond the design value. To fully utilize the HL-LHC period, significant upgrades and consolidations of all four main detectors are planned. The increased collision rate and expected pileup will result in high particle multiplicity and a challenging radiation environment, necessitating advancements in the Trigger system to maintain performance.

In this context, the scope of applications for Machine Learning, particularly Artificial Neural Network algorithms, has experienced an exponential expansion due to their considerable potential for elevating the efficiency and efficacy of data processing in this experimental setting.

Nevertheless, one frequently overlooked aspect of utilizing Artificial Neural Networks (ANNs) revolves around the imperative of efficiently processing data for online applications. This becomes particularly crucial when exploring innovative methods for selecting intriguing events at the trigger level, as seen in the pursuit of Beyond Standard Model (BSM) events. The study delves into the potential of Autoencoders (AEs), an unbiased algorithm capable of event selection based on abnormality without relying on theoretical priors. However, the distinctive latency and energy constraints within the Level-1 Trigger domain at the Compact Muon Solenoid (CMS) at CERN necessitate tailored software development and deployment strategies. These strategies aim to optimize the utilization of on-site hardware, with a specific focus on Field Programmable Gate Arrays (FPGAs). This is why a technique called Knowledge Distillation (KD) is studied in this PhD Thesis. It consists in using a large and well trained "teacher", like the aforementioned AE, to train a much smaller student model which can be easily implemented on an FPGA. The optimization of this distillation process involves exploring different aspects, such as the architecture of the student and the quantization of weights and biases, with a strategic approach that includes hyperparameter searches to find the best compromise between accuracy, latency and hardware footprint.

The strategy followed to perform Offline Response Based KD on a teacher model will be presented, together with consideration on the difference in performance of applying the quantization before or after the best architecture of the student model has been found. All the steps to obtain a firmware for FPGA from a purely pythonic model, using both the hls4ml library and proprietary software from a FPGA vendor, will also be described. Moreover, Online Response Based KD was also explored as an alternative and preliminary results will be shown.

Finally, a new teacher model was tested using an AE based on Graph Convolutional Neural Network, to search for more complex architecture able to perform

Anomaly Detection, due to the possibilities opened up by KD to implement advanced algorithms on efficient hardware.

Chapter 1 provides a global view Large Hadron Collider at CERN in Geneva, Switzerland;

Chapter 2 presents an introduction to the Compact Muon Solenoid experiment at LHC, with a particular attention to the Level 1 Trigger system and its upgrade to keep up with HL-LHC, the next phase of the particle accelerator;

Chapter 3 describes the main characteristics of FPGAs, as well as the workflow to implement designs with such a kind of electronics devices and two ways to interact with them from a host machine;

Chapter 4 introduces Machine Learning concepts and terminology, offering an overview of the mathematical formulation behind this kind of algorithms. Artificial Neural Network will also be described, together with a focus on the models used in this thesis;

Chapter 5 Gives an overview on different strategies to compress and optimize Machine Learning and specifically Neural Networks to reduce their hardware footprint and latency;

Chapter 6 presents the original findings of this project, focusing on the distillation of an AutoEncoder for Anomaly Detection in data containing a mix of Standard Model events and Beyond the Standard Model (BSM) decays, with the goal of making the model efficiently implementable on FPGAs. The chapter details the impact of two hyperparameter search strategies and provides latency results from actual hardware implementation. Additionally, it introduces preliminary results for Online Knowledge Distillation (KD) and explores an AutoEncoder based on Graph Neural Networks.

Appendices Three appendix are included in this thesis about: further tests with PYNQ which allows the use of FPGAs via a Python script, the technical setup behind a workshop in Bologna organized to teach ML on FPGA techniques, and QUnfold, a Python library to perform statistical unfolding using Quantum Algorithms on simulated and real Quantum Processing Units.

Contents

Chapter 1

The Large Hadron Collider

The Large Hadron Collider (LHC) [1]–[3] is a particle accelerator operating at CERN since 2010. Placed in a 27 km tunnel at the border between Switzerland and France, the former Large Electron–Positron Collider (LEP) tunnel, it is capable of accelerating either protons or heavy-ion beams. In the first case LHC currently reaches a center-of-mass energy of 13 TeV. The entire accelerator ring is divided into eight independent sectors. Particles travel in two separated beams on opposite directions and in extreme vacuum conditions. Beams are controlled by superconductive electromagnets, keeping them in their trajectory. Both of these will be very briefly described in this section.

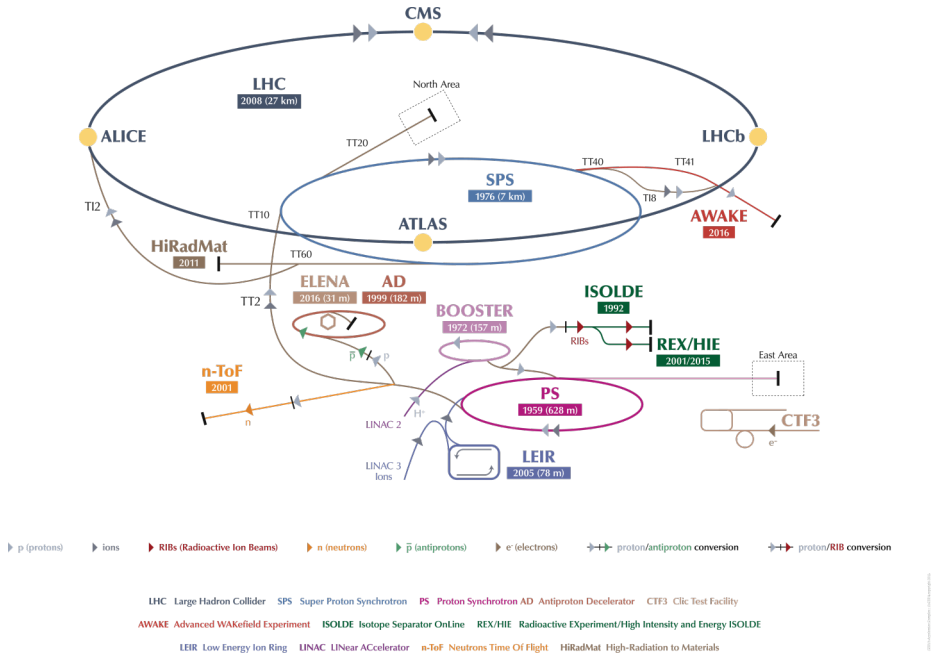


Figure 1.1: The CERN accelerator complex.

The proton injection is done employing pre-existing accelerators. This chain of accelerators, shown in Figure 1.1, comprises the following steps:

1. Protons are obtained by removing the orbiting electron on hydrogen atoms, with a process called stripping;

Quantity	Value
Circumference	26659
Magnets working temperature (K)	1.9
Number of Magnets	9593
Number of principal dipoles	1232
Number of principal quadrupoles	392
Number of radio-frequency cavities per beam	16
Nominal energy, protons (TeV)	6.5
Nominal energy, ions (TeV/Nucleon)	2.76
Magnetic field maximum density (T)	8.33
Project luminosity ($cm^{-2}s^{-1}$)	2.06×10^{34}
Number of proton packages per beam	2808
Number of proton per package (outgoing)	1.1×10^{11}
Minimum distance between packages (m)	~ 7
Number of rotations per second	11245
Number of collisions per crossing (nominal)	~ 20
Number of collisions per second (millions)	600

Table 1.1: Main LHC technical parameters.

2. A linear accelerator, called LINAC2, starts the proton acceleration bringing them to an energy up to 50 MeV;

3. Protons are then injected in the Proton Synchrotron Booster (PSB), where the energy of the beam reaches about 1.4 GeV;

4. Protons then enter the Proton Synchrotron (PS), and are accelerated up to 25 GeV;

5. The proton beam is sent to the Super Proton Synchrotron (SPS), where they reach an energy of 450 GeV;

6. Finally, protons are transferred in a bunch configuration into the two adjacent and parallel beam pipes of the LHC, circulating for several hours around the ring, with one beam in the clockwise direction and the second one in an anticlockwise direction.

Some technical parameters can be found in Table 1.1.

The Vacuum System

The LHC vacuum system [4], extending over 104 kilometers of vacuum ducts, is one of the most advanced in the world. Its primary function is to prevent collisions between beam particles and air molecules by creating an ultra-high vacuum environment (10^{-13} atm), as empty as interstellar space. Additionally, the vacuum minimizes heat exchange between components that require extremely low temperatures to operate efficiently, thus maximizing the system's overall performance.

The vacuum system is made of three independent parts:

1. an isolated vacuum system for cryomagnets;
2. an isolated vacuum system for Helium distribution line;
3. a vacuum system for beams.

The Electromagnets

The electromagnets in the accelerator [5] are designed to guide beams along their path, modifying single particles trajectories as well as align them in order to increase collision probability. To bend protons in the LHC, a magnetic field B is needed with an intensity given by the following equation

$$p[\text{TeV}] = 0.3 \cdot B[\text{T}] \cdot r[\text{km}] \tag{1.1}$$

where p is the momentum of the beam particle and r is the radius of the LHC ring.

The LHC consists of eight arc sections [6], which house the magnetic fields and vacuum chambers, and eight straight sections, where the collision points and utilities are located. These straight sections include four collision points (two of which are optimized for maximum luminosity) beam injectors, beam dump facilities, radiofrequency cavities and the collimation systems.

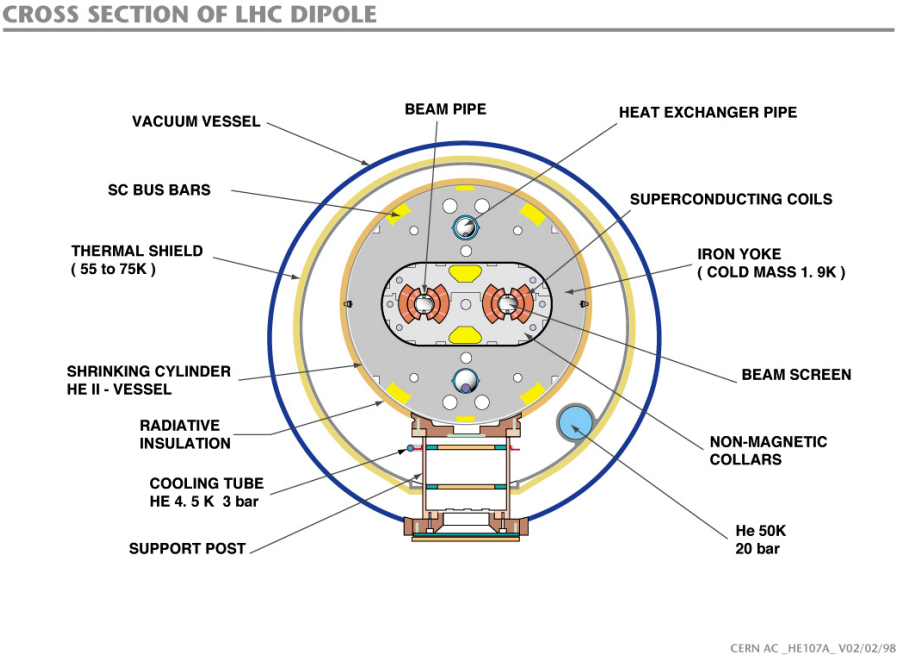


Figure 1.2: Transversal section of a dipole magnet of the LHC [7].

There are more than fifty different kind of magnets in LHC, totaling approximately 9600 magnets. The most numerous magnets (1232) are dipoles, shown schematically in Figure 1.2, generating a magnetic field with a maximum intensity of 8.3 T. In order to reach such an intense field, a current of 11 850 A is needed. To minimise power dissipation, superconducting magnets are employed, using cables made of niobium-titanium (NbTi). A system of liquid He distribution keeps the magnets at a temperature of about 1.9 K. At this incredibly low

temperatures, below that required to operate in conditions of superconductivity, helium becomes also super-fluid: this means an high thermal conductivity, thus an efficient refrigeration system for magnets.

Other important magnets are the quadrupoles (392), which help focusing the beam by squeezing it either vertically or horizontally in order to maximise the change of two protons colliding head-on. Finally, high order magnets contribute to correct imperfections of the magnetic field in the main ring magnets (dipoles and quadrupoles) and in the interaction region magnets.

Radiofrequency Cavities

Radiofrequency cavities [8] are metallic chambers where an electromagnetic field is applied. Their primary function is to organize protons into tightly packed bunches and focus them at the collision point, ensuring high luminosity and maximizing the number of collisions.

As particles pass through the cavity, they experience the force of the electromagnetic field, which propels them forward along the accelerator. When the LHC operates at nominal energy, a perfectly timed proton with the correct energy encounters zero accelerating voltage, while protons with slightly different energies are either accelerated or decelerated, grouping the particle beams into "bunches." The LHC has eight cavities per beam, each providing 2 MV at 400 MHz. These cavities operate at 4.5 K and are arranged in four cryomodules. At regime conditions, each proton beam is divided into 2808 bunches, each containing about 10^{11} protons. Away from the collision point, the bunches are a few cm long and 1 mm wide, and are compressed down to 16 nm near the latter, increasing the probability of a $p-p$ collision.

The number of bunches affects significantly the instantaneous luminosity $\mathcal{L}$ of the machine, defined as

$$\mathcal{L} = f\gamma \frac{n_b N_b^2}{4\pi\epsilon_n \beta^*} F \tag{1.2}$$

where n_b and N_b are the number of bunches and particles per bunch respectively, f represents the bunch crossing (BX) frequency, γ is the relativist Lorentz factor of the protons, ϵ_n the transverse emittance describing the shape of the beam and finally β^* is the focal length at the collision point. The F factor then takes into account the geometric reduction of the luminosity, depending on the transverse and longitudinal dimensions of the beams σ_{xy} and σ_z at the interaction point, and on the beam crossing angle θ_c:

$$F = \left(1 + \theta_c \frac{\sigma_z}{2\sigma_{xy}}\right)^{-1} \tag{1.3}$$

At full luminosity packages are separated in time by 25 ns, corresponding to a frequency of 40 MHz, or 40 million BX per second. The luminosity defined in 1.2 represents the coefficient of proportionality between the number of events produced per second dN/dt, the event rate, and the cross section of the physical process in question σ_p:

$$\frac{dN}{dt} = \mathcal{L}\sigma_p \tag{1.4}$$

To obtain the total number of collision in a defined time interval, another kind of luminosity is often used, called integrated luminosity, defined as

$$L = \int \mathcal{L} dt \tag{1.5}$$

1.1 LHC Detectors

The collisions happen at four interaction points, where the 4 main detectors are located: ATLAS and CMS are general purpose detectors, ALICE focuses on the heavy ions physics and on the study of the quark-gluon plasma, and LHCb studies the CP violation in b-physics.

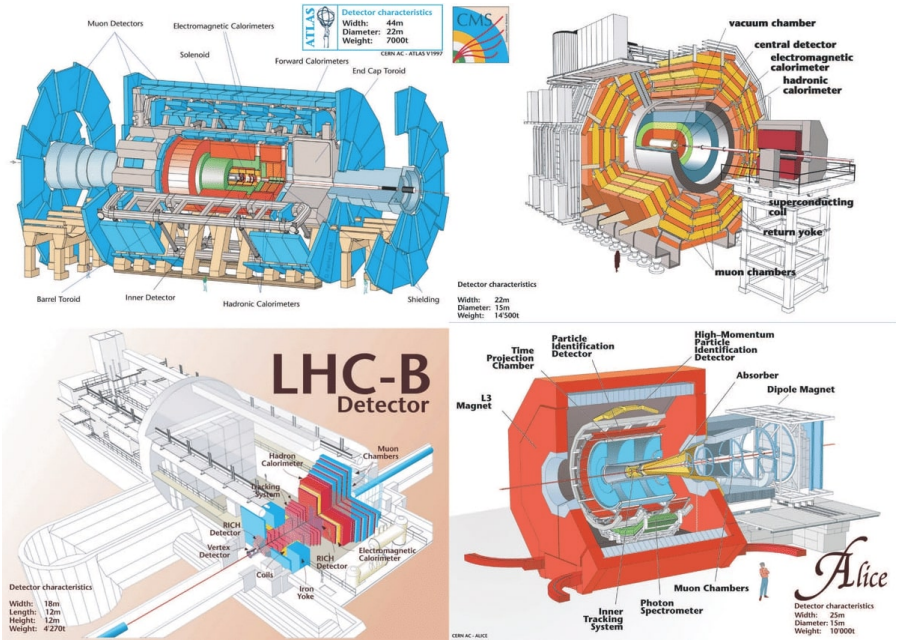


Figure 1.3: Schematic drawings of the four main LHC detectors.

Here is a small description of these experiments, except CMS which will be covered in the next chapter (Chapter 2)

1.1.1 ALICE

ALICE (**A** **L**arge **I**on **C**ollider **E**xperiment) [9] is a detector dedicated to heavy-ion physics at the LHC. It is designed to study the physics of strongly interacting matter at extreme energy densities, where a phase of matter called quark-gluon plasma is formed. By exploiting the similarities between these conditions and those just after the Big Bang, it is possible to gain insight on the origin of the Universe. It allows a comprehensive study of all kinds of known particles produced in the collision of heavy nuclei (Pb-Pb). During proton-proton runs, data is taken nonetheless in order to provide reference data for the heavy-ion programme and to a number of specific strong-interaction topics for which ALICE is complementary to the other LHC detectors.

The ALICE detector is $16 \times 16 \times 23m^3$ with a total weight of approximately 10000 tons. It consists of a central barrel part and a forward muon spectrometer. The barrel is embedded in a large solenoid magnet capable of a magnetic field up to 0.5 T.

1.1.2 ATLAS

ATLAS (**A** **T**oroidal **L**HC **A**pparatu**S**) [10] is the other general-purpose detector at the LHC. It investigates a wide range of physics, such as the search for extra dimensions and particles that could make up dark matter. The detector is forward-backward symmetric with respect to the interaction point, making it comparable to a 25m high and 44m long cylinder. A solenoid aligned on the beam axis provides a 2 T magnetic field in the inner detector, while three toroids (one for the barrel and one for each end-cap) 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.

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.

1.1.3 LHCb

The **LHC b**eauty experiment (LHCb) [11] experiment specializes in investigating the difference between matter and antimatter by studying a type of particle called the "beauty quark", or "b quark". Instead of surrounding the collision point with an enclosed detector as ATLAS and CMS, the LHCb experiment uses a series of subdetectors to mainly detect forward particles.

The 5600-tonne LHCb detector is made up of a forward spectrometer and planar detectors. It is 21 metres long, 10 metres high and 13 metres wide.

1.1.4 Other experiments

Aside from the aforementioned major LHC experiments, other smaller ones are worth a mention. One of them, LHCf [12], uses particles thrown forward by $p-p$ collisions as a source to simulate high energy cosmic rays. LHCf is made up of two detectors which sit along the LHC beamline, at 140m from either side of the ATLAS collision point. They only weight 40 kg and measures $30 \times 80 \times 10$ cm.

Another experiment placed at the LHC is called TOTEM [13]. It is designed to measure $p-p$ total elastic and diffractive cross section by measuring protons emerging at a small angle with respect to the beam lines. Detectors are spread across half a kilometre around the CMS interaction point in 26 special vacuum chamber called "roman pots", and they are connected to beam ducts, in order to reveal particles produced during the collisions.

1.2 LHC Operational History

After some major technical problems caused by a magnet quench in one of the sectors of the LHC in 2008, the collider began its research program in the spring

of 2010, starting the first phase of operations called in jargon Run 1.

Initially, during the first operational run in November 2009, the center of mass energy was $\sqrt{s} = 900$ GeV. Then, during the early part of 2010, the energy was increased up to 3.5 TeV per beam. The record of high energy collisions was reached on the end of March 2010 by colliding proton beams at a center of mass energy of 7 TeV By the end of 2011, the CMS experiment had collected a total integrated luminosity of 5.6 fb^{-1} with a record peak instantaneous luminosity of 4×10^{33} cm^{-2}s^{-1}. In 2012, the center of mass energy was increased to 8 TeV with higher instantaneous luminosities. In total, the luminosity gathered by CMS during this year amounted to 22 fb^{-1} with a record peak luminosity of 7.7×10^{33} cm^{-2}s^{-1}. In both 2011 and 2012, the LHC operated with a bunch spacing of 50 ns corresponding to a collision frequency of 20 MHz. The LHC remained in operation until February 2013, running continuously for three years and delivering a total luminosity of around 30 fb^{-1}. The CMS experiment collected a luminosity of 20 fb^{-1}, achieving the discovery of the Higgs boson together with the ATLAS Collaboration.

At the beginning of 2013, the LHC was shutdown in order to prepare the collider and run at high energy and luminosity. The accelerator was turned on in early 2015, operating at a center of mass energy of 13 TeV, and starting the second phase of operations called Run 2.

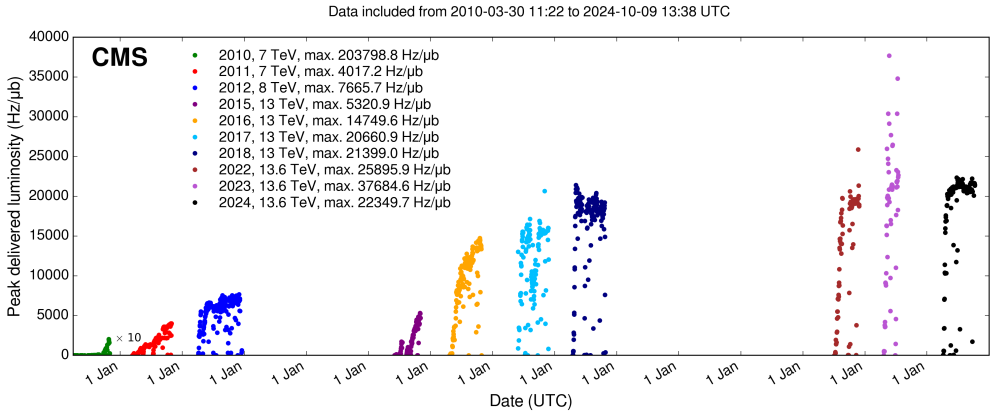


Figure 1.4: Instantaneous peak luminosity recorded by the CMS experiment per year [14]. The luminosity is given in Hz/μb i.e. 10^{30} cm^{-2}s^{-1}.

During the years 2016-2018 the majority of the Run 2 data was delivered and collected, with the full 40 MHz collision frequency. The LHC was operating proton-proton collisions from April to November of each year, with increasingly higher instantaneous luminosities. The record luminosity was 1.5×10^{34} cm^{-2}s^{-1} in 2016, and 2.1×10^{34} cm^{-2}s^{-1} in 2017 and 2018, measured by CMS as shown in Figure 1.4. The total number of collision in 2016 exceeded the number from the whole Run 1 at a higher energy per collision. The integrated luminosities measured by CMS were 41 fb^{-1} in 2016, 50 fb^{-1} in 2017 and 68 fb^{-1} in 2018, as shown in Figure 1.5.

The year 2018 was the end of Run 2. During this data taking period, it was possible to achieve good physics results, in particular precision measurements for the constraints of the SM: the masses of the Higgs and the W bosons were computed

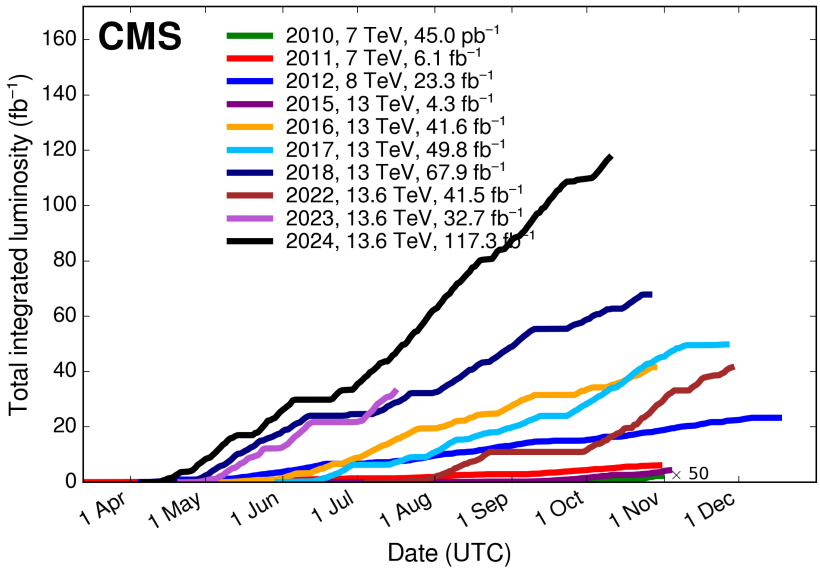


Figure 1.5: Cumulative luminosity versus day delivered to CMS during stable beams for pp collisions at nominal center-of-mass energy. This is shown for data-taking in 2010 (green), 2011 (red), 2012 (blue), 2015 (purple), 2016 (orange), 2017 (light blue), 2018 (navy blue), 2022 (brown), and 2023 (light purple), with all years plotted on the same range. [14]

with greater precision, new couplings of the Higgs were observed and an improved measurement of the CKM matrix allowed an investigation on the CP violations. A new shutdown followed Run 2.

The collider resumed delivering new data in 2022 with the new Run 3, aiming at 300 fb^{-1} in the following three years period. The actual integrated luminosities measured by CMS did not reach the expected performance with 42 fb^{-1} in 2022, 33 fb^{-1} in 2023 and 117.3 fb^{-1}, nonetheless a considerable increase was possible, as shown also by the slight increase in record instantaneous luminosity with 2.6×10^{34} cm^{-2}s^{-1} in 2022, 3.8×10^{34} cm^{-2}s^{-1} in 2023 and 2.2×10^{34} cm^{-2}s^{-1}. After Run 3 and the subsequent long shutdown, around 2026, a new era of the LHC will begin, with a complete redesign of several components of the accelerator and the surrounding experiments, in a phase of operations called High Luminosity LHC.

Such high luminosities have been possible only by squeezing the proton bunches as much as possible at the interaction point. This increases the instantaneous luminosity, but also increases the multiple collisions happening in a single bunch crossing, phenomenon called pileup. The distributions of the number of reconstructed interactions, or pileup profiles, are shown per year in Figure 1.6. In general, an high luminosity is advantageous for the physics analysis, thanks to the higher rates of rare interesting processes. However, the unavoidably larger pileup is an obstacle for the data taking and reconstruction.

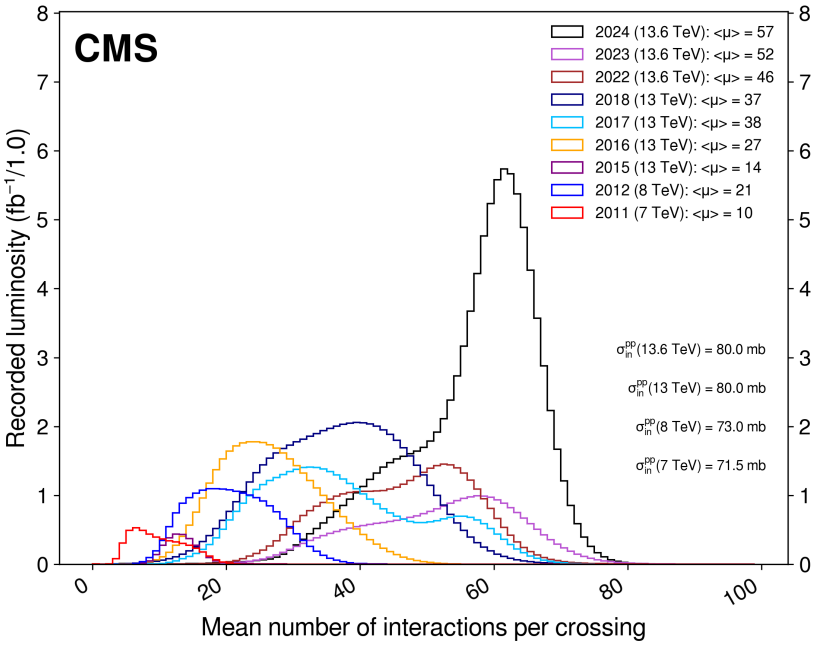


Figure 1.6: Distribution of the average number of interactions per crossing (pileup) for pp collisions in 2011 (red), 2012 (blue), 2015 (purple), 2016 (orange), 2017 (light blue), 2018 (navy blue), and 2022 (brown). The overall mean values and the minimum bias cross sections are also shown. [14]

1.3 High Luminosity LHC

In order to further increase LHC's discovery potential, as well as to improve the precision of Standard Model physics measurements, the High Luminosity LHC (HL-LHC) [15] Project was setup in 2010 to extend its operability by another decade and to increase its luminosity (and thus collision rate) by a factor of ~ 10 beyond its design value. The main objective of the HL-LHC design study was to determine a set of beam parameters and the hardware configuration that will enable the LHC to reach the following targets:

- a peak luminosity of up to $\sim 7.5 \times 10^{34}$ cm^{-2}s^{-1} with leveling, i.e. a constant luminosity at a value below the virtual maximum luminosity, in order to reduce the "luminosity burn-off" (protons consumed in the collisions) which follows a luminosity peak without leveling;

- an integrated luminosity of 250 fb^{-1} per year with the goal of 3000 fb^{-1} in about a dozen years after the upgrade. The integrated luminosity is about ten times the expected luminosity of the first twelve years of the LHC lifetime.

The overarching goals are the installation of the main hardware for the HL-LHC during the Long Shutdown 3 (LS3), scheduled for 2026-2028 (see Figure 1.7), finishing the hardware commissioning at machine re-start in 2028-2029 while taking all actions to assure a high efficiency in operation until 2035-2040.

Upgrading such a large scale, complex piece of machinery is a challenging procedure and it hinges on a number of innovative technologies. The process relies

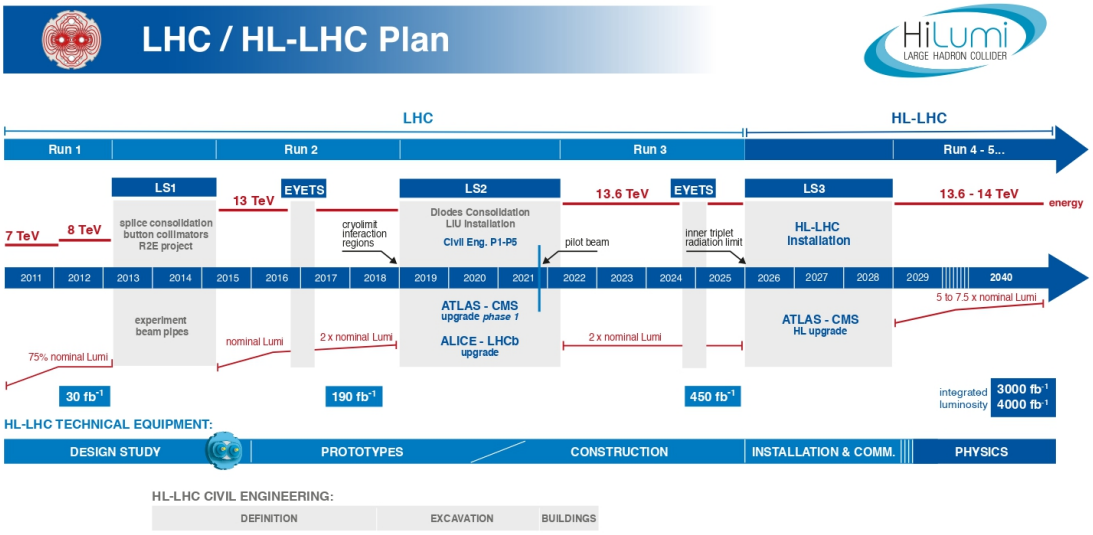


Figure 1.7: LHC/ HL-LHC Plan (last update February 2022 [16]).

on a combination of 11-12 T superconducting magnets, compact and ultraprecise superconducting radio-frequency cavities for beam rotation, as well as 100-m-long high-power superconducting links with zero energy dissipation. In addition, the higher luminosities will make new demands on vacuum, cryogenics and machine protection, and they will require new concepts for collimation and diagnostics, advanced modelling for intense beam and novel schemes of beam crossing to maximize the physics output of the collisions.

The HL-LHC physics program is designed to address fundamental questions about nature of matter and forces at the subatomic level. Although the Higgs boson has been discovered, its properties can be evaluated with much greater precision with $\sim$ 10 times larger data set [17] than the original design goal of 300 fb^{-1}. The low value of the Higgs boson mass poses the so-called hierarchy problem of the Standard Model, which might be explained by new physics and from a better understanding of electroweak symmetry breaking. The imbalance between matter and anti-matter in the universe is the big open issue for flavour physics. Finally, there may be a new weakly interacting massive particle to explain the existence of Dark Matter. The HL-LHC will also allow further scrutiny of the new landscape of particle physics in case evidence of deviations from the SM, including new particles, are found. In the absence of any such hint, the ten-fold increase in data taking will nevertheless push the sensitivity for new physics into uncharted territory.

The ATLAS and CMS detectors will be upgraded to handle an average number of pile-up events per BX of $\sim$ 200, corresponding to an instantaneous luminosity of approximately 7.5×10^{34} cm^{-2}s^{-1} for operation with 25 ns beams at 7 TeV. The detectors are also expected to handle a line density of pile-up events of 1.3 events per mm per BX. ALICE and LHCb will be upgraded to operate at instantaneous luminosity of up to 2×10^{31} cm^{-2}s^{-1} and 2×10^{33} cm^{-2}s^{-1} respectively.

Chapter 2

The CMS Detector and the challenge for fast triggering

The **C**ompact **M**uon **S**olenoid (CMS) [18], [19] is a general purpose detector, whose main goal is to explore the $p-p$ physics at the TeV scale, with a particular focus on the search for the SM Higgs boson [20]. However, thanks to its incredible versatility it allows to cover many other physical processes at the LHC energy scale, with the final goal of probing different unproven models of the elementary structure of matter. It is also well suited for the study of top, beauty, and τ physics at lower luminosity as well as operating on the heavy ions physics program. It was designed to operate in $p-p$ (Pb−Pb) collisions at a center-of-mass energy of 14 TeV (5.5 TeV), with luminosities up to 10^{34} cm^{-2}s^{-1} (10^{27} cm^{-2}s^{-1}). The cylindrical design is structured in several layers, each dedicated to detecting a specific type of particle. The detector is primarily optimized to identify and measure muons, photons, and electrons with high precision. Key components include a high-performance muon system, a central tracking system capable of excellent track reconstruction, a high-quality electromagnetic calorimeter, and a hadronic calorimeter with good energy resolution. The entire detector is enveloped by a strong magnetic field, generated by a superconducting solenoid, which enables precise measurement of muon momentum.

The CMS collaboration consists in over 4000 particle physicists, engineers, computer scientists, technicians and students from around 200 institutes and universities from more than 40 countries.

2.1 The different subdetectors

The CMS detector consists in a cylindrical barrel, built of five slices, and two disk-like endcaps. The overall detector length is 21.6 m, its diameter is around 15 m and it has a total weight of approximately 12500 tons. It is made up of different layers, as illustrated in Figure 2.1. Each of them is designed to trace and measure the physical properties and paths of different kinds of subatomic particles. Furthermore, this structure is surrounded by a huge solenoid based on superconductive technologies, operating at 4.4 K and generating a 3.8 T magnetic field.

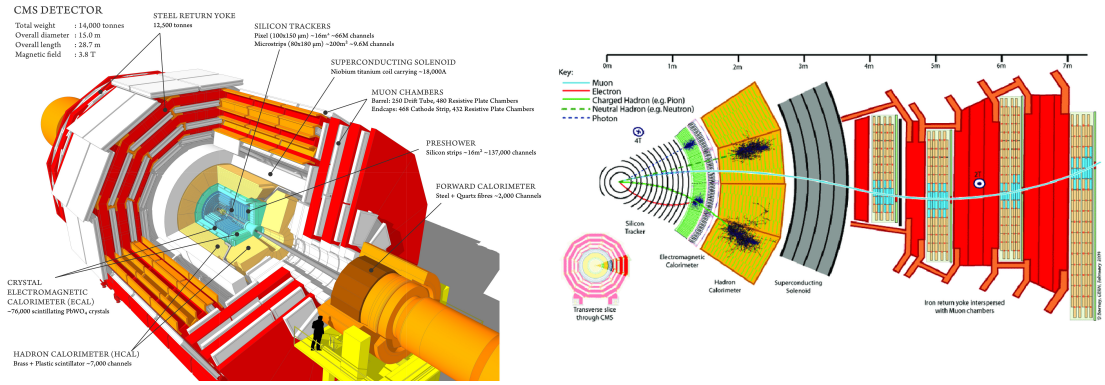


Figure 2.1: Overall view of the CMS detector.

In CMS, a right-handed coordinate system can be defined (schematically shown in Figure 2.2): 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. Due to the cylindrical symmetry of the CMS detector, a polar coordinate system is often used to describe the position and momentum of particles. It is defined by a polar angle θ measured with respect to the z-axis, defined in the range $0 \leq \theta \leq \pi$, and an azimuthal angle ϕ measured in the $x-y$ plane from the x-axis which can take values $0 \leq \phi \leq 2\pi$.

In a collision, the center of mass is boosted along the z-axis with respect to the laboratory frame. Therefore, the kinematics are usually described by the coordinates p_T, y, ϕ and m, where ϕ is the azimuthal angle, m is the invariant particle mass, p_T the transverse momentum defined as $p_T = p\,\sin\theta = \sqrt{p_x^2 + p_y^2}$, and y is the rapidity defined as

$$y = \frac{1}{2}\ln\left(\frac{E + p_z}{E - p_z}\right) \tag{2.1}$$

An ultra-relativistic approximation ($|\vec{p}| \gg m$) of the rapidity y, known as pseudorapidity η, is used in most cases:

$$y \approx \eta = \frac{1}{2}\ln\left(\frac{|\vec{p}| + p_z}{|\vec{p}| - p_z}\right) = -\ln\left(\tan\frac{\theta}{2}\right) \tag{2.2}$$

where E, $|\vec{p}|$ and p_z are the energy, the 3-momentum and the component along the z-axis of a particle.

As previously stated, in order to achieve high detection coverage of the physics produced in the collisions, the CMS is made up of various subdetectors with very different characteristics and technologies. Here is a quick rundown of them, with a slightly major focus on the so-called muon system, due to its importance and being the main aspect that makes CMS stand out with respect to other general purpose HEP detectors. [21], [22]

2.1.1 The Tracking System

The Tracker is a crucial component in the CMS design. It measures particles' momentum through their path, the greater is their curvature radius across the

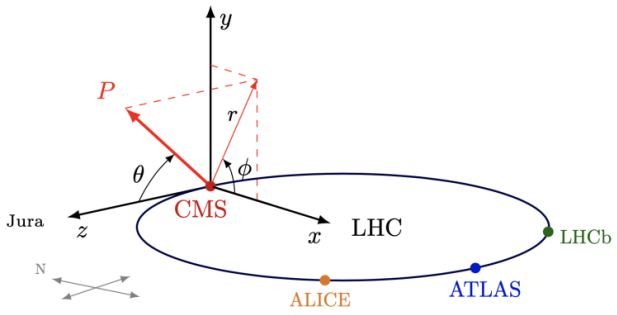


Figure 2.2: The CMS internal coordinate system.

magnetic field, the larger is their momentum. The Tracker is able to reconstruct muons, electrons and hadrons paths as well as tracks produced by short-lived particles decay, such as b quarks. It has a low degree of interference with particles and a high resistance to radiation: these are important characteristics due to the high radiation environment in which the Tracker is installed. On the other hand, the detector has to feature high granularity and fast response in order to keep up with the rate of particles crossing the Tracker (about 1000 particles from more than 20 overlapping $p-p$ interactions every 25 ns). These requirements on granularity, response time and radiation hardness lead to a design entirely based on silicon detector technology.

Considering that particle flux quickly decreases with radius, three detection regions can be identified: a fine granularity pixel detector system in its innermost parts, and silicon strips modules of different pitch in its central and external part. This design allows to have an occupancy of $\approx 1\%$ everywhere during high luminosity $p-p$ collisions, ensuring also a reasonable occupancy level during Pb-Pb ones (1% in the pixels, 210% in the silicon strip detectors). However, this high granularity leads to increased power consumption and, with the low temperatures required to allow a good functioning and to prevent radiation damage (around -10 °C), requires an efficient cooling infrastructure. At the same time, the overall material in the tracker must be minimized to reduce multiple scattering and other unwanted interactions. As a result, a careful balance in the tracker design was essential to meet these competing demands.

The pixel detector system consist of finely segmented silicon pixels, whose cell size is of $100 \times 150 \mu m^2$, placed on a silicon substrate. It is built to ensure a precise 3D vertex reconstruction. The system covers a pseudorapidity range up to $|\eta| < 2.5$ and the small pixel size allows to keep single channel occupancy per bunch crossing around 10^{-4} , even in the expected high flux scenario.

When a charged particle goes through one of this units, the amount of energy releases an electron with the consequent creation of an hole. This signal is than received by a chip which amplifies it. It is possible, in the end, to reconstruct a 3D image using bi-dimensional layers for each level.

This system was upgraded in 2017 to the current configuration [21] of 4 layers at radii between 29 mm and 160 mm in the barrel region of CMS and 3 disks in both endcaps with a radial coverage from 45 to 161 mm. The new tracker had the

innermost layer in the barrel region closer to the beam line and overall improved tracking efficiency. During LS2, the innermost layer was again replaced due to the radiation aging, providing also electronics upgrade which resolved issues with readout synchronization, noise shielding and radiation resistance experienced in Run 2. Even more upgrades [23] are planned for the so-called Phase-2 of CMS, which is planned to begin in 2029, after the Long Shutdown 3.

The two outermost regions of the tracking system are composed of several layers of silicon microstrip detectors. They consist of ten layers with about 10 million detector strips, divided into 15200 modules and scanned by 80000 microelectronic chips in a silicon area of about 200 m^2 , capable of detecting the passage of charged particles from $p-p$ collisions. Each module consists of three key components: a set of sensors, a support structure, and the electronics required for data acquisition. The sensors provide high responsiveness and excellent spatial resolution, enabling the detection of numerous particles within a confined space. They detect electrical currents generated by interacting particles and transmit the collected data. This part of the detector also needs to be kept at low temperatures (around -15 °C) to "freeze" radiation-induced damage to the silicon structure and prevent it from worsening over time.

2.1.2 Calorimeters

As previously mentioned, there are two types of calorimeters in the CMS experiment which measure the energy of particles emerging from the collisions.

Electromagnetic calorimeters measure the energy of particles subjected to the Electromagnetic interaction by keeping track of their energy loss inside the detector. The Electromagnetic Calorimeter (ECAL) of CMS [24] is a hermetic homogeneous calorimeter made of 61200 lead tungstate ($PbWO_4$) crystals, mounted in the central barrel part, with a pseudorapidity coverage up to $|\eta| = 1.48$, closed by 7324 crystals in each of the two endcaps, extending coverage up to $|\eta| = 3.0$. $PbWO_4$ scintillates when electrons and photons pass through it, i.e. it it produces light in proportion to the crossing particle's energy. ECAL is designed to reconstruct electrons and photons position and energy accurately, as well as to perform, in conjunction with the Hadron Calorimeter, precise measurement of hadronic jets. The measurement provided by the tracking system and the calorimetry are often complementary in CMS. However, while the tracker is able to identify only charged particles with a precision inversely proportional to the particle's p_T , the calorimeters can measure both charged and neutral particles with a resolution proportional to the increase of the particle's energy.

The use of these high density crystals guarantees a calorimeter which is fast, has fine granularity and it is radiation resistant: all important characteristics in the LHC environment. The low emitted light output (4.5 γ/MeV at room temperature), requires some photodetectors with an high gain operating in an high magnetic field. Therefore solutions based on Vacuum Photodiodes (VPT) and Avalanche Photodiodes (APD) are thus been adopted in endcaps and barrel respectively. As the latter has a response which is sensitive to temperature, thermal stability up to 0.1 °C is required to preserve the energy resolution. For energy

below 500 GeV, the energy resolution for ECAL can be parametrised as follows:

$$\left(\frac{\sigma}{\sqrt{E}}\right)^2 = \left(\frac{S}{\sqrt{E}}\right)^2 + \left(\frac{N}{E}\right)^2 + C^2 \tag{2.3}$$

where S is a stochastic term due to fluctuations in lateral shower containment, photostatistics and energy deposit in the pre-shower; N is the noise term related to electronics, digitisation and pileup; and C is a constant contribution caused by ECAL calibration, non-uniformities in the light collection and leakage from the back of the crystals. Studies performed during test beams [25] allowed to estimate these parameters to be: S = 2.8%, N = 12% and C = 0.30%.

The Hadronic Calorimeter

The Hadron Calorimeter (HCAL) [26] is used, together with the ECAL, to perform direction and energy measurements of hadronic jets and to estimate the amount of missing transverse energy (MET) of each event. The request to perform precise MET measurement implies the development of a very hermetic system, whose design is constrained by compactness requests and by the high magnetic field. In order to achieve such requirements, a sampling calorimeter system based on brass absorber layers alternated to active plastic scintillators has been built. The signal coming from active scintillators is read out with embedded wavelength-shifting fibers (WLS) and transported via clear fiber waveguides to hybrid photodiodes. The choice of brass as absorber material has been driven from its short interaction length (λ_I) and its non-magnetic nature.

In the barrel region, a barrel calorimeter (HB) covers an η region up to 1.4 and its readout segmentation (of $\Delta\eta \times \Delta\phi = 0.087 \times 0.087$) is tight enough to allow proper di-jet separation and mass resolution. The HB total depth increases as a function of η, raising from 5.15 λ_I at $\eta = 0$ to 10.15 λ_I at $\eta = 1.3$. Additionally, to obtain a better energy resolution of the barrel calorimeters, an outer calorimeter (HO) is placed outside the magnet coil, extending the total interaction length to about $11\lambda_I$.

In the endcap region, instead, an endcap calorimeter (HE) has been placed inside the magnet bore, covering the $1.4 < \eta < 3.2$ region. Its segmentation overlaps with the HB one and its average depth is about 10.5 λ_I . Outside the magnet a forward calorimeter (HF) covers the η region up to 5.2, guaranteeing the hermeticity of the detector. Due to the harsh radiation environment at high η, hard quartz fibers have been chosen as active medium. The energy resolution of the system (with E expressed in GeV), is dependent on the imperfect containment of the hadronic shower, resulting in a resolution sampling term up to 100% and a constant term of 5%, according to test beam studies [27]

$$\frac{\sigma}{E} \approx \left(\frac{65}{\sqrt{E}} \oplus 5\right)\% \qquad \text{for the barrel region,} \tag{2.4}$$

$$\frac{\sigma}{E} \approx \left(\frac{83}{\sqrt{E}} \oplus 5\right)\% \qquad \text{for the HE,} \tag{2.5}$$

$$\frac{\sigma}{E} \approx \left(\frac{100}{\sqrt{E}} \oplus 5\right)\% \qquad \text{for the HF.} \tag{2.6}$$

In LS2, Hybrid photo diodes (HPDs) were replaced with Silicon photomultipliers (SiPM), which have many advantages over HPDs, including high photon detection efficiency, excellent linearity, rapid recovery, better tolerance to radiation and insensitivity to magnetic fields. Radial segmentation was also increased, from 2 to 4 in the barrel region, providing improved depth measurement of hadronic showers. The new readout electronics increased readout granularity and redundancy, and improved quality of information sent to the Level-1 (L1) trigger.

2.1.3 The CMS Muon System

The muon (μ) is an elementary particle classified as a lepton, with an electric charge of -1 (+1 for antimuons) and a spin of $\frac{1}{2}$ with a mass of about 105 MeV, $\approx$ 200 times higher than the one of an electron. During $p-p$ and heavy ions collisions at LHC, muons are produced and they are mainly detected via the Tracker system, see Section 2.1.1, and the Muon System (whose original design is in Figure 2.3), a group of subdetectors dedicated to this task and placed in the outermost region of the CMS experiment.

As is implied by the experiment's middle name, the detection of muons is of central importance to CMS: precise and robust muon measurement was a central theme from its earliest design stages. The aim of the Muon System [28] is to provide a robust trigger, capable to perform BX assignment and standalone transverse momentum (p_T) measurement, perform efficient identification of muons and contribute to the measurement of the p_T of muons with energy as high as few hundreds of GeV or more. Good muon momentum resolution and trigger capability are enabled by the high-field solenoidal magnet and its flux-return yoke. The latter also serves as a hadron absorber for the identification of muons.

The experimental muon setup is made up of three different types of gaseous detectors with a different design, coping with the radiation environment and magnetic field at different values of η:

- 250 Drift Tube Chambers (DT), organized into 4 concentric stations interspersed among the layers of the flux return plates, are used in the barrel region (with $|\eta| < 1.2$) where a low residual magnetic field is present and track occupancy is low;

- The endcaps ($0.8 < |\eta| < 2.4$) are equipped with 540 Cathode Strip Chambers (CSC) with a faster and radiation resistant capability in order to cope with a higher particle flux and a non uniform magnetic field. There are 4 stations of CSCs in each endcap, with chambers positioned perpendicular to the beam line and interspersed between the flux return plates;

- To ensure redundancy and improve trigger performances, 610 Resistive Plate Chambers (RPC) complement the DT and CSC in both regions up to $|\eta| < 2.1$, due to their fast response and excellent time resolution but low spatial resolution, improving the precision in the muon trigger on the determination of the bunch crossing (BX) in which the muon has been created. RPCs are organized in 6 layers in the barrel Muon System, two in each of the first two stations, and one in each of the last two stations. In the endcap

region, the two outermost layers of CSCs have one layer of RPCs (divided in two stations) placed right after the CSC chambers. The inner endcap disk, instead, contains two layers of RPCs: one positioned on the inner side, just after the innermost CSC chambers, and the other on the outer side, along the iron yoke that forms the disk.

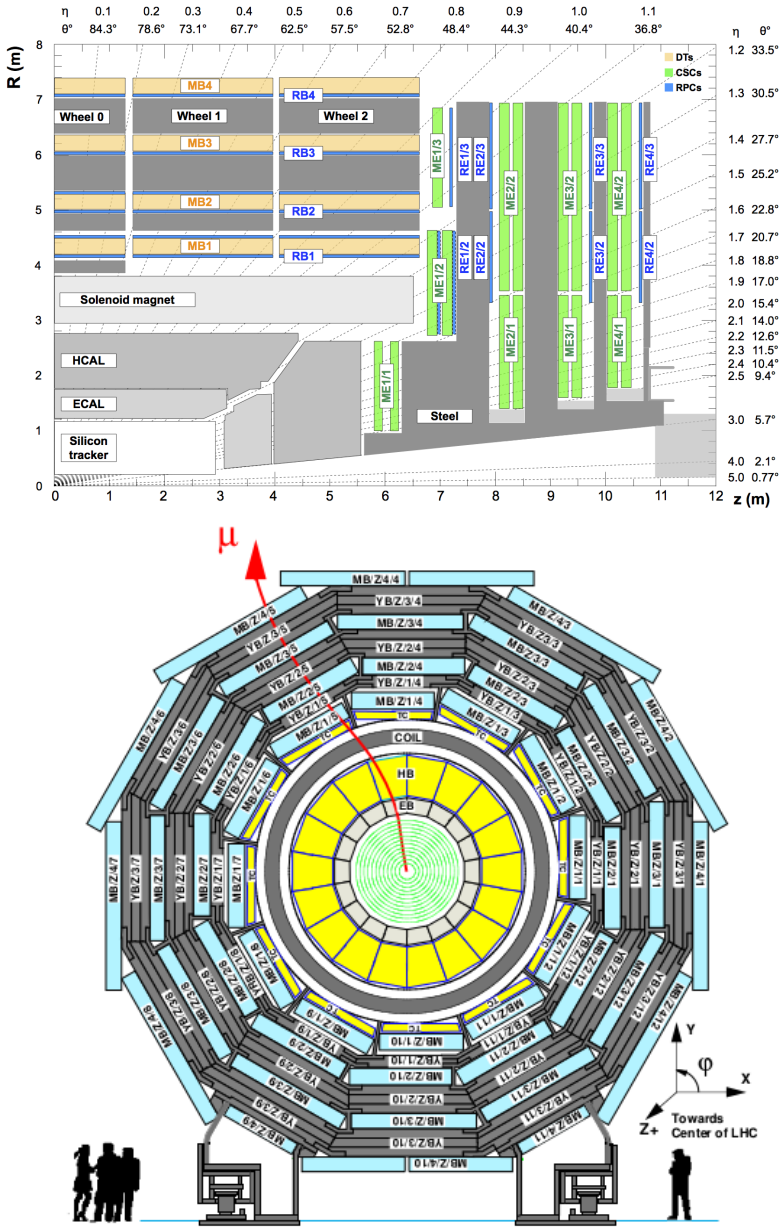
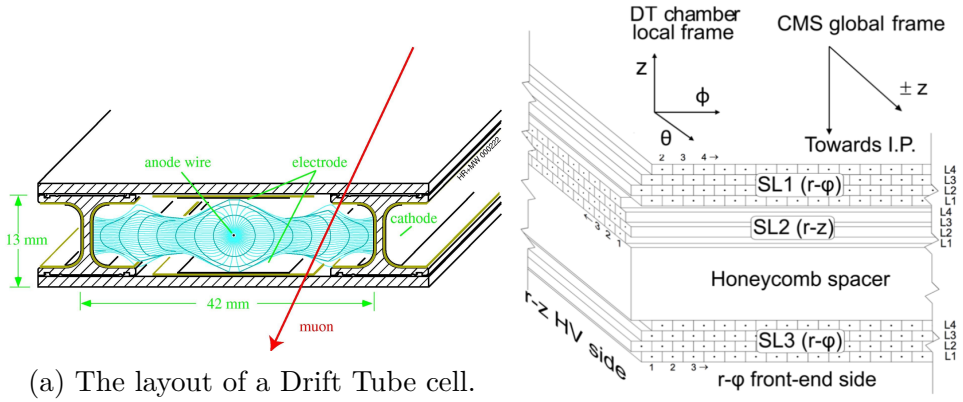


Figure 2.3: CMS detector longitudinal (top) view and barrel transverse (bottom) view. Describing LHC a reference frame is usually used in which the x-axis is pointed towards the center of the circular accelerator, the y-axis goes upward and the z-axis runs along the accelerated beam.



(a) The layout of a Drift Tube cell.

(b) Schematic view of a DT chamber.

Figure 2.4: DT chamber schematic in the Muon System (b) and a drift tube cell (a).

The Drift Tube Chambers

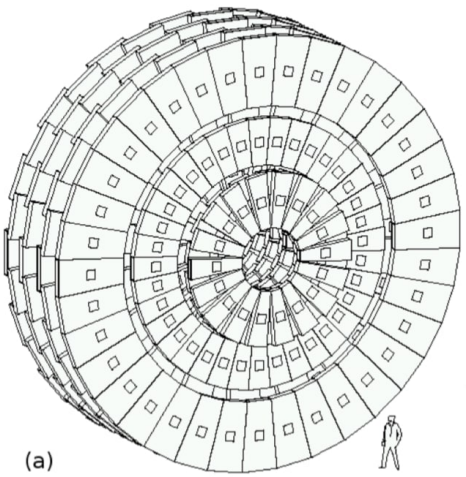
The basic detector element of the DT muon system is a drift tube cell (Figure 2.4a) of 42 mm × 13 mm and it contains a stainless steel anode wire with a diameter of 50 μm and length varying from 2 to 3 m. Cells are placed next to each other separated by "I"-shaped aluminium beams, making up layers contained in between two parallel aluminium planes. Strips of aluminium, deposited on both faces of each I-beam and electrically isolated serve as cathodes. Anode wires and cathodes are put at positive and negative voltage (typically +3600 V, -1200 V) respectively, and provide the electric field within the cell volume. The distance of the traversing track to the wire is measured by the drift time of ionization electrons; for this purpose, two additional positively-biased (+1800V) strips are mounted on the aluminium planes on both inner surfaces in the center of the cell itself, in order to provide additional field shaping and improve the space-to-distance linearity over the cell. The tubes are filled with a 85%/15% gas mixture of Ar/CO_2, which provides good quenching properties. The drift speed obtained is about 55μm/ns. Thus, a maximum drift time (half-cell drift distance) of $\sim$ 380 ns (or 15-16 BXs) is obtained. The choice of a drift chamber as tracking detector in the barrel was dictated by the low expected rate and by the relatively low intensity of the local magnetic field.

The DT system is segmented in 5 wheels along the z direction, each about 2.5 m wide and divided into 12 azimuthal sectors, covering $\sim$ 30° each. Drift tubes are arranged in 4 concentric cylinders, called stations, within each wheel, at different distance from the interaction point, and interleaved with the iron of the yoke. Each DT station consists of 12 chambers in each wheel, with the exception of the outermost station, MB4, whose top and bottom sectors are equipped of two chambers each, thus yielding a total of 14 chambers per wheel in that station. Each DT chamber is azimutally staggered with respect to the preceding inner one, in order to maximize the geometrical acceptance. The DT layers inside a chamber, as shown in Figure 2.4b, are stacked, half-staggered, in groups of 4 to form three superlayers (SL), two of them measure the muon position in the bending plane r-ϕ,

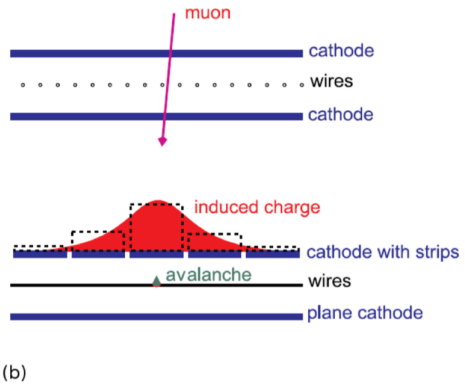
the other one measures the position along the z coordinate. However, the chambers in the outermost station, MB4, are only equipped with two ϕ superlayers. The overall CMS detector is thus equipped with a total of 250 DT chambers.

The Cathode Strip Chambers

The high magnetic field and particle rate expected in the muon system endcaps does not allow to use drift tubes detectors to perform measurements at large η values. Therefore a solution based on Cathode Strip Chambers has been adopted [29].



(a) Layout of the CSC subsystem.



(b) Schematic view of CSC chamber signal formation.

Figure 2.5: The Cathode Strip Chambers of the CMS endcap muon system.

Each endcap region of CMS has four muon station disks (from ME1 to ME4) of CSCs. These chambers are trapezoidal multi wire proportional chambers (MWPC) with segmented cathodes, capable of providing precise spatial and timing information, due to a short drift length which leads to fast signal collection, even in presence of large inhomogeneous magnetic field and high particles rates. A charged particle crossing the layers produces a signal which is collected by several adjacent cathode strips; since the strips are deployed radially, a charge interpolation provides a high resolution measurement of the ϕ-coordinate. The additional analysis of the wire signal offers the measurement of the orthogonal r-coordinate. Wire signals provide a fast response, useful for trigger purposes.

Each CSC has six layers of wires sandwiched between cathode panels. Wires run at approximately constant spacing, while cathode panels are milled to make six panels of strips running radially, one plane of strips per gas gap. Therefore, each chamber provides six measurements of the ϕ -coordinate (strips) and six measurement of the r-coordinate (wires).

ME1 has three rings of CSCs, at increasing radius, while the other three stations are composed of two rings. All but the outermost chamber of ME1 overlap in ϕ and therefore form rings with no dead area in azimuth. In station 2 to 4 there are 36 chambers covering 10° in ϕ making up the outer ring, and 18 chambers covering

20° in the inner ring, closer to the beam pipe. which are then arranged to form four disks of concentric rings placed in between the endcap iron yokes.

The Resistive Plate Chambers

Resistive Plate Chambers (RPCs) [30] are used both in the barrel and endcaps, complementing DT and CSC systems, in order to ensure robustness and redundancy to the muon spectrometer. RPCs are gaseous detectors characterized by a coarse spatial resolution, however they show a time response comparable to scintillators, and, with a sufficient high segmentation, they can measure the muon momentum at the trigger time and provide an unambiguous assignment of the BX.

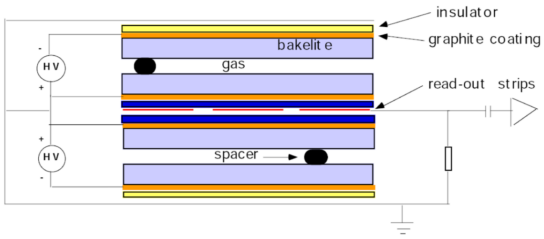


Figure 2.6: Schematic view of a CMS double gap RPC.

A RPC is formed, as shown in Figure 2.6, by two planes of material with high resistivity (Bakelite) separated by a 2mm gap filled with a mixture of freon ($C_2H_2F_4$) and isobutane ($i-C_4H_{10}$). The planes are externally coated with graphite, which forms the cathode for the high voltage (9.5 kV). The crossing particle generates an electron avalanche which induces a signal in the insulated aluminium strips placed outside the graphite cathodes ready to be read-out. CMS uses double-gap RPCs, with two gas-gap read-out by a single set of strips in the middle: this increase the signal on the read out strip, which sees the sum of the single gap signals. In the barrel the readout is segmented into rectangular strips 1-4 cm wide and 30-130 cm long, whereas the endcaps are equipped with trapezoidal shaped strips covering approximately the range $\Delta\phi = 5-6°$,$\Delta\eta = 0.1$.

In the barrel region, the system layout follows the DT segmentation and two RPC stations are attached to each side of the two innermost DT stations of a sector, whereas one single RPC is attached to the inner side of the third and fourth DT stations. This solution ensures proper detection of muons in the low p_T range within barrel trigger, which cross by multiple RPC layers before they stop in the iron yoke.

Muon chambers upgrades for Phase-2

The Phase-2 upgrade of the DT system foresees a replacement of the chamber on-board electronics, which is presently built with components that are neither sufficiently radiation hard to cope with HL-LHC conditions nor designed to cope with the expected increase of L1T rate foreseen for Phase-2 operation. DT chambers themselves will not be replaced, hence the existing detectors will operate

throughout Phase-2. In the upgraded DT architecture, time digitization (TDC) data will be streamed by the new on-board DT electronics (OBDT) directly to a new backend electronics, hosted in the service cavern, and called Barrel Muon Trigger Layer-1 (BMTL1). Event building and trigger primitive generation will be performed in the BMTL1 using the latest commercial FPGAs. This will allow building L1T TPs exploiting the ultimate detector time resolution (few ns) improving BX identification, spatial resolution and reducing the probability to produce multiple trigger segments per chamber for a given crossing muon (ghosts), with respect to the present DT local trigger.

In order to better exploit the intrinsic time resolution of the existing RPC system ($\sim$ 2 ns), and ensure the robustness of its readout throughout the HL-LHC era, the off-detector electronics (called Link System) will be replaced. Regarding L1T, the most relevant aspect of this upgrade is the increase of the readout frequency from 40 MHz to 640 MHz (reading out the detector 16 times per BX). As a consequence, each RPC hit provided to the muon track finders will have an additional time information featuring a granularity of one sixteenth of BX.

In order to increase the redundancy of the Muon System in the challenging forward region, new improved RPCs (iRPC) chambers will be installed in stations 3 and 4 (RE3/1 and RE4/1), extending the RPC pseudorapidity coverage to $|\eta| < 2.4$. The reduced bakelite resistivity and gap thickness (1.4 mm compared to 2 mm in the present RPC system) allows the detector to withstand the high rates anticipated in RE3/1 and RE4/1.

Moving on to CSCs, the on-chamber cathode boards on the inner rings of chambers ($1.6 < |\eta| < 2.4$) will be replaced in order to handle higher trigger and output data rates, together with the FPGA mezzanine boards on most of the on-chamber anode boards, in order to cope with higher L1T latency. Corresponding off-chamber boards that receive trigger and readout data will also be replaced to handle the higher data rates.

Finally, new Gas Electron Multiplier (GEM) chambers have started to be installed in the forward region $1.6 < |\eta| < 2.8$. The installation of GEM detectors allows a precise measurement of the muon bending angle in the first and second stations to be performed and used as a handle to control the muon trigger rate. The added sensitive detecting layers can increase the trigger efficiency and improve the operational resilience of the system. GEM foils have been demonstrated to be a suitable technology for the CMS forward region. A single GEM chamber is made of three GEM foils. A stack of two or six GEM chambers forms a superchamber. These superchambers will be installed in three distinct locations in the forward region ($1.6 < |\eta| < 2.8$), dubbed GE1/1 (already mounted before Run 3), GE2/1 and ME0, as shown in red and orange in Figure 2.7.

2.2 Trigger and data acquisition

At LHC, the beam crossing interval for protons is 25 ns, corresponding to a frequency of 40 MHz. Depending on luminosity, several collisions may occur at each crossing of the proton bunches. Considering that in the two general purpose experiments ATLAS and CMS each event has a size of about 1 MB and it is impossible

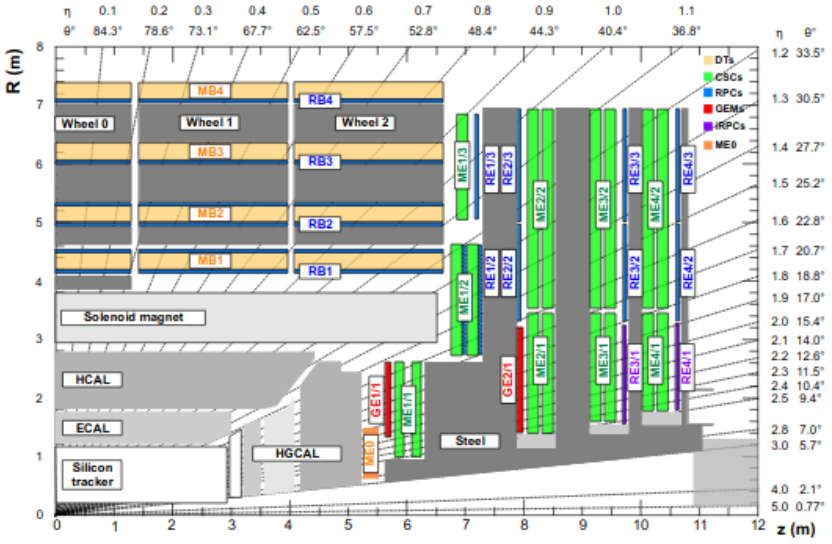


Figure 2.7: Longitudinal view of a quadrant of the CMS Phase-2 muon system. Different colors in the figure refer to different sub-detectors: DT (orange), RPC (light blue), CSC (green), iRPC (purple), GE (red), ME0 (orange).

to store and process this large amount of data, a reduction from the 40 MHz to the offline storage rate of approximately 1 kHz has to be achieved. Despite the high rejection factor, trigger algorithms have to be also quite sensitive to physical processes with very different probabilities in order to not saturate with much more common type of processes. This problem arises from the very large range of cross-sections of the processes produced at LHC, as shown in Figure 2.8

Since the time between crossings is too short to collect all the information coming from all the subdetectors and process it in a single step, an architecture based on different levels of increasing complexity has been adopted. In CMS this bandwidth reduction is performed in two main steps:

Level-1 trigger : it is based on custom electronics, and has to reduce the number of accepted events down to a maximum rate of 100 kHz, using coarse information coming from the muon detectors and calorimeters;

High Level trigger (HLT): it is based on software algorithms running on a farm of commercial CPUs. At this stage each event can take much more time for its processing, since the bandwidth has been already reduced and events are processed in parallel by different machines of the HLT. In this way the full information available from all the detectors (including the one of the silicon inner tracker) can be used, allowing a further reduction of event rate of a factor 10^{2-3}.

In Figure 2.9 a diagram of the CMS trigger chain is pictured, together with the rate of events which characterize each steps of the chain.

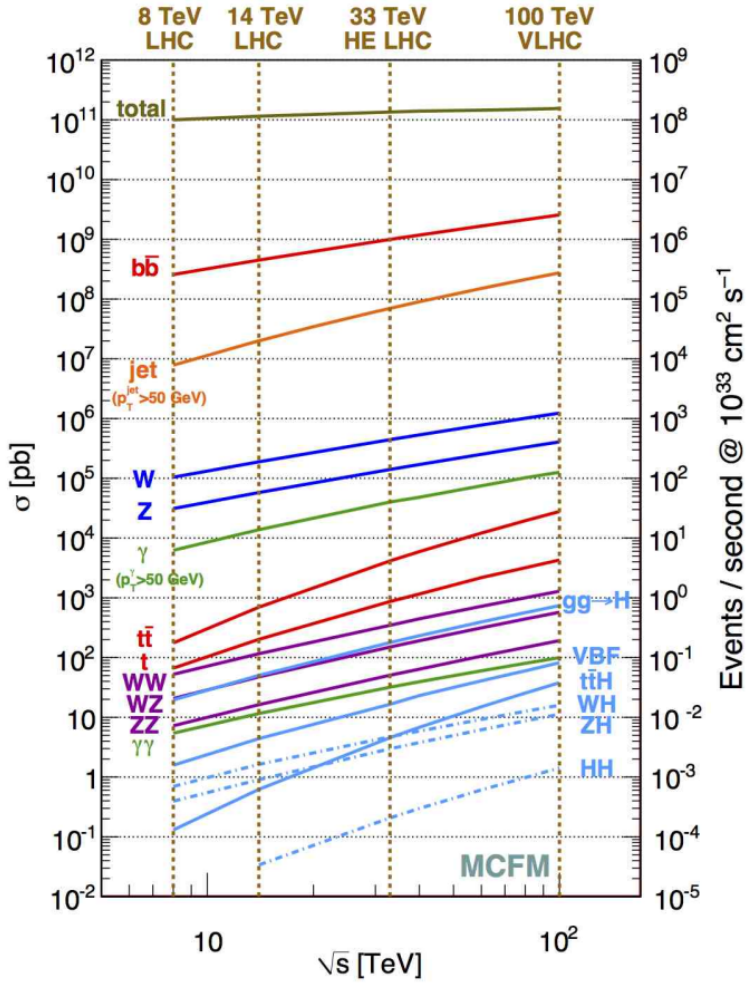


Figure 2.8: Example of the different cross sections for the main processes produced at the LHC.

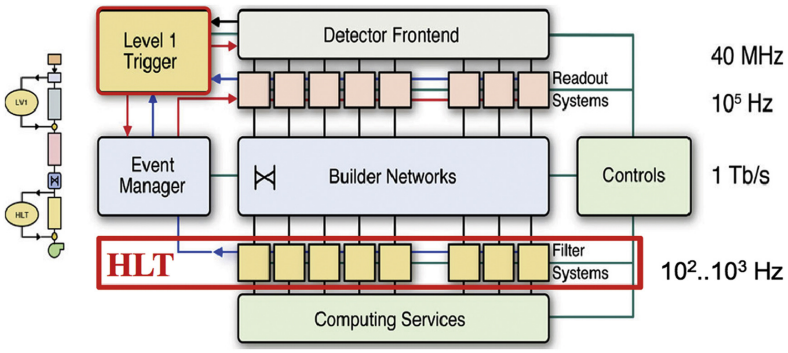


Figure 2.9: Schematic diagram of the CMS Trigger Chain.

2.2.1 The Level-1 Trigger system

The Level-1 Trigger (L1T) [31], [32] must cope with the machine frequency of 40 MHz and the time between collisions, 25 ns, is far too short for running any kind of

non trivial algorithm and for taking a decision on accepting that event. However, since dead time has to be avoided, complete information from the subdetectors is stored in First In First Out (FIFO) memories. In parallel, the trigger logic runs using a subset of the information, pipelined in small steps requiring less than 25 ns each, in order to start a new event processing every BX, even if the full processing requires a much longer time to complete. To make this possible, custom developed programmable hardware is used: Field Programmable Gate Arrays (FPGA) are used where possible, but also application-specific integrated circuits and Programmable Lookup Tables are taken into account to complete each processing step in time. At the end of the logic chain a decision is taken. If the event has to be kept, the FIFO memories containing the detector data are read and sent to the HLT. The maximum time available for the trigger logic to take a decision is determined by the amount of BXs for which the detector data can be stored into FIFOs, and corresponds to 4 μs.

The L1T at CMS is further subdivided into three major subsystems: the Muon Trigger, the Calorimeter Trigger and the Global Trigger. The first two systems process information coming from, respectively the muon spectrometer and calorimeters and do not have to perform the task of selecting events by themselves. On the other hand, they identify and perform sorting on various types of *trigger objects* (i.e. electron/photon, jets, muons) and then forward the four best candidates of each kind of trigger object to the Global Trigger where the final decision is made, as shown in Figure 2.10. This last selection can be performed considering only one type of object (e.g. selecting events where μ have a $p_T > 22$ GeV) or combining queries regarding more trigger objects.

For Run 3, a demonstration L1-scouting system has been introduced [33]. It passively receives L1 trigger data at the full LHC collision rate (nominally 40 MHz, effectively over 30 MHz) via specialized FPGA boards and saves the output for analysis. The advantage of looking at the full-rate trigger information is in allowing searches for rare processes whose signatures are difficult to detect by the trigger.

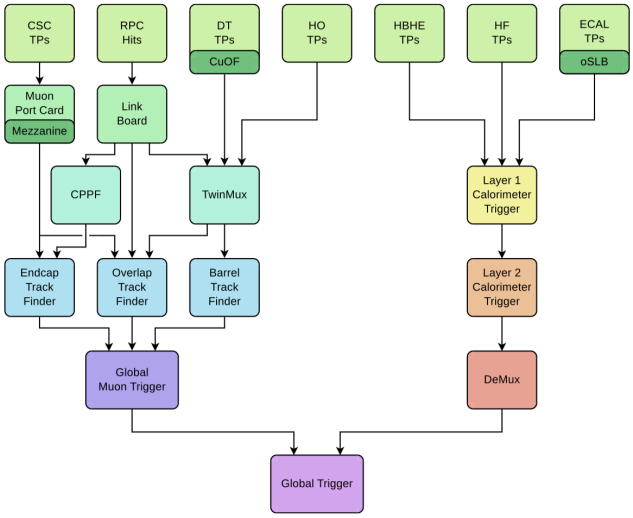


Figure 2.10: Schematic diagram of the CMS Level-1 trigger.

The Level-1 Muon Trigger

The Level-1 Muon trigger is designed to reconstruct muon position and p_T and to assign the particle's origin in terms of BX.

During the LHC Run-2 the L1T had to cope with an increase of the total event rate of roughly a factor 6 compared to the limits reached during the first LHC run (Run-1). In view of this increase in luminosity the L1T chain of CMS underwent considerable improvements.

In the upgraded L1T, the architecture of the electronics devoted to muon tracking follows a geographical partitioning. As we can see on the left of Figure 2.10, Trigger Primitives (TP) from the CSCs are sent to the Endcap Muon Track Finder (EMTF) and the Overlap Muon Track Finder (OMTF) via a mezzanine on the muon port card. Endcap RPC hits are sent via the link board to the concentrator pre-processor and fan-out (CPPF) card and barrel RPC hits are sent to the TwinMux concentrator card. DT trigger primitives are sent to the TwinMux card via a copper to optical fiber (CuOF) mezzanine. The TwinMux builds "Superprimitives", which combine the very good spatial resolution of DT trigger segments with the superior timing properties of RPC hits, improving the efficiency and the quality of the information used in the following steps.

The EMTF receives RPC hits via the CPPF card. In addition to the CSC hits, the OMTF receives DT hits and RPC hits via the CPPF and the TwinMux, which also provides DT and RPC hits to the Barrel Muon Track Finder (BMTF). The Global Muon Trigger sorts the muons, performs duplicate removal and sends the best eight muons to the Global Trigger.

2.2.2 The High Level Trigger and DAQ

The CMS High Level Trigger [34] has the task to further reduce the event rate from the L1T to ≈ 1 kHz, as required by the storage system and the offline processing of events. In order to achieve this reduction, the HLT performs an analysis similar to off-line event reconstruction relying on a farm of commercial processors.

The architecture of the CMS data acquisition system (DAQ) is shown schematically in Figure 2.9. The detector front-end electronics are read out in parallel by the Front-End System (FES) that format and store the data in pipelined buffers. These buffers must be connected to the processors in the HLT farm, and this is achieved by the large switch network (Builder Network). From the buffers, data is moved by the Front End Drivers (FEDs) to the Front End Readout Links (FRLs) which are able to get information from two different FEDs. Information coming from different FRLs is then sent to the Event Builder system in charge of building the full event. At this stage, data reaches the CMS surface buildings while beginning the reconstruction phase. After the assembly phase, each event is sent to the Event Filter where HLT algorithms, together with some Data Quality Monitoring operations, are performed. Filtered data is then separated into several online streams, whose content depends on trigger configurations (e.g. all data collected by single muon triggers), and is sent to a local storage system before being migrated to the CERN mass storage. Two systems complement this flow of data from the Front-ends to the processor farm: the Event Manager, responsible for the actual

data flow through the DAQ, and the Control and Monitor System, responsible for the configuration, control and monitor of all the elements.

The data acquisition for Run 3 satisfies similar requirements for readout as in Run 2, handling approximately 200 GB/s of data flow at rate above 100 - 110 kHz with event sizes of $\approx$ 1.6 MB (at nominal Run 3 LHC conditions). Due to end-of-life of server and network equipment used in Run 2, the system between the detector frontend and the HLT was upgraded with more recent computer and network technologies. A network of Ethernet servers used for data-to-surface transport to the DAQ system was replaced by a chassis-based 100 Gbit/s Ethernet switch, which can flexibly route all TCP traffic from readout cards to readout servers connected via 100 Gbit/s Ethernet links. Approximately 50 of these nodes, equipped with AMD Rome architecture CPUs, serve also as nodes of the so called folded event builder network architecture [35], performing the event-building (EVB), a process of collecting all disparate readout event data in one location, together with a second 100 Gbit/s chassis-based switch which supports remote Direct Memory Access (RDMA). The second chassis switch supports also other DAQ components, such as the data transfer to Tier-0 at CERN for permanent storage. Fully built events are delivered to the HLT, a cluster of 200 nodes integrated into the DAQ data flow via the same chassis-based switch. These nodes are equipped with powerful dual AMD Milan 7763 CPUs, 256 GB RAM, and two Nvidia T4 GPUs. In this way, HLT is able for the first time to run some of the reconstruction algorithms on GPUs facilitated by the CMS Software framework support for the offloading of computing workloads. Initially this is supported for Pixel, HCAL and ECAL reconstruction with approx. Approximately 40% of the CPU capacity is offloaded to GPUs.

2.2.3 Global Event Reconstruction and Particle Flow Algorithm

This sections aims at concluding the description of the CMS' data acquisition methods with an overview on the identification and reconstruction of physics objects candidates coming from each collisions event, using a particle flow (PF) technique [36].

The PF approach relies on the combination of information coming from the CMS subdetectors, in order to give a global and coherent description of the events, under the form of a reconstructed particle candidate. Events collected by CMS are centrally processed with reconstruction algorithms referred to as "event reconstruction", starting from raw data and giving as output a collection of detected particles with properties like momentum or angle.

Firstly, individual particles are classified into mutually exclusive types: muons, electrons, photons, charged hadrons, and neutral hadrons. Track trajectories are reconstructed starting from hits in the tracker and then linked to energy deposits in the ECAL (for electrons) or both the ECAL and HCAL (for charged hadrons). Photons are identified as energy clusters in the ECAL that are not matched to the extrapolation of any charged particle trajectory from the tracker. Muons are initially identified as tracks in the central tracker and then matched with either

tracks or multiple hits in the muon system, potentially associated with calorimeter deposits. Charged and neutral hadrons produce hadronic showers in the ECAL, followed by absorption in the HCAL; the resulting clusters are used to estimate their energy and direction. Additionally, an indirect measurement of non-interacting, neutral particles provided by the calorimeters is crucial for computing the MET, which could be a signature of new particles and phenomena.

Finally, higher-level physics objects, such as jets, MET, τ leptons, and lepton isolation, are built from PF candidates. Jets are clustered using the anti-k_T algorithm [37], with good momentum and spatial resolution due to the excellent ECAL granularity and high-quality tracking detectors. τ leptons, characterized by their short lifetime, are identified through their hadronic decays by reconstructing intermediate resonances. The MET vector is calculated as the opposite of the transverse momentum sum of all final state particles reconstructed in the detector.

2.3 The CMS Phase-2 Level-1 Trigger upgrade

In order to fully exploit the HL-LHC running period, major consolidations and upgrades of the CMS detector are planned [38]–[42]. The collision rate and level of expected pileup imply very high particle multiplicity and an intense radiation environment. The intense hadronic environment corresponding to ~ 200 simultaneous collisions per beam crossing, imposes serious challenges to the L1T system requirements in order to maintain performance.

The Phase-2 upgrade [43] of the Level-1 trigger system aims to not only maintain but also improve the signal selection efficiency compared to Phase-1, together with increasing the potential to identify unconventional signatures indicative of new physics. This upgrade will enhance precision in physics measurements, especially in previously challenging areas such as forward detector regions. Most importantly, state-of-the-art techniques used in offline reconstruction and analyses, such as the global event reconstruction based on particle-flow techniques, become possible at the L1 trigger, with the availability of L1 tracks delivered by the upgraded Outer Tracker, together with the benefits from the increased granularity of the calorimeter information. Thanks to the experience gained in Phase-1, the upgraded trigger system will rely on modern technologies like FPGAs and high-speed optical links to optimize data handling, ensuring adaptability to changing LHC conditions.

This upgrade of the trigger and DAQ system will keep a two-level strategy while increasing the L1T maximum rate to 750 kHz. The total latency will be increased to 12.5 μs to allow, for the first time, the inclusion of the tracker information at this trigger stage. The extended latency will enable more advanced object reconstruction and identification, along with the evaluation of complex global event quantities and correlation variables, optimizing physics selectivity, while also allowing the implementation of sophisticated algorithms such as particle-flow reconstruction techniques and machine learning-based approaches. These new features will be implemented on top of the 40 MHz scouting system, already added in Run 3, which can harvest the trigger primitives produced by sub-detectors and the trigger objects produced at various levels of the trigger system. The concept of trigger scouting has been introduced in CMS at the HLT. It is based on the use of physics

objects reconstructed as a by-product of the triggering process to perform data reduction and analysis, only storing high-level information for selected events, thus overcoming the rate to storage limitations of the DAQ. In a very similar way, the Level-1 scouting system uses L1T reconstructed objects and quantities, selecting and analyzing them on the fly at the collision rate. This system has the additional advantage of allowing systematic search of correlations among multiple contiguous bunch crossing, and can be used to scrutinize the collision events and identify potential signatures unreachable through standard trigger selection processes.

In order to successfully integrate and commission this complex upgraded L1 trigger, an approach similar to that adopted in the Phase-1 upgrade was chosen, where part of the system started to run in parallel with the established system during Run-3 operations. The muon system in place now will remain in Phase-2 and is already used to test new algorithms and gain confidence in their development.

2.3.1 Upgrade Requirements and Conceptual Design

To summarize, with the increased complexity of detectors and readout electronics under HL-LHC conditions, the CMS Level-1 trigger system will undergo significant upgrades to handle higher luminosity and pileup. These upgrades will utilize advanced FPGAs and processors to optimize the reconstruction, identification, and calibration of trigger candidates, leveraging high-granularity detector data. High-speed optical links will enable global data aggregation for precise event processing, while a flexible and modular architecture will ensure adaptability to evolving conditions and physics requirements, supporting more sophisticated selection algorithms and topologies.

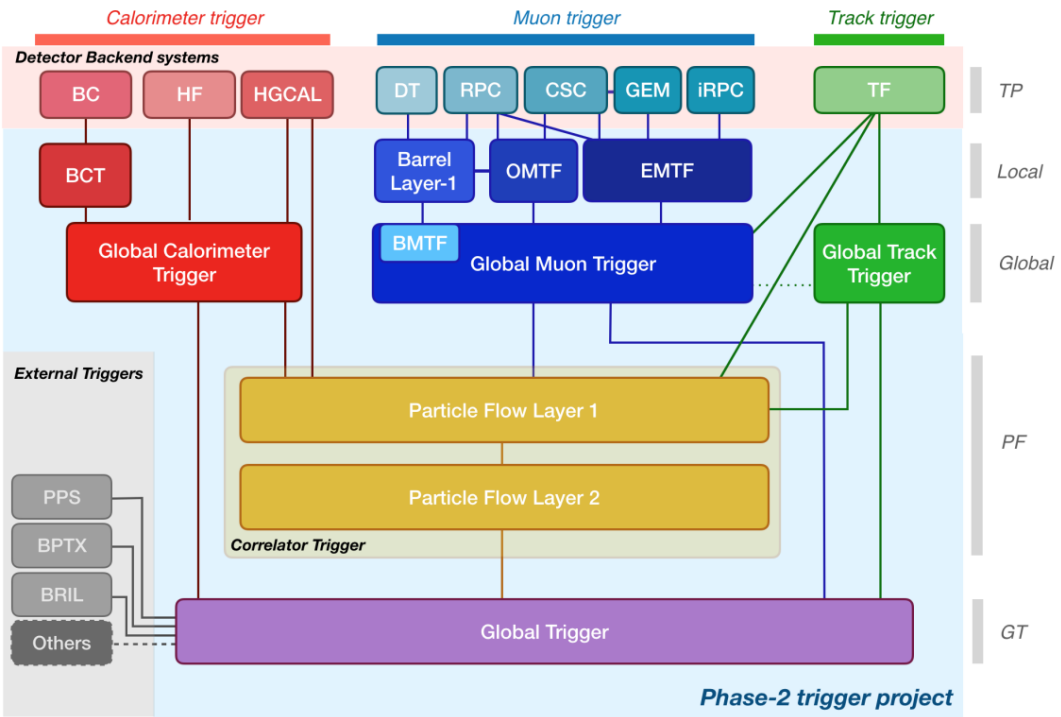


Figure 2.11: Functional diagram of the CMS L1 Phase-2 upgraded trigger design.

The functional diagram of the architecture and data flow of the Phase-2 trigger system is presented in Figure 2.11. With the 12.5 μs latency, not only is information from the calorimeters and muon detectors used (as in the Phase-1 system), but the information from the new tracker and high-granularity endcap calorimeter can also be included. The total output bandwidth considered is 750 kHz. Given the complexity and large data volume produced by the detector, a significant fraction of the computing of trigger quantities, such as trigger primitives completed by particle identification variables, takes place in the detector backend electronics.

A key feature of the proposed system is the introduction of a correlator layer that combines data from multiple sub-detectors using advanced algorithms to create higher-level trigger objects, improving selectivity to levels closer to those of the HLT.

In order to ensure design flexibility and robustness by generating complementary types of trigger objects and thus achieve the best physics selectivity, four independent data processing paths, each tailored to meet specific physics requirements and provides independent trigger criteria, are implemented:

Calorimeter Trigger path A barrel calorimeter trigger (BCT) and the backend of the new High Granularity Calorimeter (HGCAL) are used to process high-granularity information from the calorimeters to produce high-resolution clusters and identification variables to be used for later processing. Outputs from the BCT, HGCAL and the hadron forward calorimeter (HF) are sent to a global calorimeter trigger (GCT).

Track Trigger path Tracks from the Outer Tracker are reconstructed in the track finder (TF) processors as part of the detector backend. The reconstructed track parameters and track reconstruction quality flags are provided to the trigger system to achieve precise vertex reconstruction and matching with calorimeter and muon objects. This key feature maximizes the trigger efficiency while keeping the trigger rate within the allowed budget. A global track trigger (GTT) will be included, to reconstruct the primary vertices of the event along with tracker only based objects, such as jets and missing transverse momentum.

Muon Trigger path The processing of trigger primitives by muon track finder algorithms is organized as in the Phase-1 system covering the three separate regions: barrel, overlap and endcaps. Standalone muons and stubs as well as L1 tracks are sent to a global muon trigger (GMT). A muon stub contains reconstructed local information extracted from the detector hits in each of the muon stations. It includes position, bend angle, and timing, depending on the station. Beyond the removal of muon duplicates and misreconstructed muons, the main feature of the GMT is the generation of track-matched muons and L1 tracks matched to muon stubs, the so-called tracks plus muon stubs. Tracks can either be received directly from the TF or through the GTT. Interconnections established between GTT and GMT offer the possibility to provide the vertex information to the GMT algorithms if required.

Particle-Flow Trigger path The correlator trigger (CT) occupies a central role in the design. The CT implements sophisticated algorithms to produce

higher-level trigger objects, applies particle identification, and provides a sorted list of objects to the global trigger. The structure of the CT is organized in two layers with a first layer, referred to as "Layer-1" producing the particle-flow candidates, which are constructed from the matching of calorimeter clusters and tracks, and a second layer, called "Layer-2", building and sorting final trigger objects and applying additional identification and isolation criteria.

Finally, outputs from the GCT, GMT, GTT, and CT are combined in the global trigger (GT), which calculates a trigger decision based on a menu of algorithms. The GT has resources to evaluate sophisticated correlation variables among various types of objects to increase the selectivity. The Level-1 Accept signal is transmitted to the Trigger Control and Distribution System (TCDS), which distributes it to the detector backend systems, initiating the readout to the DAQ.

The division of labor achieved through the implementation of global triggers (GCT, GMT and GTT), allows the reduction of the FPGA resources required to implement the particle-flow algorithm in the CT, meaning that enough headroom is available to further optimize the algorithms.

2.3.2 Trigger algorithms for the HL-LHC

The upgraded L1 system would more closely replicate the full offline object reconstruction, instead of making use of simple subsystem variables, to make a better optimized selection. The Phase-2 trigger algorithms foreseen can thus be used to reconstruct a large variety of objects: standalone, which are reconstructed from single detector information (including tracker-only objects), standalone matched to L1 tracks, and particle-flow. The trigger decision can rely on the redundancy of these objects to achieve the best possible efficiency while keeping the trigger rate under control.

Given the discovery targets of HL-LHC, the trigger object requirements are not only driven by the need to maintain physics selection thresholds to match those of Phase-1, but also by having to provide the selection of exotic signatures, including displaced objects. The algorithm implementation in firmware greatly benefits from the introduction of High-Level-Synthesis software (see Section 3.2.2) that could be used to design advanced machine learning trained variables or even iterative processes in the core of the trigger system. This section provides an overview on some of the baseline algorithms that have been developed with the minimum requirement of meeting the challenges of the HL-LHC.

Triggering on electrons and photons

Many standalone electron and photon trigger reconstruction techniques are being investigated to optimise both the response and the position resolution for the purpose of achieving the highest possible track matching efficiency. Both identification criteria and isolation variables based on calorimeter can be used, together with tracking information, to reduce the background level. Given the intense running conditions foreseen, the algorithms have also to be designed to be pileup resilient.

An electron finder can be built in the barrel region which uses the crystal information from the ECAL. A 5×3 crystal matrix ($\Delta\phi \times \Delta\eta = 0.087 \times 0.052$) is used to define the maximum size of the electron footprint in the ECAL. As in the Phase-1 algorithm approach, the extension in ϕ is motivated by the necessity to recover energy lost through bremsstrahlung, i.e. energy lost in the material when deflected. An improved position resolution can be achieved using a weighted-energy sum around the seed crystal, with a seeding threshold of $E_T > 1$ GeV. Extra shower shape features are used as identification criteria and the matching of the clusters with tracks is performed using an extrapolation to the ECAL surface.

On the other hand, the starting point of the electron reconstruction algorithm in the endcap region would be the cluster reconstructed in the backend electronics of the HGCAL. Further identification of the electromagnetic object is performed through a multivariate approach optimized to exploit the input variables transmitted from the HGCAL. Dedicated boosted decision trees (BDTs) can be trained on signal and background to achieve an optimal signal efficiency while rejecting pileup-induced clusters. Bremsstrahlung recovery is performed as well as an energy calibration of the final e/γ candidate. The availability of tracking information improves the reconstruction of isolated photon candidates. However, reconstructing electron tracks using the TF is challenging due to the bremsstrahlung radiation that occurs as electrons pass through the tracker material. The extended tracking system, which was originally designed to reconstruct displaced trajectories, could help recover the efficiency of electron track reconstruction.

Triggering on hadronic jets, taus and energy sums

Triggering on hadronic signals has always represented a challenge for detectors operating in an intense hadronic environment. Algorithms developed for the Phase-1 Level-1 trigger system are optimized to provide thresholds adequate for physics using calorimeter-only information and their rate of triggering and, in particular, the their performance when missing energy transverse energy is involved, is strongly correlated to the number or pileup events and the filling scheme of the machine. This is why the level of pileup expected at the HL-LHC could be a problem for the performance of hadronic triggers, and it explains the study of new pileup mitigation algorithms exploiting the full capabilities of the Phase-2 detectors. As calorimeter-only algorithms are expected to have high thresholds, complementary approaches are proposed with tracker-only information, track-matched calorimeter objects and higher-level objects.

- Calorimeter-only jet finding algorithms would use barrel ECAL and HCAL information, endcap HGCAL and forward HF information. Although various configurations were considered, a simple square geometry of 7×7 trigger towers around a local maximum gives acceptable performance while keeping the pileup contribution to a minimum. The jet window definition corresponds approximately to the cone size of 0.4 used by the offline anti-k_T algorithm [37]. Similarly to the Phase-1 algorithm, a tower-by-tower pileup correction depending on the level of pileup and η would be applied prior to jet clustering.

- Tracker-only jet finding is performed on a set of tracks from the track finder

passing purity requirements to keep the trigger object resilient to pileup. Track clustering makes no use of the primary vertex information. In order to optimize the latency, primary vertex computation and jet clustering are performed in parallel. By considering a smaller z-range of tracks from the interaction point is possible to gain more robustness against pileup. The clustering of tracks in the $\eta - \phi$ plane is performed using a nearest-neighbour approach in two one-dimensional steps. The maximal jet size corresponds to $\Delta R = \sqrt{\Delta\eta^2 + \Delta\phi^2} = 0.3$, while the jet p_T is computed as the sum of each track p_T associated to it.

- The particle-flow based jet finding consists of building jets from particle-flow candidates grouped into pseudo trigger towers, equivalent to 0.083×0.087 in the $\eta - \phi$ plane, which are then clustered into a 7×7 tower window around a local minimum. The jet momentum is computed as the sum of the objects' momentum in this window and the seeds coordinates η and ϕ are associated to the jet as its position.

Moving on to τ, the benefit of developing a dedicated reconstruction and identification algorithm for those decaying hadronically (τ_h) was demonstrated in Run-2. Like for the jet finding, calorimeter-based τ_h are built from trigger towers. Given that they are narrow jets and that several decay products may be producing more than one cluster separated along the ϕ direction due to the magnetic field, a 3×5 tower window, equivalent to $\Delta\eta \times \Delta\phi = 0.261 \times 0.435$, is chosen to optimize p_T resolution. This window is used in conjunction with a 7×7 one to define the isolation regions to actually identify the τ_h while maintaining the rate under control. Although this approach performs well within the entire calorimeter acceptance, the enhanced granularity of the HGCAL detector allows the implementation of advanced identification techniques exploiting the τ shower characteristics, being its profile different from that of pileup induced particles. For example, a dynamic clustering of 3D-clusters could give optimum response, while trained BDTs could provide dedicated energy calibrations for each of the τ decay modes.

Finally, the triggering on missing transverse energy (E_T^{miss}) is a particularly challenging task for detectors operating in hadronic environment, especially when the average expected pileup is 200. This quantity is a key input for many signatures, including beyond Standard Model processes, in the L1 trigger. The use of L1 tracks is essential to achieve manageable rates for moderate thresholds. The algorithms pursued are either tracker-based or PF-based. The tracker-based approach considers tracks originating from the primary vertex and applying dedicated selection to reject misreconstructed tracks. The rate is considerably reduced using this approach. With particle-flow, the information of all sub-systems is used and further mitigation of pileup contributions is obtained with the PileUp Per Particle Identification (PUPPI), an algorithm that removes charged particles with tracks not originating at the primary vertex and downweights neutral particles based on the probability that they originate from pileup. Thresholds applied to particle-flow and PUPPI inputs in various η regions can be adjusted to optimize performance

Triggering on muons

The overall structure of the muon system for Phase-2 remains similar to the current one. The signals from the three partially-overlapping sub-detectors (CSC, DT, and RPC) are combined to reconstruct muons and measure their transverse momenta. Additional muon stations, such as iRPC, GEM and ME0, are installed in the forward region to extend the acceptance to $|\eta| = 2.4$ and $|\eta| = 2.8$ respectively. Following the approach of the Phase-1 trigger upgrade, the reconstruction of standalone muons uses information from all available sub-detectors simultaneously to build tracks in three distinct pseudorapidity regions, improving the muon reconstruction and increasing signal efficiency while reducing background rates. Given the improved sub-detector electronics readout for Phase-2, the muon chambers will provide finer information along with precise timing (≈ 1.5 ns) that can be exploited by the muon track finding algorithms. Each track finder uses an optimised track reconstruction algorithm and p_T assignment logic, and assigns a track quality corresponding to the estimated p_T resolution. Similarly to the existing system, the BMTF uses DT and RPC trigger primitives to reconstruct segments merged to obtain a muon candidate. The RPC fired strips are clustered before being used by the muon track finders.

A track finding approach based on a Kalman Filtering technique called Kalman barrel muon track finder (KBMTF) has been developed and already tested on data. As tracks can be reconstructed by KBMTFT with and without a constraint forcing them to originate from the primary vertex, displaced muons can be reconstructed with acceptable p_T resolution resulting in higher acceptance.

In the overlap region, the OMTF receives data from DT, RPC and CSC stations and reconstructs tracks by associating hits, using generated patterns from simulated events. This naive Bayes-classifier approach identifies the most likely muon p_T.

The muon endcap track finding algorithms exploit the information from up to 12 muon stations. In addition to CSC and RPC trigger primitives, the Phase-2 EMTF++ system proposed for this upgrade receives information for GEM (including ME0) and iRPC detectors. The standalone reconstruction algorithm looks for correlated CSC trigger primitives through multiple stations compatible with a muon track corresponding to predefined patterns. Consistent RPC primitives are associated to this track candidate and a trained deep neural network for p_T assignment, with and without beam constraint, is implemented.

The availability of tracks from the Outer Tracker allows another category of muons, with increased acceptance at low p_T or originating from regions with limited detector coverage, to be considered. The matching of standalone muons and tracks is performed optimally in each pseudorapidity region, so that, as in the offline or Phase-1 HLT cases, misreconstructed muons are reduced, and the p_T measurement accuracy is improved. Another complementary approach consists of propagating the tracks from the Outer Tracker into the muon detectors and associate stubs from at least two layers of the muon stations. This algorithm shows optimum performance for a large variety of physics signals while maintaining efficiency and providing robustness against detector aging. The possibility to correlate tracking and muon stubs information is used to produce trigger objects adequate to identify

heavy stable charged particles. Given the particularity of this signal, the candidate L1 track is matched to muon stubs from the same event or subsequent ones.

Global Trigger Algorithms

Global trigger algorithms refer to those that rely on correlations between physics objects or use advanced variables like invariant masses. This capability has greatly improved the selectivity of the Phase-1 trigger system, and is planned to be a key feature in the Phase-2 upgrade. These algorithms are implemented in the Global Trigger (GT), which provides customized triggers for specific physics analyses. Additionally, the Global Muon Trigger, Global Calorimeter trigger, and Global Tracker Trigger systems generate quantities based on information available upstream of the correlator trigger and the GT, allowing specific objects or variables to be directly combined with other physics data.

For instance, the GTT can calculate invariant masses of track combinations that pass quality cuts to trigger on specific light resonances. Future upgrades may include timing information from the MTD to flag out-of-time physics objects, potentially linked to Beyond Standard Model processes. Other global quantities, like centrality (important for heavy ion triggers), can also be incorporated. Furthermore, machine learning techniques are being explored as alternatives to traditional cut-based triggers, with tools like hls4ml enabling implementation on FPGAs (see Section 5.3.1). Early results show that machine learning approaches significantly improve the detection of signals such as Higgs boson production via VBF, especially in decay channels like H$\rightarrow b\bar{b}$ and invisible decays H $\rightarrow inv$ compared to standard triggers.

Chapter 3

Field Programmable Gate Arrays

In this section an introduction on the piece of hardware which shows promising results for fast Machine Learning inference, Field Programmable Gate Arrays, will be presented.

There are two main different way to tackle computation [44]: the hardware and the software approach. Computer hardware, such as application-specific integrated circuits (ASICs), provides highly optimized resources for quickly performing critical tasks, but it is permanently configured to only one application via a very expensive design and fabrication effort, which is especially costly in a scenario where the number of chips requested is of few units. Computer software, on the other hand, provides the flexibility to change applications and perform a huge number of different tasks, but is much less optimized than ASIC implementations in terms of performance, silicon area efficiency, and power usage.

Field programmable gate arrays (FPGAs) are peculiar devices that can be considered as a blend of the respective benefits mentioned above. Hardware circuits are implemented in their fabric, providing huge power, area, and performance benefits over software applications, yet can be reprogrammed cheaply and easily to implement a wide range of tasks. Just like computer hardware, FPGAs can be set up to perform a large number of operations in parallel using resources distributed across a single silicon chip. Such systems can be hundreds of times faster than microprocessor-based designs. However, unlike in ASICs, these computations are not permanently frozen by the manufacturing process. This means that an FPGA-based system can be programmed and reprogrammed many times.

3.1 The Computing Architecture

In order to efficiently implement computation on a particular hardware, it is always best to understand what goes on "under the hood". This is especially important in this context, where the boundary between software and hardware is particularly blurry.

Not unlike the typical desktop computer, where the Central Processing Unit (CPU) acts as a brain and orchestrates the other devices and peripherals attached to the machine, an FPGA is the central hub where the computation is carried out and how to interact with the peripherals is dictated in a reconfigurable computing

platform.

In very general terms, there are two main types of resources in an FPGA: *logic elements* (Section 3.1.1) and *interconnections* (Section 3.1.2): the logic elements take care of the arithmetic and logical functions, while through interconnections data is moved from one node of computation to another.

3.1.1 Logic Elements

One of the first concept learnt when starting a digital logic or computer architecture course is that any computation can be represented as a Boolean equation. In turn, any Boolean equation can be expressed as a truth table. From these simple objects, complex structures can be build that can do arithmetic, such as adders and multipliers, as well as decision-making structures that can evaluate conditional statements, such as the classic *if-then-else*. In other words, elaborate algorithms can be described simply by using truth tables.

One hardware element that can easily implement a truth table is the *lookup table*, or LUT. From a circuit implementation perspective, a LUT can be formed simply from an N-to-one multiplexer and an N-bit memory. In this way, a LUT simply enumerates a truth table. Therefore, using LUTs gives an FPGA the generality to implement arbitrary digital logic. Figure 3.1 shows a typical N-input

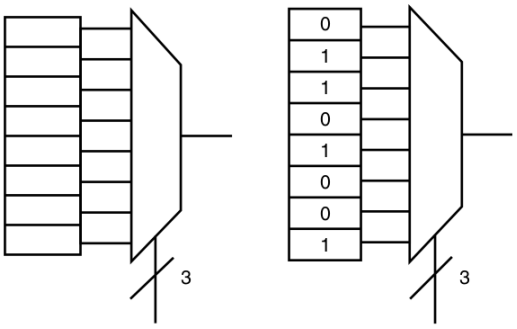


Figure 3.1: A 3-LUT schematic and the corresponding 3-LUT symbol and truth table for a logical XOR.

lookup table that could be found in FPGAs. The LUT can compute any function of N inputs by simply programming the lookup table with the truth table of the function intended for implementation. As depicted in the figure, to implement a 3-input exclusive-or (XOR) function with a 3-input LUT (also known as a 3-LUT), values are assigned to the lookup table memory so that the pattern of select bits chooses the correct row's "answer." Consequently, each "row" would produce a result of 0 except in the four instances where the XOR of the three select lines results in 1. More complicated functions, and functions of a larger number of inputs, can be implemented by aggregating several lookup tables together.

However, lookup tables are not sufficient to implement all of the functionality expected from an FPGA. Indeed, with just LUTs there is no way for an FPGA to maintain any sense of state, and therefore it is impossible to implement any form of sequential, or state-holding, logic. To remedy this situation, a simple single-bit

storage element can be added to the base logic block in the form of a *Data flip-flop* (D-FF), as shown in Figure 3.2.

Flip-flops are vital ingredients in all except purely combinational logic circuits [45]. They are basically circuits capable of holding a state (0 or 1). The D-FF is a type of flip-flop which delays the transfer of its input to its output based on a *clock* input.

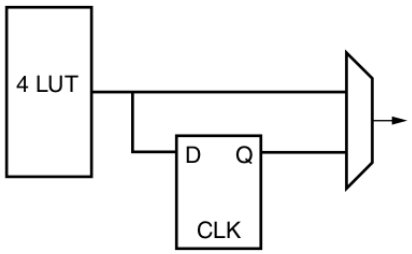


Figure 3.2: A simple lookup table logic block.

Finding the optimal size and number of LUTs per logic block is one of the direction for R&D in FPGA design. This open up the larger question of computational granularity in an FPGA. On one end of the spectrum, the rather simple structure of a small lookup table (e.g., 2-LUT) represents fine-grained computational capability. Toward the other end, coarse-grained, one can envision larger computational blocks, such as full 8-bit arithmetic logic units (ALUs), more typical of traditional CPUs. Finer-grained blocks may be more adept at bit-level manipulations and arithmetic, but require combining several to implement larger pieces of logic. Contrast that with coarser-grained blocks, which may be more optimal for datapath-oriented computations that work with standard "word" sizes (8/16/32 bits) but are wasteful when implementing very simple logical operations. The general practice in industry is to strike a balance in granularity by using rather fine-grained 4-LUT architectures and augmenting them with coarser-grained heterogeneous elements, such as *multipliers*.

Indeed implementing a multiplication with a number of the aforementioned logic blocks, albeit possible, would cause a large delay penalty or a large logic block hardware footprint. This is way these type of operations are usually delegated to an *ad-hoc* multiplier implemented into the FPGA fabric. The result is that, for a small price in silicon area, the otherwise area-prohibitive multiplication can be offloaded onto dedicate hardware that does it much more efficiently. These units are usually Digital Signal Processor (DSP) slices which are optimized to perform fixed-point arithmetic.

Looking at the logic block in Figure 3.2, it is easy to identify the programmable points. These include the contents of the 4-LUT, the select signal for the output multiplexer, and the initial state of the D-FF. Most current commercial FPGAs use volatile static-RAM (SRAM) bits connected to configuration points to configure the FPGA. Thus, simply writing a value to each configuration bit sets the configuration of the entire FPGA. In the example above, the 4-LUT would be made up of 16 SRAM bits, one per output; the multiplexer would use a single SRAM bit; and the D-FF initialization value could also be held in a single SRAM bit.

3.1.2 The Interconnection Fabric

Having defined what is commonly known as the logic block, or function block, of an FPGA, i.e. LUT and D-FF, the focus can be turned to how these computation blocks can be tiled and connected together to form the fabric that is our FPGA. Current popular FPGAs implement what is often called *island-style architecture*. As shown in Figure 3.3, this design has logic blocks tiled in a two dimensional array and interconnected in some fashion. The logic blocks form the islands and "float" in a sea of interconnect.

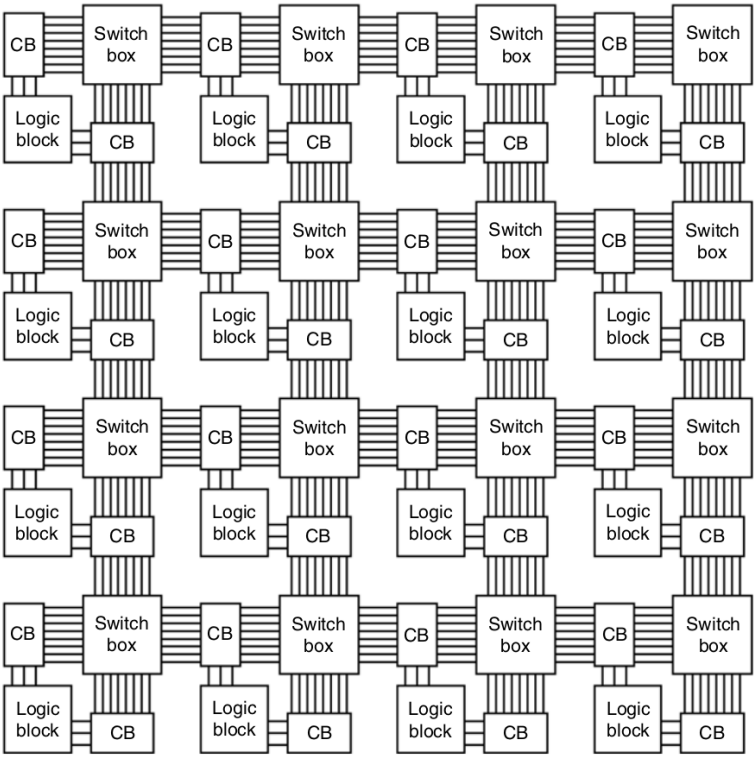


Figure 3.3: An island-style architecture with connect blocks and switch boxes to support more complex routing structures. (The difference in relative sizes of the blocks is for visual differentiation.)

With this array architecture, computations are performed spatially in the fabric of the FPGA. Large computations are broken into 4-LUT-sized pieces and mapped into physical logic blocks in the array. The interconnect is configured to route signals between logic blocks appropriately. With enough logic blocks, any kind of computation can be performed using an FPGA.

In Figure 3.3 the connection block and the switch box are introduced. The logic block accesses nearby communication resources through the connection block, which connects logic block input and output terminals to routing resources through programmable switches, or multiplexers. The connection block allows logic block inputs and outputs to be assigned to arbitrary horizontal and vertical tracks, increasing routing flexibility. The switch block appears where horizontal and vertical routing tracks converge. In the most general sense, it is simply a matrix of programmable switches that allow a signal on a track to connect to another track.

Depending on the design of the switch block, this connection could be, for example, to turn the corner in either direction or to continue straight. The design of switch blocks is an entire area of research by itself and has produced many varied designs that exhibit varying degrees of connectivity and efficiency.

In some segmented routing architectures, longer wires may also be present to allow signals to travel greater distances more efficiently. These segments may be long multiples of the length of the wires connecting adjacent blocks. The switch blocks (and perhaps more embedded switches) become points where signals can switch from shorter to longer segments. This feature allows signal delay to be less than O(N) when covering a distance of N logic blocks by reducing the number of intermediate switches in the signal path. A hierarchically approach can also be followed by creating tightly connected clusters of logic blocks which are then linked by longer wires. This strategy exploits the assumption that a well-designed circuit has mostly local connections and only a limited number of connections that need to travel long distances.

As with the logic blocks in a typical commercial FPGA, each switch point in the interconnect structure is programmable. Within the connection block, programmable multiplexers select which routing track each logic block's input and output terminals map to; in the switch block, the junction between vertical and horizontal routing tracks is switched through a programmable switch; and, finally, switching between routing tracks of different segment lengths or hierarchy levels is accomplished, again through programmable switches. For all of these programmable points, as in the logic block, FPGAs use SRAM bits to hold the user-defined configuration values.

3.2 Programming Hardware

Because of the FPGA's dual nature of software and hardware, a designer must think differently from conventional programmers. Software developers typically write sequential programs that exploit a microprocessor's ability to rapidly step through a series of instructions. In contrast, a high-quality FPGA design requires thinking about spatial parallelism, i.e. simultaneously using multiple resources spread across a chip to yield a huge amount of computation. Hardware designers have an advantage because they already think in terms of hardware implementations; even so, the flexibility of FPGAs gives them new opportunities generally not available in ASICs. FPGA designs can be rapidly developed and deployed, and even reprogrammed in the field with new functionality. Thus, they do not demand the huge design teams and validation efforts required for ASICs. Also, the ability to change the configuration, even when the device is running, yields new opportunities, such as computations that optimize themselves to specific demands on a second-by-second basis, or even time multiplexing a very large design onto a much smaller FPGA. However, because FPGAs are noticeably slower and have lower capacity than ASICs, designers must carefully optimize their design to the target device.

Modern field programmable gate arrays (FPGAs) boast an abundance of resources, including hundreds of thousands of lookup tables (LUTs), embedded mem-

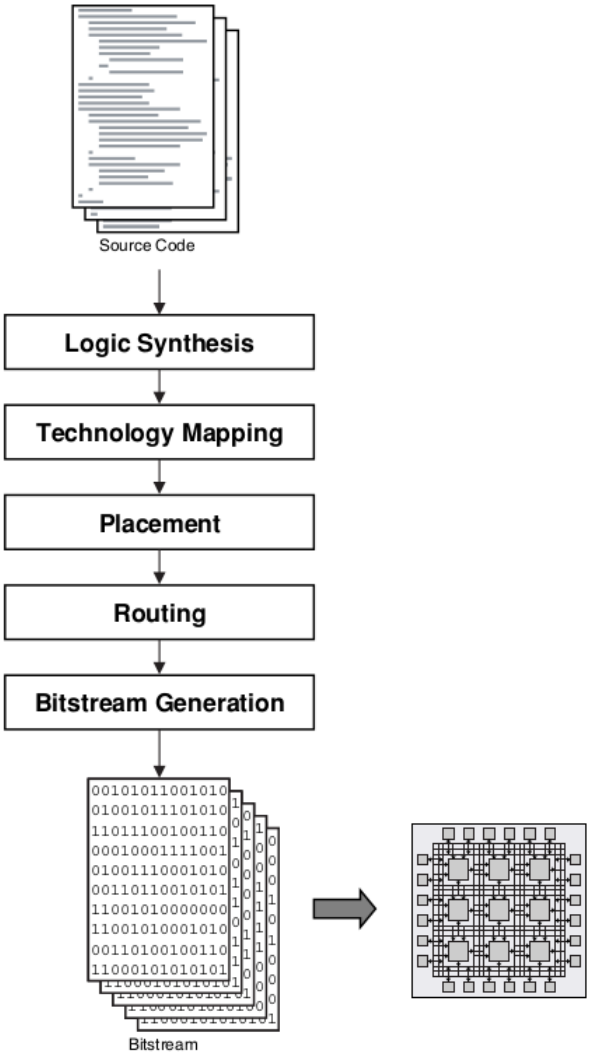


Figure 3.4: A typical FPGA mapping flow.

ories, and multipliers, all interconnected via programmable fabric. Programming these FPGAs at the level of individual elements would be almost impossible. However, with modern synthesis and layout tools, it is possible to describe a design simply by writing logical expressions, a level higher than gates, and letting the tools do the rest. Register transfer level (RTL) design is a prevalent approach for this. It allows the designer to express the design by describing the logic without the need of selecting the actual gates and their mapping to the FPGA. Very High-Speed Integrated Circuit Hardware Description Language (VHDL) is one popular programming language that supports RTL hardware descriptions. Transitioning to a higher-level approach, High-Level Synthesis represents the subsequent stage. This method involves employing a more conventional "behavioural" description of firmware, akin to C++ syntax, to generate the necessary firmware.

Because customizing an FPGA merely involves storing values to memory locations, similarly to compiling and then loading a program onto a computer, the creation of an FPGA-based circuit is a simple process of creating a bitstream to load into the device (see Figure 3.4). The abstract design produced with the de-

sired tool is optimized to fit into the FPGA's available logic through a series of steps:

1. **Logic synthesis** converts high-level logic constructs and behavioral code into logic gates;
2. **Technology mapping** separates the gates into groupings that best match the FPGA's logic resources;
3. **Placement** assigns the logic groupings to specific logic blocks and **routing** determines the interconnect resources that will carry the user's signals;
4. Finally, **bitstream generation** creates a binary file that sets all of the FPGA's programming points to configure the logic blocks and routing resources appropriately.

After a design has been compiled, the FPGA can be programmed to perform a specified computation simply by loading the bitstream into it. Typically a host microprocessor downloads the bitstream to the device. It must be kept in mind that the appropriate bitstream must be loaded every time the FPGA is powered up, as well as any time the user wants to change the circuitry when it is running. Once the FPGA is configured, it operates as a custom piece of digital logic.

3.2.1 An Example of Hardware Description Language - VHDL

Hardware Description Language (HDL) [46] is an essential Computer Aided Design (CAD) tool for the modern design and synthesis of digital systems. It offers the designer a very efficient tool for implementing and synthesizing designs on chips. Two widely used HDLs are VHDL and Verilog. After writing and testing the HDL code, the user can synthesize the code into digital logic components such as gates and flip-flops that can be downloaded into FPGAs. VHDL enjoys widespread popularity among designers in the industry [44], along with its close cousin, Verilog. Indeed, almost all modern CAD tools that perform simulation, synthesis, and layout support both. Verilog differs from VHDL primarily in the syntax it uses (VHDL is derived from Ada; Verilog, from C), but both languages are IEEE standards and are periodically reviewed to reflect changing industry realities and expectations.

VHDL is a strongly typed, Ada-based programming language that includes special constructs and semantics for describing concurrency at the hardware level. These concurrency constructs are new for most programmers and can be a source of confusion for beginners as it is quite different from the usual functional or, in general, behavioural programming of traditional languages like C or Java. However, one aspect that is shared by VHDL and, for example, C++ is being object-oriented. This can be seen in the example in the Listing 1, where a kind of object declaration is written, i.e. an object called `mux` is created providing its interface and the functional components inside this *entity* can be described later in the code. Indeed, while an entity specifies the interface of a hardware module, its internal structure and function are enclosed within the `architecture` definition.

```
entity mux is
port (
a : in std_logic;
b : in std_logic;
mux_sel : in std_logic;
c : out std_logic
);
end;
```

Listing 1: Example of the declaration of an *entity* with two inputs, a control signal and an output port using VHDL.

In a structural description of a module, the constituent submodules are declared, instantiated, and connected to each other. The instantiated components are connected to each other via internal signals by a process called port mapping which is performed on a signal-by-signal basis using the => symbol. It is analogous to assembling a set of integrated circuits (ICs) on a breadboard and wiring up the connections between the IC pins using jumper wires.

3.2.2 High Level Synthesis

Realizing the intrinsic efficiency of FPGAs in practice is an expensive proposition and tremendous design effort is expended to reach power, performance and area goals for typical designs [47]. Such efforts invariably lead to functional, performance, and reliability issues when pushing limits of design optimizations. Consequently, each generation of CAD researchers has sought to disrupt conventional design methodologies with the advent of high-level design modeling and tools to automate the design process. This pursuit to raise the abstraction level at which designs are modeled, captured, and even implemented has been the goal of several generations of CAD researchers.

Mario Barbacci noted in late 1974 that in theory one could "compile" the instruction set processor specification into hardware, thus setting up the notion of design synthesis from a high-level language specification. High-level Synthesis in later years will thus come to be known as the process of automatic generation of hardware circuit from "behavioural descriptions" (and as a distinction from "structural descriptions" such as synthesizable VHDL). Accordingly, the process was also variously referred to as a transformation "from behaviour to structure."

High-level synthesis (HLS) is an abstraction that enables a designer to focus on larger architectural questions rather than individual registers and cycle-to-cycle operations [48]. Instead a designer captures behaviour in a program that does not include specific registers or cycles and an HLS tool creates the detailed RTL micro-architecture. One of the first tools to implement such a flow was based on behavioural Verilog and generated an RTL-level architecture also captured in Verilog. Fundamentally, algorithmic HLS does several things automatically that an RTL designer does manually:

- analyzes and exploits the concurrency in an algorithm;
- inserts registers as necessary to limit critical paths and achieve a desired clock frequency;
- generates control logic that directs the data path;
- implements interfaces to connect to the rest of the system;
- maps data onto storage elements to balance resource usage and bandwidth;
- maps computation onto logic elements performing user specified and automatic optimizations to achieve the most efficient implementation.

Generally, the goal of HLS is to make these decisions automatically based upon user-provided input specification and design constraints. However, HLS tools greatly differ in their ability to do this effectively. Fortunately, there exist many mature HLS tools (e.g., Xilinx Vitis HLS [49], previously Vivado HLS) that can make these decisions automatically for a wide range of applications. However, the designer is still expected to supply the tool:

- A function specified in C, C++, or SystemC;
- A design testbench that calls the function and verifies its correctness by checking the results;
- A target FPGA device;
- The desired clock period;
- Directives guiding the implementation process.

In general, HLS tools can not handle any arbitrary software code. Indeed, many concepts that are common in software programming are difficult to implement in hardware. Still, a hardware description offers much more flexibility in terms of how to implement the computation. It typically requires additional information to be added by the designers (expressed using `#pragmas`) that provide hints to the tool about how to create the most efficient design. However, there are some limitations to obtain an efficient design, for example there is often limited support for standard libraries, system calls are typically avoided in hardware to reduce complexity, and the ability to perform recursion is often limited. On the other hand, HLS tools can deal with a variety of different interfaces (direct memory access, streaming, on-chip memories). And these tools can perform advanced optimizations (pipelining, memory partitioning, bitwidth optimization) to create an efficient hardware implementation.

The primary output of an HLS tool is a RTL hardware design that is capable of being synthesized through the rest of the hardware design flow. Additionally, the tool may output testbenches to aid in the verification process. Finally, the tool will provide some estimates on resource usage and performance. As an example, Vitis HLS generates the following outputs:

```
#define N 11
#include "ap_int.h"
typedef int coef_t;
typedef int data_t;
typedef int acc_t;
void fir(data_t *y, data_t x) {
coef_t c[N] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};

static data_t shift_reg[N];
acc_t acc;
int i;
acc = 0;
Shift_Accum_Loop:
for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
                acc += x * c[0];
                shift_reg[0] = x;
        } else {
                shift_reg[i] = shift_reg[i - 1];
                acc += shift_reg[i] * c[i];
        }
}
*y = acc;
}
```

Listing 2: Example of HLS code to implement a Finite Impulse Response filter.

- Synthesizable Verilog and VHDL;
- RTL simulations based on the design testbench;
- Static analysis of performance and resource usage;
- Metadata at the boundaries of a design, making it easier to integrate into a system.

Once an RTL-level design is available, other tools are usually used in a standard RTL design flow.

In order to show an example of HLS in action, consider the code [48] in Listing 2 where a simple Finite Impulse Response (FIR) filter is implemented without hardware optimization (for a description of this kind of digital filters see [50] Section 6.3.7). The function takes two arguments, an input sample `x`, and the output sample `y`. This function must be called multiple times to compute an entire output signal, since each time that we execute the function we provide one input sample and receive one output sample. This code is convenient for modeling a streaming architecture, since it is called as many times as needed as more data becomes available. The coefficients for the filter are stored in the `c[]` array declared inside

of the function. These are statically defined constants. Note that the coefficients are symmetric. i.e., they are mirrored around the center value `c[5] = 500`. Many FIR filter have this type of symmetry. The code uses `typedef` for the different variables. While this is not necessary, it is convenient for changing the types of data. Indeed, bit width optimization, specifically setting the number of integer and fraction bits for each variable, can provide significant benefits in terms of performance and area. The code is written as a streaming function. It receives one sample at a time, and therefore it must store the previous samples. Since this is an 11 tap filter, the previous 10 samples must be kept. This is the purpose of the `shift_reg[]` array. This array is declared `static` since the data must be persistent across multiple calls to the function. The `for` loop is doing two fundamental tasks in each iteration. First, it performs the multiply and accumulate operation on the input samples (the current input sample `x` and the previous input samples stored in `shift_reg[]`). Each iteration of the loop performs a multiplication of one of the constants with one of the sample, and stores the running sum in the variable `acc`. The loop is also shifting values through shift array, which works as a First-In-First-Out (FIFO) buffer. It stores the input sample `x` into `shift_reg[0]`, and moves the previous elements "up" through the shift array. After the for loop completes, the `acc` variable has the complete result of the convolution of the input samples with the FIR coefficient array. The final result is written into the function argument `y` which acts as the output port from this fir function. This completes the streaming process for computing one output value of an FIR. This function does not provide an efficient implementation of a FIR filter. It is largely sequential, and employs a significant amount of unnecessary control logic.

3.3 Interacting with a FPGA: how to build an accelerated application

When dealing with running computations on FPGAs, they fall into two distinct roles: as a standalone device, with everything it needs to communicate, start the computation and performing the task; or as an accelerator, where parts of a bigger programme are offloaded to the FPGA to exploit its speed and efficiency in particular tasks. The latter makes FPGAs as one of the candidates for the heterogeneous computing paradigm.

In this work the focus will be on using FPGAs as accelerators to test the capabilities of the hardware as a first step towards a real application as a ready to trigger device.

3.3.1 OpenCL

OpenCL [51] (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms. OpenCL supports a wide range of applications, ranging from embedded and consumer software to HPC solutions, through a low-level, high-performance, portable abstraction. By creating an efficient, close-to-the-

metal programming interface, OpenCL will form the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications. OpenCL is particularly suited to play an increasingly significant role in emerging interactive graphics applications that combine general parallel compute algorithms with graphics rendering pipelines. OpenCL consists of an API for coordinating parallel computation across heterogeneous processors, a cross-platform programming language, and a cross-platform intermediate language with a well-specified computation environment.

To describe the core ideas behind OpenCL, a hierarchy of models is used:

- Platform Model
- Execution Model
- Memory Model
- Programming Model

Platform Model

The model consists of a host connected to one or more OpenCL devices, shown in Figure 3.5. An OpenCL device is divided into one or more compute units (CUs) which are further divided into one or more processing elements (PEs). Computations on a device occur within the processing elements. An OpenCL application is implemented as both host code and device kernel code. The host code portion of an OpenCL application runs on a host processor according to the models native to the host platform. The OpenCL application host code submits the kernel code as commands from the host to OpenCL devices. An OpenCL device executes the commands computation on the processing elements within the device.

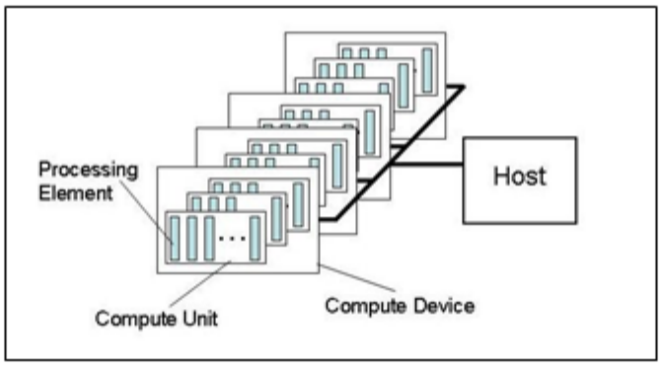


Figure 3.5: Platform Model: one host plus one or more compute devices each with one or more compute units composed of one or more processing elements.

Execution Model

The OpenCL execution model is defined in terms of two distinct units of execution: kernels that execute on one or more OpenCL devices and a host program that

executes on the host. With regard to OpenCL, the kernels are where the "work" associated with a computation occurs. A kernel executes within a well-defined context managed by the host. The context defines the environment within which kernels execute. It includes the following resources:

- *Devices*: One or more devices exposed by the OpenCL platform.
- *Kernel Objects*: The OpenCL functions with their associated argument values that run on OpenCL devices.
- *Program Objects*: The program source and executable that implement the kernels.
- *Memory Objects*: Variables visible to the host and the OpenCL devices. Instances of kernels operate on these objects as they execute.

The host program uses the OpenCL API to create and manage the context. Functions from the OpenCL API enable the host to interact with a device through a command-queue. Each command-queue is associated with a single device. The commands placed into the command-queue fall into one of three types:

- *Kernel-enqueue commands*: Enqueue a kernel for execution on a device.
- *Memory commands*: Transfer data between the host and device memory, between memory objects, or map and unmap memory objects from the host address space.
- *Synchronization commands*: Explicit synchronization points that define order constraints between commands.

Commands communicate their status through Event objects. Successful completion is indicated by setting the event status associated with a command to `CL_COMPLETE`. Unsuccessful completion results in abnormal termination of the command which is indicated by setting the event status to a negative value. In this case, the command-queue associated with the abnormally terminated command and all other command-queues in the same context may no longer be available and their behavior is implementation-defined. A command submitted to a device will not launch until prerequisites that constrain the order of commands have been resolved.

Memory Model

The OpenCL memory model describes the structure, contents, and behavior of the memory exposed by an OpenCL platform as an OpenCL program runs. The model allows a programmer to reason about values in memory as the host program and multiple kernel-instances execute.

Memory in OpenCL is divided into two parts:

- *Host Memory*: The memory directly available to the host. The detailed behavior of host memory is defined outside of OpenCL. Memory objects move between the Host and the devices through functions within the OpenCL API or through a shared virtual memory interface;

- *Device Memory*: Memory directly available to kernels executing on OpenCL devices.

Programming Model

The OpenCL framework enables applications to use a host and multiple OpenCL devices as a single heterogeneous parallel computer system. This framework comprises several key components. First, the OpenCL Platform layer allows the host program to identify OpenCL devices, understand their capabilities, and create contexts. Next, the OpenCL Runtime enables the host program to manage these contexts once they are created. Additionally, the OpenCL Compiler generates program executables that contain OpenCL kernels. It can build these executables from OpenCL C source strings, or device-specific program binary objects, depending on the device's capabilities. Some implementations may also support other kernel or intermediate languages.

3.3.2 The Python Way: PYNQ

In the last few years there has been a steady rise in the use of interpreted programming languages [52], the most popular of which is Python [53], as shown in Figure 3.6. Working with Python makes writing code much more accessible, thanks to the clearer interface and the huge number of libraries which makes even the most difficult task a matter of finding the right library, or module, and invoke the objects and functions described in them. This trend pushed AMD, and firstly its subsidiary Xilinx, to publish PYNQ [54], an open-source project which offers a Python API-based framework for utilizing AMD platforms via a Jupyter-based interface.

Programming Language	2024	2019	2014	2009	2004	1999	1994	1989
Python	1	4	8	6	10	28	22	-
C	2	2	1	2	2	1	1	1
C++	3	3	4	3	3	2	2	2
Java	4	1	2	1	1	15	-	-
C#	5	6	5	7	8	25	-	-
JavaScript	6	7	9	9	9	20	-	-
Visual Basic	7	19	-	-	-	-	-	-
SQL	8	9	-	-	7	-	-	-
PHP	9	8	6	5	6	-	-	-
Go	10	18	36	-	-	-	-	-
Objective-C	30	10	3	36	45	-	-	-
Lisp	35	30	14	20	15	13	6	3
(Visual) Basic	-	-	7	4	5	3	3	7

Figure 3.6: Positions of the top 10 programming languages from 1989 to 2024. Please note that these are average positions for a period of 12 months.

FPGA designs are represented as Python objects referred to as *overlays*, which can be accessed via a Python API. Although creating a new overlay still requires skilled developers with experience in designing programmable logic circuits, overlays are designed to be configurable and re-used as much as possible in various applications, much like software libraries.

```python
import pynq
ov = pynq.Overlay("model_binary.xclbin")
nn = ov.myproject
```

Listing 3: Programming and calling kernel function using PYNQ.

Traditionally, C or C++ have been the most common embedded programming languages. Python, on the other hand, raises the level of programming abstraction and increases programmer productivity. These options are not mutually exclusive, however. PYNQ employs CPython, which is written in C and can be extended with optimized C code while also integrating thousands of C libraries. Whenever possible, the more productive Python environment should be employed, and lower-level C code can be utilized whenever efficiency demands it.

```
auto devices = xcl::get_xil_devices();
auto fileBuf = xcl::read_binary_file(binaryFile);
cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};
OCL_CHECK(err, context = cl::Context({device},
                                     NULL, NULL, NULL, &err));
OCL_CHECK(err, q = cl::CommandQueue(context, {device},
                                    CL_QUEUE_PROFILING_ENABLE, &err));
OCL_CHECK(err, cl::Program program(context,
                                    {device}, bins, NULL, &err));
OCL_CHECK(err, krnl_vector_add = cl::Kernel(program, "vadd", &err));
```

Listing 4: OpenCL code to programme a FPGA.

PYNQ strives to work on any computing platform and operating system, which it accomplishes by utilizing a web-based architecture that is also browser-agnostic. The open-source Jupyter notebook infrastructure is used to execute an Interactive Python (IPython) kernel and a web server directly on the ARM processor of an MPSoC or the host CPU of an acceleration card.

In Listing 3 how simple it is to load a firmware on an FPGA and retrieve the kernel function inside the design is shown. This can be considered equivalent to the code in Listing 4, which is much less straightforward. Furthermore, to actually send and receive data from the FPGA and run the algorithm, the code in Listing 5 is needed. On the other hand, using PYNQ the creation of input and output buffer is done by calling the `allocate` function, which returns objects that behave like `numpy` arrays and can be moved to and from the device with `sync_to_device()` and `sync_from_device()`. Finally, the kernel function is callable providing the buffers, for example: `nn.call(input,output)`.

```
OCL_CHECK(err, l::Buffer buffer_in1(context,
    CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY, vector_size_bytes,
    source_in1.data(), &err))

OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_input},
                        0 /*0 means from host*/,NULL,&eventinp));

OCL_CHECK(err, err = myproject.setArg(0, buffer_input));
OCL_CHECK(err, err = myproject.setArg(1, buffer_output));
//[...]
OCL_CHECK(err, err = q.enqueueTask(myproject,NULL,&eventker));
// wait for all kernels to finish their operations
OCL_CHECK(err, err = q.finish());

OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_output},
                 CL_MIGRATE_MEM_OBJECT_HOST));
```

Listing 5: OpenCL code to create I/O buffers and call the kernel function of the FPGA firmware.

Chapter 4

The Artificial Neural Networks Landscape

The expression "*Machine Learning*" (ML) was first used to describe a particular type of computer algorithms in 1959 by Arthur Samuel [55] and since then, ML has become one of the pillars of computer and data science and it has been introduced in almost every aspect of everyday life. Services like Google, YouTube and Netflix improve their search engines and "recommendation" functions by implementing a complex structure of learning algorithms in order to record all users' choices and preferences and hence to build a customised environment, theoretically unique for each user. Large Language Models, like the Generative Pre-trained Transformer (GPT) behind the famous ChatGPT, have revolutionized how humans can interact with computers, giving a very realistic text response. This type of algorithms, together with models able to generate images, videos and songs, have reached a level of quality which threatens the hard work of artists and their ability to be paid for their works.

Currently, the spread of learning algorithms in many sectors finds its roots mainly in an increased quantity of data available, combined with a technological progress in storage and computational power, which can nowadays be exploited with lower maintenance and material costs.

ML tasks can be boiled down to two main categories:

Classification The computer algorithm is asked to separate data in different categories; when there are only two categories, e.g. signal/background discrimination of a measurement, it is usually called binary classification, otherwise if there are more categories it is a case of multiclass classification. To solve this task, the learning algorithm usually returns a number between 0 and 1 for each class corresponding to the probability of an input to belong to that category.

Regression In this case the algorithm is asked to predict a continuous numerical value given some inputs, e.g. an house prizing algorithm or the prediction of the transverse momentum of muons given the track information from the muon chambers [56].

These two categories do not cover more novel and specialized ML applications,

such as inferring entire probability density functions rather than just point estimates [57].

4.1 How Machines Learn

Machine Learning systems can be classified according to the amount and type of supervision they get during training [58]. There are three major categories: supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning

In supervised learning, the training data fed to the algorithm includes the desired solutions, called labels. In other words, supervised learning [59] involves observing several examples of a random vector $\mathbf{x}$ and an associated value or vector $\mathbf{y}$, then learning to predict $\mathbf{y}$ from $\mathbf{x}$, usually by estimating $p(\mathbf{y}|\mathbf{x})$. The adjective "supervised" originates form the view of the target $\mathbf{y}$ being provided by an external instructor who shows the ML system what to do.

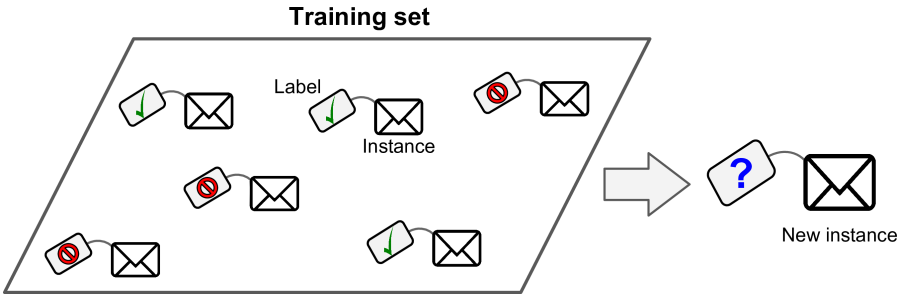


Figure 4.1: A labeled training set for supervised learning on a classification task (e.g. spam filtering).

Visual recognition, an example is shown in Figure 4.2, is an application domain that highly relies on supervised ML algorithms. For instance, a system might need to learn to identify pedestrians on a street in a automotive application for self-driving cars: to do so, it is trained with millions of short videos about street scenes, with some of the videos containing no pedestrians at all while others having up to dozens. The presence or not of pedestrian is known a priori, hence the learning is supervised: a variety of learning algorithms are trained on such data, with each having access to the correct answer. Many other decision support mechanisms that have at disposal large quantity of data on which to train a ML system, could be the base of a supervised ML approach: e.g. historical data on medical exams' output may drive a supervised ML system to learn and prompt the probability of suffering from a disease, etc.

Unsupervised learning

In unsupervised learning, on the other hand, the training data is unlabeled, see Figure 4.3. In other words, the model tries to learn without the information about the desired output. One of the most important unsupervised learning algorithm is

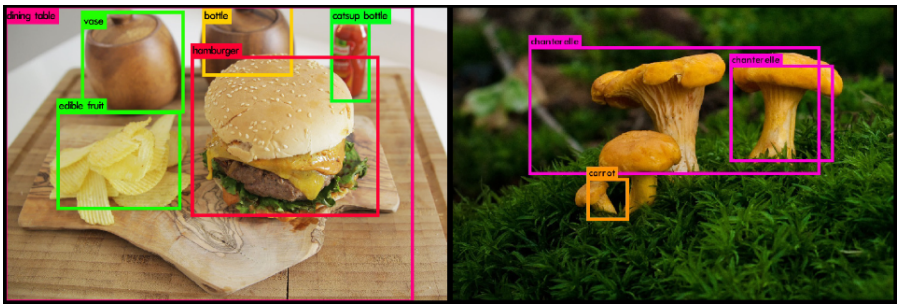


Figure 4.2: Example of image recognition [60].

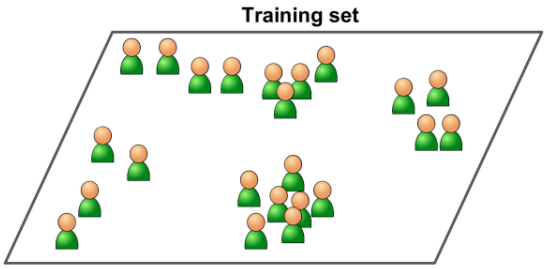


Figure 4.3: An unlabeled training set for unsupervised learning.

called clustering. Here is an example to understand what it is.

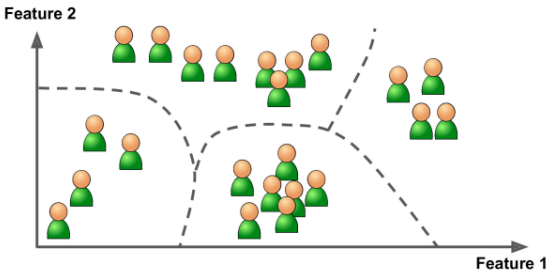


Figure 4.4: Example of clustering of an unlabeled dataset.

Consider a dataset of visitors to a website and the task is to detect groups of similar visitors (Figure 4.4). At no point information about the belonging of a certain group of the visitors is given to the clustering algorithm. Nevertheless, the model is able to find those connections without any help. For example, it might notice that 40% of the visitors are male who love comic books and generally visit the website in the evening, while 20% are young sci-fi lovers who visits during the weekends, and so on. It is also possible to use a hierarchical clustering algorithm which may divide each group into smaller groups, allowing finer content recommendation.

Visualization algorithms are also good examples of unsupervised learning algorithms: they are fed a lot of complex and unlabeled data, and they output a 2D or 3D representation of the data that can easily be plotted (Figure 4.5). These algorithms try to preserve as much structure as they can (e.g., trying to keep separate clusters in the input space from overlapping in the visualization), so the organi-

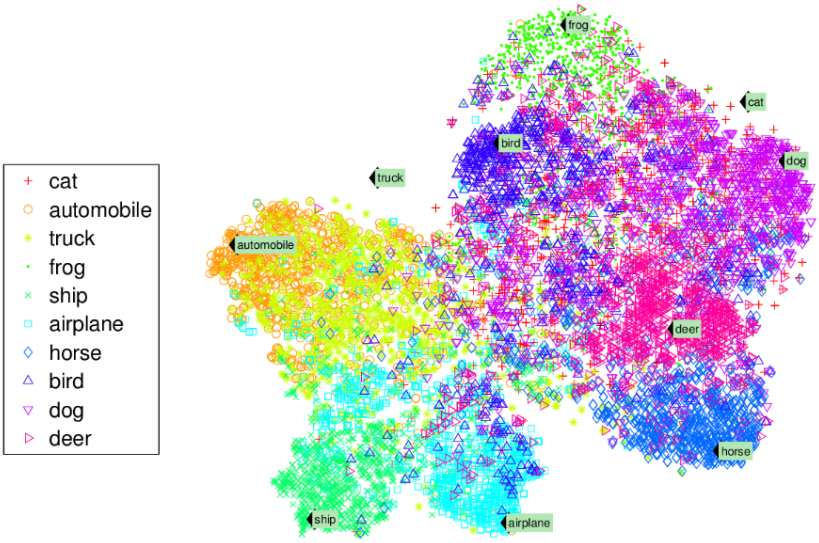


Figure 4.5: Example of a t-SNE visualization highlighting semantic clusters.

zation of the data can be understood and perhaps unsuspected patterns can be identified.

A related task is dimensionality reduction, in which the goal is to simplify the data without losing too much information. One way to do this is to merge several correlated features into one. For example, a car's mileage may be very correlated with its age, so the dimensionality reduction algorithm will merge them into one feature that represents the car's wear and tear. This is called feature extraction.

Another type of algorithm that falls under the category of unsupervised learning is Anomaly Detection, which is the main application for ML described in thesis and will be tackled in Section 4.3.1

Reinforcement learning

Reinforcement learning [61] is an important type of ML, in which an agent learns how to behave in a environment by performing actions and deciding about the next actions based on the outcome of the previous ones. In 2016 this kind of models caught the attentions of the news outlets due to Google's AlphaGo ability of besting the world champion of the game of Go, a feat that nobody thought was possible by a machine.

Reinforcement learning can be understood using the concepts of Agents, Actions, Environments, States, and Rewards (note that this description is intended to be short and provided just for completeness in describing possible ML types, and no deep discussion of its characteristics and implementation will be given, as it would go beyond the research scope of this thesis):

Agent the component that takes actions, e.g. a video game character navigating in its virtual environment, as well as a drone making a delivery;

Action one amongst the set of all possible moves/choices the agent can make. In a reinforcement learning environment, agents can choose only among a

predefined list of possible actions. E.g. in video games, the list might include moving right or left, jumping or not, jumping high or low, or crouching, or standing still; in the stock markets, the list might include buying, selling or holding any title among a list of financial product;

Environment the world through which the Agent moves. The environment takes as input the agent's current State and its selected action as input, and returns as output the agent's reward and next state;

State a concrete situation in which the agent happens to put itself. E.g. it could be a specific moment in time and/or place in space, a local and instantaneous configuration that puts the agent in contact and relation with its environment (e.g. tools, obstacles, prices);

Reward the feedback based on which the success or failure of the agent's choices is measured. E.g. in a video game, a reward could indeed be the gain of a price when a special object is captured, and similar. Every time an agent does something in the environment that foresees a possible reward, the agents sends output in form of actions to the environment and the environment returns the agent's new state as well as the obtained rewards (or lack of them);

In a nutshell, as shown in Figure 4.6, reinforcement learning judges actions by the results they produce. It is fully goal oriented, as its aim is just to learn sequences of actions that will eventually lead an agent to achieve a predefined goal, in terms of maximizing its objective function. In the video game example, the final goal could be to finish the game with the maximum score, so each additional point obtained throughout the game would affect the agent's subsequent behaviour. In a robotics example, a robot might have as a goal to travel from A to B: every millimetre step that makes it closer to the spatial objective B is counted as additional reward, so the robot will learn the direction to go and eventually reach the final destination. The implementation of reinforcement learning models requires a lot of training it-

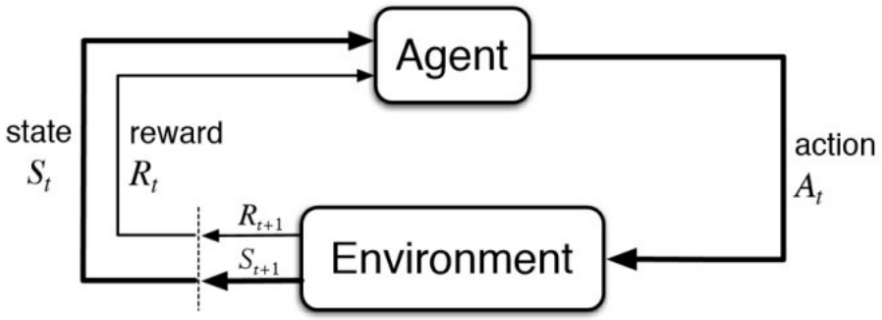


Figure 4.6: Schematic workflow of a reinforcement learning algorithm.

erations and data. For this reason, it has historically been associated with domains in which plenty of simulated data is available (e.g. video games and robotics, as in the examples above). One other aspect to mention is that - with respect to other ML types - it is far from easy to take results from academic research papers and implement them in real-world applications.

Nowadays, applications of RL can be seen e.g. in high-dimensional control problems, like robotics: they have been the subject of research (in academia and industry) for many years, and now start-ups are more and more using this ML type to build products for industrial robotics applications.

4.1.1 Machine Learning Formalism

A machine learning algorithm is an algorithm that is able to learn from data. In [62], Mitchell provides a succinct definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Machine learning tasks are usually described in terms of how the machine learning system should process an example. An example is a collection of features that have been quantitatively measured from some object or event that the machine learning system should process. An example is usually represented as a vector $\mathbf{x} \in \mathbb{R}^n$ where each entry x_i of the vector is a different feature. For example, the features of an image are usually the values of the pixels in the image. The tasks mentioned here can be the ones at the beginning of Section 4.

In this Section a more technical and mathematical description of E will be given. Roughly speaking, unsupervised learning involves observing several examples of a random vector $\mathbf{x}$ and attempting to implicitly or explicitly learn the probability distribution $p(\mathbf{x})$, or some interesting properties of that distribution [59]; while supervised learning involves observing several examples of a random vector $\mathbf{x}$ and an associated value or vector $\mathbf{y}$, then learning to predict $\mathbf{y}$ from $\mathbf{x}$, usually by estimating $p(\mathbf{y}|\mathbf{x})$. This is another way of explaining the origin of the term supervised learning as the target $\mathbf{y}$ being provided by an instructor or teacher who shows the machine learning system what to do, while in unsupervised learning, there is no instructor or teacher, and the algorithm must learn to make sense of the data without this guide.

Unsupervised learning and supervised learning are not formally defined terms. The lines between them are often blurred. Many machine learning technologies can be used to perform both tasks. For example, the chain rule of probability states that for a vector $\mathbf{x} \in \mathbb{R}^n$, the joint distribution can be decomposed as

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i | x_1, \ldots, x_{i-1}) \tag{4.1}$$

This decomposition means that one can solve the ostensibly unsupervised problem of modeling $p(\mathbf{x})$ by splitting it into n supervised learning problems. Alternatively, one can solve the supervised learning problem of learning $p(y|\mathbf{x})$ by using traditional unsupervised learning technologies to learn the joint distribution $p(\mathbf{x}, y)$, then inferring

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y'} p(\mathbf{x}, y')} \tag{4.2}$$

Though unsupervised and supervised learning are not completely formal or distinct concepts, they do help roughly categorize some of the things that can be done with machine learning algorithms.

The definition of a machine learning algorithm as an algorithm that is capable of improving a computer programme's performance at some task via experience is somewhat abstract. To make this more concrete the focus will be now on the supervised learning formalism.

A ML model is commonly defined as a parametric function $f(\mathbf{x}, \boldsymbol{\theta})$, where $\mathbf{x}$ is an element of the features domain and $\boldsymbol{\theta}$ are the parameters of the model. Let $\hat{\mathbf{y}}$ be the value the models predicts $\mathbf{y}$ should have. The output can be defined as

$$\hat{\mathbf{y}} = f(\mathbf{x}, \boldsymbol{\theta}) \tag{4.3}$$

The degree of correspondence between a model and data is defined in terms of some error (or loss) metric. This metric is usually written as a function of the model's output and the desired one: $L(\hat{\mathbf{y}}, \mathbf{y})$. The choice of the loss function is highly dependent on the type and the task of the model. Since it is quite difficult to find a perfectly suitable candidate, a computational-friendly loss function is often used, at least as a starting point.

A commonly used example of a loss function is the mean squared error (MSE), i.e. the averaged squared numerical difference between $\hat{\mathbf{y}}$ and $\mathbf{y}$ among the inputs:

$$L_{\text{MSE}} = \frac{1}{N} \sum_{(\mathbf{x},\mathbf{y}) \in D} (\mathbf{y} - f(\mathbf{x}, \boldsymbol{\theta}))^2 \tag{4.4}$$

where $(\mathbf{x}, \mathbf{y})$ is an input in a dataset D containing many labelled examples. The objective of training is to find the parameters' values that minimize the expected loss over all possible examples:

$$\boldsymbol{\theta} = \arg\min_{\theta} [\mathbb{E}_{(x,y)} L(f(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y})] \tag{4.5}$$

The task of either minimizing or maximizing some function by altering its input are generally referred to as optimization. In this context the loss function can also be called objective function. An example of optimization technique is called Gradient Descent. Generally attributed to Augustin-Louis Cauchy, it has become the predominant method in Machine Learning to find the minimum of the loss function.

In order to briefly explain how it works, consider a point $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ and a function of n real variables $G(x_1, \ldots, x_n)$. Then, starting from a point $\mathbf{x}^0$, the direction in which G decreases most rapidly is given by [63]

$$z_i = -\lambda \frac{\partial G(\mathbf{x})}{\partial x_i} \quad \text{or in vector form} \quad \mathbf{z}^0 = -\lambda \nabla_{\mathbf{x}} G(\mathbf{x}) \tag{4.6}$$

where λ is an arbitrary positive factor of proportionality. Then, the function $g(t) = G(\mathbf{x}^0 + t\mathbf{z}^0)$ has a negative derivative at $t = 0$. It will therefore be possible to find a $t > 0$ such that

$$g(t) < g(0) \tag{4.7}$$

With such a t, $\mathbf{x}^1 = \mathbf{x}^0 + t\mathbf{z}^0$ can be taken as a new starting point and continue. From this a sequence of points $\mathbf{x}^0, \mathbf{x}^1, \mathbf{x}^2, \ldots$ such that $G(\mathbf{x}^{k+1}) < G(\mathbf{x}^k)$. In this way the sequence will converge to a stationary point of G, i.e. a minimum, maximum or saddle point, as shown in Figure 4.7.

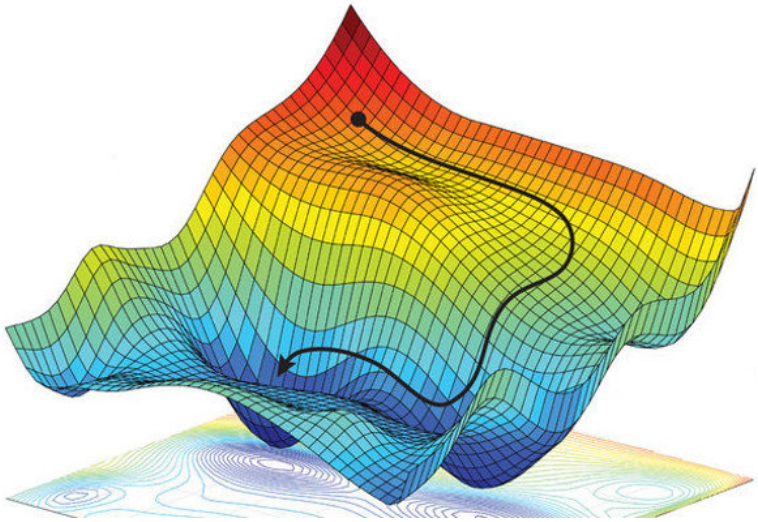


Figure 4.7: Graphical representation of gradient descent of a function as a manifold.

If t is chosen to be the smallest positive root of $g'(t) = 0$, the process has the following geometrical interpretation. Starting at $\mathbf{x}^0$, using 4.6, the direction in which the surface

$$y = G(\mathbf{x}) = G(x_1, \ldots, x_n) \tag{4.8}$$

is descending most rapidly is found. The procedure is followed until a contour is found (i.e. a horizontal section of the surface). Then, a new direction of steepest descent is taken and so on. Since the direction of steepest descent is always normal to the contour, it follows that the direction $\mathbf{z}^k$ and $\mathbf{z}^{k+1}$ are at right angles.

Nearly all of machine learning is powered by an extension of the gradient descent algorithm: stochastic gradient descent (SGD). Indeed, a recurring problem in ML is that large training sets are necessary for good generalization, but large training sets are also more computationally expensive. As mentioned before, the cost function used by machine learning algorithm often decomposes as a sum over training examples of some per-example loss function. For example, the negative conditional log-likelihood of the training data can be written as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(x,y)} L(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} L(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \boldsymbol{\theta}) \tag{4.9}$$

where L is the per-example loss $L(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = -\log p(\mathbf{y}|\mathbf{x}; \boldsymbol{\theta})$ and $\mathbf{x}^{(i)}, \mathbf{y}^{(i)}$ are the input vectors in the dataset.

For these additive cost functions, gradient descent requires computing

$$\nabla_\theta J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \boldsymbol{\theta}) \tag{4.10}$$

The computational cost of this operation is $O(m)$. As the training set size grows to billions of examples, the time to take a single gradient step becomes prohibitively long.

The insight of SGD is that the gradient is an expectation. The expectation may be approximately estimated using a small set of samples. Specifically, on each

step of the algorithm, we can sample a batch of examples $\mathbb{B} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m')}\}$ drawn uniformly from the training set. The batch size m' is typically chosen to be a relatively small number of examples. In this way a training set with billions of examples can be fitted using updates computed on only a hundred examples.

The estimate of the gradient is formed as

$$\boldsymbol{g} = \frac{1}{m'} \nabla_\theta \sum_{i=1}^{m'} L(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \boldsymbol{\theta}) \tag{4.11}$$

using examples from the batch $\mathbb{B}$. The stochastic gradient descent algorithm then follows the estimated gradient downhill:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{g} \tag{4.12}$$

where α is a scalar value known as learning rate (or step size), an example of what is called a hyperparameter, i.e. special parameters that are not changed by the training procedure but identify the characteristics of a specific model.

SGD has many important uses outside the context of machine learning. It is the main way to train large linear models on very large datasets. For a fixed model size, the cost per SGD update does not depend on the training set size m. In practice, larger model is often used as the training set size increases, but it is not always the case. The number of updates required to reach convergence usually increases with training set size. However, as m approaches infinity, the model will eventually converge to its best possible test error before SGD has sampled every example in the training set. Increasing m further will not extend the amount of training time needed to reach the model's best possible test error. From this point of view, one can argue that the asymptotic cost of training a model with SGD is $O(1)$ as a function of m.

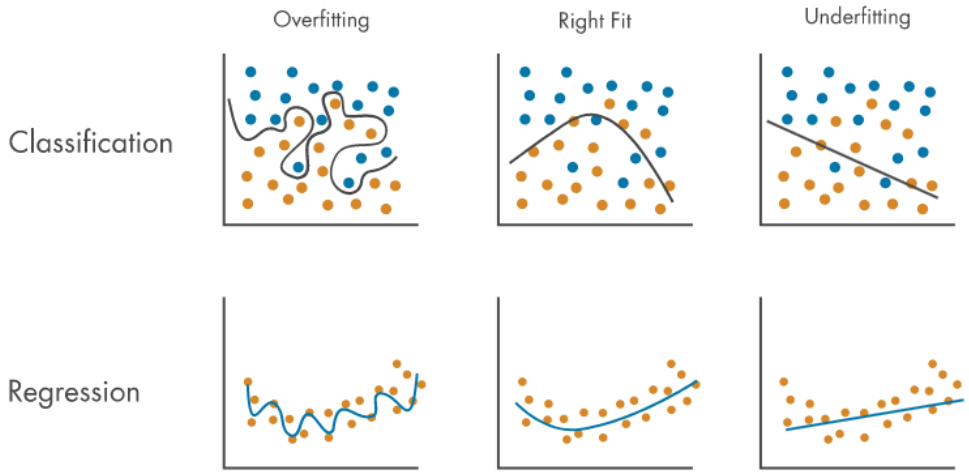


Figure 4.8: An example of underfitting and overfitting in the case of a binary classification and a regression task. Image from [64]

Thus, in practice, the learning procedure starts with all parameters picked random from a chosen distribution (e.g. normal or uniform), and at each iteration

the value of the loss function is evaluated again with the new parameters given by 4.12 until a minimum is found. However, an excess in training could result in overfitting, which describes the situation when a trained model is very accurate dealing with the training dataset but it generalize poorly over new, previously unseen data. On the other hand when a model performs poorly on both training and new data is called underfitted. In Figure 4.8 an example is shown.

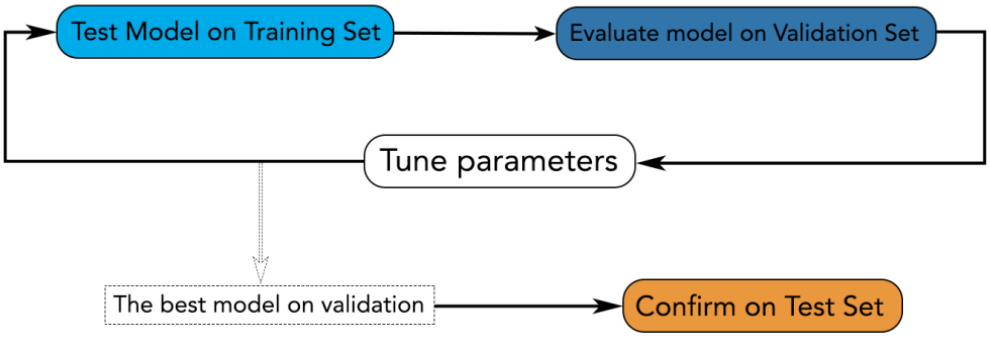


Figure 4.9: Training workflow using a validation and test subset.

In order to find the best hyperparameters of the model to avoid falling in the two cases just described, a validation set is needed, i.e. examples that the training algorithm does not observe. This subset is different from what is known as test set, composed from the same distribution as the training set, which can be used to estimate the generalization of a learner, after the entire learning procedure has completed and the model is ready to be deployed. Indeed, it is important that the test examples are not used in any way to make choices about the model, including its hyperparameters. For this reason, no example from the test set can be used in the validation set. Therefore, the validation set is always built from the training data. Specifically, the training data is split into two disjoint subsets. One of these subsets is used to learn the parameters. The other subset is the validation set, used to estimate the generalization error during or after training, allowing for the hyperparameters to be updated accordingly, as can be seen in the workflow shown in Figure 4.9. Typically, one uses about 80 percent of the training data for training and 20 percent for validation. Since the validation set is used to "train" the hyperparameters, the validation set error will underestimate the generalization error, though typically by a smaller amount than the training error does. After all hyperparameter optimization is complete, the generalization error may be estimated using the test set.

Dividing the dataset into a fixed training set and a fixed validation set can be problematic if it results in the validation set being small. A small set implies statistical uncertainty around the estimated average test error, making it difficult to claim that algorithm A works better than algorithm B on the given task. When the dataset has hundreds of thousands of examples or more, this is not a serious issue. When the dataset is too small, alternative procedures enable one to use all the examples in the estimation of the mean test error, at the price of increased computational cost. These procedures are based on the idea of repeating the

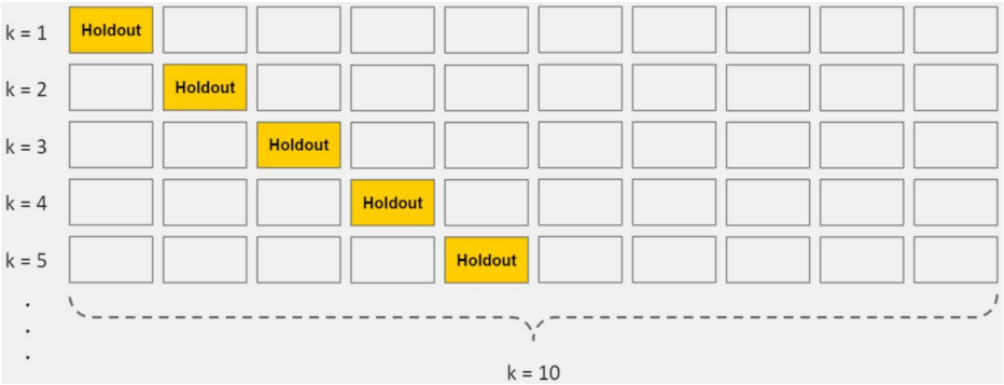


Figure 4.10: Graphical representation of 10-Fold Cross Validation.

training and validation computation on different randomly chosen subsets or splits of the original dataset. The most common of these is the k-fold cross-validation procedure, shown in Figure 4.10, in which a partition of the dataset is formed by splitting it into k non-overlapping subsets. The test error may then be estimated by taking the average test error across k trials. On trial i, the i-th subset of the data is used as the test set, and the rest of the data is used as the training set.

Choosing the metric used to define an accurate model is quite important and non-trivial. For example, consider a classification task with three classes and an entry belonging to class with index 0; this can be represented as a vector made up of the probabilities of belonging to a certain category: $[1.0, 0, 0]$. The question is whether is better a model that outputs $[0, 0, 0]$ or $[0.6, 0.4, 0.4]$. The answer is not unique and it depends on the use different use cases and types of algorithm. However, in the case of a binary classification, a very common way to establish the accuracy of a model is the Receiver Operating Characteristic (ROC) curve.

Having only two classes, the feature space can be written as $\mathbb{Y} = [0, 1]$. Let $f_i \in \mathbb{R}$ be the model output for the i-th example: the greater it is, the more likely the example is of class 1. If a binary decision has to be made using this value, a threshold t is defined, and the continuous values f_i are transformed into the definite class labels

$$y_i^{pred} = \begin{cases} 1 & \text{when } f_i > t \\ 0 & \text{when } f_i \leq t \end{cases} \tag{4.13}$$

From a statistics point of view, when classifying, a decision is made under the null hypothesis that the example is class 1 (or signal hypothesis). Such decisions will suffer from two kinds of errors: false positives (or type I) errors, where a background example is wrongly selected as signal, and false negatives (or type II) errors, when a signal example is rejected as background. By varying this threshold, the trade-off between the error types can be adjusted. The choice of the threshold (also called working point) depends on the specific classification problem and the costs of making the errors of each type. In high-energy physics, for example, the working point is usually selected with the threshold value that maximises the so called significance, defined as:

$$S = \frac{N_s}{\sqrt{n_b}} \tag{4.14}$$

where N_s is the number of selected signal examples and N_b is the number of selected background examples.

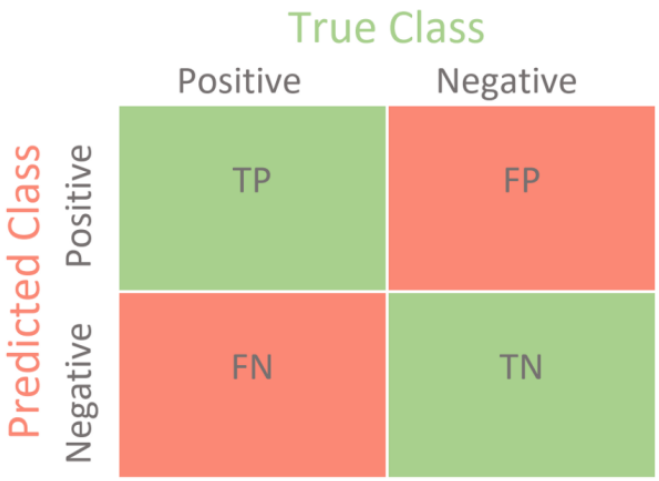


Figure 4.11: Definition of True Positive (TP), False Positive (FP), True Negative (TN) and False Negative (FN). This matrix is often called confusion matrix.

For each threshold value, the true positive rate (TPR) (also called signal efficiency or sensitivity) is computed as

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{4.15}$$

where TP and FN are the number of true positives and false negatives respectively. A simple and graphical explanation on what these terms mean is in Figure 4.11. Together with the TPR, the False Positive Rate (FPR) is also calculated:

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}} \tag{4.16}$$

with the number of false positives FP and true negatives TN (see Figure 4.11). This quantity is also called background efficiency in the High Energy Physics field. Thus, the TPR is the fraction of the signal examples that pass the selection threshold, while the FPR is the fraction of background examples that erroneously pass the selection threshold. If there is no priori information to decide the decision threshold, a common measure is to plot the TPR as a function of FPR, that is the Receiver Operating Characteristic curve.

While allowing for maximum flexibility of evaluation, such curve is not a convenient scalar performance score. To solve this, a commonly used summary statistic is the Area Under the ROC Curve (AUC). The AUC has mathematical properties that makes it attractive for the comparison of different classifiers. First of all, the AUC is a finite quantity, lying in a well-defined interval. For a perfect classifier, all predicted values for signal examples are greater than for all background examples, therefore the AUC $= 1$. For a totally random classifier, the predictions are distributed equally for signal and background examples, and the ROC curve is a straight line from $(0, 0)$ to $(1, 1)$ with an AUC $= 0.5$. An illustration of all these concepts is presented in Figure 4.12.

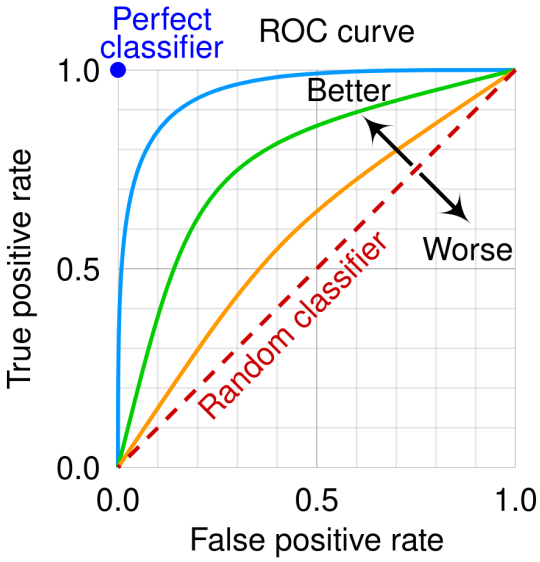


Figure 4.12: An illustration of a Receiver Operating Characteristic curve (ROC), and the area under the curve (AUC). In red, the ROC curve of the worst possible model giving random predictions (AUC= 0.5). An higher AUC corresponds to a better classifier (green, orange and blue lines). The blue dot is the perfect classifier, with 100% correct predictions. Image from [65].

The AUC is defined for a binary classifier. If there are several classes to distinguish, the single ROC AUC metric can be replaced with a set of numbers. There are two main ways to transform a n-class classification into sets of binary classification problems:

one-vs-rest : the classification is reduced to n binary classification problems. For each class, it considers the value of the metric, computed with the other classes collapsed into a single virtual class;

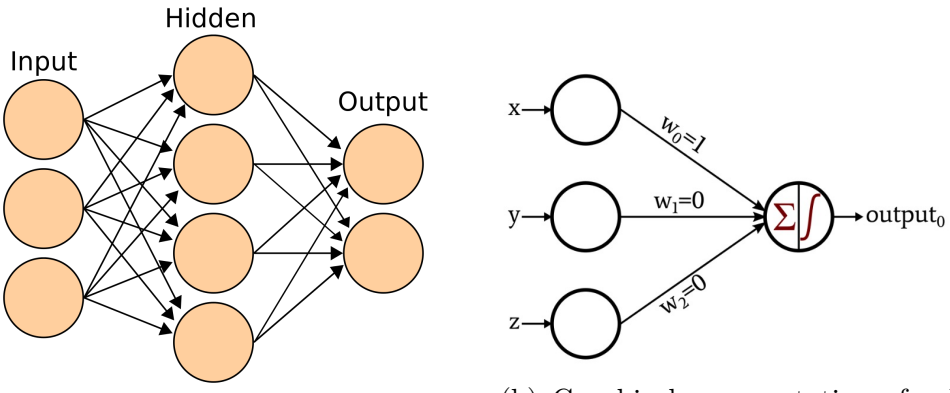
one-vs-one : the classification is reduced into $n(n-1)/2$ binary classification problems. Each class is evaluated against every other class.

4.2 Artificial Neural Networks

An artificial neural network (ANN or more commonly NN) is a computing system vaguely inspired by the biological neural connections that constitute a human brain, specifically designed to tackle non-linear learning problems. This type of architecture are included in the context of deep learning, i.e. an approach to artificial intelligence where the computer learns complex concepts by building them from simpler ones, structured in a hierarchy. This hierarchy, when represented graphically, shows many layers, thus giving rise to the term. Indeed, deep feedforward networks (a more specific name for NNs), also called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models. Their goal is to approximate some function f^*, defining a mapping $\mathbf{y} = f(\mathbf{x}, \boldsymbol{\theta})$ and learning the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation.

These models are called feedforward because information flows through the function being evaluated from $\mathbf{x}$, through the intermediate computations used to define f, and finally to the output $\mathbf{y}$. There are no feedback connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called recurrent neural networks. Feedforward networks are of extreme importance to machine learning practitioners. They form the basis of many important commercial applications. For example, the convolutional networks used for object recognition from photos are a specialized kind of feedforward network. Feedforward networks are a conceptual stepping stone on the path to recurrent networks, which power many natural language applications [59].

NNs are called networks also because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph (DAG) describing how the functions are composed together. A DAG is a directed graph $G = (V, E)$ that consists of a finite set V of vertices and a finite set E of edges, where each edge is associated with an ordered pair of vertices [66]. In a DAG, there are no directed circuits, meaning there are no closed trails where all vertices, except the end vertices, are distinct. Essentially, this means that there are no cycles in the graph. For example, going back to NNs, there might be three functions $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$ connected in a chain, to form $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$. In this case, $f^{(1)}$ is called the first layer, or input layer, $f^{(2)}$ is called second layer, and so on. The overall length of the chain gives the depth of the model. The name deep learning arose from this terminology. The final layer of a feedforward network is called the output layer. A basic example of NN is in Figure 4.13a.



(a) Neural Network basic architecture.

(b) Graphical representation of a Perceptron.

Figure 4.13: Simple diagrams representing the layout and functioning principle of Neural Networks.

The NN training consists basically in trying to match $f(\mathbf{x})$, i.e. the result from the output layer, with $f^*(\mathbf{x})$. The learning algorithm tries to use the inner layers to produce the desired output, but the training data do not say what each individual layer should do. Because the training data does not show the desired output for each of these layers and they are usually not interpretable (with some important exceptions, as explained in Section 4.3.1), they are called hidden layers.

Finally, these networks are called neural because they are loosely inspired by neuroscience. Each hidden layer of the network is typically vector valued. The dimensionality of these hidden layers determines the width of the model. Each element of the vector may be interpreted as playing a role analogous to a neuron. Rather than thinking of the layer as representing a single vector-to-vector function, the layer can also be thought as consisting of many units that act in parallel, each representing a vector-to-scalar function, called perceptrons. Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value (see Figure 4.13b). The idea of using many layers of vector-valued representations is drawn from neuroscience. The choice of the functions $f^{(i)}(\mathbf{x})$ used to compute these representations was guided by neuroscientific observations about the functions that biological neurons compute when this kind of algorithms was conceived. Modern neural network research, however, is guided by many mathematical and engineering disciplines, and the goal of neural networks is not to perfectly model the brain. It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what is known about the brain, rather than as models of brain function. An important aspect of NNs is the

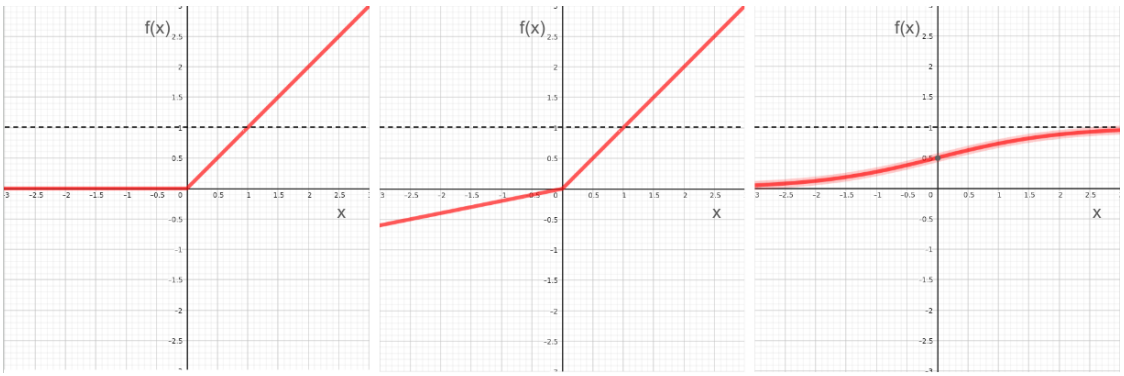


Figure 4.14: Examples of activation functions. From the left: ReLU, LeakyReLU and sigmoid function. The softmax function has the same shape of the sigmoid but it is normalized to 1 across all outputs.

ability to approximate also non-linear functions. This is made easy by the addition of non-linear activation function computed in each perceptron, after the linear combination of all inputs. There is a vast landscape of activation functions, each with their advantages and disadvantages, but the most common are (a graphical representation is in Figure 4.14):

Rectified Linear Unit : $\mathrm{ReLU}(x) = \max(0, x)$. It is zero for negative values and increases linearly for $x > 0$. This function is the simplest and obviously linear in the positive domain, in the sense that it is mainly a piecewise linear function made up of two linear pieces. Thanks to this almost linearity, ReLU units preserve many of the properties that make linear models easy to optimise with gradient-based methods [59]. A drawback of ReLU is the impossibility to learn from examples which cause the inputs of the function to be less or equal to zero;

Leaky Rectified Linear Unit : $\text{LeakyReLU}(x) = \max(0, x) + \alpha\min(0, x)$ [67]. It is a piecewise linear function with a parametrized slope for values < 0 and a slope equal to 1 for positive values. It alleviates the problems using ReLU with negative values;

Sigmoid : $\text{sigmoid}(x) = (1 + e^{-x})^{-1}$. This activation function is usually used in the last layer of a network tasked with binary classification, since its value lies in the interval $[0, 1]$. The value tends to 0 as the argument approaches negative infinity and to 1 as the argument approaches positive infinity. It is a poor choice as the activation for the hidden layers, as it suffers from a vanishing gradient issue, i.e. the gradient can become so small that it would take an enormous amount of iteration to take a good step towards a minimum;

Softmax : $\text{softmax}(\mathbf{x})_i = e^{x_i}(\sum_{j=1}^{K} e^{x_j})^{-1}$. It is useful to represent a probability distribution over a discrete variable with n possible values [59]. This can be seen as a generalization of the sigmoid function, which is used to represent a probability distribution over a binary variable. Softmax functions are most often used as the output of a multiclass classifier, to represent the probability distribution over n different classes. Using this activation guarantees not only that each output is between 0 and 1, but also that the entire vector of outputs of an entire layer sums to 1 so that it represents a valid probability distribution.

Training a NN is usually done via gradient techniques, as described in Section 4.1.1. Given a function $L(\boldsymbol{\theta})$, its minimum will be at a point where its gradient is zero. To arrive at such a point, iteratively follow the inverse gradient:

$$\boldsymbol{\theta}^{\tau+1} = \boldsymbol{\theta}^{\tau} - \eta \nabla L(\boldsymbol{\theta}^{(\tau)}) \tag{4.17}$$

where τ is the iteration number, and $\eta > 0$ is the learning rate parameter. This procedure can be very computationally expensive when considering that the model prediction for each input in the dataset would be needed to perform a step. That is why the SGD is commonly used, where, for each iteration, only a subset, called batch, of examples is randomly chosen, extracted from the whole dataset and used to compute the gradients. Once all the entries in the dataset are used, a so-called epoch is finished. After an epoch, the data is reshuffled and processed again. With this technique, the gradient is computed after each batch:

$$\boldsymbol{\theta}^{\tau+1} = \boldsymbol{\theta}^{\tau} - \eta \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(f(x_i, \boldsymbol{\theta}^{(\tau)})) \tag{4.18}$$

where $f(x_i, \boldsymbol{\theta}^{(\tau)})$ is the model prediction for the i-th example in a batch with m samples. Stochastic gradient descent follows noisy estimates of the true gradient. This slows down convergence, as shown in Figure 4.15, but eventually reaches a similar minimum with a lower computational footprint.

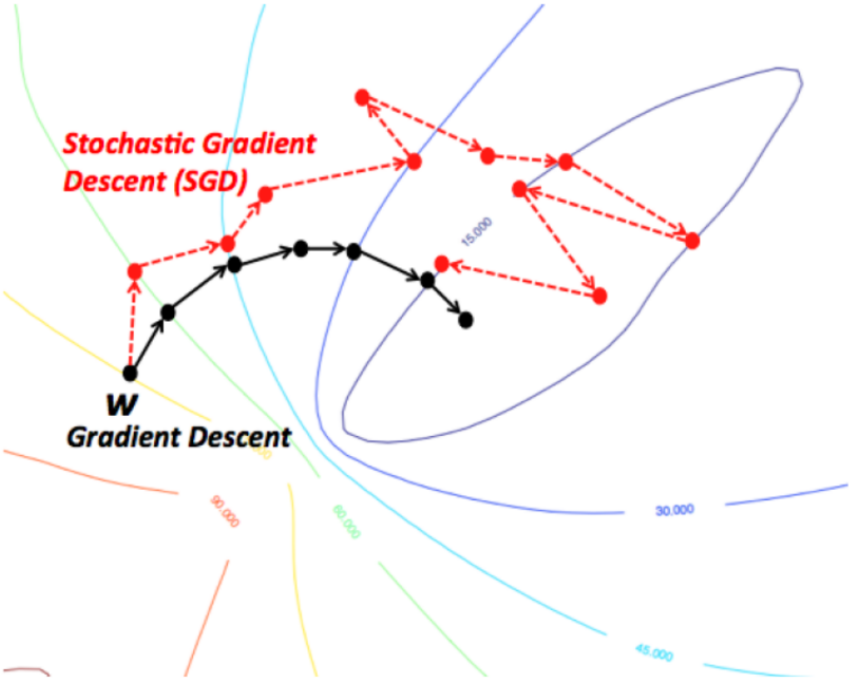


Figure 4.15: Optimising a learning algorithm with gradient descent (black line) and stochastic gradient descent (red dashed line).

Dropout and Regularization

Neural Networks, during their learning process, use the loss function to establish the set of parameters, i.e. weights and bias, to obtain an optimal output for the task at hand. To handle problems like overfitting, described in Section 4.1.1, a few techniques, called regularisation techniques, are commonly used that, acting on the loss function, helps reduce the effects of overtraining a NN.

Generally, such techniques add a term, dependent on the weights, after the loss function:

$$L_{tot}(\mathbf{x}, \boldsymbol{\theta}, \lambda) = L(\mathbf{x}, \boldsymbol{\theta}) + \lambda \cdot R(\boldsymbol{\theta}) \tag{4.19}$$

where $L(\mathbf{x}, \boldsymbol{\theta})$ is the original loss function, $R(\boldsymbol{\theta})$ is the regularisation function and λ is called regularisation rate or constant. One of the most used family of regularisation functions is the Lp norm:

$$Lp(\mathbf{x}) = ||x||_p = \left(\sum_{i=1}^{n} x_i^p \right)^{1/p} \tag{4.20}$$

of which the most commonly known is the L2 which takes the form of the usual norm of a vector:

$$L2(\mathbf{x}) = ||x|| = \sqrt{\sum_{i=1}^{n} x_i^p} \tag{4.21}$$

In practice, using this kind or regularisation means searching for a compromise between having small weights and the accuracy of the output. The balance between

these two terms is controlled with λ: when it is small, the minimisation of the loss function is dominant, otherwise finding a vector of weights with a small norm is prioritized. Another way to tackle overtraining is the dropout technique. It does

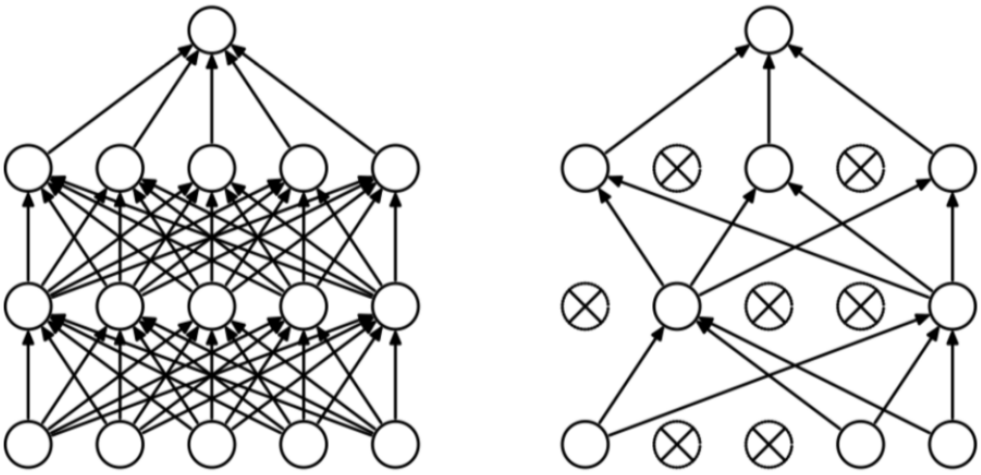


Figure 4.16: Graphical representation of the dropout technique. Each epoch the network is made slightly different, turning off a random sample of neurons.

not sees the modification of the loss function, but of the network itself. Basically, applying a dropout means removing some neurons in some or all hidden layers: during each epoch, some neurons are randomly discarded before the training step, as shown in Figure 4.16. During the real usage of a model, the network is considered in its entirety without dropping out neurons.

4.3 Examples of Artificial Neural Networks

In this section two different families of NNs are described: Autoencoders and Graph Neural Networks. These models are used for different tasks and highlight the flexibility and power of neural networks in various applications. Their structures will be explored, how they are trained, and their practical uses, in order to help understanding why ANNs are so important in today's technology landscape.

Although these model families are described here as distinct, the best results are often achieved by hybridizing predefined architectural frameworks, as tested in Section 6.4.

4.3.1 Autoencoders

An autoencoder (AE) is a neural network that is trained to attempt to copy its input to its output [59]. Internally, it usually comprises of a hidden layer h that describes a code used to represent the input. The network can be viewed as made up of two parts: an encoder function $\mathbf{h} = f(\mathbf{x})$ and a decoder that produces a reconstruction $\mathbf{r} = g(\mathbf{h})$. This architecture is presented in Figure 4.17. If an autoencoder succeeds in simply learning to set $g(f(\mathbf{x})) = \mathbf{x}$ everywhere, then it is

not especially useful. Instead, autoencoders are designed to be unable to learn to copy perfectly. Usually they are restricted in ways that allow them to copy only approximately, and to copy only input that resembles the training data. Because the model is forced to prioritize which aspects of the input should be copied, it often learns useful properties of the data.

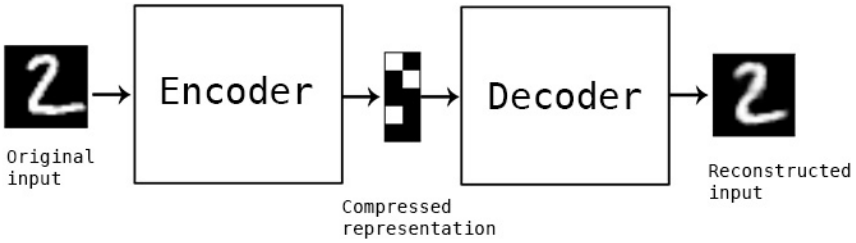


Figure 4.17: The general structure of an autoencoder, mapping an input to an output (called reconstruction) through an internal representation or code. The autoencoder has two components: the encoder and the decoder. Image from [68].

The idea of autoencoders is not new and conceptualized in the eighties. Traditionally, they were used for dimensionality reduction or feature learning. Recently, theoretical connections between AEs and latent variable models [69] have brought attentions to autoencoders as generative models. Autoencoders may be thought of as being a special case of feedforward networks and may be trained with all the same techniques, typically batch gradient descent following gradients computed by back-propagation. Unlike general feedforward networks, autoencoders may also be trained using recirculation: a learning algorithm based on comparing the activations of the network computed on the original input to the activations of the reconstructed input, i.e. the output of the model.

As said before, there needs to be a way to avoid the copy of the input in the output as it is. The most common technique, which also makes the code (or latent space) take useful properties, is to constrain the latter to have a smaller dimension than the input $\mathbf{x}$. In this case the AE can be called undercomplete. Learning an undercomplete representation forces the AE to capture the most salient features of the training data.

The learning process is described simply as minimizing a loss function

$$L(\mathbf{x}, g(f(\mathbf{x}))) \tag{4.22}$$

where L is a loss function penalizing $g(f(\mathbf{x}))$ for being dissimilar from $\mathbf{x}$, e.g. a mean squared error function, see Eq. 4.4.

Autoencoders with nonlinear encoder functions f and nonlinear decoder functions g can learn a more powerful nonlinear generalization of Principal Component Analysis [70], i.e. a traditional dimensionality reduction technique that transforms a dataset by projecting it onto a set of orthogonal (uncorrelated) axes, called principal components, which capture the most variance in the data. It simplifies the data while preserving its essential patterns and structures.

Unfortunately, if the encoder and decoder are allowed too much capacity, the AE can learn to perform the copying task without extracting useful information

about the distribution of the data. Theoretically, one could imagine that an autoencoder with a one-dimensional code but a very powerful nonlinear encoder could learn to represent each training example $\mathbf{x}^{(i)}$ with the code i. The decoder could learn to map these integer indices back to the values of specific training examples. This scenario does not occur in practice, but it illustrates clearly that an AE trained to perform the copying task can fail to learn anything useful about the dataset if the capacity of the autoencoder is allowed to become too great.

An interesting application for AEs is denoising: by changing the usual loss function shown in Eq. 4.22 with

$$L(\mathbf{x}, g(f(\tilde{\mathbf{x}}))) \tag{4.23}$$

where $\tilde{\mathbf{x}}$ is a copy of $\mathbf{x}$ that has been corrupted by some form of noise, it is possible to build a denoising autoencoder (DAE). An application of these kind of models was explored in a Bachelor's thesis I have co-supervised where a DAE was used to perform error mitigation on the output of a gate-based quantum computer [71].

Anomaly Detection

Anomaly detection (AD) aims to identify instances containing patterns that deviate from those observed in normal instances [72]. This task is crucial in various vision applications, such as manufacturing defect detection, medical image analysis, and video surveillance. Unlike typical supervised classification problems, anomaly detection presents unique challenges. Primarily, it is difficult to obtain a substantial amount of anomalous data, whether labeled or unlabeled. Additionally, the differences between normal and anomalous patterns are often fine-grained, as defective areas can be small and subtle in high-resolution images. Since the distribution of anomaly patterns is unknown in advance, models are trained to learn the patterns of normal instances. An instance is determined to be anomalous if it is not well-represented by these models (see Figure 4.18). Considering the peculiarities of this kind of task, AD is one of the popular application for autoencoders [73].

Figure 4.18: Example of how to perform anomaly detection as outliers of a data reconstruction model trained on normal data.

Autoencoders, when trained solely on normal data instances (which are the majority in anomaly detection tasks), fail to reconstruct the anomalous data samples, therefore, producing a large reconstruction error and the data samples which

produce high residual errors are considered outliers. Several variants of autoencoder architectures are proposed as illustrated in Figure 4.19 produce promising results in anomaly detection. The choice of autoencoder architecture depends on the nature of data, convolution networks are preferred for image datasets while Long short-term memory (LSTM) based models tend to produce good results for sequential data. Efforts to combine both convolution and LSTM layers where the encoder is a convolutional neural network (CNN) and decoder is a multilayer LSTM network to reconstruct input images are shown to be effective in detecting anomalies within data. The use of combined models such as Gated recurrent unit autoencoders (GRU-AE), Convolutional neural networks autoencoders (CNN-AE), Long short-term memory (LSTM) autoencoder (LSTM-AE) eliminates the need for preparing hand-crafted features and facilitates the use of raw data with minimal preprocessing in anomaly detection tasks.

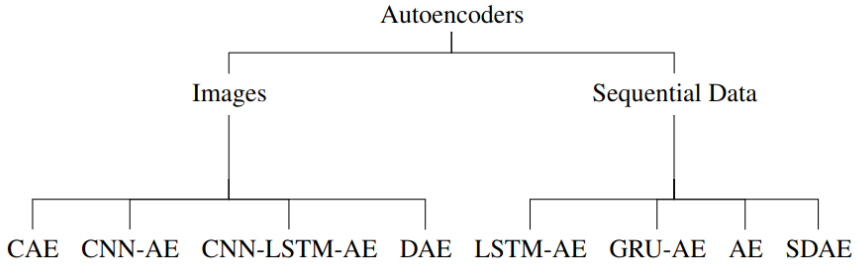


Figure 4.19: Autoencoder architectures for anomaly detection. AE: Autoencoders, LSTM: Long Short Term Memory Networks, SDAE: Stacked Denoising Autoencoder, DAE: Denoising Autoencoders, GRU: Gated Recurrent Unit, CNN: Convolutional Neural Networks, CNN-LSTM-AE: Convolution Long Short Term Memory Autoencoders, CAE: Convolutional Autoencoders

There are alternatives to reconstruction-based anomaly detection, including several common approaches. Statistical-based methods assume that normal data follows a certain probability distribution, typically Gaussian. The normal data is modeled as a Gaussian distribution, and probability theory is used to identify data points with low probability according to this estimated distribution. Distance-based methods operate on the assumption that normal data points are closely grouped, whereas outliers are farther away. These methods define a distance metric between data points and identify those that are significantly distant from others, with the Local Outlier Factor (LOF) being a popular example that uses local density to detect outliers. Clustering-based methods are based on the premise that normal data points form dense clusters, while outliers do not. Clustering is performed on the data, and any data points that do not belong to any cluster are identified as outliers.

4.3.2 Graph Neural Networks

Data can naturally be represented by graph structures in various application areas [74], including image analysis, scene description, software engineering, and natural language processing.

A graph $G = (V, E)$ consists of two sets [75]: a finite set V of elements called vertices and a finite set E of elements called edges. Each edge is identified with a pair of vertices. If the edges of a graph G are identified with ordered pairs of vertices, then G is called a directed or an oriented graph. Otherwise G is called an undirected or a nonoriented graph.

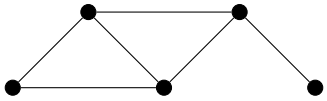
The simplest types of graph structures include single nodes and sequences. However, in many applications, information is organized into more complex graph structures such as:

acyclic graphs : A graph without circuits i.e. a finite sequence of vertices $v_1, v_2, \ldots, v_k, v_1$ such that each pair (v_i, v_{i+1}) is an edge in the graph, and no edge or intermediate vertex is repeated (Figure 4.20a).

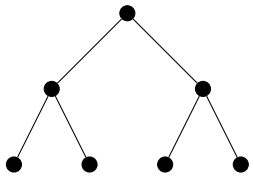
trees : a connected acyclic subgraph of a graph G (Figure 4.20b).

cyclic graphs : A graph where is it possible to follow a circuit passing through all vertices (Figure 4.20c).

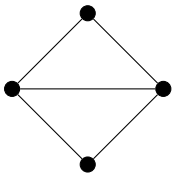
Traditionally, the exploitation of data relationships has been extensively studied within the inductive logic programming community. Recently, this research has evolved in different directions, driven by the application of relevant concepts from statistics and neural networks to these areas.



(a) Example of acyclic graph.



(b) Example of a tree of a graph.



(c) Example of a cyclic graph.

Figure 4.20: Different types of graphs which can represent the relationships between data.

Conventional machine learning applications handle graph-structured data by first converting it into a simpler representation, such as vectors of real numbers, through a preprocessing phase. This preprocessing step "compresses" the graph data into a vector format, which is then processed using tabular-based techniques. However, this approach can lead to the loss of crucial information, such as the topological dependencies between nodes, and the final outcome may be unpredictably influenced by the specifics of the preprocessing algorithm. More recently, various approaches have emerged that aim to retain the graph-structured nature of the data throughout the processing phase. The idea is to encode the underlying graph structured data using the topological relationships among the nodes of the graph, in order to incorporate graph structured information in the data processing step. However, there is a family of algorithms that extends even further and deals directly with graph structured information: Graph Neural Networks (GNN).

GNNs are based on an information diffusion mechanism. A graph is processed by a set of units, each one corresponding to a node of the graph, which are linked

according to the graph connectivity. The units update their states and exchange information until they reach a stable equilibrium. The output of a GNN is then computed locally at each node on the base of the unit state. The diffusion mechanism is constrained in order to ensure that a unique stable equilibrium always exists. GNNs can be used for the processing of general classes of graphs, e.g., graphs containing undirected links, and they adopt a general diffusion mechanism. The intuitive idea behind GNNs is that nodes in a graph represent objects or con-

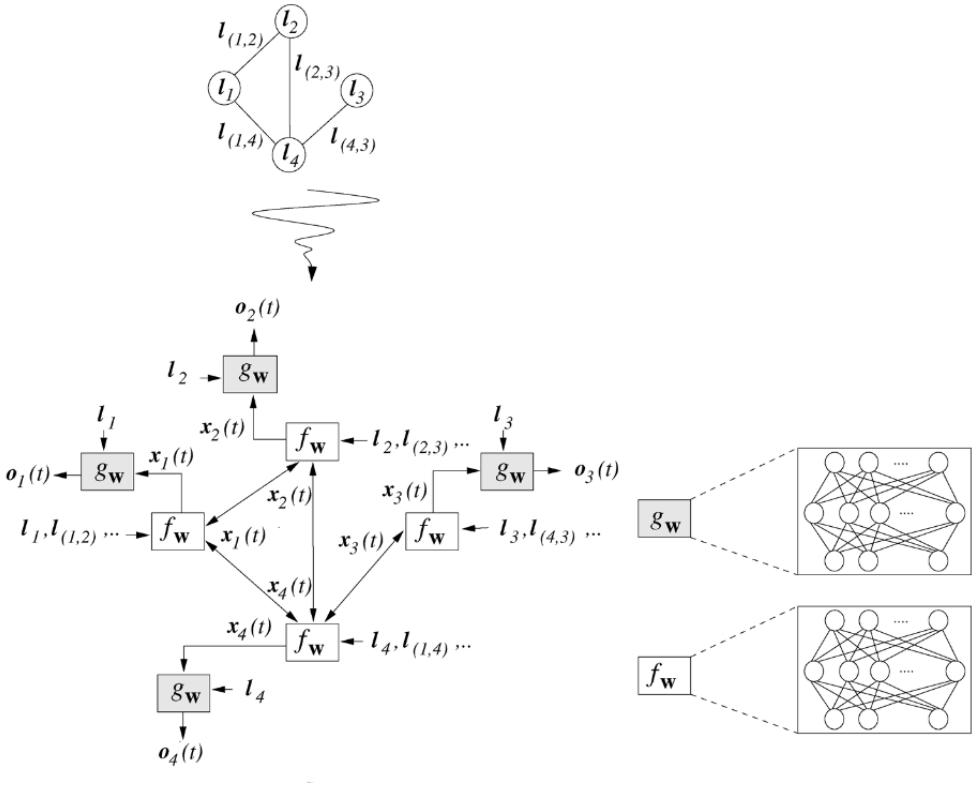


Figure 4.21: Example of a graph (on the top) and the corresponding network where the state and output of each node is computed with a Recurrent Neural Network.

cepts, and edges represent their relationships. Each concept is naturally defined by its features and the related concepts. Thus, a state $\mathbf{x}_n \in \mathbb{R}^s$ can be attached to each node n based on the information contained in the neighbourhood of n, i.e. the nodes connected to n. The state $\mathbf{x}_n$ contains a representation of the concept denoted by n and can be used to produce an output $\mathbf{o}_n$. Let $f_{\mathbf{w}}$ be a parametric function, called local transition function, that express the dependence of a node n on its neighbourhood and let $g_{\mathbf{w}}$ be the local output function that describes how the output is produced; then, $\mathbf{x}_n$ and $\mathbf{o}_n$ are defined as follows:

$$\mathbf{x}_n = f_{\mathbf{w}}(\mathbf{l}_n, \mathbf{l}_{\text{co}}[n], \mathbf{x}_{\text{ne}}[n], \mathbf{l}_{\text{ne}}[n]) \tag{4.24}$$

$$\mathbf{o}_n = g_{\mathbf{w}}(\mathbf{x}_n, \mathbf{l}_n) \tag{4.25}$$

where $\mathbf{l}_n$, $\mathbf{l}_{\text{co}}[n]$ are respectively the real valued vectors of label of n and its edges, and $\mathbf{x}_{\text{ne}}[n]$, $\mathbf{l}_{\text{ne}}[n]$ the states and the labels of the nodes in the neighbourhood of

n. At this point, [74] attests that it is possible to converge exponentially to the solutions of 4.24 by iterating:

$$\mathbf{x}_n(t+1) = f_{\mathbf{w}}(\mathbf{l}_n, \mathbf{l}_{\text{co}}[n], \mathbf{x}_{\text{ne}}[n](t), \mathbf{l}_{\text{ne}}[n]) \tag{4.26}$$

$$\mathbf{o}_n(t) = g_{\mathbf{w}}(\mathbf{x}_n(t), \mathbf{l}_n) \tag{4.27}$$

Finally, these functional representations of the state and output of each node can be substituted by Recurrent Neural Networks [59] (RNN) whose peculiarity is to be able to work better with time-series data thanks to their ability to retain a previous state of the network and use it, together with the new input, to compute the next state. These RNNs are then trained with data to reflect the relationships and information contained in it. A schematic representation of this very superficial explanation of the very complex algorithm behind GNNs is shown in Figure 4.21.

4.4 Writing a Neural Network

The growth in popularity of ML algorithms has made essential to develop frameworks which allow any user to build, train and deploy a ML model with little to no knowledge of the complex underlying algorithms. In this way, it is not necessary to be an expert in order to deploy a model able to tackle a particular problem in the scientific or commercial domain, while at the same time, offering the expert user the ability of fine tuning their models, without taking too much care in the fundamentals of the model itself. In recent years there has been an increase in the number of alternatives in ML frameworks. Among the most popular (see Figure 4.22) the two which stand out are [56]:

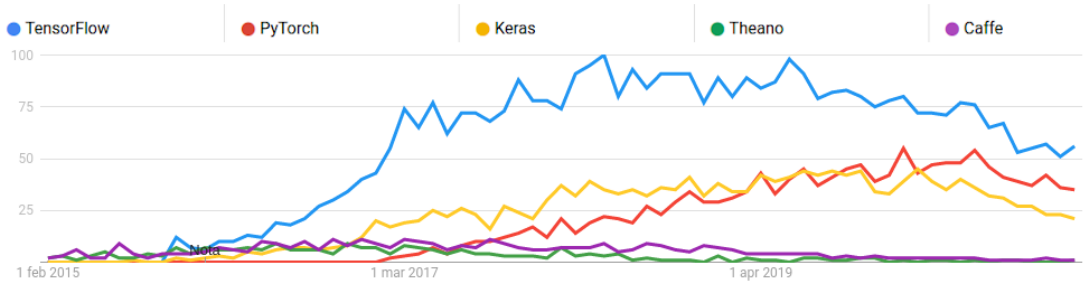


Figure 4.22: Popularity in Google searches of the most popular ML frameworks from January 2010 to February 2021 [76]. Numbers represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. A score of 0 means that there was not enough data for this term.

PyTorch [77]: an open source deep learning framework, originally developed by Meta AI but now part of the Linux Foundation [78] umbrella, built to be flexible and modular for research, with the stability and support needed for production deployment. PyTorch provides a Python package for high-level features like tensor computation with strong GPU acceleration. With the latest release of PyTorch, the framework provides graph-based execution, distributed learning, mobile deployment and quantization.

TensorFlow [79]: a library developed by Google which offers training, distributed training, and inference (TensorFlow Serving) as well as other capabilities such as TFLite (mobile, embedded), Federated Learning (compute on end-user device, share learnings centrally), TensorFlow.js, (web-native ML), TFX for platform etc. TensorFlow is widely adopted, especially in enterprise/production-grade ML. Thanks to its clean design, scalability, flexibility, easy-to-understand documentation and performance, it has reached the top of the list of the ML frameworks worldwide.

4.4.1 TensorFlow

TensorFlow is an open-source software library for dataflow programming across a range of tasks. It is a symbolic math library for numerical computation that uses data flow graphs. Despite suitable for a variety of tasks, it is particularly useful for ML applications, such as neural networks. It was developed by the Google Brain team, now merged into Google DeepMind [80] for internal use and released under the Apache 2.0 open-source license on November 2015. On September 2019, a major update to TensorFlow 2.0 has made the library even more efficient and accessible, by implementing a more intuitive Application Programming Interface (API) [79]. Currently, it is one of the most utilised ML frameworks worldwide, due to its completeness and reliable libraries.

Its architecture is flexible enough to allow users to deploy computation to one or more CPUs or GPUs in a desktop, server, or even mobile device with a single API. Moreover, through the Google Cloud Platform (GCP) it is possible to implement models built with TensorFlow on a Tensor Processing Unit (TPU), Google's custom-developed ASICs used to accelerate ML workloads [81].

In a not exhaustive list of TensorFlow's features, one could mention [82]:

- It runs on Windows, Linux and macOS, and also on mobile devices, including both iOS and Android;
- It provides a Python API which offer flexibility to create all sorts of computations, including any NN architecture one can think of;
- It includes highly efficient C++ implementations of many ML operations, particularly those needed to build NNs. There is also a C++ API to define one's own high-performance operations;
- It provides several advanced optimization nodes to search for the parameters that minimize a cost function: TensorFlow automatically takes care of computing the gradients of the functions one defines, i.e. implements automatic differentiation;
- It also comes with a native visualization tool called TensorBoard, that allows browsing through computation graph, view learning curves, and more;
- Once a model is done with TensorFlow, computations can be deployed to one or more CPUs or GPUs, local or remote via any Cloud Computing Service provider.

As an example, some code snippets showing how a model that performs linear regression can be implemented from scratch and trained in TensorFlow are shown in the following. A correspondent example will be given in the next section for a higher-level framework as well.

Data in TensorFlow, is almost always represented by tensors. However, by using TensorFlow's own functions and class constructors, it is relatively easy to obtain a tensor from more other types of data containers like Python's list or Numpy's array. Here is some data synthesized by adding Gaussian (Normal) noise to points along a line using the random generators from TensorFlow:

```
import tensorflow as tf

# The actual line
TRUE_W = 3.0
TRUE_B = 2.0
NUM_EXAMPLES = 1000
# A vector of random x values
x = tf.random.normal(shape=[NUM_EXAMPLES])
# Generate some noise
noise = tf.random.normal(shape=[NUM_EXAMPLES])
# Calculate y
y = x * TRUE_W + TRUE_B + noise
```

Then the model can be defined, together with weights and bias as variables:

```
class MyModel(tf.Module):
  def __init__(self, **kwargs):
    super().__init__(**kwargs)
    # Initialize the weights to `5.0` and the bias to `0.0`
    # In practice, these should be randomly initialized
    self.w = tf.Variable(5.0)
    self.b = tf.Variable(0.0)
  def __call__(self, x):
    return self.w * x + self.b
model = MyModel()
```

The standard L2 loss can be introduced, also known as MSE (Eq. 4.4):

```
# This computes a single loss value for an entire batch
def loss(target_y, predicted_y):
  return tf.reduce_mean(tf.square(target_y - predicted_y))
```

The training loop, written from scratch in TensorFlow, is relatively straightforward:

```
# Given a callable model, inputs, outputs, and a learning rate...
def train(model, x, y, learning_rate):

  with tf.GradientTape() as t:
    # Trainable variables are automatically tracked by GradientTape
    current_loss = loss(y, model(x))

  # Use GradientTape to calculate the gradients with respect to W and b
  dw, db = t.gradient(current_loss, [model.w, model.b])

  # Subtract the gradient scaled by the learning rate
  model.w.assign_sub(learning_rate * dw)
  model.b.assign_sub(learning_rate * db)

model = MyModel()

# Collect the history of W-values and b-values to plot later
Ws, bs = [], []
epochs = range(10)

# Define a training loop
def training_loop(model, x, y):

  for epoch in epochs:
    # Update the model with the single giant batch
    train(model, x, y, learning_rate=0.1)

    # Track this before I update
    Ws.append(model.w.numpy())
    bs.append(model.b.numpy())
    current_loss = loss(y, model(x))
```

And finally the model is actually trained by a single function call, namely `training_loop(model, x, y)`.

From this example, it is clear how, with a few lines of code, it is possible to implement a ML model without a deep understanding of the theory behind it. However, higher-level libraries, like Keras, allow an even faster deployment of models, as shown in the following section.

4.4.2 Keras

Keras [83] is a deep learning API written in Python, primarily running on the TensorFlow machine learning platform. It offers an intuitive and productive interface for solving modern ML problems. Additionally, Keras can be deployed with JAX or PyTorch as back-end alternatives. Keras provides essential abstractions

and building blocks for developing and deploying ML solutions. Designed for rapid experimentation, its simple interface reduces the number of user actions needed for common tasks and offers clear, actionable feedback when errors occur.

The core data structures of Keras are "layers" and "models". The simplest type of model is the *Sequential* model, a linear stack of layers. However, it is possible to deploy more complex architectures using the Keras *Functional* API, which allows to build arbitrary graphs of layers, or writing models entirely from scratch via subclassing.

As done in the previous section, some code snippets showing how a model that performs regression can be implemented using the Keras APIs with a TensorFlow backend, and how it results in a much simpler and cleaner code base, is shown in the following.

Building a Sequential model is as follows:

```
from tensorflow.keras.models import Sequential

model = Sequential()
```

Then, the add() method is used to stack layers, as follows:

```
from tensorflow.keras.layers import Dense
model.add(Dense(units=64,activation='relu',input_dim=100))
model.add(Dense(units=10,activation='softmax'))
```

where *units* corresponds to the number of nodes, or neurons, of that layer and *activation* is the activation function. In the definition of the first layer, the number of features in input is also specified via the *input_dim* option.

Once the layout of the model is defined, the compile() method is used to configure the learning process:

```
model.compile(loss='categorical_crossentropy', optimizer='sgd')
```

Here which loss function will be used in the learning process is specified, together with the configuration of the *optimizer* (in this example *sgd* stands for Stochastic Gradient Descent).

Now the model can be trained in batches using x_train and y_train as train set and labels:

```
model.fit(x_train,y_train, epochs=5,batch_size=32)
```

After the training is done, the model can be used easily to infer on new data:

```
classes = model.predict(test_data)
```

This example shows how simple it is to deploy a neural network with Keras. However, Keras is also a highly flexible framework suitable to iterate on state-of-the-art research ideas. It follows the principle of *progressive disclosure of complexity*: it makes it easy to get started, yet it makes it possible to handle arbitrarily advance use cases, only requiring incremental learning at each step.

4.5 Machine Learning in High Energy Physics

Standard Model has had a resounding success, including almost all subnuclear particle physics phenomena in nature. However, each problem of interest must be always accompanied by some assumptions and simplifications, including the results of proton-proton collisions at the LHC. In particular, at the LHC experiments such complexity must be multiplied by the scale of a building-sized particle detector. It is also important to mention that physical considerations and studies only allow to solve a direct problem: how does a process X look in a Y detector. However, in order to draw conclusions from any experiment, the inverse problem must be solved: given some readout from a detector Y, what is occurring in terms of a physical process X? Most of the utility of ML for particle physics comes from being able to solve this inverse problem [84], [85].

Compared to traditional, expert-designed algorithms, ML offers two main advantages. Firstly, it delivers higher quality results. Secondly, it reduces effort by replacing highly specialized, manually crafted algorithms with general methods adapted from AI research, which are also used across various fields of study. However, these benefits come at a cost. Most ML algorithms function as "black boxes," meaning their inner workings are not easily understood. The primary issue is not merely the lack of human comprehension, but the specific mathematical requirement that the training data must match the distribution of the data to which the algorithm will be applied. This is rarely the case, as ML methods generally lack formal guarantees of performance when faced with such data shifts. While this may not be a significant concern in many non-scientific applications, scientific work demands extreme precision and reliability. Although there are methods to validate data analysis techniques, expert judgment is almost always necessary to assess each case individually.

In the HEP community, the validation of ML methods follows a similar approach, with case-by-case physical considerations applied at every stage to achieve acceptable results:

Training data selection The data must encompass the entire desired phase space and all relevant physical processes;

Features selection Only the features that correspond to the physics involved are used;

Validation on different samples For an algorithm trained on simulated data, its performance must be measured on a different sample for calibration;

Manual inspection of physically-meaningful distributions Training features must be selected according to their physical distribution: in case of correlations, for instance, a possible bias may be introduced and therefore affect negatively the training procedure.

4.5.1 Event Selection: Separating Signal from Background

Selecting events that contain interesting processes is a fundamental requirement of HEP experiments and is perhaps the most established application of ML in HEP. Most analyses involve measuring the fraction of events that include a specific decay channel. Traditionally, this process involves developing an event selection algorithm, estimating its efficiency in selecting signals and rejecting background noise, and counting the events that pass the algorithm. The conventional method, known as cut-based selection, involves manually constructing a decision tree using physical considerations, Monte Carlo simulations, or both. This procedure can be fully automated using ML algorithms, both in the final statistical analysis [86] and at the initial trigger decision stage [87]. These ML tools have found high-profile application in single t quark searches [88], early Higgs boson searches [89], and the Higgs boson discovery itself [90].

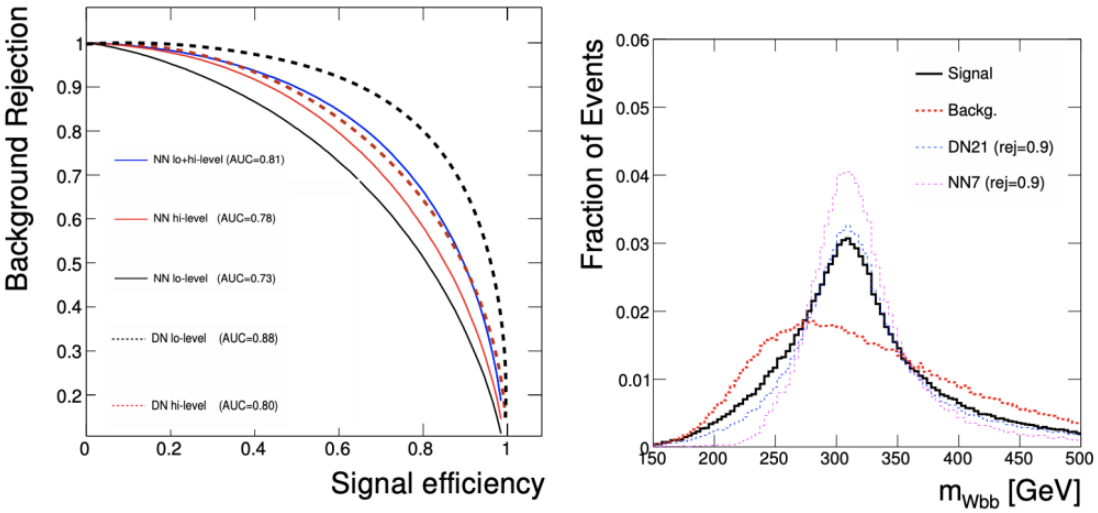


Figure 4.23: On the left, deep networks (DN) performance in signal-background classification compared to shallow networks (NN) with a variety of low- and high-level features [91]. On the right, a comparison of the distributions of invariant mass of events selected by a deep network (DN21) using only object momentum to a shallow network (NN7) that has been trained using this feature, at equivalent background rejection.

In recent years, numerous studies have shown that traditional shallow networks (with a few hidden layers) using physics-inspired engineered ("high-level") features

are outperformed by deep networks (with multiple hidden layers) utilizing higher-dimensional, minimally pre-processed ("lower-level") features. This supports the notion that feature engineering, which applies physics knowledge to construct high-level features, is often overvalued. An early study [91], for instance, compared the performance of shallow and deep networks in distinguishing a cascading decay of new exotic Higgs bosons from the dominant background. This study used a structured data set where a large set of basic low-level features (object four-momenta) was reduced to a smaller set of physics-inspired high-level engineered features. The results, as illustrated in Figure 4.23, demonstrated that the deeper network with lower-level data outperformed the shallow network with higher-level physical features.

4.5.2 Event Reconstruction

Reconstruction is the process of transforming raw detector readouts into physically meaningful objects, such as particle tracks, particle types, and vertices. In the ATLAS, CMS, and LHCb experiments, this involves three distinct operations. First, charged tracks are reconstructed using input from the tracker. Second, information from particle identification subsystems is used to assign particle types to these charged tracks. Third, calorimeters identify some neutral particles that escape the magnetic field. This process can be considered the inverse of simulation: in simulation, the response is computed from given particles, while in reconstruction, the particles causing a given detector response are identified.

ML is a natural approach to this problem due to its algorithmic nature. The simplest method involves simulating an event, using the detector response as features for an ML algorithm, and the Monte Carlo truth as the labels. However, this approach faces numerous algorithmic challenges. High-energy physics events are highly complex and structured, making standard ML algorithms ill-suited for such tasks. Consequently, each reconstruction must be handled by system experts, depending on the type of object being reconstructed.

Tracking

Track-reconstruction algorithms are among the most CPU and data-intensive of all low-level reconstruction tasks. The initial stage involves identifying hits, or points where charge is deposited on a sensing element in the detector. For the pixel sensors in the innermost layer, neighboring hits are clustered into pixel clusters, which then form track seeds. These seeds serve as starting points for a Kalman filter, which extends them into full tracks. ML has proven beneficial in various aspects of track reconstruction. For instance, when multiple tracks pass through the same pixel cluster, ATLAS uses neural networks to return measurements for each track rather than assigning them to the cluster center [92], [93].

Thanks to these algorithms and meticulous tuning, track reconstruction is now nearly 100% efficient, and spuriously reconstructed tracks are rare, indicating that the clustering aspect of tracking is largely resolved. However, reducing CPU overhead remains a significant challenge, especially within high-level trigger farms. In ATLAS and CMS, these clusters are composed of approximately 10,000 processors

that reconstruct about 100,000 events per second [94]. To manage tracking CPU costs, the experiments limit track reconstruction to specific regions of the detector. These regions are selected based on their proximity to muons or calorimeter energy deposits that align with relatively rare physical signatures, such as leptons or high-p_T jets.

Jet Tagging

ML has been extensively applied to various jet classification problems, such as identifying jets originating from heavy (c, b, t) or light (u, d, s) quarks, gluons, and W, Z, and H bosons. Traditionally, these classification problems have been categorized into flavor tagging, which differentiates between b, c, and light quarks; jet substructure tagging, which distinguishes jets from W, Z, t, and H bosons; and quark-gluon tagging.

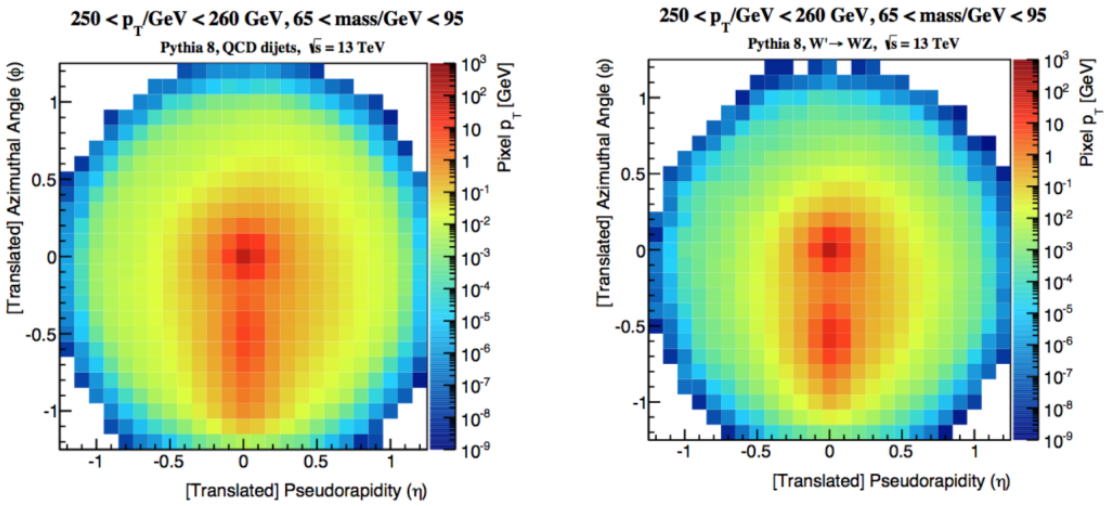


Figure 4.24: Example jet image inputs from the jet substructure classification problem. The background jets (left image) are characterized by a large central core of deposited energy caused by a single hard hadronic parton, while the signal jets (right) tend to have a subtle secondary deposition due to the two-prong hadronic decay of a high-p_T vector boson. Use of image-analysis techniques such as CNNs allow for powerful analysis of this high-dimensional input data.

In 2014, a study recognized [95] that the projective tower structure of calorimeters in nearly all modern HEP detectors resembled the pixels of an image (Figure 4.24). This similarity enabled physicists to apply advances in image classification, such as CNNs. Although the image-based approach has been successful, the irregular geometry of actual detectors necessitates some preprocessing to represent jets as images.

Both ATLAS and CMS have since implemented flavor-tagging NNs that rely on individual tracks or, in the case of CMS, particle-flow candidates. For example, CMS's DEEPJET [96] adopts a neural network that first embeds each flow candidate with a shared transformation and then combines the high-level variable candidates in a single dense network.

High p_T muons

Another application of ML techniques in physical object reconstruction involves high-p_T muon reconstruction. Currently, this measurement is performed by the $TuneP$ algorithm, which selects the best reconstruction from different refit techniques based on a small set of track-quality parameters. The refit techniques include: *inner-track* with tracker-only information (used mainly at low momentum); *tracker-plus-first-muon-station* (TPFMS) using the inner tracker and the innermost muon station containing hits; *picky* designed to handle cases where electromagnetic showers generate a high multiplicity of hits in the muon chambers; *dynamic-truncation* (DYT) for cases when radiative energy losses cause significant bending of the muon trajectory.

Adopting ML techniques, such as tree-based methods and NNs, can identify the optimal refit by considering a larger set of input variables than those used by TuneP. Training these ML models involves comparing the resolution in terms of q/p_T relative residual with the generated information from Monte Carlo simulations. Preliminary results [97] indicate a significant reduction in the q/p_T relative residual distribution tails (outliers) by approximately 60% in the barrel region and 27% in the endcap region using Boosted Decision Trees implemented with the XGBoost library. Additionally, results suggest that further improvements are possible with a larger training dataset.

4.5.3 Fast Simulation

Simulation represents the most CPU-intensive operation in HEP. Achieving fast simulations is crucial because full simulators, which accurately depict particle interactions with matter at a low level, demand significant computational resources and consume a substantial portion of current experimental computing budgets. One promising approach involves Generative Adversarial Networks (GANs). In this method, a generative model G is trained through competition with an adversary network A. While G creates simulated samples, A evaluates whether a given sample originates from the generative model or the full simulator. This adversarial setup drives G to produce samples that mimic those from the original simulation, attempting to deceive A.

However, ensuring the stability of this training process can be challenging, often requiring expert knowledge to construct an effective network. Currently, GAN approaches are applied in simulating electromagnetic showers in calorimeters [98], demonstrating computational speed-ups while maintaining a realistic energy deposition model. Similar techniques have been successful in simulating jet images [99].

4.5.4 Monitoring and Data Quality

The LHC systems and detectors are highly intricate machines equipped with monitoring systems that continuously verify parameters at all levels, from voltages to reconstructed masses from known decays. A significant challenge for these systems is distinguishing legitimate changes in data-taking conditions from equipment mal-

functions. When observed variable distributions deviate from their corresponding references, operators investigate these discrepancies. If unresolved, incidents are recorded and relevant experts are notified. The system includes automatic alarms for detecting significant discrepancies, though false alarms can occur if references are not promptly updated to reflect legitimate changes in data-taking conditions.

ML algorithms are increasingly utilized to monitor detector conditions and predict potential anomalies, a field known as anomaly detection, widely applied in data science [100], [101]. Efforts have already been initiated for the CMS Data Quality Monitoring system, leveraging unsupervised learning techniques such as dimensionality reduction and clustering.

Chapter 5

Fast Machine Learning with Model Compression

In the previous chapter an in-depth description of what a Machine Learning algorithm is was given, together with the definition and some examples of HEP application of Artificial Neural Network. However, the great results achievable with these kind of algorithms can come with a drawback. Deploying large, accurate deep learning models to resource-constrained computing environments such as FPGAs (see Section 3), mobile phones, smart cameras etc. for on-device inference poses a few key challenges [102]. Firstly, state-of-the-art deep learning models routinely have millions of parameters requiring $\mathcal{O}$(MB) storage, whereas on-device memory is limited. Furthermore, it is not uncommon for even a single model inference to invoke $\mathcal{O}(10^9)$ memory accesses and arithmetic operations, all of which consume power and dissipate heat which may drain the limited battery capacity, in the case of a mobile device, and/or test the device's thermal limits.

Addressing these challenges, an increasing amount of research aims to develop methods for compressing neural network models while minimizing any potential degradation in model quality. Latency-sensitive workloads that depend on energy-efficient on-device neural network inference are often constrained by memory bandwidth. Model compression provides the dual advantage of reducing the number of energy-intensive memory accesses and improving inference time by effectively increasing the memory bandwidth available for fetching compressed model parameters.

Some of the techniques used for model compression try to exploit the characteristics of the chip's architecture where the inference is performed, like Quantization (Section 5.1) and efficient resource reusage (explained in Section 5.3.1).

Network Pruning

Model pruning is a technique within the realm of model compression that involves removing (forcing to zero) the less salient connections (parameters) in a neural network. This approach effectively reduces the number of nonzero parameters in the model, typically resulting in little to no loss in the final model quality. By allowing for a trade-off between a slight degradation in model quality and a significant reduction in model size, model pruning can lead to substantial improvements

in inference time and energy efficiency [102].

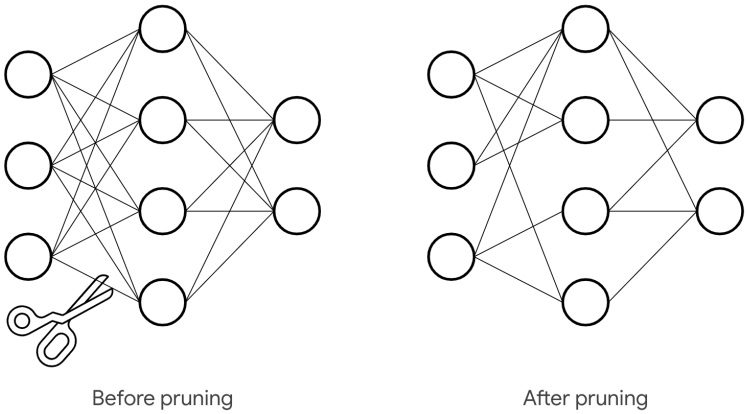


Figure 5.1: Graphical representation of the weight pruning optimization.

The pruning can be done during the training process to allow the NN to adapt to the changes while learning. The TensorFlow Model Optimization API [103] is a common way to perform this optimization. It uses an algorithm designed to iteratively remove connections, based on their magnitude during training. Fundamentally, a final target sparsity (i.e. a target percentage of weights equal to zero) is specified, along with a schedule to perform the pruning (e.g. start pruning at step 2000, stop at step 10000, and do it every 100 steps), and an optional configuration for the pruning structure (e.g. prune individual values or blocks of values). As training proceeds, the pruning routine is scheduled to execute, eliminating the weights with the lowest magnitude (i.e. those closest to zero), until the current sparsity target is reached. Every time the pruning routine is scheduled to execute, the current sparsity target is recalculated, until it reaches the final target sparsity at the end of the pruning schedule by gradually increasing it according to a smooth ramp-up function (see Figure 5.2).

5.1 Quantized Neural Networks

The term Quantization is used in this field to describe the conversion of the arithmetic used within the NN from high-precision floating-points to normalized low-precision integers (fixed-point) [104]. It can be considered as an essential step for efficient deployment of a model on a FPGA (see Section 5.3).

Floating-point representation allows the decimal point to "float" to different places within the number, depending upon the magnitude. Floating-point numbers are divided into two parts, the exponent and the mantissa, whose sum makes up the total number of bits, or *bitwidth*, used to represent a number. This scheme is very similar to scientific notation, which represents a number as $A \times 10^B$, where in this case A is the mantissa and B is the exponent. However, the base of the exponent in a floating-point number is 2, that is $A \times 2^B$. The floating-point number is standardized by IEEE/ANSI standard 754-1985. The basic IEEE floating-point number utilizes an 8-bit exponent and a 24-bit mantissa.

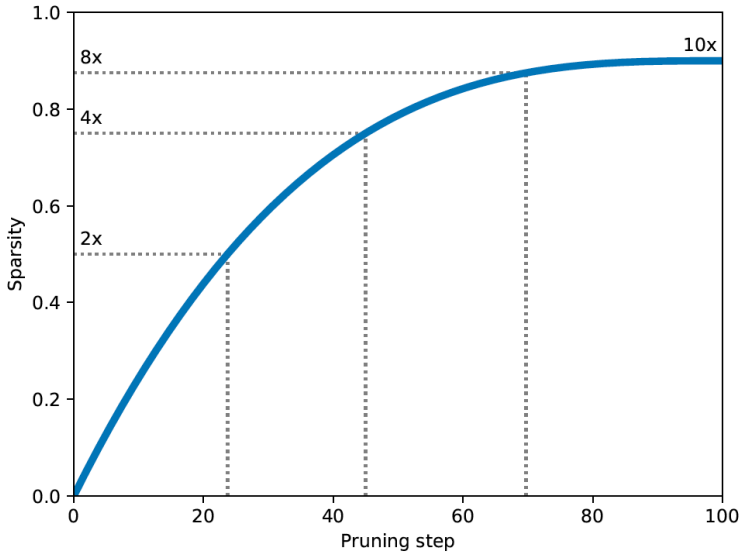


Figure 5.2: Example of sparsity ramp-up function with a schedule to start pruning from step 0 until step 100, and a final target sparsity of 90%. [103]

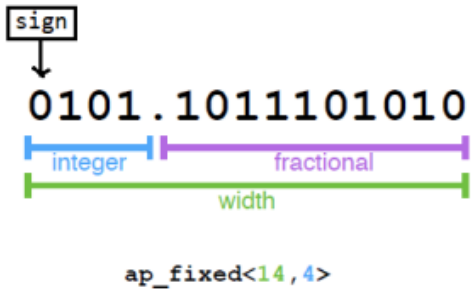


Figure 5.3: Fixed-point number representation. `ap_fixed<`*width*`,`*integer*`>` will be the C type associated to input, output and parameters of the NN model by the hls4ml library in Section 5.3.1.

Fixed-point numbers, instead, consist of two parts, integer and fractional, as shown in Figure 5.3. Compared to floating-point, fixed-point representation maintains the decimal point within a fixed position, allowing for more straightforward arithmetic operations. For example, when dealing with floating-point numbers arithmetic, one must first ensure the decimal points are aligned, by either multiplying the number with more integer bits or dividing the number with the fewest integer bits by 2 until both operands have the same number of bits in the mantissa. However, in case of two fixed-point numbers with the same precision, this is not necessary. The major drawback of the fixed-point system is that a larger number of bits is needed to represent larger numbers or to achieve a more accurate result with fractional numbers.

Briefly, integer quantization consists of approximating real values with integers

[105] according to

$$x_Q = \frac{x}{scale} \tag{5.1}$$

where

$$scale = \frac{\max(x) - \min(x)}{2^N} \tag{5.2}$$

and N is the number of bits used in the approximation. Each layer's weights and activations are given a different scale according to their extremum values. However, the so called *post-training quantization* degrades network performance.

Incorporating resource intensive models without a loss in performance poses a great challenge [106]. In recent years many developments aimed at providing efficient inference from the algorithmic point of view has been achieved. This includes the aforementioned post-training quantization with its related loss in performance and accuracy, due to the loss in precision going from a 32-bit floating precision number to a fixed precision number with less bitwidth. Therefore, a solution is to perform quantization-aware training (as suggested in [106]): a fixed numerical representation is adopted for the whole model, and the model training is performed enforcing this constraint during weight optimization.

5.1.1 QKeras

To simplify the procedure of quantizing Keras models, the *QKeras* library [107] has been developed by a collaboration between Google and CERN: it is a quantization extension to Keras that provides a drop-in replacement for layers performing arithmetic operations. This allows for efficient training of quantized versions of Keras models.

QKeras is designed using Keras' design principle, i.e. being user-friendly, modular, extensible, and "minimally intrusive" to Keras native functionality. The library includes a rich set of quantized layers, it provides functions to aid the estimation of model area and energy consumption, allowing for simple conversion between non-quantized and quantized networks. Importantly, the library is written in such a way that all the QKeras layers maintain a true drop-in replacement from Keras so that minimal code changes are necessary.

In the following, two examples of the QKeras version of native Keras objects are discussed.

The first code modification that is necessary in order to use QKeras' objects consist of typing Q in front of the original Keras data manipulation layers name and specifying the quantization type, i.e. the `kernel_quantizer` and `bias_quantizer` parameters. Only data manipulation layers, which perform some computation that may change the data input type and create variables, are changed to the QKeras version. When quantizers are not specified, no quantization is applied to the layer and it behaves like the unquantized Keras layer.

The second code change is to pass appropriate quantizers, e.g. `quantized_bits`.

In the following code snippet, QKeras is instructed to quantize the kernel and bias to a bit-width of 6 and 0 integer bits. QKeras works by tagging all variables, weights and biases created by Keras as well as the output of arithmetic layers, by

```
QDense(64, kernel_quantizer = quantized_bits(6,0),
        bias_quantizer = quantized_bits(6,0)(x))
QActivation('quantized_relu(6,0)')(x)
```

quantized functions. Quantized functions are specified directly as layers parameters and then passed to `QActivation`, which acts as a merged quantization and activation function. The `quantized_bits` quantizer used above performs mantissa quantization:

$$2^{\text{int}-b+1}\text{clip}(\text{round}(x \times 2^{\text{int}-b-1} \; - 2^{b-1}\, 2^{b-1} - 1) \tag{5.3}$$

where x is the input, b specifies the number of bits for the quantization, *int* specifies how many bits of b are to the left of the decimal point, and `clip` and `round` are function from the *Numpy* library [108].

The quantizer used for the activation functions above, `quantized_relu`, is a quantized version of ReLU (see Section 4.2). Figure 5.4 shows the quantized ReLU function for three different bit widths and two different numbers of integer bits.

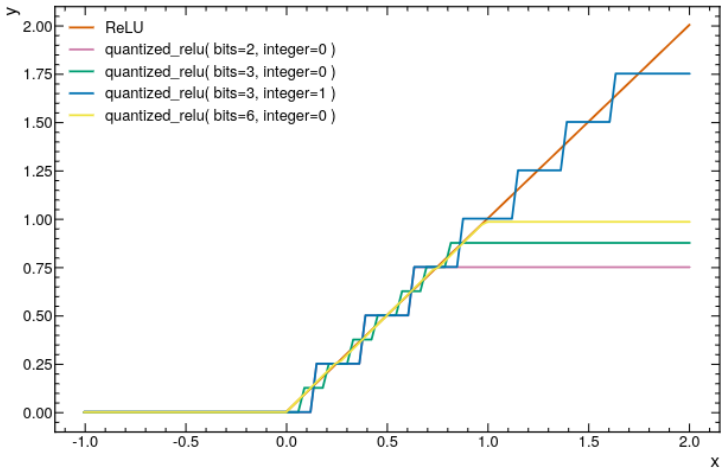


Figure 5.4: The `quantized_relu` activation function as implemented in QKeras for a 2-bit (purple), 3-bit (green and blue) and 6-bit (yellow) precision and for 0 or 1 integer bits. The unquantized ReLU function is shown for comparison (orange).

Through simple code changes like those above, a large variety of quantized models can be created. QKeras can be used to create a range of deep quantized models, trained quantization-aware and based on the same architecture as the baseline model. Finally, it is possible to create an optimally heterogeneously quantized QKeras model with a significantly reduced resource consumption, without compromising the model accuracy.

5.2 Knowledge Distillation

In large-scale machine learning, similar models are often used for both the training stage and the deployment stage, despite their differing requirements [109]. Tasks such as speech and object recognition involve training that must extract structure from large, highly redundant datasets, which does not require real-time operation and can utilize extensive computational resources. In contrast, deployment to a large number of users imposes stringent requirements on latency and computational resources. This suggests that training cumbersome models may facilitate the extraction of structure from data. These cumbersome models could be ensembles of separately trained models or single, large models trained with strong regularizers such as dropout (see Section 4.2). After training the cumbersome model, a process called "distillation" can be used to transfer the knowledge to a smaller model more suitable for deployment. This strategy, commonly known as knowledge distillation (KD) and pioneered in [110], demonstrates that the knowledge acquired by a large ensemble of models can be effectively transferred to a single, smaller model.

A conceptual block that may have hindered further exploration of this promising approach is the tendency to equate the knowledge in a trained model with its learned parameter values. This perspective makes it challenging to envision changing the model's form while preserving its knowledge. A more abstract view of the knowledge, independent of any specific instantiation, is to see it as a learned mapping from input vectors to output vectors. For complex models that classify numerous classes, the usual training objective is to maximize the average log probability of the correct answer. However, this process also results in the model assigning probabilities to incorrect answers. Even when these probabilities are very small, some incorrect answers are still relatively more probable than others. These relative probabilities reveal how the complex model generalizes. For instance, an image of a BMW might be very unlikely to be mistaken for a garbage truck, but this mistake is still far more probable than mistaking it for a carrot.

It is widely accepted that the training objective should closely reflect the user's true objective. Nonetheless, models are typically trained to optimize performance on training data, while the actual goal is to generalize well to new data. Ideally, models should be trained to generalize well, but this requires knowledge about the correct way to generalize, which is not usually available. However, when distilling knowledge from a cumbersome model into a smaller one, we can train the smaller model to generalize in the same manner as the cumbersome model. If the large model generalizes well, perhaps because it is an average of a large ensemble of models, a small model trained to generalize similarly will often perform much better on test data than a small model trained traditionally on the same training set as the ensemble.

A straightforward method to transfer the generalization ability from the cumbersome model to a smaller one is to use the class probabilities produced by the cumbersome model as "soft targets" for training the smaller model. For this transfer phase, we could use the same training set or a separate "transfer" set. When the large model is an ensemble of simpler models, we can use an arithmetic or geometric mean of their individual predictive distributions as the soft targets. High-entropy soft targets provide much more information per training case than hard targets

and reduce the variance in the gradient between training cases. Consequently, the small model can often be trained on much less data than the original large model and with a much higher learning rate.

As an example consider a multi-classification task. Neural networks typically produce class probabilities by using a softmax (see Section 4.2) output activation layer that converts all the outputs for each class of the NN before the activation function, called logits, z_i into a probability q_i by comparing each z_i with the other logits:

$$q_i = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}} \tag{5.4}$$

where the temperature T, which is normally set to 1, has been made explicit. Using a higher value for T produces a softer probability distribution over classes, i.e. a less peaked distribution for each class in the output space.

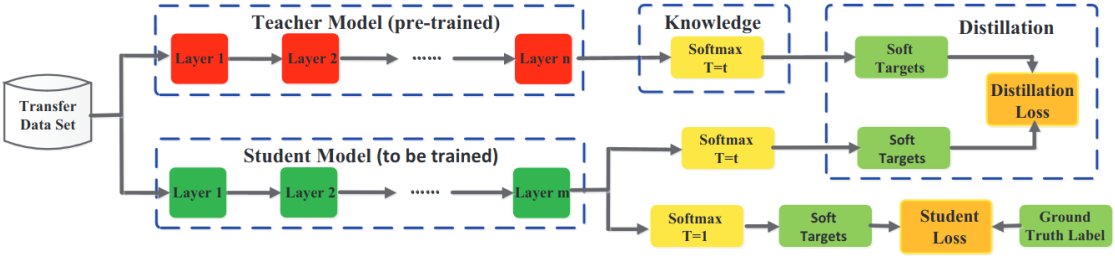


Figure 5.5: Example of a knowledge distillation scheme with two different loss functions for the student model, one targeting the distillation of the knowledge from the teacher, the other used to learn directly from data.

In the simplest form of distillation, knowledge is transferred to the distilled model by training it on a transfer set using a soft target distribution for each case. This distribution is produced by the cumbersome model with a high temperature in its softmax function. The distilled model is trained using the same high temperature, but once trained, it operates with a temperature of 1.

If the correct labels for the transfer set are known, this method can be significantly enhanced by also training the distilled model to produce the correct labels. One approach is to adjust the soft targets using the correct labels, but a more effective method is to use a weighted average of two objective functions, as shown in Figure 5.5. The first objective function is the cross-entropy with the soft targets, computed with the same high temperature in the softmax of the distilled model as was used to generate the soft targets from the cumbersome model. The second objective function is the cross-entropy with the correct labels, computed using the same logits in the softmax of the distilled model but with a temperature of 1.

In knowledge distillation, knowledge types, distillation strategies and the teacher-student architectures play the crucial role in the student learning [111]. In the next section, the focus will be on different categories of knowledge for knowledge distillation.

5.2.1 Types of Knowledge

As anticipated before, vanilla knowledge distillation uses the logits of a large deep model as the teacher knowledge. The different forms of knowledge can be put in

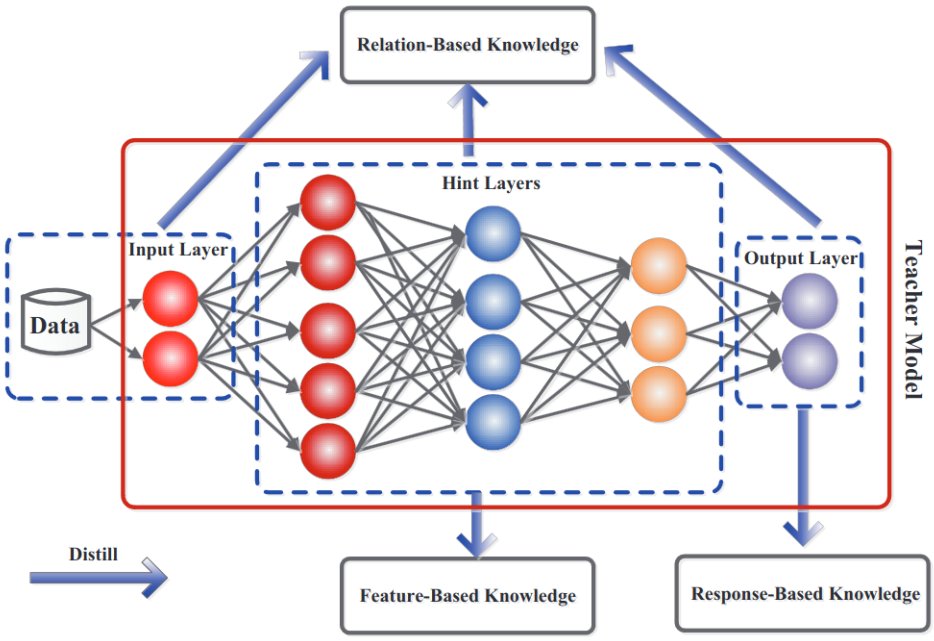


Figure 5.6: A schematic illustration of three different types of knowledge that can be transferred from a deep teacher network: response-based knowledge, feature-based knowledge and relation-based knowledge.

three main categories: response-based knowledge, feature-based knowledge, and relation-based knowledge. An intuitive example of these categories of knowledge within a teacher model is shown in Figure 5.6.

Response-Based Knowledge

Response-based knowledge usually refers to the neural response of the last output layer of the teacher model. The main idea is to directly mimic the final prediction of the teacher model. The response-based knowledge distillation is simple yet effective for model compression, and has been widely used in different tasks and applications. Given a vector of logits z as the outputs of the last fully connected layer of a deep model, the distillation loss for response-based knowledge can be formulated as

$$L_{ResD}(z_t, z_s) = \mathcal{L}_R(z_t, z_s) \tag{5.5}$$

where $\mathcal{L}_R(z_t, z_s)$ indicates a divergence loss (e.g. MSE), and z_t and z_s are logits of teacher and student respectively. A typical response-based KD model is shown in Figure 5.7. In this category the example given in the previous section falls, as a popular technique for image classification. In that case the distillation loss has the form

$$L_{ResD}(p(z_t, T), p(z_s, T)) = \mathcal{L}_R(p(z_t, T), p(z_s, T)) \tag{5.6}$$

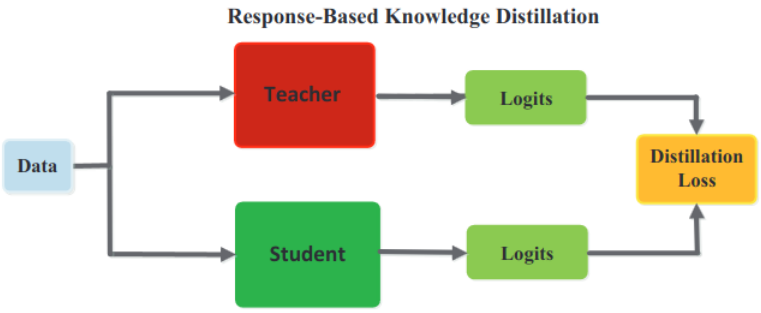


Figure 5.7: Simple diagram of a generic response-based knowledge distillation.

where $p(\cdot, T)$ is the softmax activation function with temperature T.

The idea of the response-based knowledge is straightforward and easy to understand. However, it usually relies on the output of the last layer, e.g., soft targets, and thus fails to address the intermediate-level supervision from the teacher model, which can be very important for representation learning using very deep neural networks.

Feature-Based Knowledge

Deep neural networks are good at learning multiple levels of feature representation with increasing abstraction. This is known as representation learning. Therefore, both the output of the last layer and the output of intermediate layers, i.e., feature maps, can be used as the knowledge to supervise the training of the student model. Specifically, feature-based knowledge from the intermediate layers is a good extension of response-based knowledge, especially for the training of thinner and deeper networks.

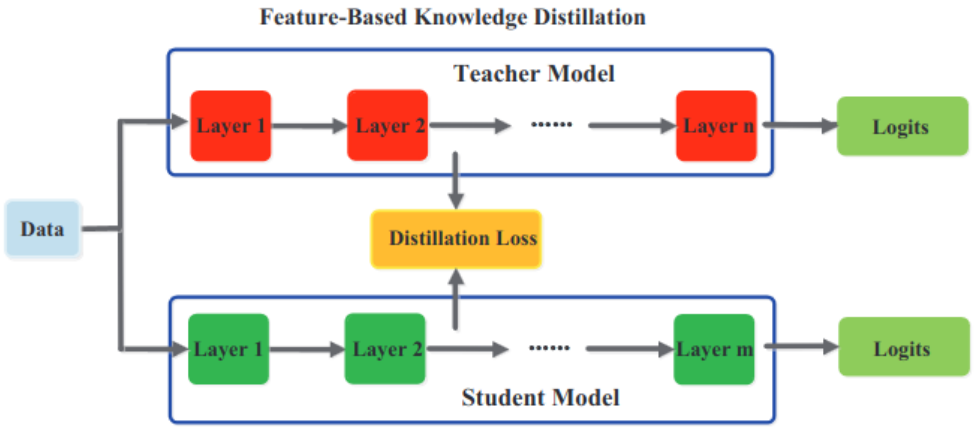


Figure 5.8: Simple diagram of a generic feature-based knowledge distillation.

The intermediate representations were first introduced in [112], to provide hints to improve the training of the student model. The main idea is to directly match the feature activations of the teacher and the student.

Generally, the distillation loss for feature-based knowledge transfer can be formulated as

$$L_{FeaD}(f_t(x), f_s(x)) = \mathcal{L}_F(\Phi_t(f_t(x)), \Phi_s(f_s(x))) \tag{5.7}$$

where $f_t(x)$ and $f_s(x)$ are the feature maps of the intermediate layers of teacher and student models respectively. The transformation functions $\Phi_t(f_t(x))$ and $\Phi_s(f_s(x))$ are usually applied when the feature maps of the two models do not have the same shape. $\mathcal{L}_F(\cdot)$ indicates the similarity function used to match the feature maps of teacher and student which can be of different kinds, from the usual L_2 and L_1 norm to specific losses used to compare distributions like the cross-entropy loss [113] or the maximum mean discrepancy loss [114]. A general feature-based KD model is shown in Figure 5.8.

Though feature-based knowledge transfer provides favorable information for the learning of the student model, how to effectively choose the hint layers from the teacher model and the guided layers from the student model remains is still an open question, as well as how to properly match feature representations of teacher and student also needs to be explored, due to the significant differences between sizes of hint and guided layers.

Relation-Based Knowledge

Both response-based and feature-based knowledge use the outputs of specific layers in the teacher model. Relation-based knowledge further explores the relationships between different layers or data samples. For example, to explore the relationships between different feature maps, in [115] it is proposed to summarize the relations between pairs of feature maps by computing the inner products between features from two layers. Using the correlations between feature maps as the distilled knowledge, knowledge distillation via singular value decomposition was proposed to extract key information in the feature maps.

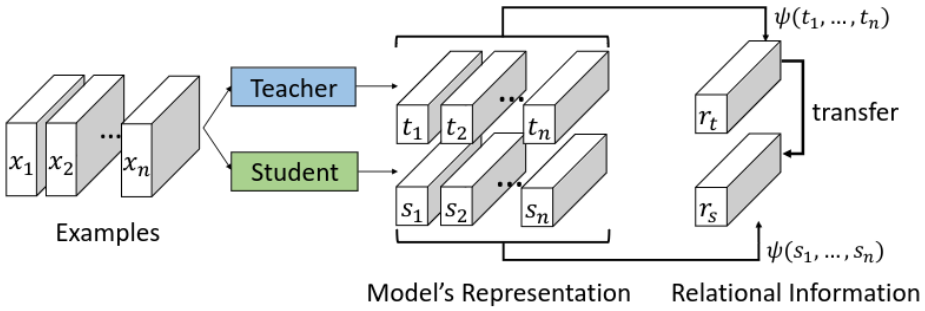


Figure 5.9: Simple diagram of a relation-based knowledge distillation.

In general, the distillation loss of relation-based knowledge based on the relations of feature maps can be formulated as

$$L_{RelD}(f_t, f_s) = \mathcal{L}_{R^1}(\Psi_t(\hat{f}_t, \check{f}_t), \Psi_s(\hat{f}_s, \check{f}_s)) \tag{5.8}$$

where f_t and f_s are the feature maps of teacher and student models. Pairs of feature maps are chosen from the teacher, $\hat{f}_t$ and $\check{f}_t$, and from the student, $\hat{f}_s$ and

$\check{f}_s$. $\Psi_t(\cdot)$ and $\Psi_s(\cdot)$ are the similarity functions for pairs of feature maps from the two models. Finally, $\mathcal{L}_{R^1}$ indicates the correlation function between the feature maps.

Traditional knowledge transfer methods often involve individual knowledge distillation. The individual targets of a teacher are directly distilled into a student. However, knowledge can also be presented by relations of the learned representations instead of the individual ones [116]; an individual data example, e.g., an image, obtains a meaning in relation to or in contrast with other data examples in a system of representation, and thus primary information lies in the structure in the data embedding space. Thus, structural knowledge can be transferred using mutual relations of data examples in the teacher's output representation. Unlike the previous approaches, a relational potential ψ is computed for each n-tuple of data examples and information is transferred through the potential from the teacher to the student. Defining $t_i = f_T(x_i)$ and $s_i = f_S(x_i)$ as the output of teacher and student on an input x_i, the distillation loss can be expressed as

$$L_{RelD} = \sum_{(x_1,\ldots,x_n)\in\chi^N} \mathcal{L}(\psi(t_1,\ldots,t_n),\psi(s_1,\ldots,s_n)) \tag{5.9}$$

where $(x_1,\ldots,x_n)$ is a n-tuple drawn from χ^N, ψ is a relational potential function that measures a relational energy of the given n-tuple, and $\mathcal{L}$ is a loss that penalizes difference between the teacher and the student.

A typical instance relation-based KD model is shown in Figure 5.9.

5.2.2 Distillation Schemes

In this section, we discuss the distillation procedures for both teacher and student models. According to whether the teacher model is updated simultaneously with the student model or not, the learning schemes of knowledge distillation can be directly divided into three main categories: offline distillation, online distillation and self-distillation, as shown in Figure 5.10.

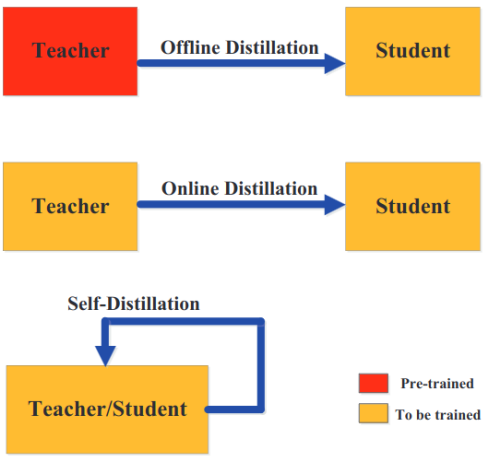


Figure 5.10: The three different distillation strategies.

Offline Distillation

In the most common use of KD, the knowledge is transferred from a pre-trained teacher model into a student model. Therefore, the whole training process has two stages, namely:

1. the large teacher model is first trained on a set of training samples before distillation;

2. the teacher model is used to extract the knowledge in the forms of logits or intermediate features, which are then used to guide the training of the student model during distillation.

The first stage in offline distillation is usually not discussed as part of knowledge distillation, i.e., it is assumed that the teacher model is pre-defined. Little attention is paid to the teacher model structure and its relationship with the student model. Therefore, the offline methods mainly focus on improving different parts of the knowledge transfer, including the design of knowledge and the loss functions for matching features or distributions matching.

Offline methods offer significant advantages due to their simplicity and ease of implementation. For instance, a teacher model can consist of multiple models trained with different software packages, potentially distributed across various machines. The knowledge from these models can be extracted and stored in a cache.

Typically, offline distillation methods involve a one-way knowledge transfer and a two-phase training process. Despite the unavoidable complexity and extensive training time required for high-capacity teacher models, the training of the student model in offline distillation is generally efficient with the teacher model's guidance. Nevertheless, there is always a capacity gap between the large teacher and the smaller student, causing the student to heavily depend on the teacher.

Online Distillation

Although offline distillation methods are simple and effective, some issues in offline distillation have attracted increasing attention from the research community, like the difficulties of transferring knowledge when the gap in complexity and size between teacher and student is very big [117]. To overcome the limitation of offline distillation, online distillation is proposed to further improve the performance of the student model.

An example of this kind of distillation is called Deep Mutual Learning [118]. In mutual learning, a group of untrained students learn simultaneously to solve a task together. Each student is trained using two types of losses: a conventional supervised learning loss and a mimicry loss that aligns each student's class posterior with the class probabilities of other students. This training method enables each student in a peer-teaching scenario to learn significantly better than in a conventional supervised learning scenario. Additionally, student networks trained through mutual learning outperform those trained via conventional distillation from a larger pre-trained teacher. Conventional distillation typically requires a teacher larger and

more powerful than the student, but mutual learning with several large networks often leads to better performance compared to independent learning.

Online distillation is a one-phase end-to-end training scheme very adaptable to a parallel computing paradigm. However, existing online methods (e.g., mutual learning) usually fails to address the high-capacity teacher in online settings, making it an interesting topic to further explore the relationships between the teacher and student model in online settings.

Self-Distillation

In self-distillation, the same networks are used for the teacher and the student models. This can be regarded as a special case of online distillation. This can be done in different ways, e.g.:

- Deeper sections of a network could transfer knowledge to its initial sections [119];
- Knowledge in the earlier epochs of the network (teacher) can be transferred into its later epochs (student) to create a more generalized network [120];
- An ensemble of students with the same architecture can be used and at each consecutive step, a new identical model is initialized from a different random seed and trained from the supervision of the earlier generation, starting from the teacher in the first step. At the end of the procedure, additional gains can be achieved with an ensemble of multiple students generations [121].

5.3 NN Inference on FPGAs

In the previous section two main ways of model compression and optimization were illustrated. However, no consideration were given on the type of hardware used to actually use the NNs, i.e. perform the inference on new data. A lot of research is being carried out on the subject of heterogeneous computing, that is the use of different kinds of hardware to perform different computing tasks in a single computing environment. In this field the use of FPGAs as inference accelerators can be placed. A more detailed description on what an FPGA is can be found in Chapter 3, but, to summarize, a Field Programmable Gate Array is a piece of hardware which can be programmed to implement electronic circuits in their fabric, providing huge power, area, and performance benefits over software applications, and at the same time they can be reprogrammed cheaply and easily with respect to ASICs.

The advantages of FPGAs can be very useful for the task of NN inference, especially when considering IoT and Edge devices, which are asked to perform quick computations with a usually low power consuming chip without interacting with a computing cloud; or, in the case studied in this thesis, when the speed and low hardware footprints are essential to perform the selection of events with a 40 MHz rate, i.e. act as a trigger in an LHC experiment.

Nonetheless, the passage from a purely software object, like a NN built using Keras (see Section 4.4.2) in Python, to a firmware implementable on a FPGA is

not without hurdles. Indeed, a Hardware Descriptive Language representation of the model must be created eventually and the knowledge needed to create this kind of description is usually not common in the researchers who deals with ML and NNs, due to the skills needed for the optimization and parameters tuning needed to create an efficient firmware.

Fortunately there are tools that help this literal translation from software to hardware, like the one used in this work which will presented in the next Section.

5.3.1 HLS4ML

As explained in Section 3.2.2, HLS is the process of automatic generation of hardware circuit from "behavioral descriptions" contained in a C or C++ program. HLS acts as a bridge between hardware and software domains, providing an improvement in productivity for hardware designers who can work at a higher level of abstraction while creating high performance hardware as well as an improvement in system performance for software designers who can accelerate the computationally intensive parts of their algorithms on a new compilation target, i.e. the FPGA.

Using HLS design methodology allows to develop algorithms at the C-level typically associated to a shorter development time. Moreover, it is easier to validate functional correctness at this level than with traditional HDLs.

The hls4ml tool [122], [123] allows physicists to rapidly prototype ML algorithms for both firmware feasibility and physics performance without extensive Verilog/VHDL experience, thus greatly decreasing the time for algorithm development cycles while preserving engineering resources. It is being developed focusing on the task of the FPGA-based triggers of the ATLAS and CMS experiments at LHC (see Section 1.1.2 and Chapter 2) with algorithm latencies in the microsecond range, fully pipelined to handle the 40 MHz LHC collision rate. For this task, solutions with either CPUs or GPUs are not possible due to the severe time limitation imposed. Such latencies are unique to LHC trigger challenges, and therefore few general tools exist for this application. Nonetheless, the hls4ml package is a general purpose tool and is designed to serve a broad range of applications in particle physics and beyond, from trigger and DAQ tasks to longer latency trigger tasks (milliseconds) and CPU-FPGA co-processor hardware.

In other words, hls4ml opens up the possibility to translate ML algorithms, built using frameworks like TensorFlow, into HLS code. In this way a trained Neural Network, defined by its architecture, weights, and biases, can be made ready for hardware synthesis with few lines of code. A schematic of a typical workflow is illustrated in Figure 5.11.

The part of the workflow illustrated in red indicates the usual software workflow required to design a NN for a specific task. The blue section of the workflow is the task of hls4ml, which translates a model into an HLS project that can be synthesized and implemented to run on an FPGA.

At a high level, FPGA algorithm design differs significantly from programming a CPU because it allows independent operations to run entirely in parallel. This capability enables FPGAs to perform trillions of operations per second while maintaining relatively low power consumption compared to CPUs and GPUs. However, these operations utilize dedicated resources on the FPGA, which cannot be dynam-

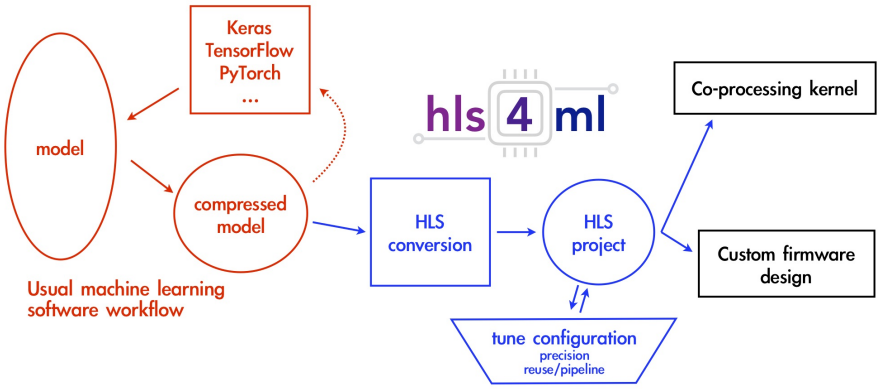


Figure 5.11: A typical workflow to translate a model into an FPGA implementation using hls4ml.

ically reallocated during execution. The main challenge in developing an optimal FPGA implementation is to balance resource usage with the latency and throughput requirements of the target algorithm. Important metrics for evaluating an FPGA implementation include:

Latency : the total time (typically expressed in units of "clocks") required for a single iteration of the algorithm to complete;

Initiation Interval : the number of clock cycles required before the algorithm may accept a new input. Initiation interval (often expressed as "II") is inversely proportional to the inference rate, or throughput; an initiation interval of 2 achieves half the throughput as an initiation interval of 1. Consequently, data can be pipelined into the algorithm at the rate of the initiation interval;

Resource Usage : usually expressed as onboard FPGA memory (BRAM), digital signal processing (arithmetic) blocks (DSPs), and registers and programmable logic (flip-flops and lookup tables).

The hls4ml tool offers numerous configurable parameters that allow users to explore and customize trade-offs between latency, initiation interval, and resource usage for their specific applications. Given that each application has unique requirements, the primary objective of the hls4ml package is to enable users to optimize these parameters through automated neural network translation and iterative FPGA design. In practice, translating a neural network with hls4ml is significantly faster (taking minutes to hours) than manually designing a specific neural network architecture for an FPGA. This rapid prototyping capability allows machine learning algorithms to be developed quickly without requiring dedicated FPGA engineering support. For physicists, this makes designing physics algorithms for the trigger or data acquisition systems much more accessible and efficient, potentially reducing the "time to physics" considerably.

Thanks to the architecture of FPGAs a new aspect of the performance of a model must be taken in consideration when implementing NNs on this kind of

hardware: parallelization. Indeed, the trade-off between latency, throughput, and FPGA resource usage is determined by the parallelization of the inference calculation. In hls4ml, this is configured with a multiplier "reuse factor" that sets the number of times a multiplier is used in the computation of a layer's neuron values. With a reuse factor of one, the computation is fully parallel. With a reuse factor of R, $1/R$ of the computation is done at a time with a factor of $1/R$ fewer multipliers. This is illustrated in Figure 5.12.

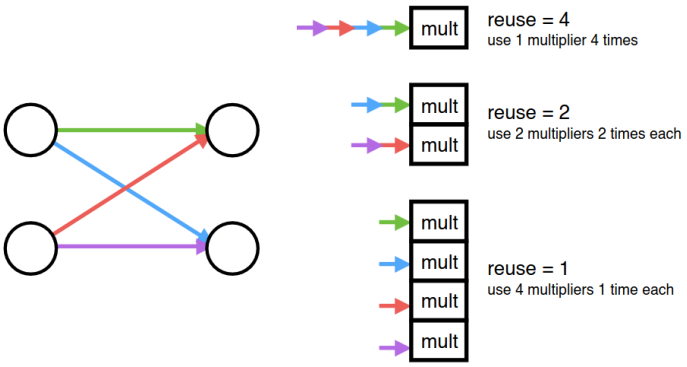


Figure 5.12: Illustration of multiplier resource usage for different values of reuse factor in the case of a two neuron pairs network linked by 4 connections.

FPGA multipliers are pipelined; therefore, the latency of one layer computation, L_m, is approximately

$$L_m = L_{mult} + (R-1) \times \mathrm{II}_{mult} + L_{activ} \tag{5.10}$$

where L_{mult} is the latency of the multiplier, II_{mult} is the initiation interval of the multiplier, and L_{activ} is the latency of the activation function computation. This expression is approximate because, in some cases, additional latency can be incurred for signal routing, for instance in the addition of multiplication results contributing to a neuron value. In this case each layer calculation is implemented independently and sequentially. The calculation of one layer cannot be initiated until the calculation of the previous layer has completed. Therefore, the total latency is equal to the sum of latencies of each layer plus the latency required to connect the layers. The number of inferences completed per unit time is inversely proportional to the reuse factor.

Some code snippets are shown in the following to explain how an already trained model can be converted into an HLS project using the hls4ml Python API.

Firstly, the model must be loaded:

```
import hls4ml
import qkeras
model = qkeras.utils.load_qmodel("quantized_model.h5")
```

Then, a *configuration* has to be created:

```
config = hls4ml.utils.config_from_keras_model(model,
                                granularity = 'name')
cfg = hls4ml.converters.create_config(part='xc7z020clg400-1',
                                backend='Vivado')
cfg['HLSConfig'] = config
```

The `config_from_keras_model()` function returns a Python *dictionary* and takes the following compulsory arguments:

- The Python object containing the NN;
- The *granularity* (`name, type` or `model`) determines the desired level of detail for parameter tuning. Opting for `name` enables the configuration of each layer and activation function independently. Conversely, texttttype is employed when a shared configuration is desired for all layers of the same type. Lastly, `model` involves utilizing a single configuration for the entire model.

By modifying the configuration dictionary it is possible to change the arithmetic precision used for weights, biases and results.

On the other hand, `create_config` creates the actual instruction to write the final HLS code, like the target FPGA and the backend used for the synthesis and implementation of the project.

After the configuration, the model can be converted by typing:

```
hls_model = hls_model = hls4ml.converters.keras_to_hls(cfg)
```

Now, typing `hls_model.compile()`, the `hls_model` can be compiled, i.e. scripts for the chosen backend are generated containing the instructions for synthesizing the model with the provided device as target hardware. It is also possible to synthesize the project inside a Python session with the `hls_model.build()` function.

It is clear from the couple of lines of code shown, how easy it is to create the HLS project, making it feasible also for people who are not experts in FPGAs or hardware. Indeed, the goal of the hls4ml package is to empower a HEP physicist to accelerate ML algorithms using FPGAs, thanks to its tools for ML models conversion into HLS. Indeed, hls4ml makes the translation of Python objects into HLS, and its synthesis, parts of an automatic workflow, allowing fast deployment times also for those who know how to write software, yet are not experts on FPGAs.

Chapter 6

Finding BSM signals with Anomaly Detection

As described in Section 2.2, the L1T in the CMS experiment employs a series of algorithms implemented as logic circuits on custom electronic boards with FPGAs. This stage filters out over 98% of events, reducing the incoming data stream to 100K events/s. Given the brief interval between collisions (25 ns) and the limited buffer capacity, all L1T algorithms must be executed within few microseconds. The second stage, known as the HLT, processes events using a computer farm equipped with commercial CPUs and GPUs. This stage runs hundreds of complex selection algorithms within $\mathcal{O}(100)$ ms. These trigger algorithms, schematically illustrated in Figure 6.1, are designed to ensure a high acceptance rate for the physics processes of interest.

In searches for new, unobserved physics phenomena, specific theory-driven scenarios are typically considered. While this supervised approach has been successful in theory-motivated searches, such as the discovery of the Higgs boson, it may pose limitations when there is no strong theoretical guidance. The ATLAS and CMS trigger systems could potentially discard valuable events, risking the loss of opportunities to discover new physics.

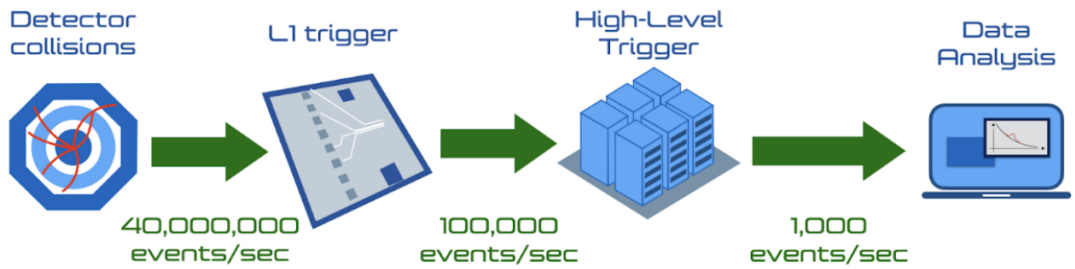


Figure 6.1: Real-time data processing flow in the ATLAS and CMS experiments: 40 million collisions occur each second and are processed by a hardware-based event selection system using algorithms implemented as logic circuits on custom electronic boards. From these events, 100,000 are selected and passed to the second stage, the High-Level Trigger, which further reduces the rate to around 1,000 events per second for offline physics analysis.

This is why recently there has been an increasing focus on unsupervised and semi-supervised methods for data selection and analysis, particularly on Anomaly Detection strategies, explained in Section 4.3.1, using Deep Learning algorithms (see Section 4.2). These approaches aim to derive a metric directly from LHC data that can rank events by their typicality. Outliers in this typicality distribution could represent a subset of data enriched with rare, possibly unobserved, physics processes. Comprehensive reviews of several proposed methods are provided in [124], [125], and related sources.

While much of this effort has been directed toward offline data analysis, a parallel initiative is needed to integrate AD algorithms into the LHC trigger system, potentially even at the L1T. This would allow an unbiased dataset to be analyzed by the AD algorithm before any event is discarded [126], [127]. Rare event topologies could then be collected in a dedicated data stream, similar to the CMS exotica hotline [128] used during the first year of LHC data collection. Studying these events could lead to the development of new theoretical models for unobserved physics phenomena, which could be tested in future data-taking campaigns. This strategy would be even more effective if included in the L1T rather than at the HLT level where some selection bias could be present.

Since each L1T event must be processed within a few microseconds, trigger decisions could be made using algorithms hard-coded into the hardware as logic circuits. Deploying DL algorithms in the L1T Field Programmable Gate Arrays could enhance both the complexity and accuracy of these algorithms. This was indeed one of the reason the hls4ml library, explored in Section 5.3.1, was introduced as a tool to convert DL models into electronic circuits. By integrating an entire neural network onto the FPGA, hls4ml prioritizes inference speed, making it ideal for small networks with $\mathcal{O}(100\text{ ns})$ latencies.

The Dataset

Proton-proton collisions at the LHC can result in the production and observation of numerous processes predicted by the Standard Model (SM) of particle physics. A brief overview of the SM particle content can be found in [129]. The occurrence rate of each process can be calculated using the SM framework and validated through experimental measurements. In this work, the events under study involves electrons (e) and muons (μ), which, along with taus (τ) and their neutrino counterparts, make up the three lepton families. While an unfiltered dataset would more accurately represent an unbiased L1T data stream, such dataset was used as a more manageable use case to test the strategy.

In the confined space of an LHC detector, electrons and muons are stable particles that do not decay inside the detector and can be directly observed as they pass through the detector material. In contrast, τs are much heavier than electrons and muons and decay rapidly into other particles. In some of these decays, electrons and muons are produced. At the LHC, the primary source of high-energy leptons is the production of W and Z bosons [130], which are among the heaviest SM particles, with masses around 80 and 90 GeV, respectively. After their production, W and Z bosons quickly decay into other particles, particularly leptons. While W and Z bosons are mostly produced directly in proton collisions, a

significant fraction of W bosons also arise from the decay of top quarks (t) and anti-top quarks ($\bar{t}$). The top quark, being heavy and highly unstable, rapidly decays into a W boson and a bottom quark, leading to events with jets or combinations of an electron, μ, τ, neutrino, and jet.

Leptons can also originate from rarer processes involving W and Z bosons, such as Higgs boson decays or multi-boson production. However, due to the low production probability of these processes, they are then excluded from this study.

Leptons can be produced as a result of the creation of light quarks (up, down, charm, strange, and bottom) and gluons, as predicted by Quantum Chromodynamics (QCD) [131]. Since quarks and gluons carry a net color charge and are subject to color confinement, they cannot exist in isolation and are not observed directly. Instead, they combine to form color-neutral hadrons through a process called hadronization, which produces a collimated spray of hadrons known as a jet. Jets are typically defined by algorithms that cluster these particles, such as the anti-k$_t$ algorithm [37]. Leptons are rarely produced within jets and where they generally arise from the decay of unstable hadrons. However, QCD multi-jet production is the most common process at the LHC, making its contribution significant and taken into account in analyses.

The processes mentioned above are the primary contributors to the e or μ data stream, which refers to the set of collision events selected based on the presence of an e or μ with energy exceeding a specified threshold. The dataset considered as not anomalous includes the simulation of such a stream. Additionally, benchmark examples of potential new lepton-production processes are provided. These involve the production of hypothetical, yet unobserved particles, serving as examples of data anomalies that could be useful for evaluating the performance of an AD algorithm.

The dataset used for this study is a refined version of the high-level-feature (HLF) dataset used in [126]. Proton-proton collisions are generated using the `PYTHIA8` event-generation library [132], fixing the center-of-mass energy to the LHC Run-II value (13 TeV) and the average number of overlapping collisions per beam crossing (pileup) to 20. Events generated by `PYTHIA8` are processed using the DELPHES library [133] to simulate detector efficiency and resolution effects. The upgraded design of the CMS detector, intended for the High-Luminosity LHC phase [134], is used as a benchmark, specifically utilizing the CMS HL-LHC detector card distributed with DELPHES. The DELPHES particle-flow (PF) algorithm combines data from various detector components to produce a list of reconstructed particles, referred to as PF candidates. For each particle, the algorithm provides measured energy and flight direction, classifying particles into one of three categories: charged particles, photons, or neutral hadrons. Additionally, separate lists of reconstructed electrons and muons are provided. Jets are clustered from the reconstructed PF candidates, using the FASTJET [135] implementation of the anti-k$_t$ jet algorithm [37].

As anticipated before, many SM processes would contribute to the considered single-lepton dataset. For simplicity, the list of relevant SM processes was restricted to the four with the highest production cross sections, namely:

- Inclusive W boson production, where the W boson decays to a charged lepton

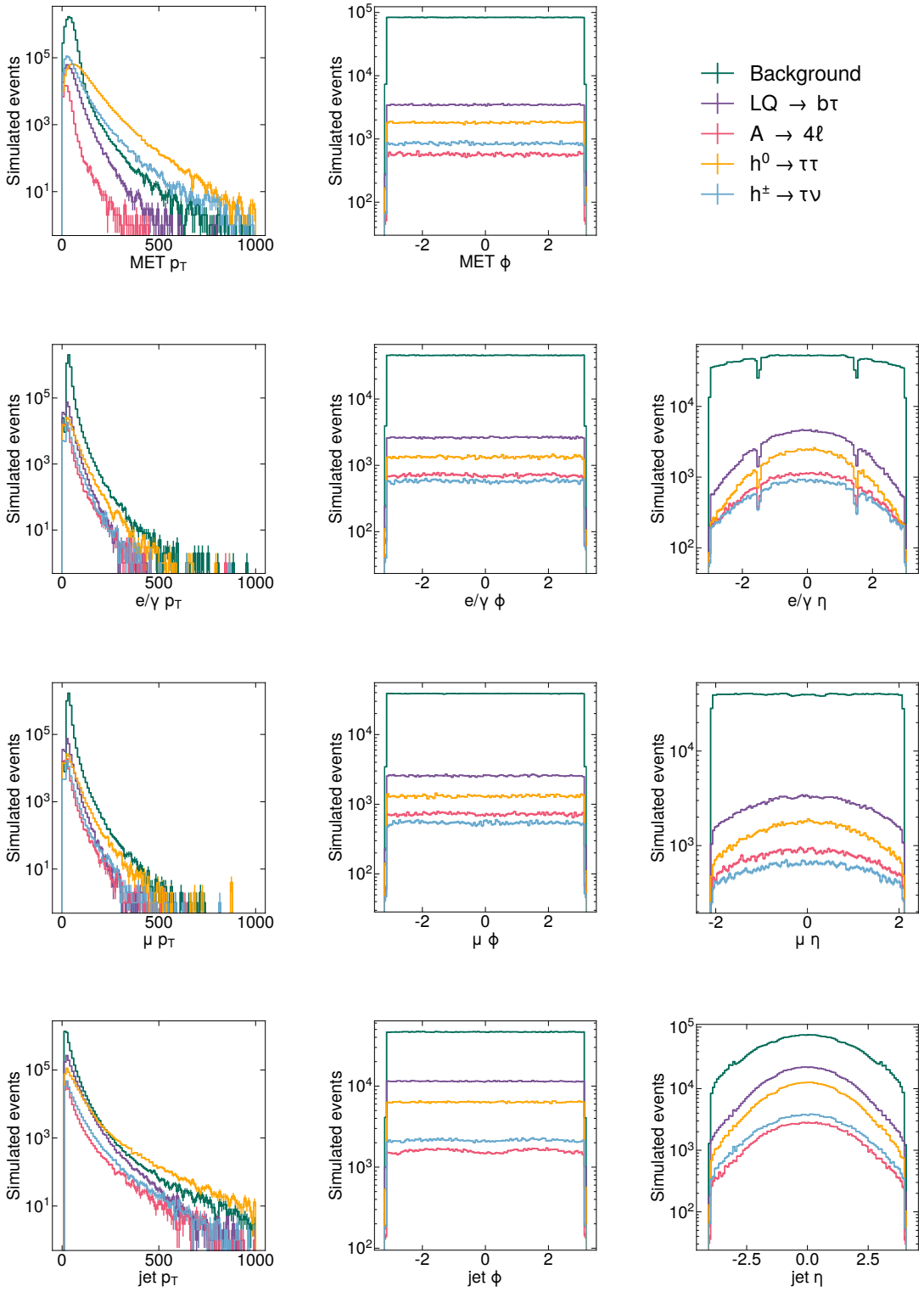


Figure 6.2: Distribution of the p_T (left), ϕ (center) and η (right) coordinates of the physics objects entering the dataset, for missing transverse energy, MET (top row), electrons (second row), muons (third row) and jets (bottom row).

(ℓ) and a neutrino (ν): $W \to \ell\nu$ ($\ell = e, \mu, \tau$);

- Inclusive Z boson production: $Z \to \ell\ell$ ($\ell = e, \mu, \tau$);
- $t\bar{t}$ production;
- QCD multijet production.

The samples are combined to create a SM cocktail dataset by scaling down the high-statistics samples ($t\bar{t}$, W, and Z) to match the lowest-statistics sample (QCD, the most computationally expensive to generate), based on their production cross sections (estimated at leading order with `PYTHIA`) and selection efficiencies, as shown in Table 6.1.

Process	Acceptance	L1 trigger efficiency	Cross section [nb]	Event fraction	Events /month
W	55.6%	68%	58	59.2%	110M
QCD	0.08%	9.6%	$1.6 \cdot 10^5$	33.8%	63M
Z	16%	77%	20	6.7%	12M
$t\bar{t}$	37%	49%	0.7	0.3%	0.6M

Table 6.1: Acceptance and L1 trigger (i.e. p_T and Iso requirement) efficiency for the four studied SM processes. The total cross section before the trigger, the expected number of events per month and the fraction in the SM cocktail are listed.

Each event is described by a list of four-momenta for high-level reconstructed objects: muons, electrons, and jets. To simulate the limited bandwidth of a typical L1T system, only the top 4 muons, 4 electrons, and 10 jets are considered, ranked by decreasing p_T. Events with fewer particles are zero-padded to ensure consistent input size, as is done in actual L1T systems. Each particle is defined by its p_T, η, and ϕ values (for the definitions of η and ϕ see Section 2). Additionally, the MET is included, represented by its magnitude and ϕ coordinate, calculated as the vector opposite to the sum of the transverse momenta of all reconstructed particles in the event.

Once generated, events are filtered requiring a reconstructed electron or a muon with $p_T > 23$ GeV within $|\eta| < 3$ and $|\eta| < 2.1$, respectively. Up to ten jets with $p_T > 15$ GeV within $|\eta| < 4$ are included in each event, together with up to four muons with $|\eta| < 2.1$ and $p_T > 3$ GeV, up to four electrons with $|\eta| < 3$ and $p_T > 3$ GeV, and the MET. Given these requirements, the four SM processes listed above provide a realistic approximation of a L1T data stream [136].

In addition to the four SM processes listed above (provided in [137]), the following Beyond the Standard model (BSM) signals are considered to benchmark anomaly-detection capabilities:

- A leptoquark (LQ) [138] with mass 80 GeV, decaying to a b quark and a τ lepton [139];
- A neutral scalar boson (A) with mass 50 GeV, decaying to two off-shelf Z bosons, each forced to decay to two leptons: $A \to 4\ell$ [140];

- A scalar boson with mass 60 GeV, decaying to two τs: $h^0 \to \tau\tau$ [141];
- A charged boson with mass 60 GeV decaying to a τ and a ν: $h^{\pm} \to \tau\nu$ [142].

The distributions of the features for the SM processes and for the chosen BSM models are shown in Figure 6.2. All expected features are observed, e.g., the detector ϕ symmetry, the detection inefficiency in η in the transition regions between detector components, and the different p_T distributions for the different processes.

In total, the background sample consists of 8 million events. Of these, 50% are used for training, 40% for testing and 10% for validation. The new physics benchmark samples are only used for evaluating the performance of the models.

6.1 Knowledge Distillation for Fast BSM events search

To tackle the challenge described in the previous section, in [143] an Autoencoder (described in Section 4.3.1) is proposed as an unbiased algorithm able to perform Anomaly Detection at the trigger level without a theoretical prior. Indeed, as discussed in [126], one can train an AE on a given data sample by minimizing a measure of the distance between the input and the output, which takes the role of loss function.

The architecture is part of the CNN family [144] and it is shown in Figure 6.3. The encoder takes as input the single-channel 2D array of four-momenta including the two MET-related features (magnitude and ϕ angle) and zeros for MET η, resulting in a total input size of $19 \times 3 \times 1$. It should be emphasised that the dataset does not contain image data, rather the tabular data is being treated as a 2D image to make it possible to use the CNN architecture. In [143] a direct implementation on FPGA was considered meaning that some of the features of the model were thought to optimize the algorithm on hardware. Specifically, the input is zero-padded in order to resize the image to $20 \times 3 \times 1$, which is required in order to parallelize the network processing in the following layer on the FPGA, given that 19 is a prime number and the rows of the "image" could not be divided and sent to a number of parallel processors.

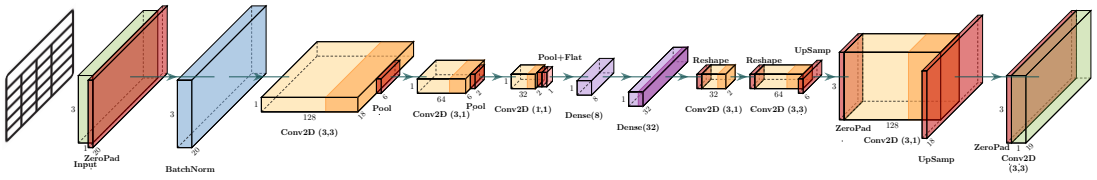


Figure 6.3: Network architecture for the CNN AE teacher model.

After padding, the input is scaled by a batch normalization layer, which regularize the input and helps in the training procedure [145]; and then it is processed by a stack of three CNN blocks, each including a 2D convolutional layer followed by a ReLU (see Section 4.2) activation function. The first layer has 128 3×3

kernels, the second layer 64 3×1 kernels and the third one 32 1×1 kernels. All layers have no bias parameters and a stride set to one. The output of the third CNN block is flattened and passed to a densely connected (DNN) layer, with 8 neurons and no activation, which represents the latent space. The decoder takes this as input to a dense layer with 32 nodes and ReLU activation, and reshapes it into a $2 \times 1 \times 16$ table. The following architecture mirrors the first half, or encoder, architecture with 3 CNN blocks with the same number of filters as in the encoder and with ReLU activation. Both are followed by an upsampling or reshaping layer, in order to mimic the result of a transposed convolutional layer. Finally, one convolutional layer with a single filter and no activation function is added. Its output is interpreted as the AE reconstructed input.

The model is implemented in TensorFlow (described in Section 4.4.1), and trained on the background dataset by minimizing the MSE loss with the Adam [146] optimizer. In order to aid the network learning process, the p_T are normalized to make the quantities $\mathcal{O}(1)$.

This Autoencoder, henceforth called teacher, was the subject of a Knowledge Distillation procedure to obtain a Fully Connected Neural Network much more suitable for hardware deployment thanks to its simplicity and potential low latency. In order to start the study of this kind of procedure from something easily approachable, an Offline Response-based distillation was chosen (see Section 5.2). In practice this means that the response in terms of Mean Squared error between input and output was retrieved from the teacher and used as truth associated to each input of the training set for the student models.

6.1.1 Hyperparameter Search

The teacher model was designed following the usual procedure when creating a Neural Network model: basically trying variations of the architecture and/or hyperparameters until a satisfying accuracy is reached on the validation set of the data available. However, it becomes less viable as a workflow if the number of configurations becomes too large or difficult to explore manually without a bias or a hint of the direction to take to get better results.

In general, the objective of a learning algorithm $\mathcal{M}$ is to find a function that minimizes some expected loss $\mathcal{L}(y; f)$ over training samples x with an associated ground truth y via the optimization of a set of parameters θ. Considering the different options and configurations of the different pieces that make up $\mathcal{M}$, the set of the chosen hyperparameters λ defines the actual model that is being trained. Thus, a way to practically choose λ as to minimize the error of $\mathcal{M}$ is needed or, in other words, a solution to the hyperparameter optimization problem [147]:

$$\lambda^{(*)} = \underset{\lambda \in \Lambda}{\operatorname{argmin}} \, \mathbb{E}[\mathcal{L}(y; \mathcal{M}(x))] \tag{6.1}$$

The critical step in hyper-parameter optimization is to choose the set of λs. The most widely used strategy is a combination of grid search and manual search. If Λ is a set indexed by K configuration variables, then grid search requires the selection of a set of values for each variable $(L^{(1)} \dots L^{(K)})$. This means that, by assembling every possible combination of values, the number of trials in a grid

search is $S = \prod_{k=1}^{K} |L^{(k)}|$ elements. This product over K sets makes grid search suffer from the so-called curse of dimensionality because the number of joint values grows exponentially with the number of hyperparameters.

Manual search is used to identify regions in Λ that are promising and to develop the intuition necessary to choose the sets $L^{(k)}$. Despite the doubts regarding the degree of reproducibility of such an approach, and the computational expense of grid searches, there are several reasons why these strategies prevail as the state of the art despite a lot of research into this kind of optimization [147]:

- Manual optimization gives researchers some degree of insight into the behaviour of the loss with different hyperparameters;
- There is no technical overhead or barrier to manual optimization;
- Grid search is simple to implement and parallelization is trivial;
- Grid search is reliable in low dimensional spaces (e.g. 1D, 2D).

As a way to fuse the benefits of grid and manual search, a random search can be performed instead, that is, independent draws from the configurations spaces as would be spanned by a regular grid to create the trial set of hyperparameters.

This excursus about hyperparameter search in general was done to introduce the first part of the work presented in this thesis. Indeed, this research was done not only to test knowledge distillation, but also to try to answer a question related to the actual procedure to follow when trying to obtain a small model implementable on an FPGA. In particular, when optimizing a NN for hardware inference, one must consider not only finding the optimal architecture but also identifying the best quantization, as defined in Section 5.1.

This raises an important question: Is there a significant difference between searching for the best candidate architecture with the quantization process in mind from the outset, versus approaching the task without factoring in quantization at first? More specifically, how do the results compare when one first identifies the optimal architecture and subsequently determines the most effective quantization strategy, as opposed to conducting a hyperparameter search that concurrently optimizes for both the architecture and the quantization parameters?

The strategy to trying to find an answer to this point is relatively straightforward. Two different approaches for hyperparameter search can be set up:

PhaseSearch Split the procedure in two phases where the first sees the optimization of the architecture and then the best candidates are optimized for quantization;

CoSearch Write a single optimization workflow where the configuration spaces of both architecture and quantization are explored at once.

For both cases, a random search was selected as the optimization method, with configuration spaces constrained by the strict requirements of hardware implementation. The number of layers and nodes per layer was kept as low as possible to reduce FPGA resource usage and achieve models with low latency and energy consumption. Simultaneously, the number of bits for quantization was limited by

model accuracy, hardware constraints, and further restricted to powers of two to efficiently utilize the memory available on the chosen device. This means that a very small set of hyperparameters was chosen:

- The model could have 3, 4, 5 or 6 layers;
- In each layer the network could have 8, 16, 32, 64 nodes;
- Every layer could be quantized differently among two different bitwidth, i.e. 8 total bits with 2 bits for the integer part of the number; or 16 total bits with an integer part 6 bits wide.

Nevertheless, with these possibilities the total number of different models possible, considering the quantization, would be 299520. In order to make the task feasible and focus on a smaller set of the configuration space, an upper and a lower ceiling were put in place of the total number of parameters (e.g. weights and biases) of respectively 6000 and 5000. This reduced the number of candidates resulting from the search to 1042. The thresholds, albeit they could seem too much restricting at first glance, were chosen after a coarser search with less options showed that the best candidates lived in this interval of number of parameters.

The hyperparameter search was written in Python using the KerasTuner [148] module, developed by the Keras team. It is a general-purpose hyperparameter tuning library. While it integrates seamlessly with Keras workflows, it is not restricted to them and can be used to tune scikit-learn [149] models or other machine learning frameworks. Given its usefulness, a very brief rundown on how it was used in this work for the CoSearch procedure will be presented, as an example for the reader to understand how to implement it for their research.

The first thing to do is to write a function which returns a compiled Keras model with `hp` as one of the arguments. This will be used by the function used to launch the actual search. Inside this build function the hyperparameters to optimize are defined with options about, for example, their type or minimum and maximum value. In Listing 6 a very simplified version of the code used in this work for such a function is shown.

This code begins by defining the input layer using `input_shape` to specify the structure of the input tensor. Following this, the input is flattened into a one-dimensional array that can be processed by fully connected layers.

Next, the code applies a quantized batch normalization layer, *QBatchNormalization*, which uses quantized bits for the normalization parameters, including beta, gamma, mean, and variance. This is achieved through the use of `quantized_bits` in the quantizer arguments. A hyperparameter, `config_indx`, is then defined, which selects a specific architecture configuration from a predefined list.

Another hyperparameter, `bits_indx`, is defined using `hp.Int("bits_indx", ...)`, which determines the specific quantization configuration from the list, effectively tuning the bit-width precision used for quantization for each layer.

Subsequently, a loop iterates through the selected model configuration, where the quantized fully connected layers are created using `QDense` and the number of units `nodes`. Both the weights and biases are quantized with bit precision parameters derived from the configuration applicable to the current candidate. After each

```python
def build_model(hp):
  # Input layer
  inputs = keras.Input(shape=input_shape)
  # Flattening the 19*3 table
  x = layers.Flatten()(inputs)
  # First BatchNormalization common to all models
  x = QBatchNormalization(beta_quantizer='quantized_bits(8,6)', \
     gamma_quantizer='quantized_bits(8,6)', mean_quantizer='quantized_bits(8,6)', \
     variance_quantizer='quantized_bits(8,6)')(x)
  # Defining the architecture hyperparameter as index in the list with all possible configurations
  config_index = hp.Int("config_indx", min_value=0, max_value=len(model_configurations)-1, step=1)
  bits_index = 0
  selected_bits_conf = []

  temp_idx = []

  # Defining the quantization hyperparameter as index in the list with all possible configurations
  bits_index = hp.Int("bits_indx", min_value=0, max_value=len(quant_configurations), step=1)

  # Number of hidden layers of the MLP is a hyperparameter.
  for i, nodes in enumerate(model_configurations[config_index]):
    # Number of nodes of each layer are different hyperparameters
    # Quantized Fully Connected layer with parametrized quantization and number of nodes
    x = QDense(units=nodes, \
            kernel_quantizer = quantized_bits(quant_configurations[bits_index][i][0], \
                              quant_configurations[bits_index][i][1],alpha=1), \
            bias_quantizer = quantized_bits(quant_configurations[bits_index][i][0], \
                              quant_configurations[bits_index][i][1],alpha=1), \
            kernel_initializer = 'he_normal', kernel_regularizer = L2(0.0001))(x)
    # Quantized ReLU activation function with parametrized quantization
    x = QActivation(activation = quantized_relu(quant_configurations[bits_index][i][0], \
                                quant_configurations[bits_index][i][1],negative_slope=0.25))(x)

  # The last layer contains 1 unit, which represents the learned loss value
  # It has a separate quantization hyperparameter
  final_quant_idx = hp.Int("final_quant_idx", min_value=0, max_value = 2, step=1)
  final_quant = final_quant_configurations[final_quant_idx]

  outputs = QDense(1,kernel_quantizer = quantized_bits(final_quant[0],final_quant[1],alpha=1), \
    bias_quantizer = quantized_bits(final_quant[0],final_quant[1],alpha=1), \
    kernel_initializer = keras.initializers.RandomUniform(minval=0, maxval=5,seed=1234))(x)

  outputs = QActivation(activation=quantized_relu(final_quant[0],final_quant[1], \
                                    negative_slope=0.25))(outputs)

  # Building the model with Keras
  hyper_student = keras.Model(inputs=inputs, outputs=outputs)

  # Compiling the model
  hyper_student.compile(
    optimizer=Adam(lr=3E-3, amsgrad=True),
    loss=distillation_loss
    )
  hyper_student.summary()
  return hyper_student
```

Listing 6: Simplified version of the code used in this work for the hyperparameter search of a quantized densely connected neural network.

dense layer, a quantized ReLU activation function, `QActivation(activation = quantized_relu(...))`, is applied, which also uses quantized bits and a negative slope of 0.25.

In addition to tuning the intermediate layers, the final layer of the network is

subject to its own quantization with a separate hyperparameter.

Finally, the Keras model is built using `keras.Model(..., ...)`, linking the input and output layers. The model is compiled with the Adam optimizer (`learning rate = 3E-3` with AMSGrad [150] enabled) and the loss function. The fully constructed and compiled model is returned at the end.

```
hypermodel = HyperStudent(x_train.shape, distillation_loss, param_threshold = (5000,6000))
tuner = keras_tuner.RandomSearch(
      hypermodel,
      max_trials = len(hypermodel.model_configurations),
      directory = 'output/hyper_tuning',
      )
tuner.search_space_summary()
# Using callbacks for early stopping and reducing the learning rate during training
callbacks = [
    EarlyStopping(monitor = 'val_loss', patience = 3, verbose = 1),
    ReduceLROnPlateau(monitor = 'val_loss', factor = 0.1, patience = 2, verbose = 1, min_lr = 1e-9)
    ]
tuner.search(
    x = x_train,
    y = y_train,
    epochs = 4,
    batch_size = 2048,
    validation_data = (x_val,y_val),
    callbacks = callbacks
    )

tuner.results_summary()

best_hps = tuner.get_best_hyperparameters()
```

Listing 7: Simplified version of the code to launch an hyperparameter search with KerasTuner for a model described in the HyperStudent class.

In Listing 7 the process to actually launch the search is shown. The HyperStudent class contains the build function described above with options for the loss and configurations space. The `tuner` is built with the constructor of the chosen algorithm, `RandomSearch` in this case, providing the model and other configurations, e.g. maximum number of trials and directory for the logs. The `search` method is very similar to the `fit()` method in basic Keras, and it basically start the training of all candidates. Finally the results can be retrieved as a summary or as a dictionary of the best hyperparameters.

CoSearch and PhaseSearch were launched using this kind of python script, with the only difference that, in the latter, there was an initial search with only the hyperparameters concerning the architecture, i.e. number of layers and number of nodes per each layer, followed by one for the quantization.

The results of the two search strategies for the 1042 student candidates are presented in Figure 6.4. These figures compare the distributions of MSEs between the student models and the teacher model, focusing on the different anomalies the students are expected to detect.

A quick glance at the box plot reveals that both workflows produce student models with similar levels of accuracy relative to the teacher model. However, the models generated through CoSearch tend to be distributed slightly closer to 0, indicating a marginally higher proportion of models with better performance

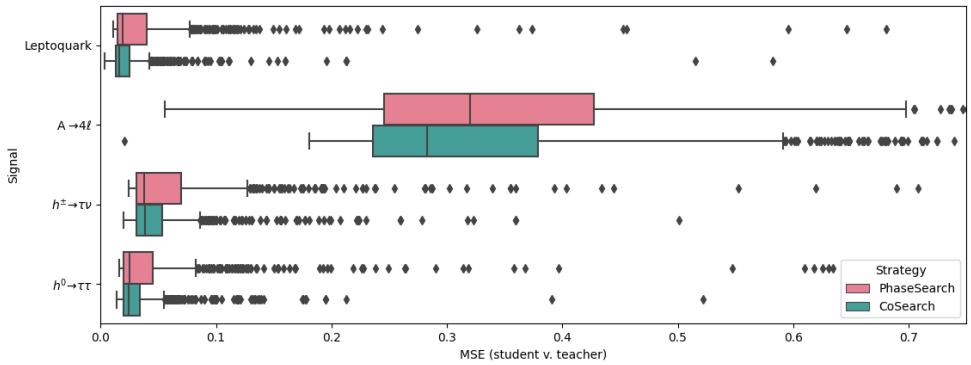


Figure 6.4: Distributions of MSE scores with respect to the teacher model of the students produced by the PhaseSearch (pink) and CoSearch (green) procedures for the 4 BSM signals under study.

compared to those from PhaseSearch. This trend is further supported by the medians and percentiles in Table 6.2, where CoSearch results consistently show smaller values than those from PhaseSearch.

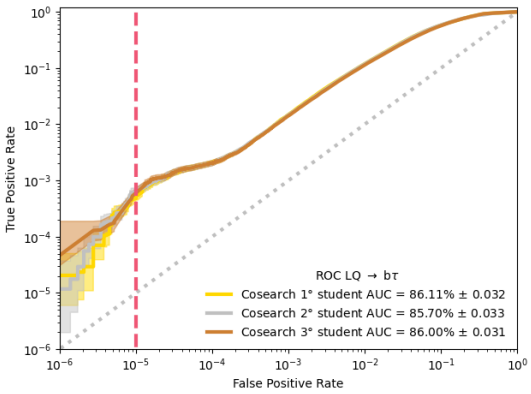
		PhaseSearch (MSE)	CoSearch (MSE)
Background	Median	0.003526	0.002124
	25%ile	0.001888	0.001625
	75%ile	0.010446	0.004469
Leptoquark	Median	0.019514	0.016468
	25%ile	0.014612	0.013378
	75%ile	0.042097	0.025582
$A \to 4\ell$	Median	0.387098	0.330177
	25%ile	0.282444	0.266956
	75%ile	0.573183	0.468822
$h^{\pm} \to \tau\nu$	Median	0.038740	0.039622
	25%ile	0.031283	0.031831
	75%ile	0.075944	0.055021
$h^{0} \to \tau\tau$	Median	0.026144	0.024405
	25%ile	0.020311	0.019853
	75%ile	0.048265	0.034714

Table 6.2: Robust estimators of MSE scores distributions produced by the PhaseSearch and CoSearch procedures for the 4 BSM signals under study.

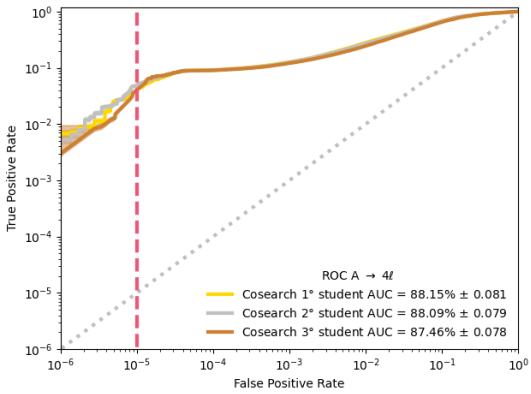
In Figures 6.5 and 6.6 the ROC curve for the three models with the highest AUC on average over all four signals are shown.

Comparing ROCs and AUCs

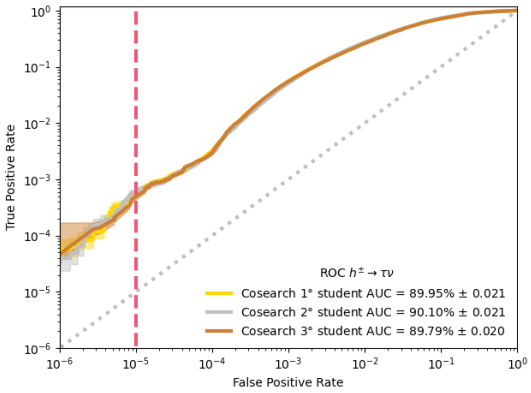
In order to compare and evaluate the best models, a robust way to assess the uncertainty of the metric used to create the rankings must be found. In this case the data at hand are the Receiver Operating Characteristic curve (made up of the True Positive rate vs the False Positive rate at different thresholds) and the



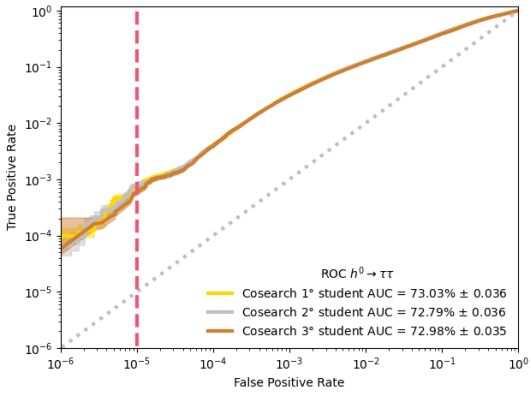
(a) Leptoquark $\to b\tau$



(b) Neutral scalar boson $A \to 4\ell$



(c) Charged boson $h^{\pm} \to \tau\nu$



(d) Scalar boson $h^0 \to \tau\tau$

Figure 6.5: ROC curves of the three best student produced by the CoSearch procedure according to the average over the four BSM signals under study of the AUC.

associated Area under the Curves and a way to compute an error to these quantities is the Clopper-Pearson interval [151] for the former and Delong's algorithm [152] for the latter.

The Clopper-Pearson (CP) interval is an exact method used to calculate the confidence interval (CI) for a binomial proportion (success rate), i.e. it is a way to bound the proportion within a specified confidence level, such as 95%. If X is the number of successes in n independent trials with success probability p, then the CP interval provides a confidence interval for p using the quantiles or inverse cumulative distribution function B of the Beta distribution:

$$\text{CI} = \left[B\left(\frac{\alpha}{2}; X, n - X + 1\right), B\left(1 - \frac{\alpha}{2}; X + 1, n - X\right) \right] \tag{6.2}$$

where α is the significance level (e.g. 0.05 for a 95% CI).

This links to the ROC because the TPRs and FPRs used to draw it are based on binary decisions which are inherently binomial proportions, i.e. the probability of success calculated from the outcome of a series of success-failure experiments. Thus, in each point of the ROC an upper and lower confidence interval can be

computed using the `scipy` [153] library in python, as shown in Listing 8, and used to draw a band around the curve like in the Figures 6.5, 6.6, 6.9, 6.13, and 6.15.

```python
from scipy.stats import beta
def clopper_pearson(total, passed, level, bUpper):
    alpha = (1.0 - level) / 2.0

    if bUpper:
        return np.where(
            passed == total,
            1.0,
            beta.ppf(1 - alpha, passed + 1, total - passed)
        )
    else:
        return np.where(
            passed == 0,
            0.0,
            beta.ppf(alpha, passed, total - passed + 1)
        )
```

Listing 8: Function used to compute the Clopper-Pearson confidence interval for each point of the ROCs shown in this thesis.

To explain Delong's algorithm, the starting point is the definition of the sample version of the AUC:

$$\hat{\theta} = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} \mathcal{H}(X_i - Y_j) \tag{6.3}$$

given $X_1 \ldots, X_m$ and $Y_1 \ldots, Y_n$ two independent and identically distributed samples drawn from two populations, and

$$\mathcal{H}(t) = \begin{cases} 1 & t > 0 \\ \frac{1}{2} & t = 0 \\ 0 & t < 0 \end{cases} \tag{6.4}$$

i.e. the Heaviside function. Let $\hat{\boldsymbol{\theta}} = \{\hat{\theta}^{(1)}, \ldots, \hat{\theta}^{(k)}\}$ be a vector of statistics representing the areas under the ROC curves derived from different readings $X_1^{(r)} \ldots, X_m^{(r)}$ and $Y_1^{(r)} \ldots, Y_n^{(r)}$ with $1 \leq r \leq k$ fo k different experiments. For the rth element of the vector, define the "structural component"

$$V_{10}(X_i^{(r)}) = \frac{1}{n} \sum_{j=1}^{n} \mathcal{H}(X_i^{(r)} - Y_j^{(r)}), \qquad i = 1, \ldots, m \tag{6.5}$$

and

$$V_{01}(Y_j^{(r)}) = \frac{1}{m} \sum_{j=1}^{m} \mathcal{H}(X_i^{(r)} - Y_j^{(r)}), \qquad j = 1, \ldots, n \tag{6.6}$$

Also define two matrices $S_{10} = [s_{10}^{(r,s)}]_{k \times k}$ and $S_{01} = [s_{01}^{(r,s)}]_{k \times k}$ such that

$$s_{10}^{(r,s)} = \frac{1}{m-1} \sum_{i=1}^{m} \left[V_{10}\left(X_i^{(r)}\right) - \hat{\theta}^{(r)}\right] \left[V_{10}\left(X_i^{(s)}\right) - \hat{\theta}^{(s)}\right] \tag{6.7}$$

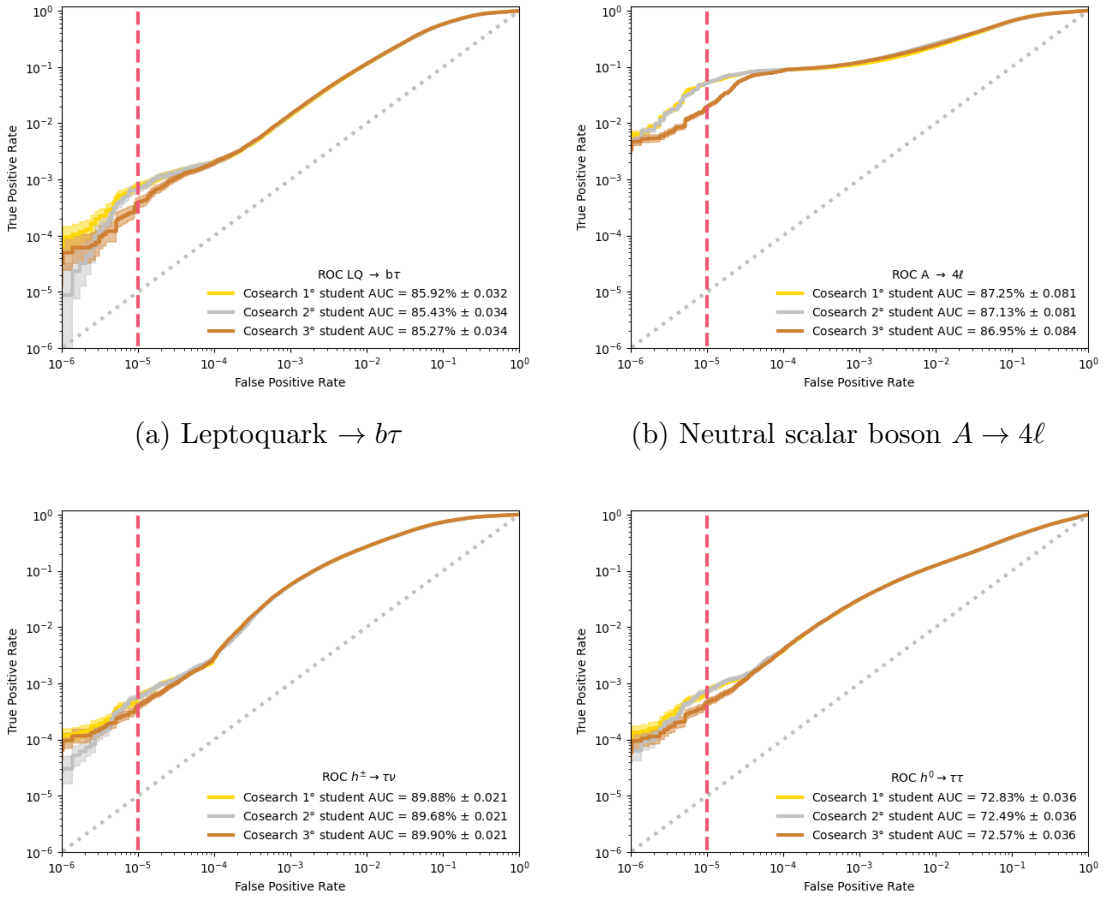
and

$$s_{01}^{(r,s)} = \frac{1}{n-1} \sum_{j=1}^{n} \left[V_{01}\left(Y_j^{(r)}\right) - \hat{\theta}^{(r)}\right] \left[V_{01}\left(Y_j^{(s)}\right) - \hat{\theta}^{(s)}\right] \tag{6.8}$$

Then, Delong proposed a variance-covariance matrix estimator for the vector $\hat{\boldsymbol{\theta}}$ as

$$\mathbf{S} = \frac{1}{m}\mathbf{S}_{10} + \frac{1}{n}\mathbf{S}_{01}. \tag{6.9}$$

When the vector $\hat{\boldsymbol{\theta}}$ contains only one element, that is $r = s = 1$ in Eq. 6.7 and Eq. 6.8, the covariance estimator reduces to a variance estimator $\mathbb{V}(\hat{\theta})$. In [152] a variation of the original algorithm is proposed to reduce the computing complexity from $\mathcal{O}[kmn + k^2(m+n)]$ to $\mathcal{O}[k(m+n)\log(m+n) + k^2(m+n)]$ and its implementation available on Github at [154] was the one used in this work.



(a) Leptoquark $\rightarrow b\tau$

(b) Neutral scalar boson $A \rightarrow 4\ell$

(c) Charged boson $h^{\pm} \rightarrow \tau\nu$

(d) Scalar boson $h^0 \rightarrow \tau\tau$

Figure 6.6: ROC curves of the three best student produced by the PhaseSearch procedure according to the average over the four BSM signals under study of the AUC.

Using these tools from the field of statistics, it is possible to say with more certainty, from looking at Figures 6.5 and 6.6, that CoSearch and PhaseSearch ultimately yield nearly equivalent optimal models. However Figure 6.4 suggests that optimizing both architecture and quantization simultaneously, as part of a unified hyperparameter search, increases the likelihood of producing a high-performing model with fewer iterations.

6.2 FPGA Implementation of a Fast Neural Network for Anomaly Detection

In the previous section the procedures to find the most accurate candidate for implementation on hardware were described. From Figures 6.5 and 6.6 it is possible to select the best performing student to start the workflow which will produce a running inference machine on an FPGA for the detection of Beyond the Standard Model signals against a theoretically explainable background within the boundaries of the Standard Model.

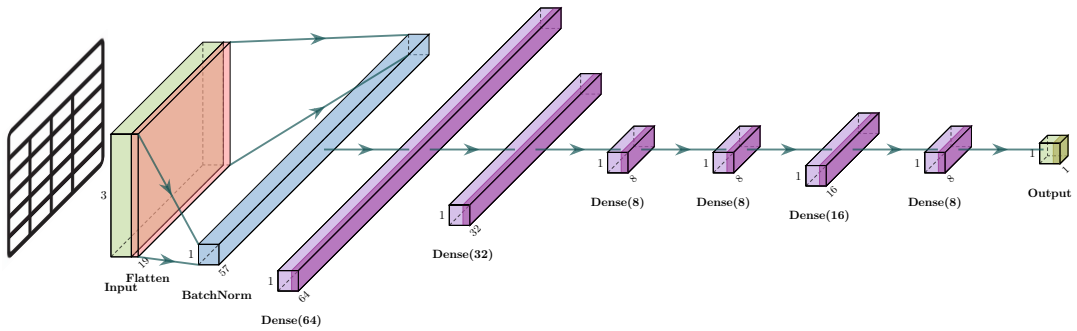
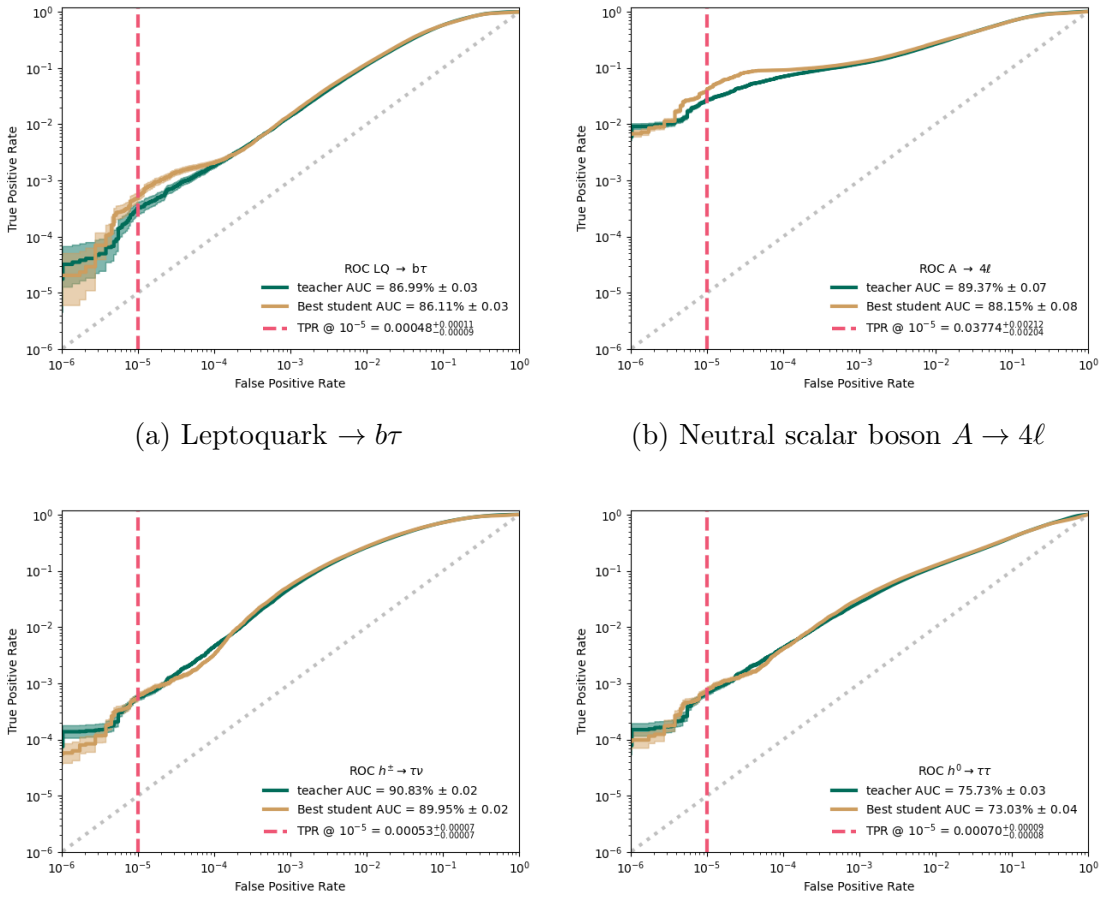


Figure 6.7: Network architecture for the Neural Network for AD to be implemented on FPGA.

The candidate was chosen from the CoSearch group and it is made up of an input layer, a flatten layer to reshape the data into a 1-D vector of numbers, a Batch Normalization layer and 6 hidden fully connected layers, in turn built with the following characteristics: All the hidden layers, and the output one, are

# Nodes	# Bits (Integer part)
64	16 (6)
32	16 (6)
8	16 (6)
8	16 (6)
16	16 (6)
8	16 (6)

followed by a Quantized ReLU activation function with the same quantization as the preceding layer and negative slope $\alpha = 0.25$. Their weights are also initialized with a He normal distribution [155], i.e. a truncated normal distribution centered on 0 with standard deviation $\sigma = \sqrt{\frac{2}{fan\_in}}$ where fan_in is the number of input units in the weight tensor. Finally, a regularizer was added to each layer from the L2 family, described in 4.2, with a very small factor of 0.0001. These settings were chosen as the ones which produced the best results after testing a number of different initializers, activation functions and regularizers. A visual representation of the network is in Figure 6.7.



(a) Leptoquark $\to b\tau$

(b) Neutral scalar boson $A \to 4\ell$

(c) Charged boson $h^{\pm} \to \tau\nu$

(d) Scalar boson $h^0 \to \tau\tau$

Figure 6.8: ROC curves of the best overall student according to the average over the four BSM signals under study of the AUC. It was produced by the CoSearch procedure.

It is worth noting that only 16-bit quantization is used, which is expected given its higher resolution compared to 8-bit quantization, allowing for a larger range of possible values for the model parameters. However, the second- and third-best models in CoSearch utilize one and two layers, respectively, with reduced bitwidth. Given the very similar performance among these three models, it suggests that achieving optimal performance with lower bitwidth than the maximum available is not only possible but plausible. For this thesis, however, the overall best model was selected, without considering bitwidth variations, in order to prioritize obtaining the most accurate fast AD algorithm.

6.2.1 Using HLS4ML to create a firmware

Having chosen the desired model to implement in hardware, the next step is to create the High Level Synthesis able to replicate its functionality in a so-called kernel which will be the computing unit in the overall firmware that will be implemented on the FPGA. This task is made easier for a person with a background in physics by the hls4ml library described in Section 5.3.1. Using this python package, the

process is simplified by leaving out the chore of writing the code in C++ related to the layers and activation functions of the neural network, specifically optimized to create a HLS kernel.

In order to start the conversion, the first step is to import the model using the chosen ML library. In this case it is QKeras due to the fact that the model under development was quantized directly during its training:

```
model = load_qmodel("coquantization_best_student.h5")
```

Listing 9: Importing a Neural Network model using QKeras.

Once the model is accessible a configuration dictionary has to be created:

```
config = hls4ml.utils.config_from_keras_model(model, granularity = 'name',default_reuse_factor=5)
```

Listing 10: How to create a default configuration for the HLS code produced by hls4ml.

The `config_from_keras_model` infers a lot of the parameters needed for the writing of the HLS code from the model itself. However, with the `granularity = 'name'` argument it is possible to overwrite the default configuration for each of the elements of the neural network. In this case, this was needed to impose the right number of bits for the output of the activation functions and the internal parameters of the BatchNormalization layer, which were manually optimized, as shown in Listing 11.

```
config['LayerName']['input_1']['Precision']['result'] = 'ap_fixed<16,9>'

config['LayerName']['q_activation']['Precision'] = 'ap_fixed<16,6,RND_CONV,SAT>'
config['LayerName']['q_activation_1']['Precision'] = 'ap_fixed<16,6,RND_CONV,SAT>'
config['LayerName']['q_activation_2']['Precision'] = 'ap_fixed<16,6,RND_CONV,SAT>'
config['LayerName']['q_activation_3']['Precision'] = 'ap_fixed<16,6,RND_CONV,SAT>'
config['LayerName']['q_activation_4']['Precision'] = 'ap_fixed<16,6,RND_CONV,SAT>'
config['LayerName']['q_activation_5']['Precision'] = 'ap_fixed<16,6,RND_CONV,SAT>'

config['LayerName']['q_batch_normalization']['Precision']['beta'] = 'ap_fixed<16,6,RND_CONV,SAT>'
config['LayerName']['q_batch_normalization']['Precision']['gamma'] = 'ap_fixed<16,6,RND_CONV,SAT>'
config['LayerName']['q_batch_normalization']['Precision']['mean'] = 'ap_fixed<16,6,RND_CONV,SAT>'
config['LayerName']['q_batch_normalization']['Precision']['variance'] = \
                                                    'ap_fixed<16,6,RND_CONV,SAT>'
config['LayerName']['q_batch_normalization']['Precision']['result'] = \
                                                    'ap_fixed<16,6,RND_CONV,SAT>'
config['LayerName']['q_batch_normalization']['Precision']['scale'] = 'ap_fixed<16,6,RND_CONV,SAT>'
config['LayerName']['q_batch_normalization']['Precision']['bias'] = 'ap_fixed<16,6,RND_CONV,SAT>'
```

Listing 11: In depth configuration of the quantization of activation functions and a batch normalization layer with the hls4ml library.

The terms `RND_CONV` and `SAT` refer to different quantization strategies. `RND_CONV` instructs the compiler to use convergent rounding, where values are rounded to the

nearest representable value. In cases where the value is exactly midway between two possible values ("ties"), the number is rounded to the nearest even value, ensuring that the least significant bit after rounding is set to zero. `SAT`, on the other hand, directs the compiler to handle overflow by capping values at the maximum representable value, and negative overflow by assigning the minimum representable value.

Tracing, which involves saving the output of individual elements within the model, was activated for all layers (Listing 12). This was done to monitor and detect any potential unwanted behaviors, such as overflow or insufficient precision, that could arise within the network. By capturing the intermediate outputs, we can more effectively diagnose and address issues related to numerical stability and precision limitations throughout the model's layers.

```
for layer in config['LayerName'].keys():
    config['LayerName'][layer]['Trace'] = True
```

Listing 12: Activating tracing for all layers of a Neural Network to be translated to HLS with hls4ml.

The code to be produced by hls4ml have to be suitable for the development using the Vitis platform, which is specific for the creation of firmware for accelerator cards like the AMD/Xilinx Alveo U50. To set up this feature another configuration has to be written with information on the FPGA part number, the backend of hls4ml to use where the right templates and script are described, and the interface desired. This can be seen in Listing 13.

```
cfg = hls4ml.converters.create_config(part='xcu50-fsvh2104-2-e', backend='VivadoAccelerator')

cfg['HLSConfig'] = config
cfg['AcceleratorConfig']['Driver']    = 'python'
cfg['AcceleratorConfig']['Board']     = 'alveo-u50'
cfg['AcceleratorConfig']['Interface'] = 'axi_stream'
cfg['IOType']= 'io_parallel'
cfg['KerasModel'] = model
cfg['OutputDir'] = 'HLS_Project'
```

Listing 13: Hls4ml configuration for an accelerator platform like the AMD/Xilinx Alveo U50.

Finally, the project can be compiled to create all the scripts and code in HDL ready to be used by the Vivado/Vitis suite for development of FPGA firmware:

```
hls_model = hls4ml.converters.keras_to_hls(cfg)

hls_model.compile()
```

Fixing a bug in hls4ml

While using hls4ml to produce the code for the Vitis platform, a bug was encountered in the translation of the activation layers of the model to HLS code. The function used in the student network is the quantized ReLU with a negative slope of 0.25. This means that its output values can be also negative but they follow a straight line in the third quadrant of the cartesian plane with an angular coefficient of 0.25 instead of 1 used with positive inputs.

```
def parse_qactivation_layer(keras_layer, input_names, input_shapes, data_reader):
# [...]
# Parsing all supported layers
# [...]
    else:
        layer['class_name'] = 'Activation'
        layer['activation'] = activation_config['class_name'].replace('quantized_', '')
# [...]
```

Listing 14: Snippet of the old version of the script in the hls4ml library parsing the activation layers of models built using QKeras.

However, in the code used to parse the quantized activation layers, this kind of function was not listed and this caused the use of the default version of the non-quantized version, as seen in Listing 14, where the code simply suppressed the `'quantized_'` part of the name, making it a regular ReLU, instead of a LeakyReLU which is described by a different class of objects in TensorFlow.

```
  elif activation_config['class_name'] == 'quantized_relu' and \
                               activation_config['config']['negative_slope'] != 0:
        layer['class_name'] = 'LeakyReLU'
        layer['activation'] = activation_config['class_name'].replace('quantized_', 'leaky_')
        layer['activ_param'] = activation_config['config']['negative_slope']
```

Listing 15: `if` case added in the script in the hls4ml library parsing the activation layers of models built using QKeras to include the correct translation of quantized LeakyReLUs.

This issue was resolved by adding a new case to the `if-else` statement, as shown in 15. The added code triggers when the layer's name is `quantized_relu` and the negative slope is non-zero, effectively identifying it as a LeakyReLU. At this point, the `quantized` part of the name is replaced with `leaky`. Finally, the configuration for the negative slope is stored in the dictionary that holds all the layer information to be parsed.

To finalize the fix, an additional adjustment was made in the script responsible for retrieving the initial configuration from the imported model for conversion. Initially, the output of the `quantized_relu` function was assumed to be unsigned. However, this assumption does not hold if the ReLU has a negative slope. Therefore, support for signed output was added, as shown in Listing 16.

```
if quantizer['class_name'] in ('quantized_relu', 'quantized_relu_po2'):
            if quantizer['config']['negative_slope'] != 0.0:
                signed = True
            else:
                signed = False
                integer -= 1
```

Listing 16: Fix in the script of the hls4ml library to retrieve the right configuration of the output of the `quantized_relu` activation function. Originally the case of a signed output was not allowed, making it impossible to produce a negative value, which are instead possible for LeakyReLUs.

Together with the necessary suite of tests, these fixes were part of a Pull Request on the official hls4ml repository [156] and the merge was approved to make it part of the published code base [157].

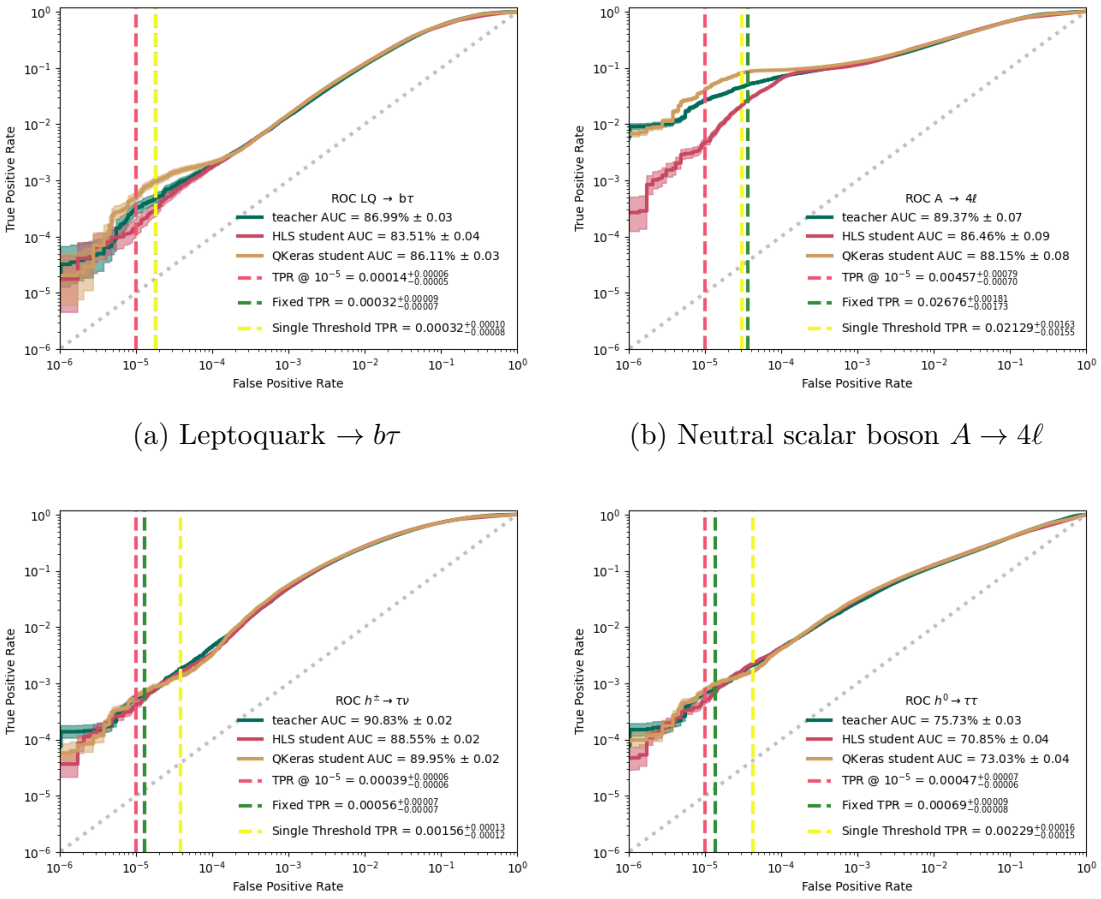
6.2.2 Accuracy of BSM signal detection

After creating the HLS code to implement the functionalities of the quantized Neural Network at hand, the accuracy in detecting the anomalies, here represented by the 4 BSM signals under study, must be checked again before the actual hardware implementation.

In the previous sections, model performance was evaluated using TPRs at a fixed FPR as a benchmark, focusing on low FPR levels. It is important to note that the purity of signal detection is tied to a specific threshold in the neural network output, which defines the boundary between background and signal. The aim of this study is to explore the potential deployment of such an algorithm at the Level-1 trigger of CMS, enabling the selection of signals without reliance on theoretical assumptions. This requires selecting a single threshold that ensures high purity in the positive dataset, regardless of the signals in the input sample. To determine the optimal cut value, performance was analyzed across the four BSM signals, with the threshold corresponding to the lowest FPR where the relative difference between the TPRs of the student and teacher models was less than 1%. This resulted in four different thresholds, each producing slightly higher TPRs compared to an FPR of 10^{-5}. The smallest of these thresholds was ultimately chosen as the final cut point to increase the likelihood of capturing rare events, given the nature of the signals targeted by this approach.

All of this is shown in Figure 6.9 which shows the ROC curves comparing the best student model implemented using QKeras (brown) and HLS (red) against the teacher model (green). Three key TPRs are highlighted at different FPRs. The first TPR corresponds to an FPR of 10^{-5}; the second TPR occurs where the student model's performance is closest to that of the teacher, illustrating the lowers FPR where the student approximates the teacher's accuracy in a satisfying matter. Lastly, a third TPR is shown at a common threshold applied across four signals, chosen based on the highest loss value where one of the signals achieves a TPR most similar to the teacher model.

In Table 6.3, a compendium of the numerical values obtained for the test signals



(a) Leptoquark $\rightarrow b\tau$

(b) Neutral scalar boson $A \rightarrow 4\ell$

(c) Charged boson $h^{\pm} \rightarrow \tau\nu$

(d) Scalar boson $h^0 \rightarrow \tau\tau$

Figure 6.9: ROC curves comparing the best student model using QKeras (brown) and the HLS implementation (red) against the teacher model (green). Three TPRs are highlighted for different false positive rates FPRs: one at 10^{-5}, another where the student's performance is closest to the teacher's, and a third at a common threshold across the four signals, selected based on the highest loss value where one signal's TPR most closely matches the teacher's.

is listed, together with the significance. This metric, described in [158], is used to evaluate the sensitivity of the experiment by analyzing the ratio of the signal strength s to the background fluctuation $\sqrt{b}$.

In particle physics experiments one often searches for processes that have been predicted but not yet seen. The statistical significance of an observed signal can be quantified by means of a p-value or its equivalent Gaussian significance. It is useful to characterize the sensitivity of an experiment by reporting the expected (e.g., mean or median) significance that one would obtain for a variety of signal hypotheses. Finding both the significance for a specific data set and the expected significance can involve Monte Carlo calculations that are computationally expensive.

In [158] an approximate method it is proposed by which one can obtain both the significance for given data as well as the full sampling distribution of the

Process	True Positive Rate @ 10^{-5} FPR	True Positive Rate Single threshold	False Positive Rate Single threshold	Significance
LQ $\to b\tau$	$0.00014^{+0.00006}_{-0.00005}$	$0.00032^{+0.00010}_{-0.00008}$	1.83×10^{-5}	0.061173
$A \to 4\ell$	$0.00457^{+0.00079}_{-0.00070}$	$0.02129^{+0.00163}_{-0.00155}$	3.00×10^{-5}	0.700552
$h^{\pm} \to \tau\nu$	$0.00039^{+0.00006}_{-0.00006}$	$0.00156^{+0.00013}_{-0.00012}$	3.81×10^{-5}	0.694085
$h^0 \to \tau\tau$	$0.00047^{+0.00007}_{-0.00006}$	$0.00229^{+0.00016}_{-0.00015}$	4.26×10^{-5}	0.925252

Table 6.3: Results from the HLS representation of the student model to be implemented on FPGA for the 4 BSM signals under study. Three TPRs are highlighted for different FPRs: one at 10^{-5}, another where the student's performance is closest to the teacher's, and a third at a common threshold across the four signals, selected based on the highest loss value where one signal's TPR most closely matches the teacher's. Finally, the significance described in [158] is reported to assess the statistical confidence in the signal selection by the model.

significance under the hypothesis of different signal models, all without recourse to Monte Carlo. In this way one can find, for example, the median significance and also a measure of how much one would expect this to vary as a result of statistical fluctuations in the data. A useful element of the method involves estimation of the median significance by replacing the ensemble of simulated data sets by a single representative one, referred to here as the "Asimov" data set, defined as a data set such that when one uses it to evaluate the estimators for all parameters describing a data set under study, one obtains the true parameter values.

The case under study in this thesis falls under the umbrella of counting experiments, however it can be considered as a special case where the background b is much larger than the signal s. If b is regarded as known, the observed number of events n follows a Poisson distribution with a mean of $s + b$. The likelihood function for observing n events, given s and b, is:

$$L(\mu) = \frac{(\mu s + b)^n}{n!} e^{-(\mu s + b)} \tag{6.10}$$

To assess the presence of a signal, the test statistics q_0 is used. It can quantify the lack of agreement with having only background events in the data and it can be written as:

$$q_0 = \begin{cases} -2 \ln \frac{L(0)}{L(\tilde{\mu})} & \tilde{\mu} \geq 0 \\ 0 & \tilde{\mu} < 0 \end{cases} \tag{6.11}$$

where $\tilde{\mu} = n - b$. For sufficiently large b the following asymptotic formula can be used for the significance

$$Z_0 = \sqrt{q_0} = \begin{cases} \sqrt{2(n \ln \frac{n}{b} + b - n)} & \tilde{\mu} \geq 0 \\ 0 & \tilde{\mu} < 0 \end{cases} \tag{6.12}$$

To approximate the median significance assuming the hypothesis that there is a signal ($\mu = 1$), n is replaced by the value $s + b$, obtained by considering an Asimov data set:

$$\text{med}[Z_0|1] = \sqrt{q_{0,\text{A}}} = \sqrt{2((s+b)\ln(1 + s/b) - s)} \tag{6.13}$$

and, expanding logarithm in s/b one finds

$$\text{med}[Z_0|1] = \frac{s}{\sqrt{b}}(1 + \mathcal{O}(s/b)) \tag{6.14}$$

Although $\text{med}[Z_0|1] \approx s/\sqrt{b}$ has been widely used for cases where $s+b$ is large, one sees here that this final approximation is strictly valid only for $s \ll b$.

Based on the significance values presented in Table 6.3, it could be concluded that the current model is not yet capable of detecting BSM signals with sufficient confidence against the background while keeping the number of false positives as low as possible. However, the primary focus of this work is on demonstrating the feasibility of deploying such complex architectures on hardware to achieve extremely low latencies. The main objective of this section is to highlight that it is possible to distill a network and obtain a similar performance between the distilled, quantized, and hardware-translated network and the original teacher model. These results underscore the need for further research into more accurate and sophisticated models for this task. Importantly, knowledge distillation has proven effective in enabling the deployment of such advanced models on FPGAs. A preliminary exploration of a novel architecture is provided in Section 6.4, marking an initial step in this direction. Furthermore a Master's thesis was supervised on the development of more complex Variational Autoencoders to perform Anomaly Detection for signal hunting, however on a more simple background and signal dataset [159]. This work was also presented at the International Symposium on Grids & Clouds (ISGC) 2023 in Taipei [160].

6.2.3 Towards synthesis and implementation

Now that the model written in HLS code is ready, the next step is to create a firmware to load onto the FPGA. Being a proof of concept, and due to the hardware availability, the target platform for the workflow in this thesis is an Alveo U50, manufactured by AMD/Xilinx [161]. This means that the firmware will not be a standalone product, running on a FPGA without the need of a host computer, but it will take the image of an accelerated application which will be launched on the card containing the FPGA by the host's CPU via a PCI Express connection, as schematically shown in Figure 6.10, where HBM stands for High Bandwidth Memory and it is the on chip memory of this kind of accelerator cards.

To create this kind of application on an AMD device, the workflow sees the use of the Vitis Design Suite [162]. The process starts with the creation of a project where the target platform is chosen, as seen in Figure 6.11. This will give the information to Vitis on how to build the part of the firmware which is independent from the custom application described in HLS, instead it contains the needed circuitry to sustain the kernel containing the neural network and perform all the communication from and to the FPGA.

At this stage, the source files generated by the hls4ml library must be imported. These include header and implementation files that handle the various data types and layers supported by the library. Custom files, on the other hand, define the weights, biases, and layers, with the appropriate data types specified in the configuration (Listing 17). Additionally, they include the necessary `#include` directives

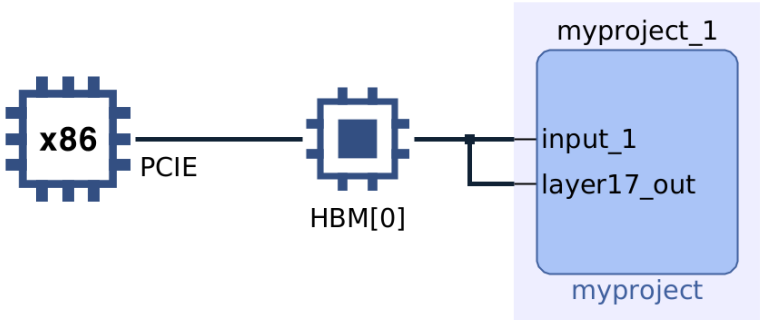


Figure 6.10: System schematics of an accelerated function running on a FPGA (on the right) which is launched by a host PC with a traditional processor via a PCIe connection, and interacting with the on-device memory (HBM).

```
// Insert fixed sizes
#define N_INPUT_1_1 19
#define N_INPUT_2_1 3
#define N_INPUT_3_1 1
// [...]
#define N_LAYER_14 8
#define N_LAYER_14 8
#define N_LAYER_16 1
#define N_LAYER_16 1

// Insert layer-precision
typedef ap_fixed<16,9> input_t;
typedef ap_fixed<16,6,AP_RND_CONV,AP_SAT> layer3_t;
typedef ap_fixed<16,7> weight4_t;
typedef ap_fixed<16,7> bias4_t;
// [...]

#endif
```

Listing 17: Configuration file defining the data types, layers, weights, and biases for the neural network architecture.

for files containing the neural network's parameters and the configuration of activation functions and layers, which are structured as `structs` (Listing 18). Lastly, the main script contains the core function that will be invoked to perform inference on the inputs provided to the FPGA (Listing 19). All of this code in C++ can be modified if needed, giving great flexibility in the use of hls4ml, in the fact that it could be also used as an optimal starting point even if a desired feature is not yet implemented.

In Listing 19 there is the actual instantiation of the neural network. All pieces of the model are described as template functions which are able to deal with the different types of inputs and outputs accordingly. Their arguments are the actual pointers with the data coming in and out of the layer or activation function, together with the necessary weights and/or other parameters. In this way the entire functionality of the model is encapsulated in a single `void` function, with all the inner workings neatly split in functional units, which can then be translated in different areas of the chip and run independently, only driven by the data available.

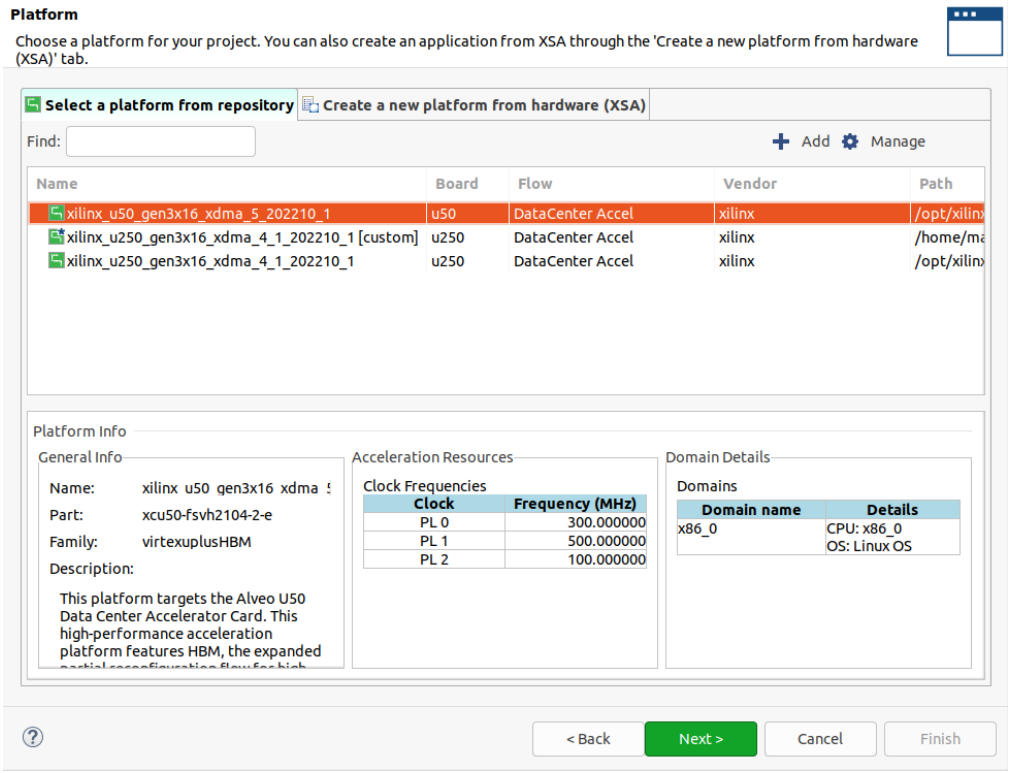


Figure 6.11: Snapshot of one of the steps to create a project with the Vitis Design Suite. Here the target platform is chosen in order to create a specific accelerated application to run on the FPGA.

At the beginning of the code for the kernel function there is also the description of its inputs and outputs using HLS' `pragmas`. This tells the Vitis' compiler to create and reserve the correct ports for the entire IP core. In this case these arrays are also reshaped and partitioned in order to make them more optimized to be stored in the memory of the FPGA. Focusing on the type of interface, the Advanced eXtensible Interface (AXI4) [163] was chosen, as it is a standard interface which allows the sending and receiving of structured data in batches and it is easily managed by the Vitis compiler to make a ready to go firmware to use with OpenCL or Python using the PYNQ library (see Section 3.3).

The hls4ml tool also generates an additional file that creates a buffer between the main function and the rest of the firmware. However, due to compatibility issues with the platform, it was decided to bypass this buffer and directly use the kernel function. As a result, minor modifications were made to the generated code to make the function in Listing 19 standalone. This function was then designated as the main function for the entire project.

Latency (ns)	BRAM	BRAM(%)	DSP	DSP(%)	FF	FF (%)	LUT	LUT (%)
790	2	~ 0	1002	16	62969	3	207144	23

Table 6.4: Latency and hardware footprint of the kernel containing the neural network in terms of the memory units, computing units and registers used.

With the kernel ready the synthesis, implementation and bitstream generation

```
// Layer includes
#include "nnet_utils/nnet_activation.h"
#include "nnet_utils/nnet_batchnorm.h"
#include "nnet_utils/nnet_dense.h"
// [...]

// Weights and bias values includes
#include "weights/s3.h"
#include "weights/b3.h"
// [...]

// Insert layer-config

// QDense
struct config4 : nnet::dense_config {
    static const unsigned n_in = 57;
    static const unsigned n_out = 64;
    static const unsigned io_type = nnet::io_parallel;
    static const unsigned reuse_factor = 5;
    // [...]
};

// QActivation
struct leaky_relu_config5 : nnet::activ_config {
    static const unsigned n_in = 64;
    static const unsigned table_size = 1024;
    static const unsigned io_type = nnet::io_parallel;
    static const unsigned reuse_factor = 5;
    typedef q_activation_table_t table_t;
};

// [...]
```

Listing 18: Header file containing `#include` directives for the neural network's parameter values, activation functions, and layer configurations structured as `structs`.

workflow was launched. This then produced the actual file containing the entire firmware to be used to program the FPGA to perform the task of Anomaly Detection.

At this stage, some insights into the hardware footprint of the neural network implementation are available. Table 6.4 presents the usage of BRAMs, DSPs, Flip-Flops, and Lookup Tables, along with the corresponding percentages of total resources available on the platform. The resource usage is minimal, demonstrating that the distillation process was highly effective in producing a compact implementation of a complex algorithm. This reduction in complexity is achieved with a negligible impact on accuracy, underscoring the success of the approach. It is important to note that, while the Alveo U50 lacks the characteristics of standalone, on-board FPGAs typically used for Level-1 triggering, it is one of the smallest cards in the accelerator space. Achieving such low resource consumption on this platform highlights the efficiency of this implementation, making it a noteworthy success in terms of both performance and resource management.

Finally the power consumption and thermal information of the card running the produced firmware is shown in Table 6.5. All values are well within the range of ideal operation. This reinforce the idea that the distillation was very successful and it could be used to reduce the complexity of model performing difficult tasks

```
#include "myproject.h"
#include "parameters.h"

void myproject(
    input_t input_1[N_INPUT_1_1*N_INPUT_2_1*N_INPUT_3_1],
    result_t layer17_out[N_LAYER_16]
) {
    // Insert IO and some reshaping and partition to increase performance
    #pragma HLS ARRAY_RESHAPE variable=input_1 complete dim=0
    #pragma HLS ARRAY_PARTITION variable=layer17_out complete dim=0
    #pragma HLS INTERFACE m_axi port=input_1,layer17_out
    #pragma HLS PIPELINE

#ifndef __SYNTHESIS__
    static bool loaded_weights = false;
    if (!loaded_weights) {
        // Load weights
        nnet::load_weights_from_txt<q_batch_normalization_scale_t, 57>(s3, "s3.txt");
        nnet::load_weights_from_txt<q_batch_normalization_bias_t, 57>(b3, "b3.txt");
        nnet::load_weights_from_txt<weight4_t, 3648>(w4, "w4.txt");
        nnet::load_weights_from_txt<bias4_t, 64>(b4, "b4.txt");
        // [...]
    }
#endif
    // Insert layers for network instantiation
    auto& layer2_out = input_1;
    layer3_t layer3_out[N_SIZE_0_2];
    #pragma HLS ARRAY_PARTITION variable=layer3_out complete dim=0
    nnet::normalize<input_t, layer3_t, config3>(layer2_out, layer3_out, s3, b3); // qbatchnorm

    layer4_t layer4_out[N_LAYER_4];
    #pragma HLS ARRAY_PARTITION variable=layer4_out complete dim=0
    nnet::dense<layer3_t, layer4_t, config4>(layer3_out, layer4_out, w4, b4); // qdense

    layer5_t layer5_out[N_LAYER_4];
    #pragma HLS ARRAY_PARTITION variable=layer5_out complete dim=0
    nnet::leaky_relu<layer4_t, layer5_t, leaky_relu_config5>(layer4_out, 0.25, layer5_out);
    // [...]
    layer16_t layer16_out[N_LAYER_16];
    #pragma HLS ARRAY_PARTITION variable=layer16_out complete dim=0
    nnet::dense<layer15_t, layer16_t, config16>(layer15_out, layer16_out, w16, b16); // dense_fin

    nnet::leaky_relu<layer16_t, result_t, leaky_relu_config17>(layer16_out, 0.25, layer17_out);
}
```

Listing 19: Main script implementing the function for performing inference on the inputs provided to the FPGA.

but at the same time the constraints in terms of number of layers and nodes in the fully connected neural network could be lifted a bit.

6.2.4 Running on FPGA

After exporting the bitstream from Vitis, contained in a `xclbin` file, it is the moment to actually run the algorithm on hardware. As briefly explained in Section 3.3, there are two main ways to launch an accelerated application on a FPGA card: one uses OpenCL code to write a programme running on the host machine which will communicate with the PCIe peripheral using the Xilinx Runtime Drivers (xrt), the other takes advantage of the PYNQ library which allows the programming and use of FPGAs with a completely Python API. There is obviously a difference in tim-

Total On-Chip Power	13.178 W
FPGA Power	12.803 W
HBM Power	0.375 W
Design Power Budget	60 W
Power Budget margin	46.822 W
Junction Temperature	64.9 °C
Thermal Margin	35.1 °C (41.3 W)
Ambient Temperature	55.0 °C

Table 6.5: Power footprint of the kernel containing the neural network. Other than the power needed to run the model, the temperature of the chip is shown and the margins left from the maximum possible for correct functioning.

ing performance between these two approaches (also studied in Appendix A), due to the difference in the programming languages and different layers of abstraction needed to run the two applications [164]. However, the level of expertise needed to create an application using OpenCL and to write a simple Python script is also very different, making the second alternative a good way to open up the world of FPGAs and accelerated applications to new people and of different scientific backgrounds.

In this section a brief explanation on how to write the C++ application to use the FPGA will be given, with short snippets of simplified code coming from the actual one used to run the AD algorithm on FPGA. Obviously the first step is to import the relevant libraries, other than the general ones used in C++, for example, for IO operations and manipulation, advanced mathematical operations, and data handling. In this case the OpenCL includes could be the `cl_ext_xilinx.h` header for dealing with data streams, or `xcl2.hpp`, a library of Xilinx-provided helper functions to wraparound some of the required initialization functions.

The core component of the application is the `main` function, as is standard in all C++ or C programs, which primarily operates on the host machine's CPU. This allows it to be handled like a typical application, especially in terms of input arguments and their management. In this case, the application is designed to accept the bitstream file name, the number of inputs, and an optional debug mode flag as command-line arguments when the program is launched (Listing 20).

Once the input arguments are read, and the data is loaded using the desired way of reading file and put in a data container, the OpenCL code begins. Integral part of writing code with this standard is the use of objects which represent the different parts needed to run the application on the device. For example in Listing 21, objects are instantiated representing:

- Device (the FPGA);
- Context (an ensemble of devices, in this case it will contain only one device);
- Queue of commands that will be given to the card;
- Programming operation;
- Kernel containing the function described in the previous Section.

```
int main(int argc, char **argv) {

  // Check input arguments
  if (argc < 2 || argc > 4) {
    std::cout << "Usage: " << argv[0] << " <XCLBIN File> <#samples(optional)> <debug(optional)>"
                                                                                << std::endl;
    // [...]
  }
  // Read FPGA binary file
  auto binaryFile = argv[1];
  unsigned int num_samples = 1;
  // Check if the user defined the # of samples
  if (argc >= 3){
    user_size = true;
    num_samples = std::stoi(argv[2]);
    // [...]
  }
```

Listing 20: Argument handling in the main function of an example of OpenCL application to interact with a FPGA to run code on the device.

It is also useful to use an error handler to keep track of any error that could come up in each step of the application.

```
  // OpenCL Host Code Begins.
  // OpenCL objects
  cl::Device device;
  cl::Context context;
  cl::CommandQueue q;
  cl::Program program;
  cl::Kernel myproject;

  cl_int err;
```

Listing 21: Instantiation of objects in OpenCL representing the different parts needed to run the application on a device.

Programming the FPGA is simplified thanks to utility APIs provided by Xilinx/AMD. For instance, the `xcl::get_xil_devices` function retrieves a list of devices connected to a Xilinx platform, while `xcl::read_binary_file` loads a binary file and returns a pointer to its buffer. With these utilities, FPGA programming essentially involves calling the constructor of the `program` object, passing in the `context`, the list of devices, and the output from the file-reading function. This is shown in Listing 22. Finally the kernel can be initialized with the `program` object and it is ready to be queued as a command to the device.

As anticipated before, to perform tasks on the FPGA, a queue is needed where the different kernels inside a single firmware could be called in succession. In this case only one kernel is present, containing the neural network, and so the process is relatively simpler, as shown in Listing 23. Firstly, the input and output have to be represented by pointers of the correct length. In this case the `data` method of `vectors` was used to return it from the data containers. Then the input has to be loaded onto the input buffer of the device, while the output buffer must be made ready to accept the output of the kernel and store it at a pointer. Now these

```
auto devices = xcl::get_xil_devices();
auto fileBuf = xcl::read_binary_file(binaryFile);
cl::Program::Binaries bins{{fileBuf.data(), fileBuf.size()}};

for (unsigned int i = 0; i < devices.size(); i++) {
  device = devices[i];
  cl::Program program(context, {device}, bins, NULL, &err);

  OCL_CHECK(err, myproject  = cl::Kernel(program, "myproject" , &err));
  }
```

Listing 22: Programming a FPGA device using OpenCL code.

buffers can be set as the two arguments of the kernel. This is because the function inside it was written to accept two arguments, namely the data going in the neural network and its result. After queuing the copy of the input data on the device, the kernel call can be put on the list and, after a waiting function to be sure to give enough time for the computation, the output can also be told to be copied back to the host machine's memory.

```
for (unsigned int k = 0; k < num_samples; ++k)
{
  // [...]
  // Device-to-host communication
  OCL_CHECK(err, cl::Buffer buffer_input(context,
                                    CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
                                    size_bytes_in, input_pointer, &err));
  OCL_CHECK(err, cl::Buffer buffer_output(context,
                                    CL_MEM_USE_HOST_PTR | CL_MEM_WRITE_ONLY,
                                    size_bytes_out, output_pointer, &err));
  // Setting Kernel Arguments
  OCL_CHECK(err, err = myproject.setArg(0, buffer_input));
  OCL_CHECK(err, err = myproject.setArg(1, buffer_output));
  // Copy input data to device global memory
  OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_input},
                           0 /* 0 means from host*/,NULL,&eventinp));
  OCL_CHECK(err, err = q.finish());
  // Launching the Kernel
  OCL_CHECK(err, err = q.enqueueTask(myproject,NULL,&eventker));
  // wait for the kernel to finish their operations
  OCL_CHECK(err, err = q.finish());
  // Copy Result from Device Global Memory to Host Local Memory
  OCL_CHECK(err, err = q.enqueueMigrateMemObjects({buffer_output},
                       CL_MIGRATE_MEM_OBJECT_HOST,NULL,&eventout));
  OCL_CHECK(err, err = q.finish());
```

Listing 23: Code needed to run an application, or kernel, on a FPGA device using OpenCL.

This is a schematic explanation of how to write an OpenCL code to use an accelerator card containing a FPGA. In Listing 23 `events` handlers were used to extract information about the three different phases of the runtime, namely input injection, kernel execution, and output extraction. These were used as shown in Listing 24 to have the timings needed to perform inference on this device and compare it to a GPU, the standard architecture used nowadays when talking about machine learning.

```
    eventker.getProfilingInfo(CL_PROFILING_COMMAND_START, &time_start);
    eventker.getProfilingInfo(CL_PROFILING_COMMAND_END, &time_end);

    nanoSecondsker = time_end-time_start;
```

Listing 24: Functions used to extract the wall time of the three phase making up the launch of an application on a device: input injection, kernel execution, and output extraction.

The results obtained for the latency of the inferences are shown in Figure 6.12. For comparison the latency of the same kernel deployed using PYNQ is shown as well (to have an idea on how it was done see [165], the script is very simple and does not need an in depth explanation.)

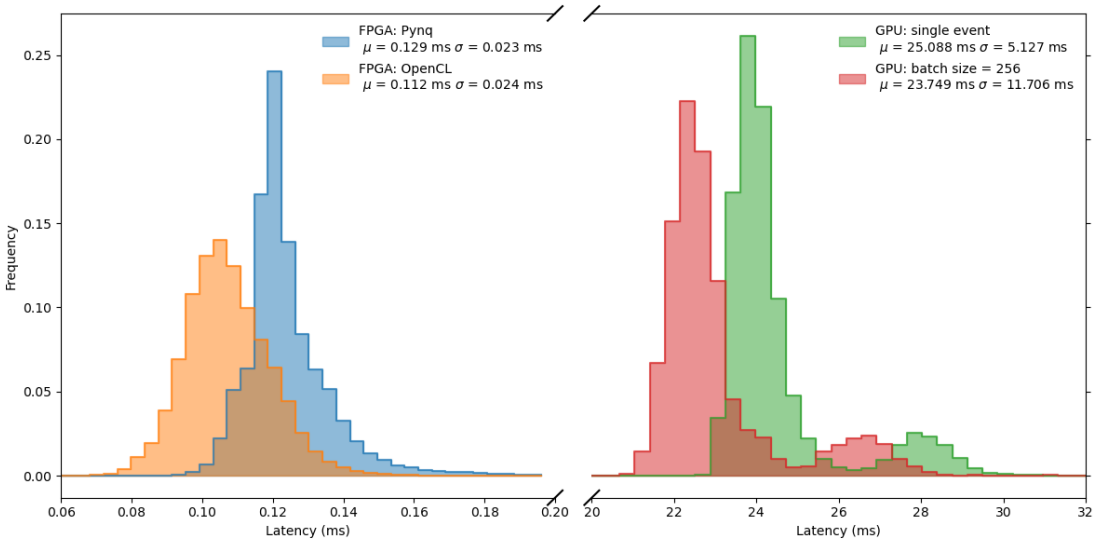


Figure 6.12: Latency for Anomaly Detection inference using a FPGA with an OpenCL application (orange) and a Python script with PYNQ (blue), compared to the performance of an Nvidia RTX 2060 GPU with both single event and batched inputs.

The plot demonstrates that using OpenCL to write an application for interacting with the FPGA results in a shorter inference time, averaging 0.112 ms compared to 0.129 ms with the Python approach. However, the most significant outcome is the comparison with inference times using an Nvidia RTX 2060 GPU, which shows an increase of two orders of magnitude for both single-event inference, like the FPGA, and a batch size of 256. Since the average inference time for the larger batch size is shorter than for single events, it can be inferred that increasing the batch size would further reduce total inference time when using a GPU. However, the GPU's higher latency and unpredictability, evidenced by a small cluster of events with longer processing times (the cause of which remains unclear), make it unsuitable for real-time analysis at the trigger level. Additionally, the use of custom buffer mechanisms makes it more feasible to integrate a scouting system

with this AD algorithm into the existing workflow for exploring new physics at CMS and other large LHC experiments. This suggests that the ideal kernel would need to be highly efficient for single-event processing, while the management of incoming data buffers would require tailored research to meet the accelerator and data acquisition system timing constraints.

6.3 A test with Online Distillation

The anomaly detection model implemented in the last section was a product of an offline response-based knowledge distillation. This is the simplest and fastest to implement type of KD, where it is simply a matter of taking the output of a pre-trained teacher student and use it as the "truth" to be learned by the student model.

Another approach which has shown some degrees of success [111] is the online distillation, i.e. perform the training of the small and relatively simpler neural network at the same time of the more complex and bigger model. This means that the training steps required to implement this kind of procedure are not the standard and readily available, but the actual code running when the models are trained has to be written by hand. Thankfully, TensorFlow gives all the tools to make this relatively simple, needing only a bit of tuning and adapting to its formalism, and it support the plugging in of a custom training step without losing all the other commodities, like the use of callbacks to monitor the process or the usual model interface, which characterize Keras' models.

In practice, to perform the training of two models at the same time, the `train_step` function, the one called when the `fit` method is used with a TensorFlow neural network, has to be rewritten. As shown in Listing 25, the function is still preceded by the `@tf.function` decorator like the default one, which allows the creation of a static computational graph, increasing the speed of execution. Then, the gradients of both models are independently tracked with the use of two different `tf.GradientTape()` and computed on a `total_loss` obtained by adding up the reconstruction loss of the teacher, the student loss with respect to it, and a weighted loss on the latent space of the teacher Autoencoder with respect to a central hidden layer of the student.

Finally the gradients are applied to the trainable variables using their respective optimizers, which will then give the next set of parameters for the following step.

The training step just described was implemented in each epoch of the overall training procedure as shown in Listing 26. With these two modifications in the `fit` function, a hyperparameter search was launched, but due to the more complexity of the problem, and linked increase in training times, it was reduced to 47 total pairs of teacher and student. The teacher in each of them was kept with the same architecture as the one used for the offline distillation, while the quantization and architecture was changed for the students.

The results of the three best model pairs are shown in Figure 6.13, ranked by the student's average AUC across the four BSM signals. While the results are promising, with values relatively close to those achieved through offline distillation, the added complexity of the co-training procedure does not yield a sufficient pay-

```python
@tf.function
def train_step(inputs):
    x = inputs[0]
    y = inputs[1]
    with tf.GradientTape() as t_tape, tf.GradientTape() as s_tape:
        tlatent, reconstruction = teacher_model(x, training=True)
        slatent, loss_prediction = student_model(x, training=True)
        t_loss, loss_signal  = teacher_loss(y, reconstruction)
        s_loss = student_loss(loss_signal, loss_prediction)
        l_loss = latent_loss(tlatent, slatent[-1, 0:8])
        total_loss = t_loss + s_loss + latent_loss_factor*l_loss

    grad_of_teacher = t_tape.gradient(total_loss, teacher_model.trainable_variables)
    grad_of_student = s_tape.gradient(total_loss, student_model.trainable_variables)

    teacher_optimizer.apply_gradients(zip(grad_of_teacher, teacher_model.trainable_variables))
    student_optimizer.apply_gradients(zip(grad_of_student, student_model.trainable_variables))

    return t_loss, s_loss
```

Listing 25: Custom training step for a teacher-student model in TensorFlow, where both models are optimized together to perform online Knowledge Distillation.

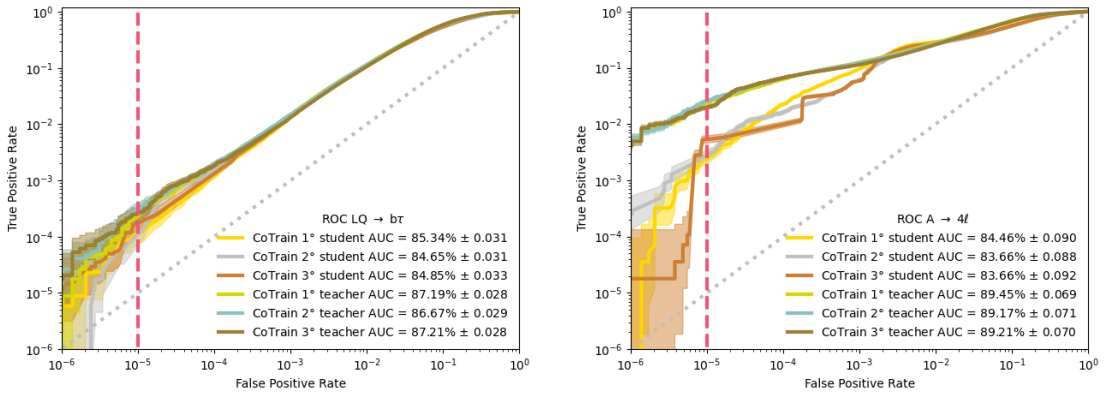
```python
for epoch in range(epochs):
    train_epoch_tloss_avg = tf.keras.metrics.Mean()
    train_epoch_sloss_avg = tf.keras.metrics.Mean()

    for i, (x, y) in enumerate(train_dataset):

        t_loss, s_loss = train_step((x,y))

        train_epoch_tloss_avg.update_state(t_loss)
        train_epoch_sloss_avg.update_state(s_loss)

    state = (train_epoch_tloss_avg.result().numpy(),
             train_epoch_sloss_avg.result().numpy(),
             val_epoch_tloss_avg.result().numpy(),
             val_epoch_sloss_avg.result().numpy())

    state_accumulator.append(state)
```

Listing 26: `for` loop performing the training step for all the data provided and storing the losses after each epoch.

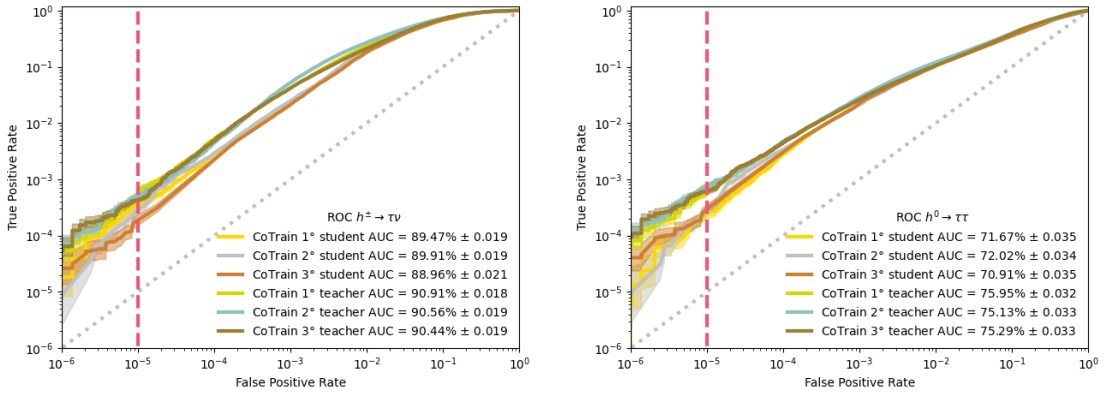
off. Moreover, the results are highly unstable, requiring precise tuning of the loss weights to ensure the teacher learns effectively before the student begins to overfit with an undertrained teacher. In conclusion, while the strategy shows potential, it requires further refinement to be a viable alternative to the offline approach.

6.4 An alternative to CNNs - A GNN for AD

As mentioned at the end of Section 6.2.2, there is a need in finding an optimal neural network for anomaly detection and the promising results in knowledge distillation open up the possibility of exploring new and more complex architectures for this task. This is the idea behind testing the idea of using Graph Neural Networks (see



(a) Leptoquark $\rightarrow b\tau$ (b) Neutral scalar boson $A \rightarrow 4\ell$



(c) Charged boson $h^{\pm} \rightarrow \tau\nu$ (d) Scalar boson $h^0 \rightarrow \tau\tau$

Figure 6.13: ROC curves of the three best pairs of student-teacher produced using an online Knowledge Distillation procedure. The rank was given according to the average over the four BSM signals under study of the AUC.

Section 4.3.2) as the encoder and decoder of an autoencoder trained for anomaly detection.

The idea comes from [166], where a new neural network module called EdgeConv is proposed for CNN-based high-level tasks on point clouds including classification and segmentation. This module acts on graphs dynamically computed in each layer of a network and it is differentiable and can be plugged into existing architectures.

But first the concept of point cloud must be introduced: they are basically scattered collections of points in 2D or 3D, which makes them the simplest way to represent shapes. Indeed, with the advent of fast 3D point cloud acquisition, recent pipelines for graphics and vision often process point clouds directly, bypassing expensive mesh reconstruction or denoising due to efficiency considerations or instability of these techniques in the presence of noise. Traditionally, point cloud classification and segmentation were carried out employing handcrafted features to capture geometric properties, however, more recently the success of deep neural networks for image processing has motivated a data-driven approach to learning

features on point clouds. Deep point cloud processing and analysis methods are developing rapidly and could outperform traditional approaches in various tasks.

Inspired by the PointNet model [167], the idea is to exploit local geometric structures of data by constructing a local neighborhood graph and applying convolution-like operations on the edges connecting neighboring pairs of points, in the spirit of GNNs. From this edge convolution the EdgeConv name comes from. Unlike graph CNNs, the graph here is not fixed but rather is dynamically updated after each layer of the network. That is, the set of k-nearest neighbors of a point changes from layer to layer of the network and is computed from the sequence of representation of data inside the model. Proximity in feature space differs from proximity in the input, leading to non-local diffusion of information throughout the point cloud.

6.4.1 The formalism behind the Edge Convolution

A point cloud is represented by an F-dimensional vector with n points, denoted by $\mathbf{X} = \{x_1, \ldots, x_n\} \subseteq \mathbb{R}^F$. In the implementation explored in this study $F = 3$, i.e. the three quantities p_T, η, and ϕ of the particles, jets and MET in the dataset. This could be interpreted as each point containing 3D coordinates. In a deep neural network architecture, each subsequent layer operates on the output of the previous layer, so more generally the dimension F represents the feature dimensionality of a given layer when inside the model.

A directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is computed representing local point structure, where $\mathcal{V} = \{1, \ldots, n\}$ and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ are the vertices and edges respectively. In this case $\mathcal{G}$ is constructed as the k-nearest neighbours (k-NN) graph of $\mathbf{X}$ in $\mathbb{R}^F$. The graph includes self-loops, meaning each node also points to itself. hen, the edge features are defined as $\mathbf{e}_{ij} = h_\Theta(x_i, x_j)$, where $h_\Theta : \mathbb{R}^F \times \mathbb{R}^F \to \mathbb{R}^{F'}$ is a nonlinear function with a set of learnable parameters Θ. Now the EdgeConv operation can be defined as a channel-wise symmetric aggregation operation $\square$ (e.g. $\sum$ or the maximum function) on the edge features associated with all the edges emanating from each vertex. Thus, the output at the i-th vertex is given by

$$\mathbf{x}'_i \doteq \square_{j:(i,j)\in\mathcal{E}} h_\Theta(\mathbf{x}_i, \mathbf{x}_j) \tag{6.15}$$

As an analogy to convolution in images, the $\mathbf{x}_i$ can be regarded as the central pixel, and $\{\mathbf{x}_j : (i,j) \in \mathcal{E}\}$ as a patch around it. The EdgeConv operation will produce a F'-dimensional point cloud with n points from a F-dimensional one with the same number of points.

The choice of the edge function and the aggregation operation has a crucial influence on the properties of EdgeConv. The option chosen for this implementation is an asymmetric edge function $h_\Theta(\mathbf{x}_i, \mathbf{x}_j) = h_\Theta(\mathbf{x}_i, \mathbf{x}_j - \mathbf{x}_i)$. This explicitly combines global shape structure, captured by the coordinates of the centers $\mathbf{x}_i$, with local neighborhood information, captured by $\mathbf{x}_j - \mathbf{x}_i$. In particular, the operator can be written as:

$$e'_{ijm} = \mathrm{ReLU}(\theta_m \cdot (\mathbf{x}_j - \mathbf{x}_i) + \phi_m \cdot \mathbf{x}_i) \tag{6.16}$$

where $\Theta = (\theta_1, \ldots, \theta_M, \phi_1, \ldots, \phi_M)$. This can be implemented in Keras using a 2-dimensional convolutional layer with a 1 by 1 kernel followed by a ReLU activation function.

Before it was anticipated that the graph is updated after each layer of the network. The way of doing that is by using the nearest neighbours in the feature space produced by each layer. Such a dynamic graph update is the reason why this architecture could be called Dynamic Graph CNN (DGCNN). With dynamic graph updates, the receptive field is as large as the diameter of the point cloud, while being sparse. At each layer there is a different graph $\mathcal{G}^{(l)} = (\mathcal{V}^{(l)}, \mathcal{E}^{(l)})$, where the l-th layer edges are of the form $(i, j_{i1}), \ldots, (i, j_{ik_l})$ such that $\mathbf{x}^{(l)}_{j_{i1}}, \ldots, \mathbf{x}^{(l)}_{j_{ik_l}}$ are the k_l points closest to $\mathbf{x}^{(l)}_i$. Put differently, the architecture learns how to construct the graph $\mathcal{G}$ used in each layer rather than taking it as a fixed constant constructed before the network is evaluated. That is done by computing a pairwise distance matrix in feature space and then take the closest k points for each single point.

6.4.2 Implementation and results

In the Python implementation of the DGCNN Autoencoder using TensorFlow, as mentioned earlier, the graph is updated at each layer by computing the pairwise distances between points in 3D space. This is accomplished within the `EdgeComp` function, which calculates the standard Euclidean distance using code similar to that in Listing 27, based on the formula:

$$\text{distance}(p_i, p_j) = ||p_i - p_j||^2 = ||p_i||^2 + ||p_j||^2 - 2 \cdot p_i^T p_j \tag{6.17}$$

```
point_cloud_transpose = tf.transpose(point_cloud, perm=[0, 2, 1])
point_cloud_inner = tf.linalg.matmul(point_cloud, point_cloud_transpose)
point_cloud_inner = -2*point_cloud_inner
point_cloud_square = tf.math.reduce_sum(tf.square(point_cloud), axis=-1, keepdims=True)
point_cloud_square_tranpose = tf.transpose(point_cloud_square, perm=[0, 2, 1])
adj_matrix = point_cloud_square + point_cloud_inner + point_cloud_square_tranpose
```

Listing 27: Python code to compute the pairwise distance of each point in 3D space, used to find the k-nearest neighbours and create the 3D graph inside the Dynamic Graph CNN.

In Listing 28 the first few lines of the code used to build the encoder of the model are shown. The decoder follows a very similar architecture in order to make a final model as symmetric as possible to facilitate the reconstruction of the original point cloud from the encoded latent space. There is an element in this build function that is to note: the `input_transform_net` object.

This cluster of layers (whose build function is in Listing 29) was introduced to align the input point set to a canonical space by applying an estimated 3×3 matrix. This matrix is calculated by concatenating each point's coordinates with the coordinate differences between that point and its k nearest neighbors. Essentially, this process mirrors the operations performed throughout the rest of the network. Introducing this transformation allows the points in the cloud to be

```python
def build_encoder(self):
    inputs = tf.keras.layers.Input((self.cloud_shape,3,1))
    edge_feature = EdgeComp(k=self.k)(inputs)

    transform = input_transform_net(edge_feature, K=3)

    point_cloud_transformed = MatMult()([inputs,transform])
    edge_feature = EdgeComp(k=self.k)(point_cloud_transformed)

    x = tf.keras.layers.Conv2D(64, kernel_size=(1,1), use_bias=True, padding='valid')(edge_feature)
    x = tf.keras.layers.Activation('relu')(x)
    x = tf.keras.layers.BatchNormalization()(x)
```

Listing 28: First part of the function dedicated to the description of the encoder in the Dynamic Graph CNN Autoencoder.

```python
def input_transform_net(edge_feature,K=3):
    num_point = edge_feature.shape[1]

    x = tf.keras.layers.Conv2D(64,kernel_size=(1,1), use_bias=True, padding='valid')(edge_feature)
    x = tf.keras.layers.Activation('relu')(x)
    x = tf.keras.layers.BatchNormalization()(x)

    # [...] Two more Conv2D with activation and BatchNormalization

    x = tf.keras.layers.MaxPooling2D(pool_size=(num_point,1),strides=(2,2))(x)
    x = tf.keras.layers.Flatten()(x)

    x = tf.keras.layers.Dense(512,activation='relu')(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Dense(256,activation='relu')(x)
    x = tf.keras.layers.BatchNormalization()(x)

    transform = tf.keras.layers.Dense(K*K)(x)
    transform = tf.keras.layers.Reshape((K,K))(transform)
    return transform
```

Listing 29: Input transformation net described as a collection of Keras layers. This object was used at the beginning of the encoder, after the first edge convolution, to align an input point set to a canonical space by applying an estimated 3×3 matrix.

projected from their original p_T, η, and ϕ space into a new 3D space, where graph operations should be performed more effectively. The transformation is trainable and learned simultaneously with the rest of the neural network, enabling the model to optimize this alignment dynamically.

The code written for this implementation was inspired by the one made for [166], however all the TensorFlow had to be translated from v1.x to v2, which follows a fundamentally different programming paradigm, making the translation not trivial [168].

Eventually, the DGCNN obtained had an encoder with the architecture shown in Figure 6.14. Just from this it can be said that the DGCNN is much more complex than the CNN AE previously used, given also the presence of links between different parts of the network, making it not completely sequential. The complex schematic

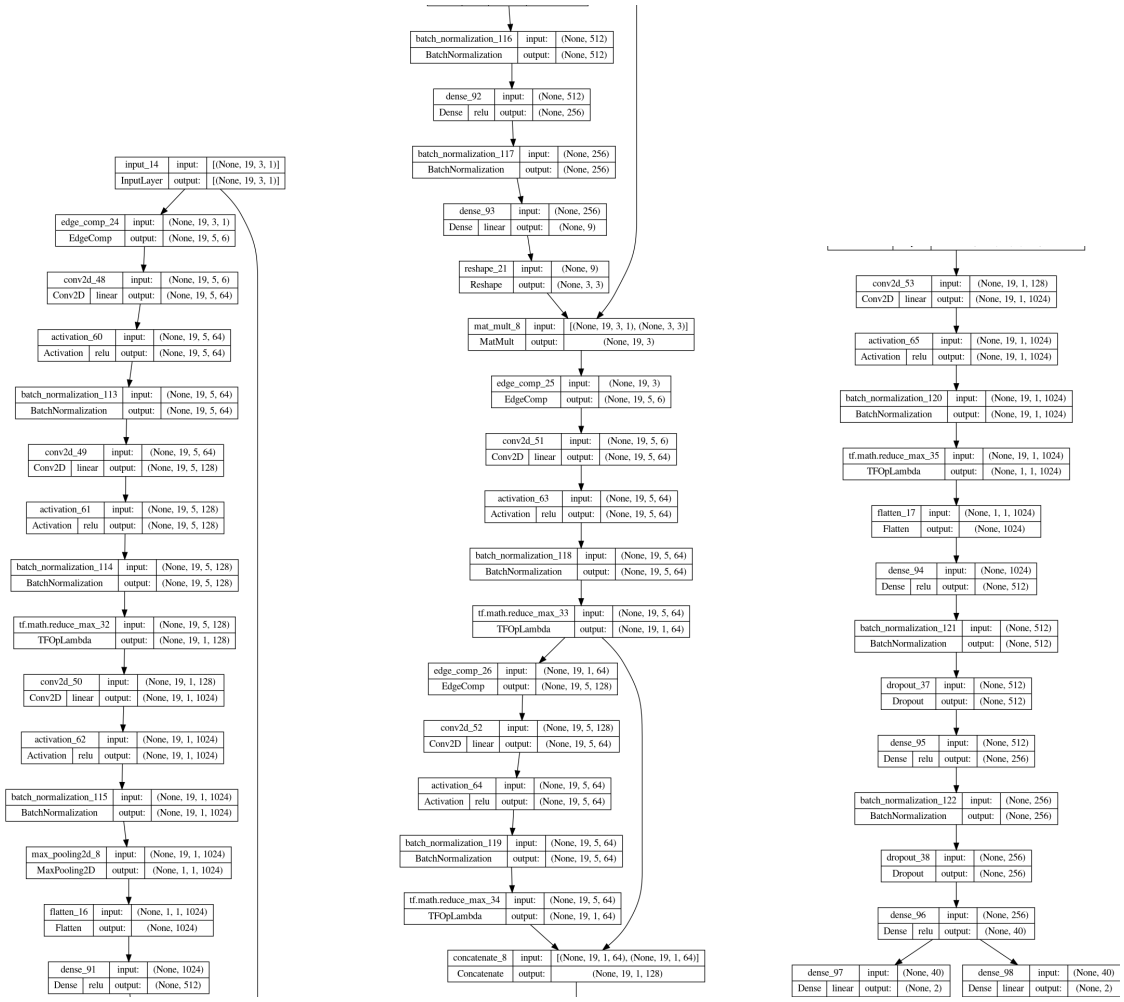
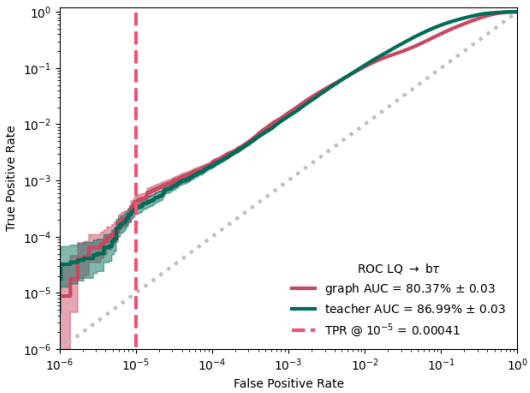


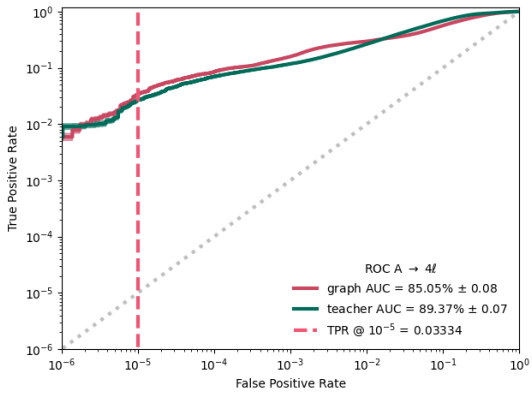
Figure 6.14: Encoder architecture of a Dynamic Graph CNN Autoencoder built to perform Anomaly Detection.

and the related increase in the number of parameters and operations needed for such a model, makes it a perfect candidate to test the Knowledge Distillation procedure in the future.

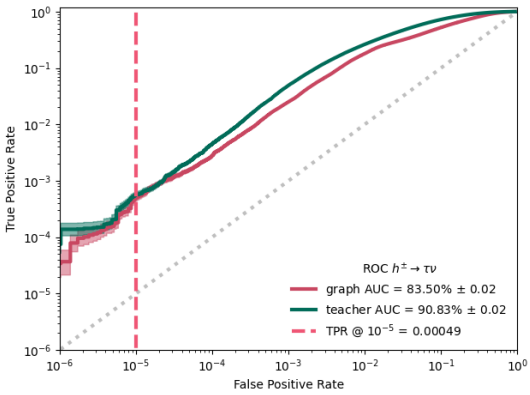
In order to support the last statement, the anomaly detection scores as ROC curves used throughout all this thesis are shown in Figure 6.15, compared to the teacher previously used. The performance obtained is comparable but still slightly worse than the CNN AE, however it must be kept in mind that this was a very preliminary test, and a lot of the effort was made towards the upgrade of the code to TensorFlow 2, rather than the optimization of the hyperparameters or the architecture in general.



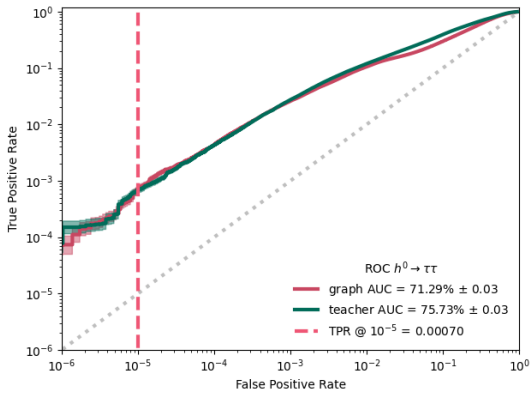
(a) Leptoquark $\rightarrow b\tau$



(b) Neutral scalar boson $A \rightarrow 4\ell$



(c) Charged boson $h^{\pm} \rightarrow \tau\nu$



(d) Scalar boson $h^0 \rightarrow \tau\tau$

Figure 6.15: ROC curves of the Graph Autoencoder trained on the point cloud compared to the Convolutional AE used in the rest of this thesis as a teacher for KD.

Conclusions

This thesis has explored the application of Machine Learning techniques, specifically Autoencoders (AEs), to enhance data analysis and event selection at the Compact Muon Solenoid experiment placed in the Large Hadron Collider at CERN, with a focus on efficiently implementing these methods on hardware suited for online applications, such as Field Programmable Gate Arrays (FPGAs). The increasing versatility of Artificial Neural Networks (ANNs) in high-energy physics has demonstrated significant potential, but the challenge of meeting the strict latency and energy constraints of the Level-1 Trigger requires specialized approaches.

To address these challenges, this work investigated the use of Knowledge Distillation (KD) as a method for compressing large, well-trained models into smaller, more hardware-efficient versions. The distillation process was optimized, considering various student architectures and the quantization of weights and biases, finally achieving a balance between accuracy, latency, and hardware footprint. The performance of Offline Response-Based KD was examined, along with the impact of applying quantization before or together with determining the optimal student model architecture, reaching the conclusion that the latter has a higher probability of giving optimal results with fewer trials. A detailed account of the steps required to convert a Python-based model into firmware for FPGA, using both the hls4ml library and proprietary FPGA software, was provided.

Additionally, the thesis presented preliminary exploration into Online Response-Based KD as a potential alternative approach. Finally, a more advanced teacher model, based on a Graph Convolutional Neural Network Autoencoder, was tested for Anomaly Detection, highlighting the potential for KD to facilitate the implementation of more sophisticated algorithms on resource-constrained hardware.

This research has demonstrated that KD can play a critical role in enabling efficient, high-performance Machine Learning models for real-time data processing in environments like the Level-1 Trigger at CMS, paving the way for future advancements in trigger systems at the LHC and beyond.

Appendix A

Machine Learning inference using PYNQ environment in a AWS EC2 F1 Instance

A.1 Introduction

Machine Learning has become in recent years one of the pillars of computer and data science and it has been introduced in almost every aspect of everyday life and research fields alike. Currently, the spread of learning algorithms in many sectors finds its roots mainly in an increased quantity of data available, combined with a technological progress in storage and computational power, which can nowadays be delivered with lower maintenance and building costs.

In order to reach the full potential of ML algorithms, new computing solutions are being developed and tested like never before since the rise of the x86 architecture as the *de facto* standard for general purpose computing. This is done to find the perfect combination of fast prediction times and low energy consumption needed to deploy ML efficiently in a variety of use cases, from IoT devices to data centers applications and scientific research.

This work focuses on a specific type of hardware called Field Programmable Gate Array (described in Chapter 3) which promises low latencies and unprecedented power efficiency. In order to facilitate the translation of ML models to fit in the usual workflow for programming FPGAs, a variety of tools have been developed. One example is the HLS4ML toolkit, developed by the HEP community, which allows the translation of Neural Networks built using tools like TensorFlow to a High-Level Synthesis description (e.g. C++) in order to implement this kind of ML algorithms on FPGAs.

The analysis described in this appendix concentrate on a new way to interact and retrieve results from FPGAs: PYNQ (Section A.2). This Python package allows to use a simple Python script to program the FPGA and use the function included in its design in a similar way to usual function calls.

Performance tests on a regressor model used as benchmark, will be presented in Section A.3, where the consistency in the predictions of the NN with respect to using an OpenCL application, will be verified.

Summing up, this paper describes the work done to produce a complete and as simple as possible workflow to implement algorithms of interest to the HEP field, namely Neural Networks, on FPGAs. A case study from the CMS experiment at CERN was used as an example to test the different tools employed and as a benchmark to take some preliminary measurements regarding latency and accuracy of the algorithm.

A.1.1 AWS EC2 F1 Instance

In order to test the capabilities of the implementation workflow presented in this work, cloud computing resources, more specifically Amazon Web Services' EC2 F1 instances [169], equipped with Xilinx FPGA acceleration cards, have been used. F1 instances are equipped with tools to develop, simulate, debug, and compile a hardware acceleration code.

Using F1 instances to deploy hardware accelerations can be useful in many applications to solve complex science, engineering, and business problems that require high bandwidth, enhanced networking, and very high compute capabilities. Examples of target applications that can benefit from F1 instance acceleration are genomics, search/analytics, image and video processing, network security, electronic design automation (EDA), image and file compression and big data analytics.

F1 instances provide diverse development environments: from low-level hardware developers to software developers who are more comfortable with C/C++ and OpenCL environments. Once an FPGA design is complete, it can be registered as an *Amazon FPGA Image* (AFI), and deployed to every F1 instance needed.

To deploy a design on these instances, the bitstream must be uploaded to an S3 Bucket [170] and request the creation of an AFI using a script included in the official github repository of the AWS EC2 FPGA Hardware Development Kit [171]. This will produce a `awsxclbin` file that can be used to program Amazon's FPGAs.

A.2 The PYNQ project

PYNQ [54] is an open-source project from Xilinx®, a prominent FPGA producer. It provides a Jupyter-based framework with Python APIs for using Xilinx platforms and AWS-F1 instances.

FPGA designs are presented as Python objects called *overlays* that can be accessed through a Python API. Creating a new overlay still requires developers with expertise in designing programmable logic circuits. Overlays, like software libraries, are designed to be configurable and re-used as often as possible in many different applications.

To date, C or C++ are the most common embedded programming languages. In contrast, Python raises the level of programming abstraction and programmer productivity. These are not mutually exclusive choices, however. PYNQ uses CPython which is written in C, and integrates thousands of C libraries and can be extended with optimized code written in C. Wherever practical, the more productive Python environment should be used, and whenever efficiency dictates, lower-level C code can be used.

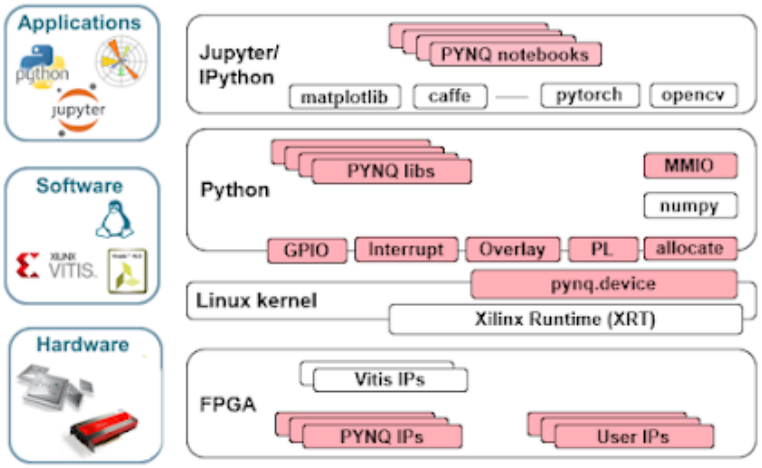


Figure A.1: PYNQ's components in the different level of abstraction needed for running applications on FPGAs.

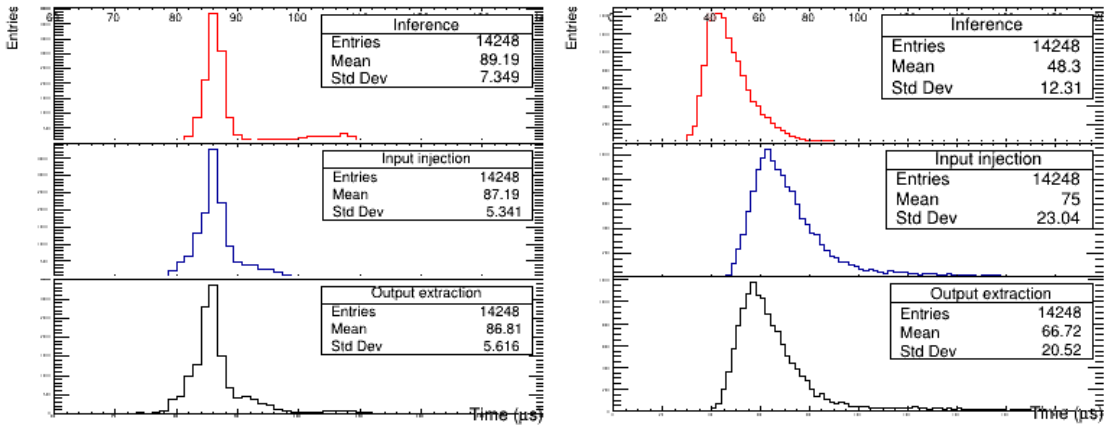
PYNQ aims to work on any computing platform and operating system. This goal is achieved by adopting a web-based architecture, which is also browser agnostic. It incorporates the open-source Jupyter notebook infrastructure to run an Interactive Python (IPython) kernel and a web server directly on the ARM processor of a MPSoC or host's CPU of an acceleration card. The web server brokers access to the kernel via a suite of browser-based tools that provide a dashboard, bash terminal, code editors and Jupyter notebooks. The browser tools are implemented with a combination of JavaScript, HTML and CSS and run on any modern browser. PYNQ's main components are summed up in Figure A.1.

A description on how to use PYNQ and a comparison with writing an OpenCL application can be found in Section 3.3. By looking at both approaches, it is evident how writing Python code including the PYNQ package is less complicated than the alternative.

A.3 Neural Network performance on FPGA

Two main aspects have been considered to study the performance of using the PYNQ package to carry out Neural Network inference on an FPGA: latency and inference accuracy. The model built for this research is the next iteration of the regressor designed for the Master's thesis [56]. Its purpose was to find an alternative algorithm to perform transverse momentum (p_T) assignment to muons in the context of the Level-1 trigger at the Compact Muon Solenoid experiment at CERN. This NN has been implemented with the following structure: the first hidden layer has 35 neurons and receives the information directly from the input layer of 27 different features with the ReLU selected as activation function. The second layer is identical to the first one but contains 20 neurons and this is repeated for other 4 additional hidden layers with 25, 40, 20 and 15 neurons, respectively. In the end, the output layer (with only one node) closes the network.

Also a pattern recognition classifier trained and tested using the Iris dataset from the UCI Machine Learning Repository [172] was tested, but only to verify



(a) PYNQ (b) OpenCL

Figure A.2: Distribution of the times needed to inject data in the FPGA, perform NN inference and extract the output using the PYNQ package in Python (left) and an OpenCL application (right).

that this workflow could also be applied to classification algorithms, albeit this example being a very simple one.

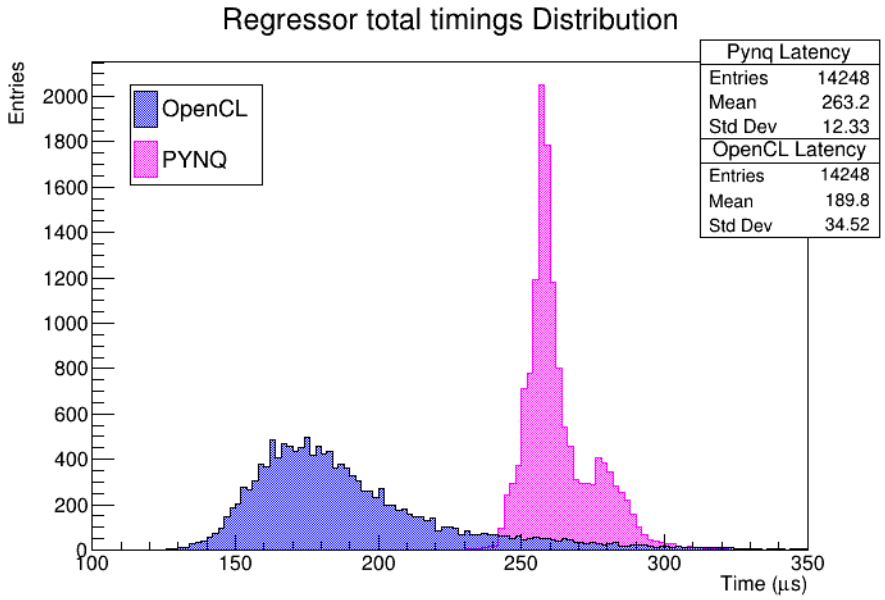


Figure A.3: Total inference time distribution (input injection + inference + output extraction) using PYNQ (pink) and an OpenCL application (blue).

For the first metric, the *wall* time has been measured for the three main tasks that are executed by the host-FPGA pair for each inference that is requested. In Figure A.2 the time distribution for the input injection on the FPGA card (blue), the actual inference (red) and output extraction (black) is shown for the entire validation dataset using PYNQ on the left and the OpenCL application on the right. In the PYNQ case, a degree of consistency can be seen between the different tasks. This can be explained by a common overhead caused by Python's nature

as an interpreted language, which can also be considered as the main cause for the overall larger total processing time, shown in Figure A.3, with respect to the application compiled in C++.

Nonetheless, the main objective of using PYNQ is offering an easier interface and less steep learning curve in dealing with accelerating algorithms using FPGAs. This means that, to achieve the full potential of this type of hardware, the traditional approach using C/C++ application is still the way to follow.

A.3.1 p_T resolution histogram

To study the accuracy of the NN model implemented on the F1 instance p_T resolution histograms were used. For each entry of the dataset, the histograms were built using the following relation:

$$\frac{\Delta p_T}{p_T} = \frac{p_{T_{est}} - p_{T_{sim}}}{p_{T_{sim}}} \tag{A.1}$$

where $p_{T_{est}}$ is the estimation of the transverse momentum, given by the model prediction or the actual algorithm used in the Level-1 trigger at CMS to perform this task, and $p_{T_{sim}}$ is the "true" transverse momentum associated to each entry of the validation set. Even though this metric makes a quick and easy to understand comparison possible, it is important to keep in mind that this resolution is asymmetric, i.e. its range can go from -1 to infinite. This means that, for a constant actual spread, the standard deviation associated to its distribution is affected by the value of its mean: the smaller it is, the smaller the standard deviation gets.

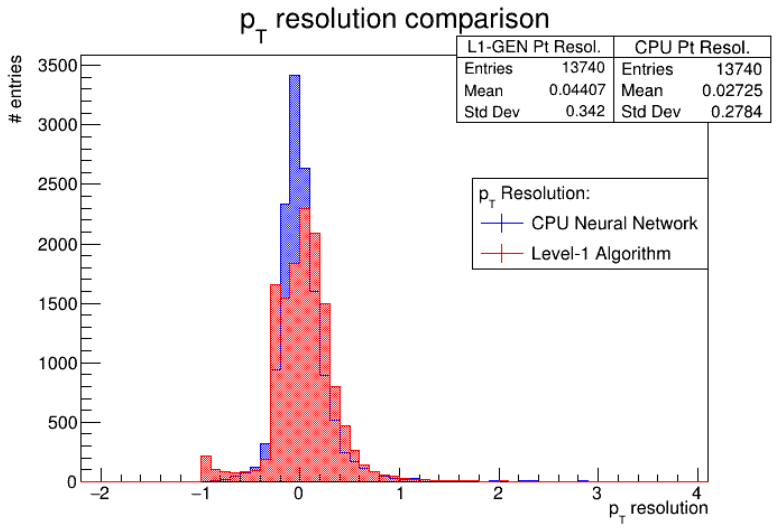


Figure A.4: Transverse momentum resolution histograms computed for the machine learning model (blue) and Level-1 trigger (red) based momentum assignment.

Firstly, the resolution of the model before the implementation on the FPGA must be checked (Figure A.4). The red histogram describes the resolution distribution of the Level-1 trigger system while the blue one shows the resolution of the predictions made by the network model running on a consumer CPU.

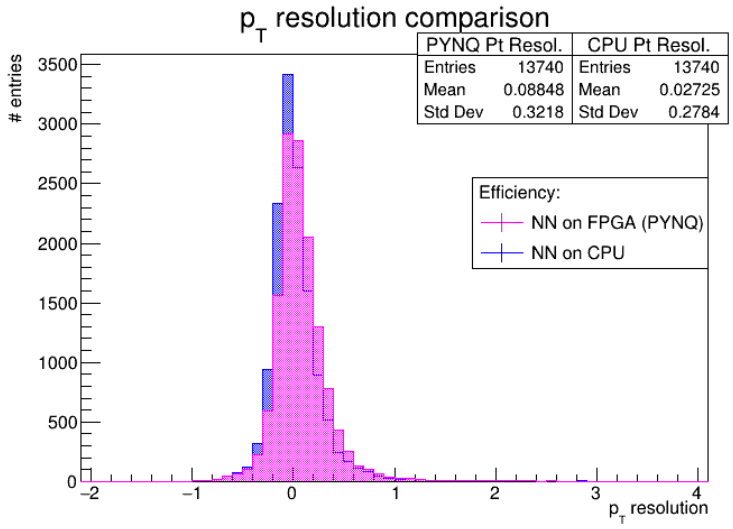


Figure A.5: Transverse momentum resolution histograms computed for the machine learning model (blue) and Level-1 trigger (red) based momentum assignment.

In particular, it is possible to notice a less broad distribution for the ML resolution, resulting in an overall improvement, yet small, with respect to the Level-1 trigger system. Another noticeable detail is the small peak corresponding to the value -1: this happens when the p_T assigned by the trigger is significantly underestimated with respect to the true p_T. The Machine Learning based momentum assignment is therefore less prone to large p_T underestimation.

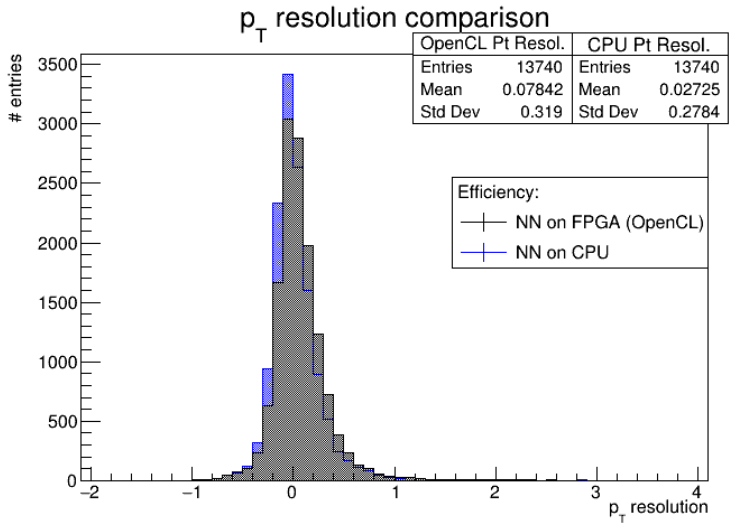


Figure A.6: Transverse momentum resolution histograms computed for the machine learning model (blue) and Level-1 trigger (red) based momentum assignment.

Having verified the accuracy of the NN model, its implementation on the FPGA available in the F1 instance can be analyzed. In Figure A.5 the p_T resolution histogram obtained by performing the inference using the PYNQ environment is shown over the model resolution described before. It is clear that the model infer momenta with a resolution distribution which is narrower when the computation

is carried out on a CPU. When the assignment is performed on an FPGA, slightly worse results are produced, with a small bias towards higher values of $\Delta p_T/p_T$. This could be the effect of the loss in precision the input features have to go through due to the conversion to fixed-point representation needed to perform computations efficiently in an FPGA [56]. Nevertheless, the hardware approach still appears compatible, or in case of higher momenta, even better than the Level-1 trigger based momentum assignment.

For a final comparison, in Figure A.6 there is the resolution histogram obtained by performing the inference on the FPGA using an OpenCL application. As expected the result is very similar to the PYNQ one, however there is a small difference which can be explained by a different implementation of floating point numbers to fixed point precision conversion.

Appendix B

Scalable training on scalable infrastructures for programmable hardware

Machine Learning has gained significant prominence in recent years within the field of computer science. This is evident through the proliferation of educational initiatives, workshops, and courses aimed at enhancing skills in this domain. In 2020, the AI Index [173], an independent effort associated with the Stanford Institute for Human-Centered Artificial Intelligence (HAI), conducted a survey targeting top-ranked universities across the globe. This survey focused on four key aspects of AI education: undergraduate and graduate program offerings, education in AI ethics, and faculty diversity and expertise. The survey received responses from 18 universities spanning 9 countries. The results of the survey indicate a noteworthy increase in the quantity of AI courses offered, concentrating on the practical development and deployment of AI models, as well as an uptick in AI-specialized faculty.

The survey also delved into advanced-level courses, specifically those intended for graduate students seeking to acquire the necessary skills for constructing and implementing practical AI models. Over the course of the last four academic years, these offerings saw a substantial 41.7% surge, rising from 151 courses in the 2016–17 academic year to 214 in the 2019–20 academic year.

In response to the escalating demand for AI courses and degree programs, there was a significant rise in the number of tenure-track faculty members with a primary research emphasis on AI at the surveyed universities. The count of AI-focused faculty increased by 59.1%, expanding from 105 individuals in the 2016–17 academic year to 167 in the 2019–20 academic year.

The European Commission's Joint Research Center (JRC) evaluated advanced digital skills education across 27 European Union member states and an additional six countries: the United Kingdom, Norway, Switzerland, Canada, the United States, and Australia. The Commission counted a total of 1,680 specialized AI programs across all considered countries during the 2019–20 academic year. Notably, the United States boasted a higher count of specialized AI programs compared to other regions, although the EU closely followed, particularly in terms of AI-specialized Master's programs.

B.1 The course: *Machine learning techniques with FPGA devices for particle physics experiments*

The introduction of FPGAs has significantly transformed the landscape of digital logic design and deployment [44]. By blending the performance attributes of ASICs with the adaptability of microprocessors, FPGAs have enabled novel applications and even replaced ASICs and digital signal processors in some conventional roles. However, harnessing the potential of FPGAs requires a comprehensive grasp of both hardware and software considerations. This entails not only accounting for the hardware components required for computations but also incorporating the software workflow that facilitates the design process. Although FPGAs offer the advantages of software flexibility and hardware efficiency, achieving optimal results demands a more intricate programming approach compared to microprocessors, despite FPGAs' superior speed and energy efficiency. Effectively leveraging FPGAs necessitates a foundational comprehension of both software and hardware principles. This includes, as depicted in Figure B.1, familiarity with digital logic design, hardware description languages like Verilog or VHDL, as well as basic computer programming knowledge encompassing data structures and algorithms. An ideal user profile would integrate expertise in electrical engineering, computer science, and computer engineering. Condensing such a vast array of concepts within a single course or workshop presents a significant challenge.

In the previous section, the abundance of new courses focused on Machine Learning and Artificial Intelligence is evident, yet the same is not observed when considering the integration of AI and FPGAs. Despite the potential benefits of combining these advanced technologies, such as reduced latency and energy consumption, particularly in fields like High Energy Physics [122], there have been limited endeavors to educate individuals in this intersection.

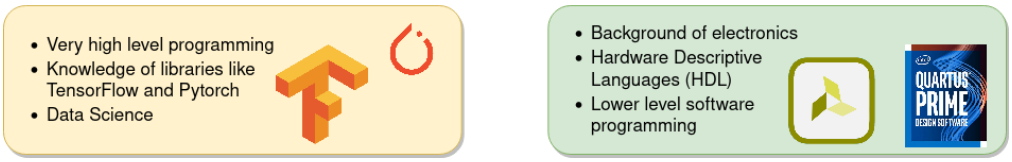


Figure B.1: Different set of skills needed to be proficient in both the world of AI and FPGAs.

From the gap in tutorials on ML on FPGAs, the idea of a course called *Machine learning techniques with FPGA devices for particle physics experiments* [174] came up, in order to give a start in understanding and experimenting the various tools that allow the connection between the world of AI and FPGAs.

The course took place from 2^{nd} to 4^{th} November 2022 and it was organized by the Bologna division of the Italian National institute for Nuclear Physics (INFN) with the technical support of CNAF, the main data processing and computing technology research center of INFN. This effort was funded by the INFN Training program. It represented a first step towards a greater focus on education in this field

in Italy. The course featured leading international lecturers who are involved in the development of tools to make hardware more approachable at a higher level. The program also received support from the AMD/Xilinx University Program (XUP).

A lot of topics were addressed in the dense two days of lectures and more than half of the duration of the course was spent on tutorials:

- Introduction to efficient use of Machine Learning in HEP;
- Crash course on what FPGAs are;
- *HLS4ML* and how to translate Python to something implementable in hardware (see Section 5.3.1)
- *Vitis-AI*, the AMD/Xilinx solution to Artifical Intelligence on programmable hardware;
- A new kind of computer architecture (multi-core and heterogeneous) which dynamically adapts to the specific computational problem rather than being static: the *BondMachine* (see Section B.1.1)
- How Quartus and Intel make ML on FPGA possible;

In the next Section a small description of the *BondMachine* is given, as it was one of the two topics of major interest for the high-energy physics community, together with hls4ml, explained in section 5.3.1.

B.1.1 The BondMachine

BondMachine (BM) [175] is an open-source framework that enables the creation of computational systems with co-designed hardware and software. This approach maximizes the use of existing resources in terms of concurrency and heterogeneity. The unique feature of BM is the creation of a dynamic architecture that adapts to the specific problem, rather than being static. The hardware is customized to meet the software requirements, implementing only the necessary processing units, resulting in significant advantages in terms of energy consumption and performance. Furthermore, BM is vendor and board independent, allowing for the creation of clusters of heterogeneous FPGAs.

Compared to the use of Hardware Description Language (HDL) code, BM introduces an architecture abstraction layer with minimal overhead, allowing for the use of a standard computational model. This toolkit makes full use of the main features of FPGAs and can be used as an High-Level tool to generate custom firmware for accelerated computation.

The BM architecture is particularly suitable for computational models and graphs. The project's flagship activity involves generating firmware with the aim of developing accelerated systems on FPGA to solve different computational problems with a particular focus on machine learning inference [176]. The firmware for accelerated inference generated starting from an high-level trained model with standard machine learning libraries, is highly customizable according to the needs of the specific problem. Different hardware and software optimization techniques

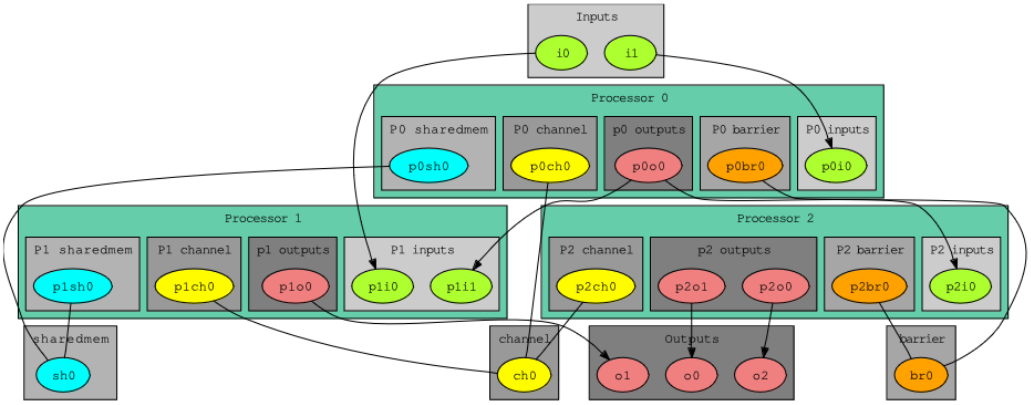


Figure B.2: An example of a BondMachine architecture. This specific BM is made of two inputs and three outputs interconnected between the input/output registers of the processors. Shared objects, such as memory, Channel and Barrier, are connected among the processors.

have been implemented, starting from the choice of the numerical precision, up to the collapse and pruning of the processors, in order to reduce the resource usage and the energy consumption while improving the inference speed at the same time.

B.2 A scalable classroom using Cloud Computing

The course aimed to provide an avenue for participants to gain hands-on experience with FPGA technology and the workflows that will be used to create a functional ML design. However, the development of ML algorithms and FPGA firmware requires specific software and libraries, which means a dedicated development machine must be available to attendees. On the other hand, despite the desire to use actual hardware to test the firmware, it is typically not possible for multiple individuals to access FPGAs simultaneously for programming. At the same time it is evident that providing a board for each attendee would be cost-prohibitive and impractical. As a result, the solution was to utilize FPGAs in the cloud.

A system with two different machines was set up (Figure B.3): a *Development machine* and a *Deployment machine*.

The *Development machines* consist in CentOS 7 Virtual Machines (VM) created in the INFN Cloud infrastructure. By utilizing Anaconda [177], a Python environment was made accessible which contained all the necessary tools to manipulate data and construct Neural Networks such as TensorFlow, Keras, QKeras [106] for quantization and optimization, and HLS4ML. To access the machines, SSH with X11 support has been used. The Vivado Design Suite was installed to enable the creation of FPGA firmware, equipped with the relevant libraries to target the available board in the deployment machine. To guarantee remote access to the machines, a public floating IP (FIP) address has been assigned to each VM. In order to let the users play with the available resources and services after the

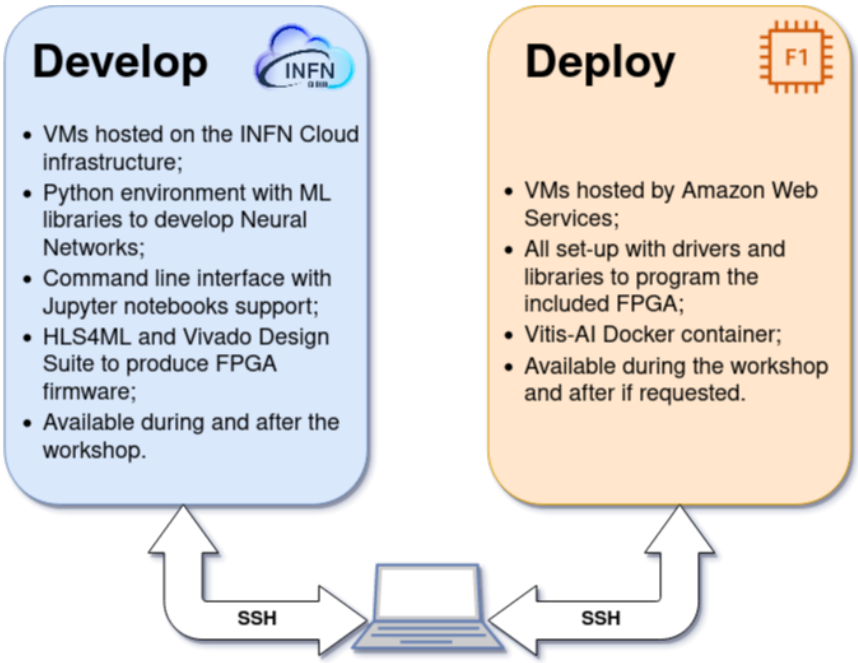


Figure B.3: Layout of the two virtual machines made available to each attendee of the course.

working period of the workshop, they were maintained for a few weeks after the workshop ended.

The *Deployment machines* hosted by AWS are EC2 F1 instances [169], equipped with Xilinx FPGA acceleration cards. F1 instances are equipped with tools to develop, simulate, debug, and compile a hardware acceleration code, including an FPGA Developer Amazon Machine Image (AMI) and supporting hardware level development on the cloud. In order to test the Vitis-AI toolkit [178], the Docker Daemon was added to the AMI.

Using F1 instances to deploy hardware accelerations can be useful in many applications to solve complex science, engineering, and business problems that require high bandwidth, enhanced networking, and very high compute capabilities. A variety of target applications can benefit from F1 instance acceleration, including but not limited to genomics, search/analytics, image and video processing, network security, electronic design automation (EDA), image and file compression, and big data analytics.

F1 instances provide diverse development environments: from low-level hardware developers to software developers who are more comfortable with C/C++ and OpenCL environments. Once an FPGA design is complete, it can be registered as an Amazon FPGA Image (AFI), and deployed to every F1 instance needed.

The course has been used as a test to exploit the potential benefits of a seamless integration between INFN Cloud and a cloud provider like AWS. The proposed sketch of how this integration could work are listed hereafter:

- The users would authenticate themselves on the INFN Cloud using a federated authentication system;

- They would then select the type of resource they need, even FPGAs from various vendors;
- If the desired FPGA resource is not available on INFN Cloud, it could be instantiated on AWS transparently;
- The user would be provided with an endpoint to connect to, without the need for a different authentication or interface.

B.3 Expanding INFN Cloud Services with HPC Bubbles

This proof of concept is part of the effort by the people behind INFN Cloud to continuously expand the services that they can offer and keep up with the ever-growing interest in heterogeneous computing.

Indeed, INFN spearheaded the *Terabit network for Research and Academic Big data in Italy* (TeRABIT) initiative, which is backed by the Italian National Recovery and Resilience Plan (NRRP) [179]. The project's objective revolves around establishing a distributed, highly interconnected hybrid Cloud-HPC computing environment. This entails the integration of the distributed INFN infrastructure with PRACE-Italy's HPC resources, facilitated by a high-speed network provided by the National computer network for universities and research (GARR)

Within this framework, INFN is expanding its INFN Cloud infrastructure by deploying distributed HPC infrastructures known as "HPC Bubbles" These HPC Bubbles encompass various clusters featuring CPUs, CPUs + GPUs, and CPUs + FPGAs, along with swift storage capabilities. The plan encompasses achieving integration both among the distributed HPC bubbles themselves and between these bubbles and the INFN Cloud infrastructure. Moreover, integration is sought between the HPC bubbles and conventional HPC systems, with a notable focus on the Leonardo@CINECA system.

The overarching aim is to establish a scalable "Edge-Cloud Continuum" that leverages AI technologies. This continuum is designed to empower users to flexibly process substantial volumes of big data, offering a dynamic and adaptable approach to data processing.

Acknowledgments

We would like to express our gratitude to Dr. Thea Aarrestad (ETH), Dr. Vladimir Loncar (CERN), Dr. Jennifer Ngadiuba (CALTECH), and Dr. Sioni Summers (CERN) for their invaluable support and constructive feedback. Additionally, we would like to acknowledge the financial support provided by the INFN Training Commission and the technical assistance offered by the AMD/Xilinx University Program. Furthermore, we extend our appreciation to Mariella Gangi and Antonella Monducci for their organizational support.

Appendix C

QUnfold - A Python library for unfolding using Quantum Computing

C.1 Introduction

In High-Energy Physics (HEP), the analysis of experimental data often involves correcting for distortions introduced by the measurement process. These distortions stem from various systematic effects such as the finite resolution of detectors, their limited efficiency and geometric acceptance, as well as background noise. These factors combine to obscure the true underlying distribution of the physical quantities of interest. The process of correcting these distortions to recover the true distribution is known as the statistical unfolding problem.

The importance of unfolding cannot be overstated, as it is crucial for accurate comparison between experimental results and theoretical predictions. Without proper unfolding, the biases and smearing effects inherent in the data would lead to incorrect conclusions about the physical phenomena being studied.

In this context, the emerging technology of quantum computing represents an opportunity to enhance the unfolding performance and potentially yield more accurate results. QUnfold [180] is a Python package designed to tackle the unfolding problem by leveraging the capabilities of Quantum Annealing (QA), explained briefly in Section C.2. The regularized log-likelihood maximization formulation of the unfolding problem is translated into a Quantum Unconstrained Binary Optimization (QUBO) problem, solvable by D-Wave [181] QA systems.

C.1.1 Challenges in Unfolding

The unfolding problem is mathematically challenging due to the ill-posed nature of the inverse problem it represents.

Consider an unknown probability density function $f(z)$ of a physical observable and the corresponding $g(\mu)$, the expected distribution to be measured in the experimental setup. To model the distortion effects introduced by the measurement process, the detector response function (usually estimated empirically by Monte

Carlo simulations) is defined as

$$r(\mu|z) = m(\mu|z)\ \epsilon(z) \tag{C.1}$$

where $m(\mu|z)$ is the migration (or resolution) function, normalized such that

$$\int m(\mu|z)\ dz = 1, \tag{C.2}$$

and $\epsilon(z)$ is the detection efficiency. This function $r(\mu|z)$ represents the probability to observe μ, including the effect of limited efficiency, given that the true value of the observable was z. Therefore, the expected distribution $g(\mu)$ can be expressed as

$$g(\mu) = \int r(\mu|z) f(z)\, dz \tag{C.3}$$

This convolution represents the *folding* process of the true distribution with the response function, and thus the inverse task of estimating $f(z)$ from $g(\mu)$ is called unfolding (or deconvolution).

Considering a binned version of the same problem, the observable histogram can be represented by an integer-valued vector $\boldsymbol{z} = (z_1, \ldots, z_M)$. Also, the integral in Equation (C.3) breaks into a sum over M bins and the expected number of entries to be observed in bin i becomes:

$$\mu_i = \sum_{j=1}^{M} R_{ij} z_j \qquad \text{or} \qquad \boldsymbol{\mu} = R\boldsymbol{z} \tag{C.4}$$

Thus, the unfolding problem essentially reduces to an inversion of the response matrix R setting the estimator $\hat{\boldsymbol{\mu}} = \boldsymbol{d}$, where $\boldsymbol{d}$ represents the actual observed histogram. However, the matrix R may be nearly singular, leading to solutions that are highly sensitive to statistical fluctuations in the observed data. To mitigate this issue, it is convenient to write the problem using the equivalent log-likelihood maximization formulation and introducing an additional regularization term:

$$\max_{\boldsymbol{z}} \left(\log L(\boldsymbol{z}|\boldsymbol{d}) + \lambda S(\boldsymbol{z})\right) \tag{C.5}$$

The likelihood $L(\boldsymbol{z}|\boldsymbol{d})$ is typically a product of Poisson distributions $P(d_i; \mu_i)$ of the number of entries in bin i and the function $S(\boldsymbol{z})$ represents the regularization term controlled by the λ multiplicative parameter.

The logarithm of Poisson terms product can be replaced by the L_2-norm of the difference between the reconstructed and the observed histogram, which corresponds to take the Gaussian approximation in the limit of a large number of entries. Moreover, the regularization term $S(\boldsymbol{z})$ is usually defined as a second derivative operator (i.e. Tikhonov regularization), related to the average curvature of the true distribution and approximated by finite differences using the discrete Laplacian operator D acting on the true histogram.

Eventually, the log-likelihood maximization problem in Equation (C.5) can be rewritten as a quadratic minimization problem:

$$\min_{\boldsymbol{z}} \left(||R\boldsymbol{z} - \boldsymbol{d}||^2 + \lambda ||D\boldsymbol{z}||^2\right) \tag{C.6}$$

C.1.2 Existing Unfolding Methods

Several approaches have been developed to address the unfolding problem:

1. **Bin-by-bin Correction**: The simplest approach involves applying a correction factor to each bin of the observed distribution independently. This method is easy to implement but often fails to account for correlations between bins, making it unsuitable for complex unfolding problems.

2. **Matrix Inversion**: Direct inversion of the response matrix $\mathbf{R}$ can be used, but this approach is typically unstable due to noise amplification and often requires strong regularization.

3. **Regularized Unfolding**: Techniques such as Tikhonov regularization or iterative methods (e.g., the Richardson-Lucy algorithm) impose smoothness constraints on the solution to obtain stable and physically meaningful results.

4. **Bayesian Unfolding**: This method integrates prior knowledge about the expected true distribution into the unfolding process, using a probabilistic framework to balance the observed data with prior expectations.

Despite these advances, traditional methods often struggle with the computational complexity and instability inherent in unfolding problems, especially as the size and complexity of datasets increase.

C.2 Quantum Annealing as a New Paradigm

Quantum Annealing (QA) offers a fundamentally different approach to solving optimization problems, which has shown promise in addressing challenges like those posed by unfolding. QA operates by exploiting quantum mechanical principles to explore the solution space more efficiently than classical algorithms.

C.2.1 Fundamentals of Quantum Annealing

QA is a quantum algorithm used to solve optimization problems by finding the minimum of a cost function, which is represented as the ground state of a quantum Hamiltonian. Unlike classical optimization methods that may get trapped in local minima, QA leverages quantum tunneling to escape these local minima, potentially finding better solutions more efficiently.

The general form of the Hamiltonian for a Quantum Annealing process is:

$$H(t) = A(t)H_{\text{init}} + B(t)H_{\text{fin}}, \tag{C.7}$$

where H_{init} is the initial Hamiltonian, representing a simple problem with a known ground state, and H_{fin} is the final Hamiltonian, encoding the problem we want to solve. The functions $A(t)$ and $B(t)$ control the evolution from the initial to the final Hamiltonian over time t. According to the adiabatic theorem, if this evolution is slow enough, the system remains in its ground state, and at the end of the process,

it should be in the ground state of H_{fin}, which corresponds to the optimal solution of the problem.

The target problem is often expressed as a Quadratic Unconstrained Binary Optimization (QUBO) problem, where the cost function takes the form:

$$H_{\text{QUBO}} = \sum_i a_i x_i + \sum_{i,j} b_{ij} x_i x_j, \tag{C.8}$$

where $x_i \in \{0, 1\}$ are binary variables, and $a_i, b_{ij} \in \mathbb{R}$ are coefficients that define the problem. QA searches for the configuration of binary variables that minimizes this cost function.

C.2.2 Advantages of Quantum Annealing

QA offers several advantages over classical optimization methods:

- **Global Optimization**: The ability to explore multiple solution paths simultaneously through quantum superposition can lead to finding global minima more effectively than classical algorithms, which are often limited by local minima.
- **Scalability**: QA is particularly well-suited for large-scale optimization problems, as the quantum annealers are designed to handle thousands of interacting variables efficiently.
- **Robustness to Noise**: Quantum systems inherently operate in a regime where noise and decoherence are present, but the QA process can be robust against these factors, making it potentially more stable than classical methods in certain scenarios.

Given these advantages, QA presents a promising approach to the unfolding problem, where the complexity and instability of the solution space pose significant challenges.

C.3 The QUnfold Framework

In order to run simulated or quantum annealing-based algorithms, the first fundamental step is to reformulate the given optimization task as a Quadratic Unconstrained Binary Optimization (QUBO) problem.

C.3.1 QUBO problem formulation

Consider the set of all possible binary vectors $\boldsymbol{x} \in \{0, 1\}^n$ with n being the number of bits. The function $f_Q : \{0, 1\}^n \to \mathbb{R}$ assigns a real value to each binary vector $\boldsymbol{x}$ through

$$f_Q(\boldsymbol{x}) = \boldsymbol{x}^T Q \boldsymbol{x} = \sum_{i=1}^{n} \sum_{j=1}^{n} x_i Q_{ij} x_j \tag{C.9}$$

where $Q \in \mathbb{R}^{n\times n}$ is the so-called QUBO real-valued matrix. The optimization problem consists of finding the binary vector $\boldsymbol{x}^*$ of the function f_Q, namely:

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} f_Q(\boldsymbol{x}) \tag{C.10}$$

With respect to the unfolding problem, it is straightforward to show that, if the linear coefficients vector $\boldsymbol{a} \in \mathbb{R}^M$ and the quadratic interaction matrix $B \in \mathbb{R}^{M\times M}$ are defined as

$$\begin{aligned} \boldsymbol{a} &= -2R^T\boldsymbol{d} \\ B &= R^TR + \lambda D^TD \end{aligned} \tag{C.11}$$

the minimization problem of Equation (C.6) can be rewritten as

$$\min_{\boldsymbol{z}} \left(\boldsymbol{a}\ \boldsymbol{z} + \boldsymbol{z}^T B\ \boldsymbol{z}\right) \tag{C.12}$$

This is defining the unfolding formulated as an optimization problem over the integer-valued vector $\boldsymbol{z}$, representing the underlying true histogram.

However, to get a QUBO function in the form of Equation (C.9), each integer variable z_i (i.e. number of entries in bin i) must be encoded by a binary vector (or bitstring) $\boldsymbol{x}_i$ of length l_i. In order to do so, we follow the encoding strategy proposed in [182], generalized for the case of different resolutions for each bin. Thereby, different resolutions correspond to different lengths of the precision vectors $\boldsymbol{p}_i = (2^0, 2^1, \ldots, 2^{l_i-1})$ for each bin $i = 1, \ldots, M$. Then, each integer variable is encoded by

$$z_i = \boldsymbol{p}_i \cdot \boldsymbol{x}_i \tag{C.13}$$

The linear and quadratic terms in Equation (C.12) can now be written as

$$\boldsymbol{a}\cdot\boldsymbol{z} = \sum_{i=1}^{M} a_i\, \boldsymbol{p}_i\cdot\boldsymbol{x}_i = \boldsymbol{a}_{\text{bin}}\ \boldsymbol{x} \tag{C.14}$$

$$\boldsymbol{z}^T B\boldsymbol{z} = \sum_{i=1}^{M}\sum_{i=j}^{M} \boldsymbol{x}_i^T\ \boldsymbol{p}_i^T\ B_{ij}\ \boldsymbol{p}_j\ \boldsymbol{x}_j = \boldsymbol{x}^T B_{\text{bin}}\ \boldsymbol{x} \tag{C.15}$$

where $\boldsymbol{x} \in \{0,1\}^n$ is the full vector of binary variables, built concatenating all the bitstrings $\boldsymbol{x}_i$. The vector $\boldsymbol{a}_{\text{bin}} \in \mathbb{R}^n$ and matrix $B_{\text{bin}} \in \mathbb{R}^{n\times n}$ represent respectively the linear coefficients and the quadratic interactions of the binary variables defined in the encoding procedure. Note that the scaling of the required number of bits $n = \sum_{i=1}^{M} l_i$ is linear with the number of bins and logarithmic with the number of entries in each bin.

Finally, the matrix $Q \in \mathbb{R}^{n\times n}$ defining the QUBO problem function of Equation (C.9) is constructed by simply summing the vector of linear coefficients $\boldsymbol{a}_{\text{bin}}$ to the diagonal of the interaction matrix B_{bin}.

C.3.2 Implementation in QUnfold

The QUnfold package is an open-source Python library designed to implement the QA-based unfolding approach described above. It is built on top of the D-Wave Ocean SDK [183], which provides access to D-Wave's quantum annealers, and NumPy, which is used for linear algebra computations.

QUnfold is designed for ease of use and integration with existing HEP analysis workflows. Users can define the response matrix and observed data, set the regularization parameters, and choose between different solvers (simulated annealing, quantum annealing, or hybrid approaches).

The package is publicly available on GitHub at [180], making it accessible to the broader scientific community. It is designed to handle real-scale HEP data, making it suitable for use in large-scale experiments such as those conducted at the LHC.

C.4 Validation and Performance Evaluation

C.4.1 Simulated Data Analysis

To validate the effectiveness of QUnfold, we applied it to simulated data for the $t\bar{t}$ process, a well-known physics process studied extensively at the LHC. The process involves the production of a top quark and an anti-top quark, which then decay into leptons and b-jets:

$$pp \to t\bar{t} \to W^+ b W^- \bar{b} \to l^+ \nu_l \, l^- \bar{\nu}_l b\bar{b}, \tag{C.16}$$

where p are protons, W are weak bosons, b are bottom quarks, l are leptons, and ν are neutrinos.

We generated approximately 2.5 million events using the Madgraph generator [184] to obtain the truth-level distribution and used the Delphes framework [133] to simulate the detector response, providing the measured data. We then applied QUnfold to recover the true distribution from the measured data.

C.4.2 Comparison with Classical Methods

The performance of QUnfold was benchmarked against two classical unfolding methods: Matrix Inversion (MI) and Iterative Bayesian Unfolding (IBU), both of which are implemented in the RooUnfold framework [185]. We conducted toy Monte Carlo (MC) experiments to compute the covariance matrix and evaluate the quality of the unfolding using the χ^2 test statistic.

The comparison focused on key observables such as the transverse momenta (p_T) and the pseudo-rapidity (η) of the leading and sub-leading leptons. The results, shown in Figure C.1, demonstrate that QUnfold performs comparably to or better than classical methods, particularly in scenarios where the classical methods struggle with instability or noise.

C.4.3 Computational Performance

In addition to accuracy, we evaluated the computational performance of QUnfold, particularly its scalability and resource requirements. Quantum Annealing was tested on D-Wave's Advantage QPU, which currently features over 5000 qubits. The hybrid solver was also tested, combining classical and quantum resources to optimize performance.

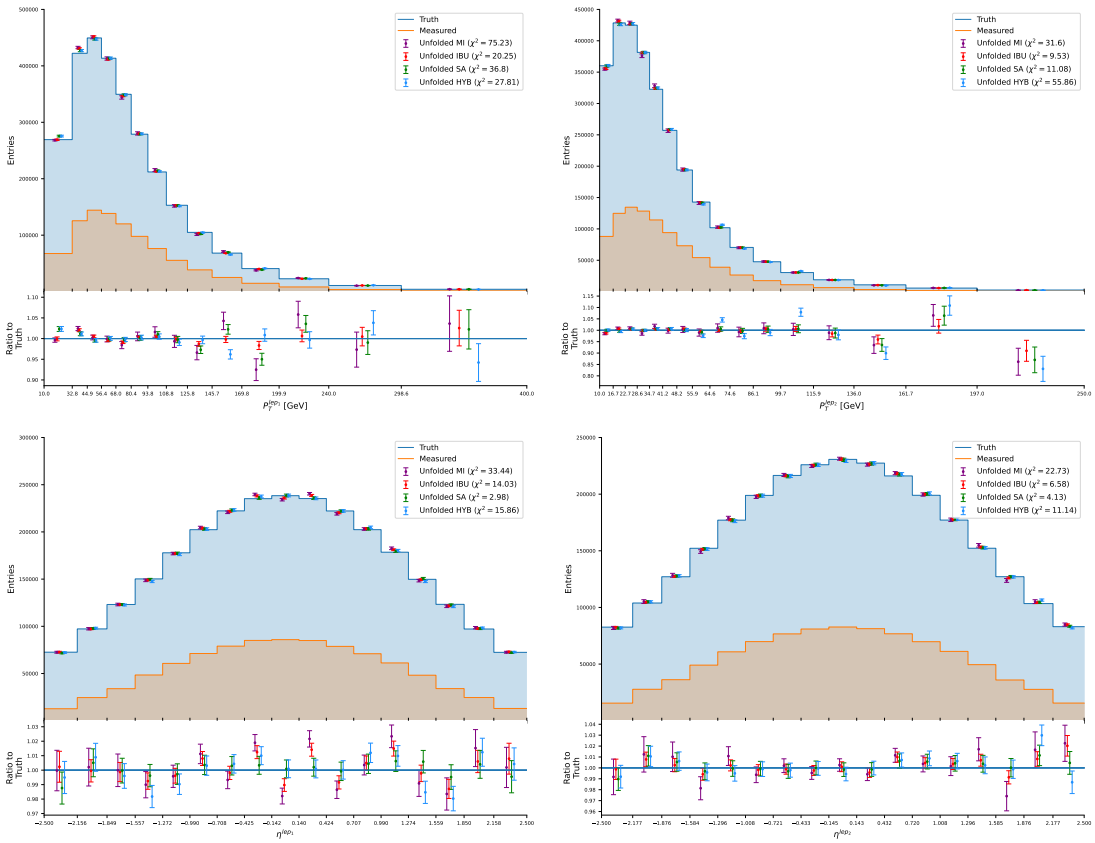


Figure C.1: Comparison of different unfolding methods for key observables.

The results indicate that while pure quantum solutions offer significant potential, hybrid approaches currently provide a more stable solution due to the limitations of current quantum hardware. However, as quantum hardware continues to evolve, we expect the performance of pure quantum solutions to improve substantially.

C.5 Conclusion and Future Work

In this paper, we introduced QUnfold, a novel approach to statistical unfolding in High-Energy Physics using Quantum Annealing. By reformulating the unfolding problem as a QUBO problem, QUnfold leverages the power of quantum computing to address the inherent challenges of unfolding, such as instability and noise sensitivity.

Our validation on simulated data for the $t\bar{t}$ process shows that QUnfold performs on par with, or in some cases better than, traditional unfolding methods. The computational performance, particularly in hybrid configurations, also demonstrates the practicality of using quantum approaches in real-scale HEP analyses.

Future work will focus on further optimizing the binarization process and the QUBO matrix pre-conditioning to enhance the annealing performance. We also plan to explore fully quantum approaches on small data samples to assess the pure power of quantum annealing.

The continued evolution of quantum hardware, particularly improvements in qubit quality, will undoubtedly enhance the solutions presented in this work, paving the way for more widespread adoption of quantum computing techniques in HEP and beyond.

Finally, a Bachelor's thesis was supervised on the topic of new applications of QUnfold and testing its features [186].

Bibliography

[1] O. S. Brüning, P. Collier, P. Lebrun, *et al.*, *LHC Design Report*. Geneva: CERN, 2004, vol. 1. DOI: 10.5170/CERN-2004-003-V-1.

[2] O. S. Brüning, P. Collier, P. Lebrun, *et al.*, *LHC Design Report*. Geneva: CERN, 2004, vol. 2. DOI: 10.5170/CERN-2004-003-V-2.

[3] M. Benedikt, P. Collier, V. Mertens, *et al.*, *LHC Design Report*. Geneva: CERN, 2004, vol. 3. DOI: 10.5170/CERN-2004-003-V-3.

[4] European Organization for Nuclear Research (CERN). (2024). Pulling together: Superconducting electromagnets, [Online]. Available: https://home.web.cern.ch/science/engineering/vacuum-empty-interstellar-space.

[5] European Organization for Nuclear Research (CERN). (2024). A vacuum as empty as interstellar space, [Online]. Available: https://home.web.cern.ch/science/engineering/pulling-together-superconducting-electromagnets.

[6] L. Evans and P. Bryant, "LHC Machine", *Journal of Instrumentation*, vol. 3, no. 08, Aug. 2008. DOI: 10.1088/1748-0221/3/08/S08001.

[7] J. L. Caron. (1998). Cross section of LHC dipole, [Online]. Available: https://cds.cern.ch/record/841539.

[8] European Organization for Nuclear Research (CERN). (2024). Accelerating: Radiofrequency cavities, [Online]. Available: https://home.web.cern.ch/science/engineering/accelerating-radiofrequency-cavities.

[9] K. Aamodt, A. Abrahantes Quintana, R. Achenbach, *et al.* [ALICE Collaboration], "The ALICE experiment at the CERN LHC", *Journal of Instrumentation*, vol. 3, no. 08, Aug. 2008. DOI: 10.1088/1748-0221/3/08/s08002.

[10] G. Aad, E. Abat, J. Abdallah, *et al.* [ATLAS Collaboration], "The ATLAS experiment at the CERN large hadron collider", *Journal of Instrumentation*, vol. 3, no. 08, Aug. 2008. DOI: 10.1088/1748-0221/3/08/s08003.

[11] A. Augusto Alves Jr., L. M. Andrade Filho, A. F. Barbosa, *et al.* [LHCb Collaboration], "The LHCb detector at the LHC", *Journal of Instrumentation et al.*, vol. 3, no. 08, S08005, Aug. 2008. DOI: 10.1088/1748-0221/3/08/s08005.

[12] The LHCf Collaboration, "The LHCf detector at the CERN large hadron collider", *Journal of Instrumentation*, vol. 3, no. 08, Aug. 2008. DOI: 10.1088/1748-0221/3/08/s08006.

[13] The TOTEM Collaboration, *Journal of Instrumentation*, vol. 3, no. 08, Aug. 2008. DOI: 10.1088/1748-0221/3/08/s08007.

[14] The CMS Collaboration. (2024). CMS Luminosity public plots, [Online]. Available: https://twiki.cern.ch/twiki/bin/view/CMSPublic/LumiPublicResults.

[15] G. Apollinari, I. Béjar Alonso, O. Brüning, *et al.*, "High-Luminosity Large Hadron Collider (HL-LHC)", Tech. Rep. CERN-2017-007-M, 2017. DOI: 10.23731/CYRM-2017-004.

[16] The HL-LHC Project. [Online]. Available: https://hilumilhc.web.cern.ch/content/hl-lhc-project.

[17] P. Campana, M. Klute, and P. Wells, "Physics goals and experimental challenges of the proton–proton high-luminosity operation of the lhc", *Annual Review of Nuclear and Particle Science*, vol. 66, no. 1, 2016. DOI: 10.1146/annurev-nucl-102115-044812.

[18] The CMS Collaboration, "The CMS experiment at the CERN LHC", *Journal of Instrumentation*, vol. 3, no. 08, Aug. 2008. DOI: 10.1088/1748-0221/3/08/s08004.

[19] The CMS Collaboration, "CMS, the Compact Muon Solenoid: Technical proposal", Dec. 1994. [Online]. Available: http://cds.cern.ch/record/290969.

[20] M. Della Negra *et al.* [CMS Collaboration], "CMS: The Compact Muon Solenoid: Letter of intent for a general purpose detector at the LHC", Oct. 1992.

[21] S. Morović, "Cms detector: Run 3 status and plans for phase-2", *ArXiv e-prints*, 2023. arXiv: 2309.02256 [hep-ex].

[22] G. Pásztor [CMS Collaboration], "The Phase-2 Upgrade of the CMS Detector", *PoS*, vol. LHCP2022, p. 045, 2023. DOI: 10.22323/1.422.0045.

[23] The Tracker Group [CMS Collaboration], "The CMS Phase-1 Pixel Detector Upgrade", CERN, Tech. Rep. CMS-NOTE-2020-005, 2020. [Online]. Available: https://cds.cern.ch/record/2745805.

[24] The CMS Collaboration, "The CMS electromagnetic calorimeter project: Technical Design Report", Tech. Rep. CERN-LHCC-97-033, 1997. [Online]. Available: https://cds.cern.ch/record/349375.

[25] The CMS Collaboration, "Energy calibration and resolution of the CMS electromagnetic calorimeter in pp collisions at $\sqrt{s} = 7$ TeV", *Journal of Instrumentation*, vol. 8, no. 09, Sep. 2013. DOI: 10.1088/1748-0221/8/09/P09009.

[26] The CMS Collaboration, "The CMS hadron calorimeter project: Technical Design Report", Tech. Rep. CERN-LHCC-97-031, 1997. [Online]. Available: https://cds.cern.ch/record/357153.

[27] S. Abdullin, V. Abramov, B. Acharya, *et al.* [CMS Collaboration], "The CMS barrel calorimeter response to particle beams from 2 to 350 GeV/c", *The European Physical Journal C*, vol. 60, no. 3, pp. 359–373, 2009, ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-009-0959-5.

[28] J. G. Layter [CMS Collaboration], "The CMS muon project", Tech. Rep. CERN-LHCC-97-032, 1997. [Online]. Available: https://cds.cern.ch/record/343814.

[29] J. Hauser [CMS Collaboration], "Cathode strip chambers for the cms endcap muon system", *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 384, no. 1, pp. 207–210, 1996, ISSN: 0168-9002. DOI: 10.1016/S0168-9002(96)00905-9.

[30] G. Wrochna, "The RPC system for the CMS experiment at LHC", in *3rd International Workshop on Resistive Plate Chambers and Related Detectors (RPC 95)*, 1995, pp. 63–77.

[31] A. Tapper and D. Acosta [CMS Collaboration], "CMS Technical Design Report for the Level-1 Trigger Upgrade", Tech. Rep. CERN-LHCC-2013-011, Jun. 2013. [Online]. Available: https://cds.cern.ch/record/1556311.

[32] A. Sirunyan, A. Tumasyan, W. Adam, *et al.* [CMS Collaboration], "Performance of the CMS Level-1 trigger in proton-proton collisions at $\sqrt{s} = 13$ TeV", *Journal of Instrumentation*, vol. 15, no. 10, Aug. 2020. DOI: 10.1088/1748-0221/15/10/P10017.

[33] R. Ardino, C. Deldicque, M. Dobson, *et al.* [CMS Collaboration], "A 40 mhz level-1 trigger scouting system for the cms phase-2 upgrade", *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 1047, 2023, ISSN: 0168-9002. DOI: 10.1016/j.nima.2022.167805.

[34] S. Cittolin, A. Rácz, and P. Sphicas [CMS Collaboration], "CMS The TriDAS Project: Technical Design Report, Volume 2: Data Acquisition and High-Level Trigger", Tech. Rep. CERN-LHCC-2002-026, 2002. [Online]. Available: http://cds.cern.ch/record/578006.

[35] J.-M. André, U. Behrens, J. Branson, *et al.* [CMS Collaboration], "The cms event-builder system for lhc run 3 (2021-23)", *EPJ Web Conf.*, vol. 214, 2019. DOI: 10.1051/epjconf/201921401006.

[36] A. Sirunyan, A. Tumasyan, W. Adam, *et al.* [CMS Collaboration], "Particle-flow reconstruction and global event description with the cms detector", *Journal of Instrumentation*, vol. 12, no. 10, Oct. 2017, ISSN: 1748-0221. DOI: 10.1088/1748-0221/12/10/p10003.

[37] M. Cacciari, G. P. Salam, and G. Soyez, "The anti-k_t jet clustering algorithm", *Journal of High Energy Physics*, vol. 2008, no. 04, Apr. 2008, ISSN: 1029-8479. DOI: 10.1088/1126-6708/2008/04/063.

[38] Contardo, Didier and Ball, Austin [CMS Collaboration], "A MIP Timing Detector for the CMS Phase-2 Upgrade", CERN, Tech. Rep. CERN-LHCC-2019-003, Mar. 2019. [Online]. Available: https://cds.cern.ch/record/2667167.

[39] D. Contardo and A. Ball [CMS Collaboration], "The Phase-2 Upgrade of the CMS Endcap Calorimeter", CERN, Tech. Rep. CERN-LHCC-2017-023, Nov. 2017. [Online]. Available: https://cds.cern.ch/record/2293646.

[40] D. Contardo and A. Ball [CMS Collaboration], "The Phase-2 Upgrade of the CMS Muon Detectors", CERN, Tech. Rep. CERN-LHCC-2017-012, Sep. 2017. [Online]. Available: https://cds.cern.ch/record/2283189.

[41] D. Contardo and A. Ball [CMS Collaboration], "The Phase-2 Upgrade of the CMS Barrel Calorimeters", CERN, Tech. Rep. CERN-LHCC-2017-011, Sep. 2017. [Online]. Available: https://cds.cern.ch/record/2283187.

[42] D. Contardo and A. Ball [CMS Collaboration], "The Phase-2 Upgrade of the CMS Tracker", CERN, Tech. Rep. CERN-LHCC-2017-009, Jun. 2017. [Online]. Available: https://cds.cern.ch/record/2272264.

[43] F. Hartmann and A. Ball [CMS Collaboration], "The Phase-2 Upgrade of the CMS Level-1 Trigger", CERN, Tech. Rep. CERN-LHCC-2020-0041, Apr. 2020. [Online]. Available: http://cds.cern.ch/record/2714892.

[44] S. Hauck and A. DeHon, *Reconfigurable computing the theory and practice of FPGA-based computation*, 1st edition., ser. Systems on silicon. Boston, Massachusetts, USA: Morgan Kaufmann, 2008, ISBN: 0-12-370522-3.

[45] B. H.-G. John Crowe, *Introduction to Digital Electronics*, ser. Essential Electronics Series. Oxford, Oxfordshire, UK: Butterworth - Heinemann, 1998, ISBN: 9780340645703.

[46] N. Botros and N. Botros, *HDL with Digital Design*. Herndon, Virginia, USA: Mercury Learning & Information, 2015, ISBN: 9781938549816.

[47] P. Coussy and A. Morawiec, *High-Level Synthesis: from Algorithm to Digital Circuit*. New York City, New York State, USA: Springer Publishing Company, 2008, ISBN: 1402085877.

[48] R. Kastner, J. Matai, and S. Neuendorffer, *Parallel Programming for FPGAs*. 2018. arXiv: 1805.03648.

[49] Advanced Micro Devices Inc. (2024). Vitis high-level synthesis tool, [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/high-level-design.html.

[50] P. Horowitz and W. Hill, *The art of electronics; 3rd ed.* Cambridge, Cambridgeshire, UK: Cambridge University Press, 2015, ISBN: 9780521809269.

[51] Khronos OpenCL Working Group, *The OpenCL Specification*. 2024. [Online]. Available: https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.

[52] The TIOBE organization. (2024). TIOBE Index for May 2024, [Online]. Available: https://www.tiobe.com/tiobe-index.

[53] Python Software Foundation. (2024). Python programming language, [Online]. Available: https://www.python.org.

[54] Advanced Micro Devices Inc. (2024). The PYNQ Documentation, [Online]. Available: https://pynq.readthedocs.io/en/latest.

[55] A. L. Samuel, "Some studies in machine learning using the game of checkers. i", in *Computer Games I*. New York City, New York State, USA: Springer Publishing Company, 1988, pp. 335–365, ISBN: 978-1-4613-8716-9. DOI: 10.1007/978-1-4613-8716-9_14.

[56] M. Lorusso, "Fpga implementation of muon momentum assignment with machine learning at the cms level-1 trigger", Master's Thesis, 2021. [Online]. Available: https://amslaurea.unibo.it/23211/.

[57] F. Wick, U. Kerzel, M. Hahn, *et al.*, "Demand forecasting of individual probability density functions with machine learning", *Operations Research Forum*, vol. 2, no. 3, Jul. 2021, ISSN: 2662-2556. DOI: 10.1007/s43069-021-00079-8.

[58] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. Sebastopol, California, USA: O'Reilly Media, 2019, ISBN: 978-1-4920-3264-9.

[59] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org.

[60] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger", *ArXiv e-prints*, 2016. arXiv: 1612.08242.

[61] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 2018. [Online]. Available: https://mitpress.mit.edu/9780262039246/.

[62] T. M. Mitchell, *Machine learning*. New York City, New York State, USA: McGraw-hill, 1997, ISBN: 0070428077. [Online]. Available: http://www.cs.cmu.edu/~tom/mlbook.html.

[63] H. B. Curry, "The method of steepest descent for non-linear minimization problems", *Quarterly of Applied Mathematics*, vol. 2, no. 3, pp. 258–261, 1944. DOI: https://doi.org/10.1090/qam/10667.

[64] The MathWorks, Inc. (2024). What is overfitting?, [Online]. Available: https://www.mathworks.com/discovery/overfitting.

[65] Cmglee. (2024). Receiver Operating Characteristic (ROC) curve, [Online]. Available: https://commons.wikimedia.org/wiki/File:Roc_curve.svg.

[66] K. Thulasiraman and M. N. S. Swamy, *Graphs: Theory and Algorithms*. New York City, New York State, USA: Wiley-Interscience, 1992, ISBN: 978-0-4715-1356-8.

[67] A. L. Maas, A. Y. Hannun, and A. Y. Ng. (2013). Rectifier nonlinearities improve neural network acoustic models, [Online]. Available: https://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf.

[68] F. Chollet. (2016). Building autoencoders in keras, [Online]. Available: https://blog.keras.io/building-autoencoders-in-keras.

[69] J. Galbraith, I. Moustaki, D. Bartholomew, and F. Steele, *The Analysis and Interpretation of Multivariate Data for Social Scientists*. Milton Park, Oxfordshire, UK: Taylor & Francis, 2002, ISBN: 978-1-5848-8295-4.

[70] I. T. Jolliffe and J. Cadima, "Principal component analysis: A review and recent developments", *Philos. Trans. A Math. Phys. Eng. Sci.*, vol. 374, no. 2065, Apr. 2016. DOI: 10.1098/rsta.2015.0202.

[71] M. Vassallo, "Machine learning for quantum error mitigation", Bachelor's Thesis, 2024. [Online]. Available: http://amslaurea.unibo.it/32365.

[72] C. L. Li, K. Sohn, J. Yoon, and T. Pfister, "Cutpaste: Self-supervised learning for anomaly detection and localization", *ArXiv e-prints*, 2021. arXiv: 2104.04015.

[73] R. Chalapathy and S. Chawla, "Deep learning for anomaly detection: A survey", *ArXiv e-prints*, 2019. arXiv: 1901.03407.

[74] F. Scarselli, M. Gori, A. C. Tsoi, *et al.*, "The graph neural network model", *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009. DOI: 10.1109/TNN.2008.2005605.

[75] K. Thulasiraman and M. N. S. Swamy, *Graphs: Theory and Algorithms*. New York City, New York State, USA: John Wiley & Sons, 1992, ISBN: 978-1-1180-3310-4. DOI: 10.1002/9781118033104.

[76] Google LLC. (2021). Google trends, [Online]. Available: https://trends.google.com/trends.

[77] The PyTorch Foundation. (2024). PyTorch Website, [Online]. Available: https://pytorch.org/.

[78] The Linux Foundation. (2024), [Online]. Available: https://www.linuxfoundation.org.

[79] Google LLC. (2024). TensorFlow Website, [Online]. Available: https://www.tensorflow.org.

[80] Google DeepMind. (2024), [Online]. Available: https://deepmind.google.

[81] Google Cloud Platform. (2024). Introduction to cloud tpu, [Online]. Available: https://cloud.google.com/tpu/docs/intro-to-tpu.

[82] L. Giommi, "Prototype of machine learning as a service for cms physics in signal vs background discrimination", PhD thesis. [Online]. Available: http://amslaurea.unibo.it/15803/.

[83] The Keras Team. (2024). Keras documentation, [Online]. Available: https://keras.io.

[84] D. Guest, K. Cranmer, and D. Whiteson, "Deep learning and its application to lhc physics", *Annual Review of Nuclear and Particle Science*, vol. 68, pp. 161–181, 2018, ISSN: 15454134. DOI: 10.1146/annurev-nucl-101917-021019.

[85] D. Bourilkov, "Machine and deep learning applications in particle physics", *International Journal of Modern Physics A*, vol. 34, no. 35, Dec. 2019. DOI: 10.1142/s0217751x19300199. arXiv: 1912.08245.

[86] P. Abreu, W. Adam, T. Adye, *et al.* [Delphi Collaboration], "Classification of the hadronic decays of the Z0 into b and c quark pairs using a neural network", *Physics Letters B*, vol. 295, no. 3, pp. 383–395, 1992, ISSN: 0370-2693. DOI: 10.1016/0370-2693(92)91580-3.

[87] J. Köhne, J. Fent, W. Fröchtenicht, *et al.*, "Realization of a second level neural network trigger for the H1 experiment at HERA", *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 389, no. 1, pp. 128–133, 1997, ISSN: 0168-9002. DOI: 10.1016/S0168-9002(97)00062-4.

[88] V. Abazov, B. Abbott, A. Abdesselam, *et al.* [DØ Collaboration], "Search for single top quark production at DØ using neural networks", *Physics Letters B*, vol. 517, no. 3, pp. 282–294, 2001, ISSN: 0370-2693. DOI: 10.1016/S0370-2693(01)01009-7.

[89] T. Aaltonen, J. Adelman, B. Álvarez González, *et al.* [CDF Collaboration], "Search for the higgs boson using neural networks in events with missing energy and b-quark jets in $p\bar{p}$ collisions at $\sqrt{s} = 1.96$ TeV", *Phys. Rev. Lett.*, vol. 104, p. 141 801, 14 Apr. 2010. DOI: 10.1103/PhysRevLett.104.141801.

[90] V. Khachatryan, A. M. Sirunyan, A. Tumasyan, *et al.* [CMS Collaboration], "Observation of the diphoton decay of the higgs boson and measurement of its properties", *The European Physical Journal C*, vol. 74, no. 10, Oct. 2014, ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-014-3076-z. [Online]. Available: http://dx.doi.org/10.1140/epjc/s10052-014-3076-z.

[91] P. Baldi, P. Sadowski, and D. Whiteson, "Searching for exotic particles in high-energy physics with deep learning", *Nature Communications*, vol. 5, no. 1, Jul. 2014, ISSN: 2041-1723. DOI: 10.1038/ncomms5308.

[92] M. Aaboud, G. Aad, B. Abbott, *et al.* [ATLAS Collaboration], "Performance of the atlas track reconstruction algorithms in dense environments in lhc run 2", *The European Physical Journal C*, vol. 77, no. 10, Oct. 2017, ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-017-5225-7.

[93] A. Salzburger [ATLAS Collaboration], "Optimisation of the atlas track reconstruction software for run-2", *Journal of Physics: Conference Series*, vol. 664, no. 7, p. 072 042, Dec. 2015. DOI: 10.1088/1742-6596/664/7/072042.

[94] J. Miguens, "The ATLAS Run-2 Trigger: Design, Menu, Performance and Operational Aspects", *PoS*, vol. ICHEP2016, 2017. DOI: 10.22323/1.282.0244.

[95] J. Cogan, M. Kagan, E. Strauss, and A. Schwartzman, "Jet-images: Computer vision inspired techniques for jet tagging", *Journal of High Energy Physics*, vol. 2015, no. 2, Feb. 2015, ISSN: 1029-8479. DOI: 10.1007/jhep02(2015)118.

[96] The CMS Collaboration. (2017). Performance of heavy flavour identification algorithms in proton-proton collisions at 13 TeV at the CMS experiment, [Online]. Available: https://cds.cern.ch/record/2263801.

[97] T. Diotalevi. (2020). High-pt muon refit with ml, [Online]. Available: https://indico.cern.ch/event/973558/.

[98] M. Paganini, L. de Oliveira, and B. Nachman, "Accelerating science with generative adversarial networks: An application to 3d particle showers in multilayer calorimeters", *Phys. Rev. Lett.*, vol. 120, 4 Jan. 2018. DOI: 10.1103/PhysRevLett.120.042003.

[99] L. de Oliveira, M. Paganini, and B. Nachman, "Learning particle physics by example: Location-aware generative adversarial networks for physics synthesis", *Computing and Software for Big Science*, Sep. 2017, ISSN: 2510-2044. DOI: 10.1007/s41781-017-0004-6.

[100] T. Diotalevi, D. Bonacorsi, A. Falabella, *et al.*, "Collection and harmonization of system logs and prototypal Analytics services with the Elastic (ELK) suite at the INFN-CNAF computing centre", *PoS*, vol. ISGC2019, 2019. DOI: 10.22323/1.351.0027.

[101] L. Giommi, D. Bonacorsi, T. Diotalevi, *et al.*, "Towards Predictive Maintenance with Machine Learning at the INFN-CNAF computing centre", *PoS*, vol. ISGC2019, 2019. DOI: 10.22323/1.351.0003.

[102] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression", *ArXiv e-prints*, 2017. arXiv: 1710.01878 [stat.ML].

[103] TensorFlow API Documentation. (2024). Pruning comprehensive guide, [Online]. Available: https://www.tensorflow.org/model_optimization/guide/pruning/comprehensive_guide.

[104] A. Taylor, "The basics of FPGA mathematics", *Xilinx Xcell Journal*, no. 80, 2012. [Online]. Available: https://www.xilinx.com/publications/archives/xcell/Xcell80.pdf.

[105] E. Meller, A. Finkelstein, U. Almog, and M. Grobman, "Same, same but different: Recovering neural network quantization error through weight factorization", in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97, PMLR, Jun. 2019, pp. 4486–4495. [Online]. Available: http://proceedings.mlr.press/v97/meller19a.html.

[106] C. N. Coelho Jr., A. Kuusela, S. Li, *et al.*, "Automatic deep heterogeneous quantization of Deep Neural Networks for ultra low-area, low-latency inference on the edge at particle colliders", *ArXiv e-prints*, Jun. 2020. arXiv: 2006.10159 [physics.ins-det].

[107] QKeras Github Repository. (2024), [Online]. Available: https://github.com/google/qkeras.

[108] C. R. Harris, K. J. Millman, S. J. van der Walt, *et al.*, "Array programming with numpy", *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020, ISSN: 1476-4687. DOI: 10.1038/s41586-020-2649-2.

[109] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network", *ArXiv e-prints*, 2015. arXiv: 1503.02531 [stat.ML].

[110] C. Buciluundefined, R. Caruana, and A. Niculescu-Mizil, "Model compression", in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06, New York, NY, USA: Association for Computing Machinery, 2006, pp. 535–541, ISBN: 1595933395. DOI: 10.1145/1150402.1150464.

[111] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey", *International Journal of Computer Vision*, vol. 129, no. 6, pp. 1789–1819, Mar. 2021, ISSN: 1573-1405. DOI: 10.1007/s11263-021-01453-z.

[112] A. Romero, N. Ballas, S. E. Kahou, *et al.*, "Fitnets: Hints for thin deep nets", *ArXiv e-prints*, 2015. arXiv: 1412.6550 [cs.LG].

[113] J. Liu, D. Wen, H. Gao, *et al.*, "Knowledge representing: Efficient, sparse representation of prior knowledge for knowledge distillation", *ArXiv e-prints*, 2019. arXiv: 1911.05329 [cs.CV].

[114] Z. Huang and N. Wang, "Like what you like: Knowledge distill via neuron selectivity transfer", *ArXiv e-prints*, 2017. arXiv: 1707.01219 [cs.CV].

[115] J. Yim, D. Joo, J. Bae, and J. Kim, "A gift from knowledge distillation: Fast optimization, network minimization and transfer learning", in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. DOI: 10.1109/CVPR.2017.754.

[116] W. Park, D. Kim, Y. Lu, and M. Cho, "Relational knowledge distillation", *ArXiv e-prints*, 2019. arXiv: 1904.05068 [cs.CV].

[117] S.-I. Mirzadeh, M. Farajtabar, A. Li, *et al.*, "Improved knowledge distillation via teacher assistant", *ArXiv e-prints*, 2019. arXiv: 1902.03393 [cs.LG].

[118] Y. Zhang, T. Xiang, T. M. Hospedales, and H. Lu, "Deep mutual learning", *ArXiv e-prints*, 2017. arXiv: 1706.00384 [cs.CV].

[119] L. Zhang, J. Song, A. Gao, *et al.*, "Be your own teacher: Improve the performance of convolutional neural networks via self distillation", *ArXiv e-prints*, 2019. arXiv: 1905.08094 [cs.LG].

[120] C. Yang, L. Xie, C. Su, and A. L. Yuille, "Snapshot distillation: Teacher-student optimization in one generation", *ArXiv e-prints*, 2018. arXiv: 1812.00123 [cs.CV].

[121] T. Furlanello, Z. C. Lipton, M. Tschannen, *et al.*, "Born again neural networks", *ArXiv e-prints*, 2018. arXiv: 1805.04770 [stat.ML].

[122] J. Duarte *et al.*, "Fast inference of deep neural networks in FPGAs for particle physics", *Journal of Instrumentation*, vol. 13, no. 07, 2018. DOI: 10.1088/1748-0221/13/07/P07027. arXiv: 1804.06913 [physics.ins-det].

[123] FastML Team, *Hls4ml*, version v0.8.1, Dec. 2023. DOI: 10.5281/zenodo.10407911.

[124] G. Kasieczka, B. Nachman, D. Shih, *et al.*, "The lhc olympics 2020 a community challenge for anomaly detection in high energy physics", *Reports on Progress in Physics*, vol. 84, no. 12, Dec. 2021, ISSN: 1361-6633. DOI: 10.1088/1361-6633/ac36b9.

[125] T. Aarrestad, M. van Beekveld, M. Bona, *et al.*, "The dark machines anomaly score challenge: Benchmark data and model independent event classification for the large hadron collider", *SciPost Physics*, vol. 12, no. 1, Jan. 2022, ISSN: 2542-4653. DOI: 10.21468/scipostphys.12.1.043.

[126] O. Cerri, T. Q. Nguyen, M. Pierini, *et al.*, "Variational autoencoders for new physics mining at the large hadron collider", *Journal of High Energy Physics*, vol. 2019, no. 5, May 2019, ISSN: 1029-8479. DOI: 10.1007/jhep05(2019)036.

[127] O. Knapp, G. Dissertori, O. Cerri, *et al.*, "Adversarially learned anomaly detection on cms open data: Re-discovering the top quark", *ArXiv e-prints*, 2020. arXiv: 2005.01598 [hep-ex].

[128] F. Poppi, "Is the bell ringing?. Exotica : à l'affût des événements exotiques", no. 46/2010, p. 14, 2010. [Online]. Available: http://cds.cern.ch/record/1306501.

[129] C. N. Coelho Jr., A. Kuusela, S. Li, *et al.*, "A New Map of All the Particles and Forces", *QuantaMagazine*, Oct. 2020. [Online]. Available: https://www.quantamagazine.org/a-new-map-of-the-standard-model-of-particle-physics-20201022.

[130] P. D. Group, P. A. Zyla, R. M. Barnett, *et al.*, "Review of Particle Physics", *Progress of Theoretical and Experimental Physics*, vol. 2020, no. 8, Aug. 2020, ISSN: 2050-3911. DOI: 10.1093/ptep/ptaa104.

[131] R. K. Ellis, W. J. Stirling, and B. R. Webber, *QCD and Collider Physics*. Cambridge, Cambridgeshire, UK: Cambridge University Press, 1996. DOI: 10.1017/CBO9780511628788.

[132] T. Sjöstrand, S. Ask, J. R. Christiansen, *et al.*, "An introduction to pythia 8.2", *Computer Physics Communications*, vol. 191, Jun. 2015, ISSN: 0010-4655. DOI: 10.1016/j.cpc.2015.01.024.

[133] J. de Favereau, C. Delaere, P. Demin, *et al.*, "DELPHES 3: a modular framework for fast simulation of a generic collider experiment", *Journal of High Energy Physics*, vol. 2014, no. 2, Feb. 2014, ISSN: 1029-8479. DOI: 10.1007/jhep02(2014)057.

[134] D. Contardo, M. Klute, J. Mans, *et al.*, "Technical Proposal for the Phase-II Upgrade of the CMS Detector", Tech. Rep., 2015. DOI: 10.17181/CERN.VU8I.D59J.

[135] M. Cacciari, G. P. Salam, and G. Soyez, "Fastjet user manual: (for version 3.0.2)", *The European Physical Journal C*, vol. 72, no. 3, Mar. 2012, ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-012-1896-2.

[136] E. Govorkova, E. Puljak, T. Aarrestad, *et al.*, "Lhc physics dataset for unsupervised new physics detection at 40 mhz", *ArXiv e-prints*, 2021. arXiv: 2107.02157 [physics.data-an].

[137] T. Aarrestad, E. Govorkova, J. Ngadiuba, *et al.*, *Unsupervised New Physics detection at 40 MHz: Training Dataset*, version v2, Zenodo, Oct. 2022. DOI: 10.5281/zenodo.5046428.

[138] B. Diaz, M. Schmaltz, and Y.-M. Zhong, "The leptoquark hunter's guide: Pair production", *Journal of High Energy Physics*, vol. 2017, no. 10, Oct. 2017, ISSN: 1029-8479. DOI: 10.1007/jhep10(2017)097.

[139] T. Aarrestad, E. Govorkova, J. Ngadiuba, *et al.*, *Unsupervised New Physics detection at 40 MHz: LQ $\to$ b tau Signal Benchmark Dataset*, version v2, Zenodo, Oct. 2022. DOI: 10.5281/zenodo.7152599.

[140] T. Aarrestad, E. Govorkova, J. Ngadiuba, *et al.*, *Unsupervised New Physics detection at 40 MHz: A $\to 4\ell$ Signal Benchmark Dataset*, version v2, Zenodo, Oct. 2022. DOI: 10.5281/zenodo.7152590.

[141] T. Aarrestad, E. Govorkova, J. Ngadiuba, *et al.*, *Unsupervised New Physics detection at 40 MHz: $h^0 \to \tau\tau$ Signal Benchmark Dataset*, version v2, Zenodo, Oct. 2022. DOI: 10.5281/zenodo.7152614.

[142] T. Aarrestad, E. Govorkova, J. Ngadiuba, *et al.*, *Unsupervised New Physics detection at 40 MHz: $h^\pm \to \tau\nu$ Signal Benchmark Dataset*, version v2, Zenodo, Oct. 2022. DOI: 10.5281/zenodo.7152617.

[143] E. Govorkova, E. Puljak, T. Aarrestad, *et al.*, "Autoencoders on field-programmable gate arrays for real-time, unsupervised new physics detection at 40 mhz at the large hadron collider", *Nature Machine Intelligence*, vol. 4, no. 2, Feb. 2022, ISSN: 2522-5839. DOI: 10.1038/s42256-022-00441-3.

[144] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[145] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", in *Proceedings of the 32nd International Conference on Machine Learning*, F. Bach and D. Blei, Eds., ser. Proceedings of Machine Learning Research, vol. 37, PMLR, Jul. 2015, pp. 448–456. [Online]. Available: https://proceedings.mlr.press/v37/ioffe15.html.

[146] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *ArXiv e-prints*, 2017. arXiv: 1412.6980 [cs.LG].

[147] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization", *Journal of Machine Learning Research*, vol. 13, no. 10, pp. 281–305, 2012. [Online]. Available: http://jmlr.org/papers/v13/bergstra12a.html.

[148] T. O'Malley, E. Bursztein, J. Long, *et al.*, *Kerastuner*, 2019. [Online]. Available: https://github.com/keras-team/keras-tuner.

[149] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python", *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[150] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of adam and beyond", *ArXiv e-prints*, 2019. arXiv: 1904.09237 [cs.LG].

[151] C. J. Clopper and E. S. Pearson, "The use of confidence or fiducial limits illustrated in the case of the binomial", *Biometrika*, vol. 26, no. 4, pp. 404–413, Dec. 1934, ISSN: 0006-3444. DOI: 10.1093/biomet/26.4.404.

[152] X. Sun and W. Xu, "Fast implementation of delong's algorithm for comparing the areas under correlated receiver operating characteristic curves", *IEEE Signal Processing Letters*, vol. 21, no. 11, pp. 1389–1393, 2014. DOI: 10.1109/LSP.2014.2337313.

[153] P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python", *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: 10.1038/s41592-019-0686-2.

[154] Yandex School of Data Analysis, *Roc_comparison*, 2024. [Online]. Available: https://github.com/yandexdataschool/roc_comparison.

[155] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", *ArXiv e-prints*, 2015. arXiv: 1502.01852 [cs.CV].

[156] M. Lorusso. (2024). Leaky quantized relu fix, [Online]. Available: https://github.com/fastmachinelearning/hls4ml/pull/961.

[157] V. Loncar. (2024). Support negative_slope in quantized_relu, [Online]. Available: https://github.com/fastmachinelearning/hls4ml/pull/987.

[158] G. Cowan, K. Cranmer, E. Gross, and O. Vitells, "Asymptotic formulae for likelihood-based tests of new physics", *The European Physical Journal C*, vol. 71, no. 2, Feb. 2011, ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-011-1554-0.

[159] L. Valente, "A variational autoencoder application for real-time anomaly detection at cms", Master's thesis, Alma Mater Studiorum - University of Bologna, 2023. [Online]. Available: http://amslaurea.unibo.it/28788.

[160] L. Valente, L. Anzalone, M. Lorusso, and D. Bonacorsi, "Joint Variational Auto-Encoder for Anomaly Detection in High Energy Physics", *PoS*, vol. ISGC&HEPiX2023, p. 014, 2023. DOI: 10.22323/1.434.0014.

[161] Advanced Micro Devices, Inc. (2024). Alveo u50 product page, [Online]. Available: https://www.amd.com/en/products/accelerators/alveo/u50/a-u50-p00g-pq-g.html.

[162] Advanced Micro Devices, Inc. (2024). Amd vitis™ integrated design environment, [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-ide.html.

[163] *AMBA AXI Protocol Specification*, ARM Ltd., 2023. [Online]. Available: https://developer.arm.com/documentation/ihi0022/k/?lang=en.

[164] M. Lorusso, D. Bonacorsi, R. Travaglini, *et al.*, "Accelerating Machine Learning inference using FPGAs: the PYNQ framework tested on an AWS EC2 F1 Instance", *PoS*, vol. ICHEP2022, p. 243, 2022. DOI: 10.22323/1.414.0243.

[165] M. Lorusso, D. Bonacorsi, D. Salomoni, and R. Travaglini, "Machine Learning inference using PYNQ environment in a AWS EC2 F1 Instance", *PoS*, vol. ISGC2022, p. 001, 2022. DOI: 10.22323/1.415.0001.

[166] Y. Wang, Y. Sun, Z. Liu, *et al.*, "Dynamic graph cnn for learning on point clouds", *ArXiv e-prints*, 2019. arXiv: 1801.07829.

[167] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation", *ArXiv e-prints*, 2017. arXiv: 1612.00593 [cs.CV].

[168] TensorFlow API Documentation. (2024). TensorFlow 1.x vs TensorFlow 2 - Behaviors and APIs, [Online]. Available: https://www.tensorflow.org/guide/migrate/tf1_vs_tf2.

[169] Amazon Web Services. (2024). Amazon EC2 F1 Instances, [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1.

[170] Amazon Web Services. (2024). Amazon S3 Object Storage, [Online]. Available: https://aws.amazon.com/s3.

[171] Amazon Web Services, *AWS EC2 FPGA Development Kit*, 2023. [Online]. Available: https://github.com/aws/aws-fpga.

[172] R. A. Fisher, *Iris Dataset*, UCI Machine Learning Repository, 1936. DOI: 10.24432/C56C76.

[173] D. Zhang, S. Mishra, E. Brynjolfsson, *et al.*, *The AI Index 2021 Annual Report*. AI Index Steering Committee, Human-Centered AI Institute, Stanford University, Mar. 2021.

[174] National Institute for Nuclear Physics (INFN). (2022). Machine learning techniques with fpga devices for particle physics experiments, [Online]. Available: https://agenda.infn.it/event/15116.

[175] M. Mariotti, D. Magalotti, D. Spiga, and L. Storchi, "The bondmachine, a moldable computer architecture", *Parallel Computing*, vol. 109, 2022, ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2021.102873.

[176] M. Mariotti, L. Storchi, D. Spiga, *et al.*, "The BondMachine toolkit: Enabling Machine Learning on FPGA", *PoS*, vol. ISGC2019, p. 020, 2019. DOI: 10.22323/1.351.0020.

[177] Anaconda Inc. (2024). Anaconda Data Science Platform, [Online]. Available: https://www.anaconda.com.

[178] Advanced Micro Devices, Inc. (2024). AMD Vitis™ AI Software, [Online]. Available: https://www.amd.com/en/products/software/vitis-ai.html.

[179] D. Salomoni, A. Alkhansa, M. Antonacci, *et al.*, "Infn and the evolution of distributed scientific computing in italy", *EPJ Web of Conf.*, vol. 295, 2024. DOI: 10.1051/epjconf/202429510004.

[180] S. Gasperini, M. Lorusso, and G. Bianco, *Qunfold*, 2024. [Online]. Available: https://github.com/Quantum4HEP/QUnfold.

[181] D-Wave Quantum Inc. (2024). D-Wave Systems, [Online]. Available: https://www.dwavesys.com.

[182] B. Krakoff, S. M. Mniszewski, and C. F. A. Negre, "Controlled precision QUBO-based algorithm to compute eigenvectors of symmetric matrices", *PLOS ONE*, vol. 17, no. 5, I. Hen, Ed., May 2022, ISSN: 1932-6203. DOI: 10.1371/journal.pone.0267954.

[183] D-Wave Quantum Inc. (2024). D-Wave Ocean Software Documentation, [Online]. Available: https://docs.ocean.dwavesys.com.

[184] J. Alwall, R. Frederix, S. Frixione, *et al.*, "The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations", *Journal of High Energy Physics*, vol. 2014, no. 7, Jul. 2014, ISSN: 1029-8479. DOI: 10.1007/jhep07(2014)079.

[185] L. Brenner, P. Verschuuren, R. Balasubramanian, *et al.*, "Comparison of unfolding methods using roofitunfold", *ArXiv e-prints*, 2020. arXiv: 1910.14654.

[186] V. Brugnami, "Qunfold a quantum annealing-based unfolding tool for hep: A case study on entangled tt? pair production at the atlas experiment", Bachelor's Thesis, 2024. [Online]. Available: http://amslaurea.unibo.it/32515.