Alma Mater Studiorum · University of Bologna

School of Science
Department of Physics and Astronomy
Master Degree in Physics

FPGA implementation of Muon Momentum assignment with Machine Learning at the CMS Level-1 Trigger

Supervisor:
Prof. Daniele Bonacorsi

Co-supervisor:
Dr. Riccardo Travaglini
Dr. Carlo Battilana
Dr. Tommaso Diotalevi

Submitted by:
Marco Lorusso

Academic Year 2019/2020

Abstract

With the advent of the High-Luminosity phase of the LHC (HL-LHC), the instantaneous luminosity of the Large Hadron Collider at CERN is expected to increase up to $\sim 7.5 \cdot 10^{34} cm^{-2} s^{-1}$. Therefore, new strategies for data acquisition and processing will be necessary, in preparation for the higher number of signals produced inside the detectors, that would eventually make the trigger and readout electronics currently in use at the LHC experiments obsolete. In the context of an upgrade of the trigger system of the Compact Muon Solenoid (CMS), new reconstruction algorithms, aiming for an improved performance, are being developed. For what concerns the online tracking of muons, one of the figures that is being improved is the accuracy of the transverse momentum (p_T) measurement.

Machine Learning techniques have already been considered as a promising solution for this problem, as they make possible, with the use of more information collected by the detector, to build models able to predict the p_T with an improved precision.

In this Master Thesis, a step further in increasing the performance of the p_T assignment is taken by implementing such models onto a type of programmable processing unit called Field Programmable Gate Array (FPGA). FPGAs, indeed, allow a smaller latency with a relatively small loss in accuracy with respect to traditional inference algorithms running on a CPU, both important aspects for a trigger system. The analysis carried out in this work uses data obtained through Monte Carlo simulations of muons crossing the barrel region of the CMS muon chambers, and compare the results with the p_T assigned by the current (Phase-1) CMS Level 1 Barrel Muon Track Finder (BMTF) trigger system. Together with the final results, the steps needed to create an accelerated inference machine for muon p_T are also presented.

Chapter 1 provides a global view of the CMS experiment at LHC;

Chapter 2 presents an introduction of the CMS Muon System, with a particular attention to the Level 1 Trigger system of the barrel region;

Chapter 3 introduces Machine Learning concepts and terminology, offering an overview of a common workflow in the creation of a Supervised Learning algorithm, introducing also the concept of *quantizing* a Neural Network;

Chapter 4 describes the main characteristics of FPGAs, together with a brief description of the workflow to implement designs with such a kind of electronics devices;

Chapter 5 presents the original results of this project: the realization of two Neural Networks (NN) for the estimation of the particles' momentum, the steps followed to implement a NN on an FPGA, a description of the target hardware used in this thesis and the results obtained performing the p_T assignment task using an FPGA and a consumer CPU.

Sommario

Con l'avvento della fase ad alta luminosità del Large Hadron Collider (HL-LHC) presso il CERN di Ginevra, si prevede che la luminosità istantanea aumenterà fino a $\sim 7.5 \cdot 10^{34} cm^{-2} s^{-1}$. Pertanto, saranno necessarie nuove strategie per l'acquisizione e l'elaborazione dei dati, in preparazione ad un maggior numero di segnali prodotti all'interno dei vari rivelatori, che renderanno obsoleti il sistema di trigger e l'elettronica di acquisizione dati attualmente in uso presso gli esperimenti lungo LHC. Nell'ambito dell'aggiornamento del sistema di trigger del Compact Muon Solenoid (CMS), nello specifico, si stanno sviluppando nuovi algoritmi di ricostruzione che mirano a migliorarne le prestazioni. Per quanto riguarda il tracciamento online dei muoni, uno degli aspetti che si sta migliorando è la precisione della misura del momento trasverso (p_T).

Le tecniche di Machine Learning sono ritenute una soluzione promettente per questo problema, in quanto permettono, con l'uso di più informazioni raccolte dal rivelatore rispetto al sistema di trigger, di costruire modelli in grado di prevedere con una migliore precisione il p_T.

In questa tesi di Laurea Magistrale, viene perseguito un ulteriore passo avanti verso l'aumento delle prestazioni nell'assegnazione di p_T, implementando tali modelli su un dispositivo logico programmabile chiamato Field Programmable Gate Array (FPGA). Le FPGA, infatti, promettono una minore latenza con una perdita di precisione relativamente piccola rispetto ai tradizionali algoritmi di inferenza eseguiti su una CPU, entrambi aspetti importanti per un sistema di trigger. L'analisi effettuata in questo lavoro utilizza i dati ottenuti attraverso simulazioni Monte Carlo di muoni che attraversano la regione del barrel delle camere a muoni di CMS. I risultati sono quindi confrontati con il p_T assegnato dall'attuale (Fase-1) sistema di trigger di Livello-1 in CMS chiamato Barrel Muon Track Finder (BMTF). Insieme ai risultati finali, sono presentati anche i passi necessari per implementare su una FPGA un modello di Machine Learning per l'assegnamento del p_T dei muoni.

Contents

Chapter 1

The Large Hadron Collider

The Large Hadron Collider (LHC) [1, 2, 3] is the particle accelerator operating at CERN since 2010. Placed in a 27 km tunnel, the former 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 two proton beams run in separate beam pipes; the bending, needed to make the collisions possible, is achieved using compact twin-bore superconducting magnets, cooled at 2.1 K with superfluid helium, yielding a magnetic field of 8.3 T.

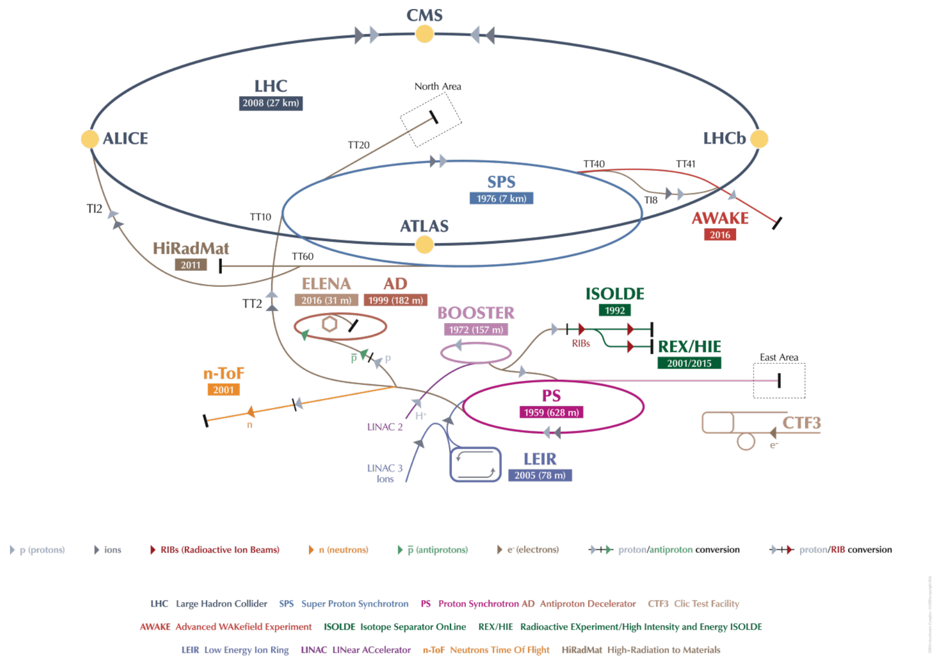


Figure 1.1: The CERN accelerator complex.

The proton injection is done employing pre-existing accelerators. This chain of accelerators, shown in Figure 1.1, comprises the Linac (Linear Accelerator), the PSB (Proton Synchrotron Booster), the PS (Proton Synchrotron) and the SPS (Super Proton Synchrotron). Given all these steps, an injection energy into LHC of 450 GeV is reached. The LHC beam filling requires about two hours. Though LHC can keep the same beams circulating for a longer period of time, data is usually taken only in the first 10 hours, then the beams are dumped and a new fill is

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 ($\mathrm{cm}^{-2}\mathrm{s}^{-1}$)	$2.06\ \times10^{34}$
Number of proton packages per beam	2808
Number of proton per package (outgoing)	$1.1\ \times10^{11}$
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.

started in order to restart at maximum beam intensity to maximize the integrated luminosity collected by the detectors.

The beams have a bunched structure, shown in Figure 1.2, with a crossing frequency of $f = 40$ MHz, corresponding to an inter-collision time of 25 ns. They are made up of 39 trains, each one containing 72 bunches with $N = 1.1 \times 10^{11}$ protons each. The number of missing packets between a train and the following one allows an absolute synchronization of the electronic systems. The inter-collision time is usually used as a time unit, commonly called *bunch crossing* (BX). Other technical parameters can be found in Table 1.1.

LHC started with a first phase of data taking (Run-1) in 2009, ending in 2013. In these years, it operated at a maximum centre of mass energy of $\sqrt{s} = 8$ TeV for proton-proton collision, reaching a maximum instantaneous luminosity of 7.67 $\times10^{33} cm^{-2} s^{-1}$ and an average number of interactions per BX equal to 21 [4]. At the end of this run, the total integrated luminosity hit 23.30 $\times10^{39} cm^{-2}$. The second LHC run (Run-2) lasted from 2015 to 2018. During this period LHC operated at a maximum center of mass energy of 13 TeV and the peak number of collisions per BX and the instantaneous luminosity reached the values of approximately 60 and $2.2{\times}10^{34} cm^{-2} s^{-1}$ respectively. The next run (Run-3) is expected to start in 2022, after a shutdown for the detectors' upgrades.

1.1 LHC Detectors

The collisions happen at four interaction points, where the 4 main detectors are located: ATLAS (**A** **T**oroidal **LHC** **A**pparatu**S**) and CMS (**C**ompact **M**uon **S**olenoid) are general purpose detectors, ALICE (**A** **L**arge **I**on **C**ollider **E**xperiment) focuses

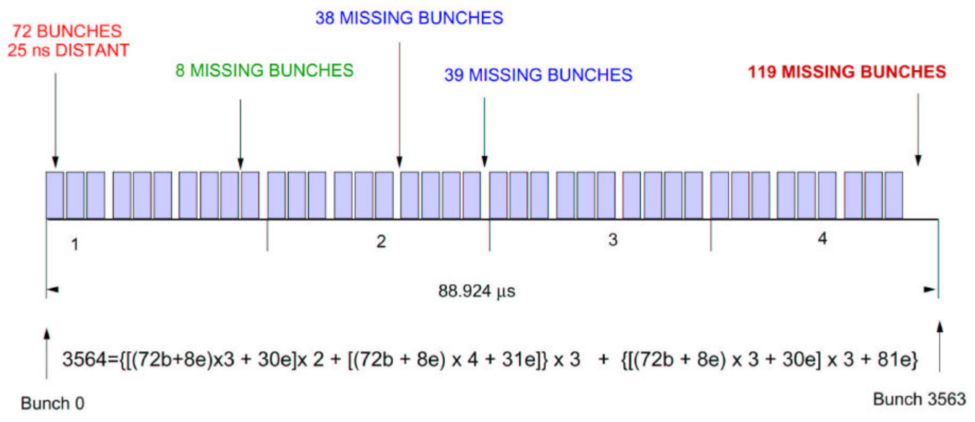


Figure 1.2: The LHC beam structure.

on the heavy ions physics and on the study of the quark-gluon plasma, and LHCb (**LHC b**eauty experiment) studies the CP violation in b-physics.

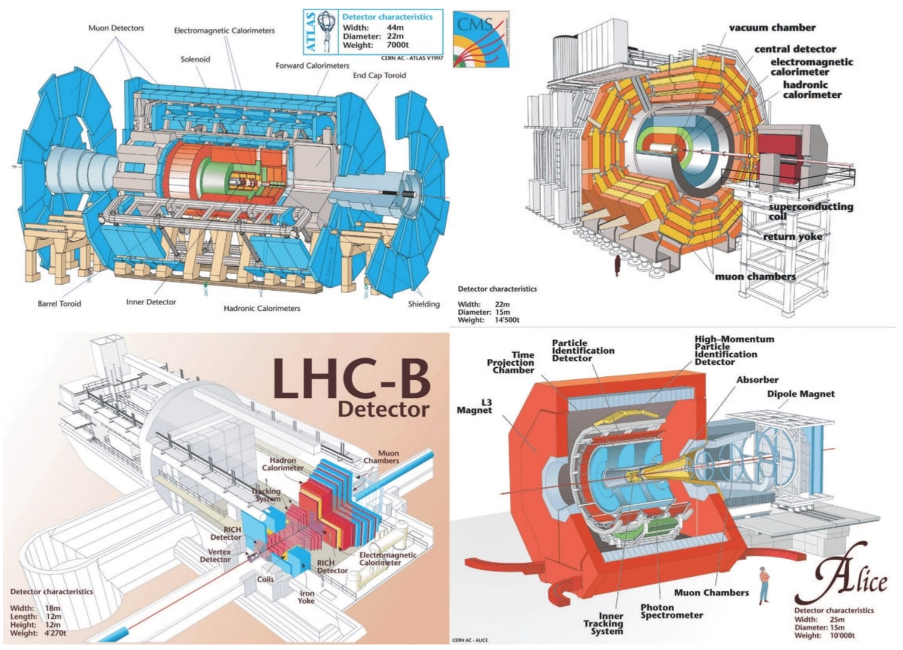


Figure 1.3: Schematic drawings of the four main LHC detectors.

1.1.1 CMS

CMS [5, 6] is a general-purpose detector at LHC. It is built around a cylindrical solenoid magnet which produces a 3.8 T magnetic field, about 100000 times the magnetic field of the Earth. The field is confined by a steel yoke that forms the bulk of the detector's 14000-tonne weight. The description of the apparatus in further details will be covered in section 1.2.

1.1.2 ALICE

ALICE (A Large Ion Collider Experiment) [7, 8] 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 similiarities 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.

1.1.3 ATLAS

ATLAS (A Toroidal LHC ApparatuS) [9, 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.

1.1.4 LHCb

The Large Hadron Collider beauty (LHCb) [11, 12] 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.

1.1.5 Other experiments

Aside from the aforementioned major LHC experiments, other smaller ones are worth a mention. One of them, LHCf [13, 14], 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 [15, 16]. 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 The Compact Muon Solenoid detector

The Compact Muon Solenoid (CMS) is a general purpose detector 21.6 m long with a diameter of 15 m and a weight of about 14000 tons. It is composed by a cylindrical barrel and two endcaps (1.4).

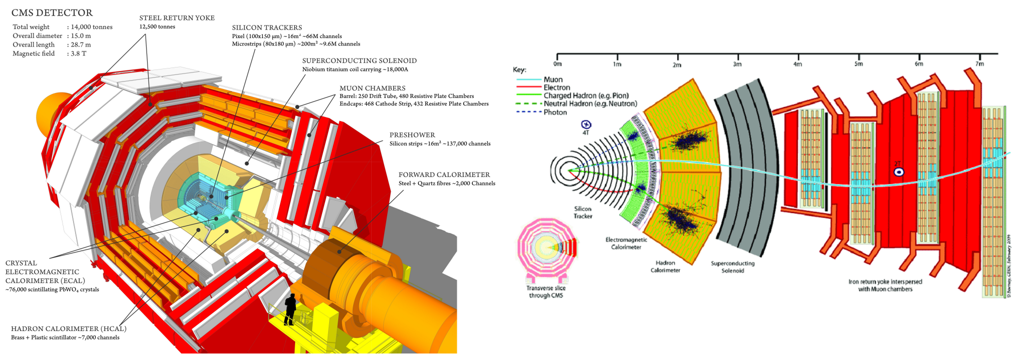

CMS DETECTOR
Total weight : 14,000 tonnes
Overall diameter : 15.0 m
Overall length : 28.7 m
Magnetic field : 3.8 T
STEEL RETURN YOKE
12,500 tonnes
SILICON TRACKERS
Pixel (100x150 μm) ~16m² ~66M channels
Microstrips (80x180 μm) ~200m² ~9.6M channels
SUPERCONDUCTING SOLENOID
Niobium titanium coil carrying ~18,000A
MUON CHAMBERS
Barrel: 250 Drift Tube, 480 Resistive Plate Chambers
Endcaps: 468 Cathode Strip, 432 Resistive Plate Chambers
PRESHOWER
Silicon strips ~16m² ~137,000 channels
FORWARD CALORIMETER
Steel + Quartz fibres ~2,000 Channels
CRYSTAL ELECTROMAGNETIC CALORIMETER (ECAL)
~76,000 scintillating PbWO₄ crystals
HADRON CALORIMETER (HCAL)
Brass + Plastic scintillator ~7,000 channels
0m
1m
2m
3m
4m
5m
6m
7m
Key:
Muon
Electron
Charged Hadron (e.g. Pion)
Neutral Hadron (e.g. Neutron)
Photon
4T
2T
Silicon Tracker
Electromagnetic Calorimeter
Hadron Calorimeter
Superconducting Solenoid
Iron return yoke interspersed with Muon chambers
Transverse slice through CMS
D.Barney, CERN, February 2004

Figure 1.4: Overall view of the CMS detector.

A superconducting solenoid generates a magnetic field of 3.8 T inside the detector. In its innermost region, a high quality central silicon inner tracker, capable of achieving a track reconstruction with very high resolution, is installed. A homogeneous electromagnetic calorimeter and a sampling hermetic hadron calorimeter are installed to perform energy measurements on electrons, photons and jets from hadrons. Finally, a redundant and highly performant muon system wraps the detector: it provides detection of muon tracks, and contribute to the measurement of their transverse momentum and angular position. Its four muon stations are sliced into an iron yoke that returns the flux of the magnetic field, acts as an absorber for traversing particles that are not muons and neutrinos, and hosts and mechanically supports the stations.

In CMS, a right-handed coordinate system can be defined: 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. Beside these coordinates, two angles are defined: the polar angle θ with respect to the z-axis, defined in the range $0 \leq \theta \leq \pi$, and the azimuthal angle ϕ with respect to the y-axis in the $x-y$ plane going from 0 to 2π (2.1). The following description is based on such coordinate system.

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

$$y = 1/2 \ln\left(\frac{E+p_z}{E-p_z}\right) \approx \eta = 1/2 \ln\left(\frac{|\vec{p}|+p_z}{|\vec{p}|-p_z}\right) = -\ln\left(\tan\frac{\theta}{2}\right) \tag{1.1}$$

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

Tracker

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 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. The Tracker is presently composed of a pixel detector with four barrel layers at radii between 4.4 cm and 10.2 cm and a silicon strip tracker with 10 barrel detection layers extending outwards to a radius of 1.1m. Each system is completed by endcaps which consist of 3 disks in the pixel detector and 3 plus 9 disks in the strip tracker on each side of the barrel, extending the acceptance of the tracker up to a pseudorapidity of $|\eta| < 2.5$.

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 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. 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. Photodetectors, that have been especially designed to work in the strong magnetic field, are glued onto the back of each crystal to detect the scintillation light and convert it to an electrical signal that is amplified and sent out for further processing and analysis.

The energy measurement for hadrons is carried out by the Hadron Calorimeter (HCAL), which is specifically built for measuring strong-interacting particles. It is a sampling calorimeter, i.e. it is made up of series of alternating passive and active layers. The passive layers are designed to make particles lose energy, while the active layers take snapshots of the showers of particles caused by hadrons traversing the calorimeter. The blue-violet light emitted by these kind of scintillators is then absorbed by optic cables of about 1 mm of diameter, shifting its wavelength into the green region of the electromagnetic spectrum and finally converted into digital data by photodetectors. The optic sum of light produced along the path of a particle, and performed by specifically designed photo-diodes, is correlated to its energy.

The HCAL also provides tools for an indirect measurement of non-interacting particles like neutrinos, which are produced in electroweak interactions. Indeed, HCAL is built to encapsulate as hermetically as possible the interaction point in order to detect “invisible” particles by looking for missing energy and momentum from the reconstruction of the collision event.

Unlike ECAL, HCAL has two more components called *forward sections*, located at the ends of CMS, designed to detect particles moving with a low scattering angle, extending the total coverage of the HCAL system to $|\eta| < 5.191$. These sections are built with more radiation resistant materials to protect them from the higher energy deposited in the regions close to the beam pipe.

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) [17] 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 2024-2026 (see Figure 1.5), finishing the hardware commisioning at machine re-start in 2026-2027 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 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

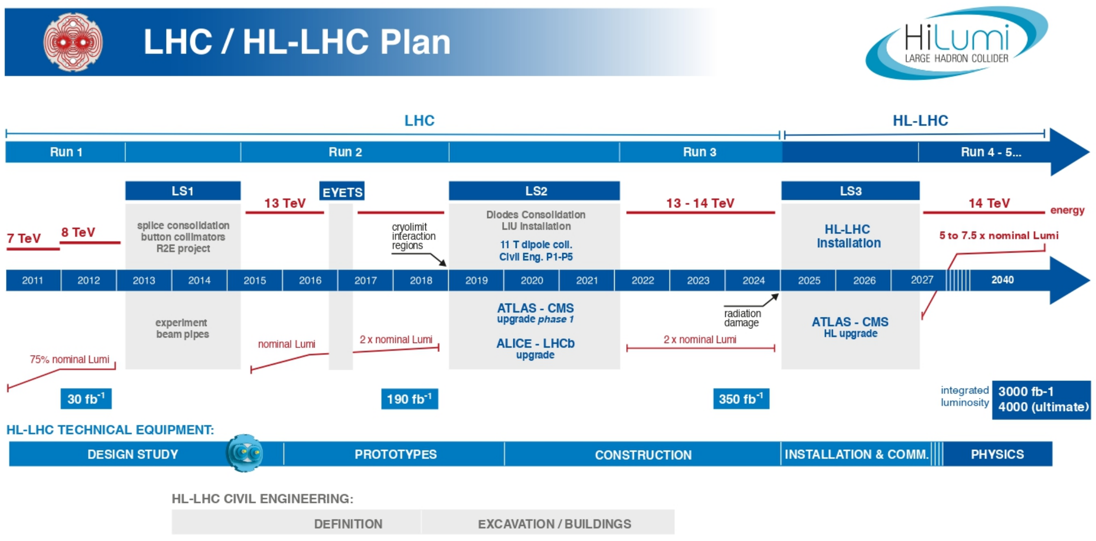


Figure 1.5: LHC/ HL-LHC Plan (last update August 2020 [18]).

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 [19] 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 ~ 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 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 1.2, and the Muon System (Figure 2.1), a group of subdetectors dedicated to this task and placed in the outermost region of the CMS experiment.

2.1 The experimental apparatus

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 [20] 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.

In the barrel region, where the background is small, the muon rate is low, and the 3.8 T magnetic field is uniform and mostly contained in the steel return yoke, drift chambers with rectangular drift cells are used. The barrel Drift Tube (DT) chambers cover the psudorapidity region $|\eta| < 1.2$ and are organized into 4 concentric stations interspersed among the layers of the flux return plates.

In the 2 endcap regions of CMS, where the muon rates and background levels are larger and the magnetic field gets large and non-uniform, the Muon System uses Cathode Strip Chambers (CSC). With their fast response time, fine segmentation and radiation resistance, the CSCs are used to track muons between $|\eta|$ values of 0.9 and 2.4. There are 4 stations of CSCs in each endcap, with chambers positioned perpendicular to the beam line and interspersed between the flux return plates.

Resistive Plate Chambers (RPC) are deployed throughout the central and forward regions ($\eta < 1.8$): they offer a fast response and an excellent time resolution providing an unambiguous BX assignment to muons, thus are mainly dedicated for triggering. Even though the RPCs have a lower spatial resolution than DTs and

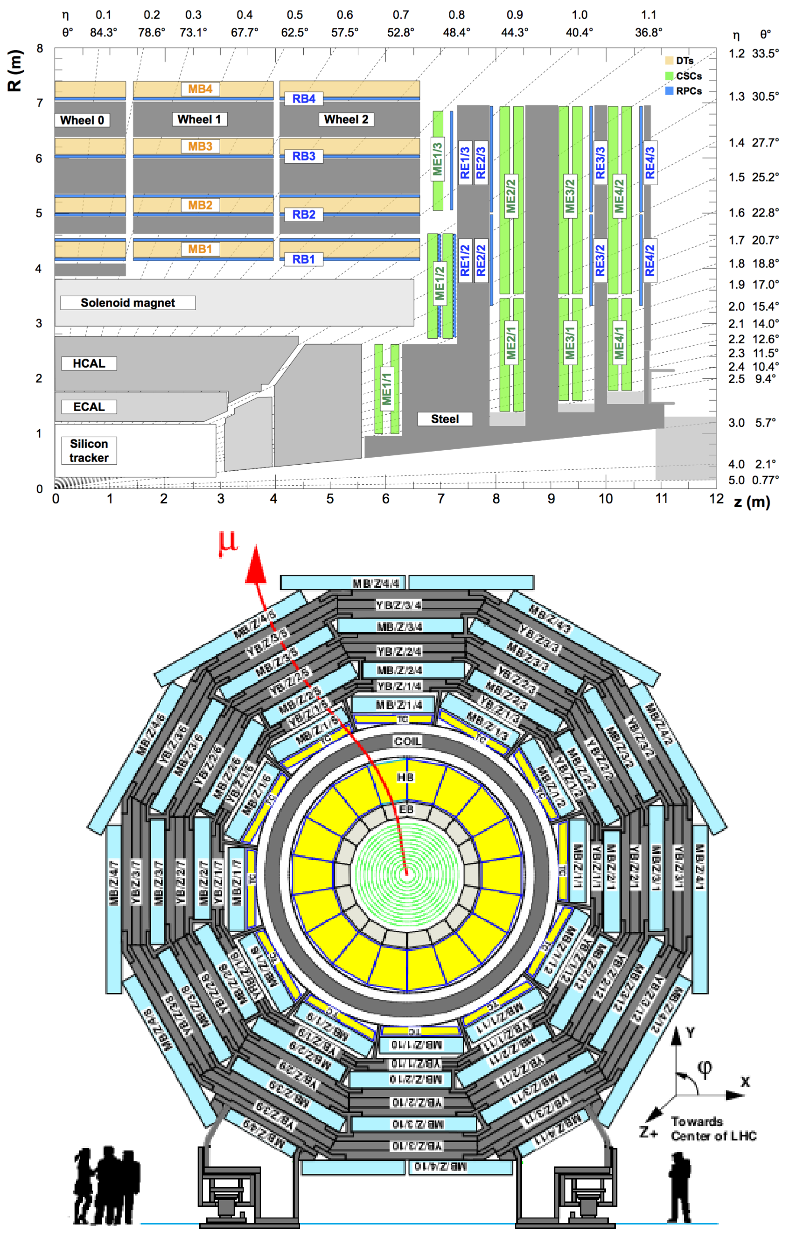
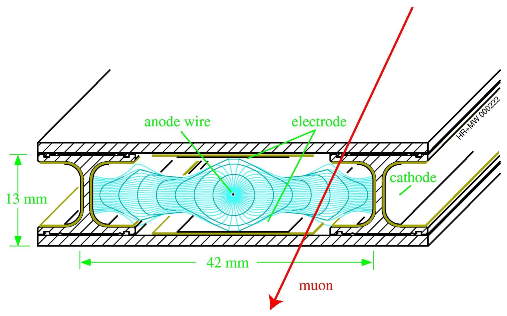


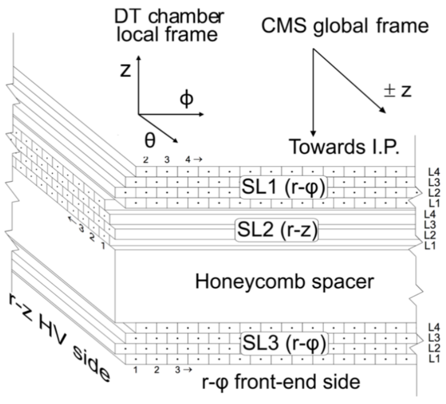
Figure 2.1: CMS detector longitudinal (top) view and barrel transverse (bottom) view. Describing LHC a reference frame is usually used in which the x-axis is pointed towards the center of the circular accelerator, the y-axis goes upward and the z-axis runs along the accelerated beam.

CSCs, they are also used to complement them in the offline reconstruction. A total of 6 layers of RPCs are embedded 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 a layer of RPCs (divided in two stations) placed right after the CSC chambers. The inner endcap disk houses instead two layers of RPCs. One is positioned in its inner side, after the innermost layer CSC chambers, the other is placed in the outer side of the iron yoke of forming the disk.

2.1.1 The Drift Tube Chambers



(a) The layout of a Drift Tube cell.



(b) Schematic view of a DT chamber.

Figure 2.2: DT chamber schematic in the Muon System (b) and a drift tube cell (a).

The basic detector element of the DT muon system is a drift tube cell (Fig. 2.2a) of 42 mm $\times$ 13 mm and it contains a stainless steel anode wire with a diameter of 50 μm and lenght 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 preceeding inner one, in order to maximize the geometrical acceptance. The DT layers inside a chamber, as shown in Figure 2.2b, 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.

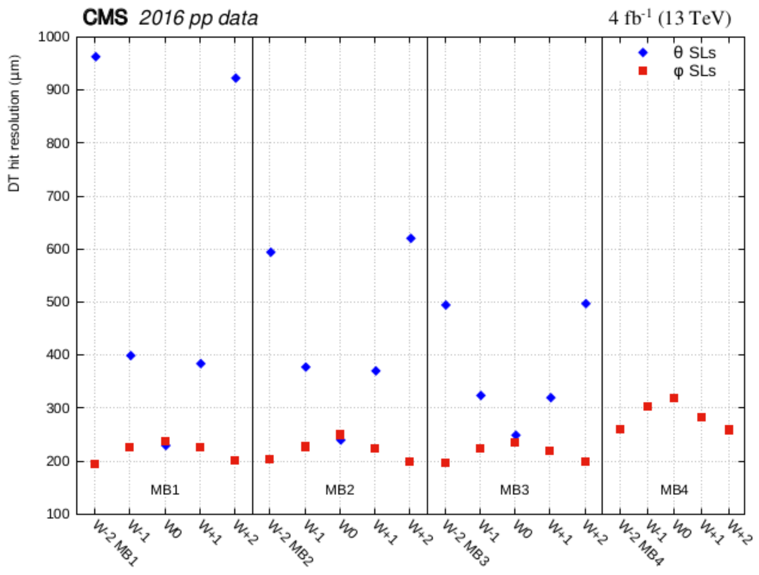


Figure 2.3: Reconstructed hit resolution for DT ϕ (squares) and θ (diamonds) superlayers measured with 2016 data [21].

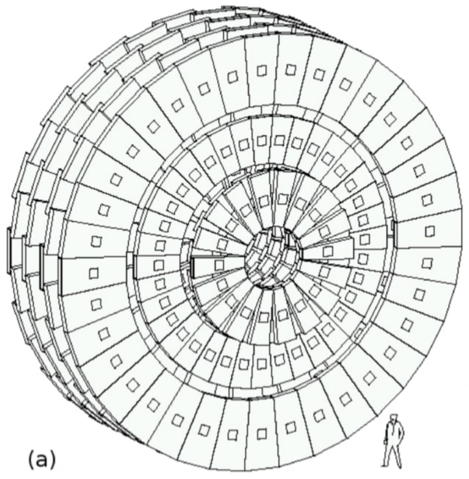
Figure 2.3 shows the spatial resolution of DT hits sorted by station, wheel and wire orientation (ϕ or θ) [21]. The resolution in the ϕ superlayers (i.e. in the bending plane) is better than 250 μm in MB1, MB2 and MB3, and better than 300 μm in MB4. In the θ superlayers, the resolution varies from about 250 to 600 μm except in the outer wheels of MB1.

Within every station, both θ and ϕ superlayers show symmetric behavior with respect to the $z = 0$ plane, as expected from the detector symmetry. In the central wheel (wheel 0), where tracks from the interaction region are mostly perpendicular to all layers, the resolution is the same for θ and ϕ superlayers. From wheel 0 toward the forward region, tracks from the interaction region have increasing values of $|\eta|$; this affects θ and ϕ superlayers in opposite ways. In the θ superlayers the increasing inclination angle degrades the linearity of the distance-drift time relation,thus worsening the resolution. In contrast, in ϕ superlayers the inclination angle increases the track path within the tube (along the wire direction), thus increasing the ionization charge and improving the resolution. The resolution of the ϕ superlayers is worse in MB4 because no θ measurement is available, so no corrections can be applied to account for the muon time of-flight and the signal propagation time along the wire.

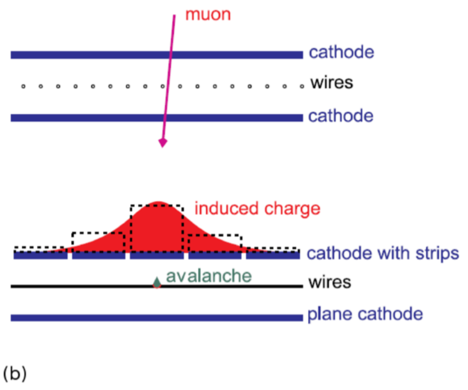
2.1.2 The Cathode Strip Chambers

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 (CSC) has been adopted [22].



(a) Layout of the CSC subsystem.



(b) Schematic view of CSC chamber signal formation.

Figure 2.4: 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 spatial resolution of the CSC strip measurement depends on the relative position at which a muon crosses a strip: it is better for a muon crossing near a strip edge than at the center because then more of the induced charge is shared between that strip and its neighbor, allowing a better estimate of the center of the charge distribution. The design specifications for the spatial resolutions in the CSC system were 75 μm for the chambers in the inner ring of ME1 and 150 μm for

Station/Ring	Spatial Resolution (μm)
ME1/1a	45
ME1/1b	52
ME1/2	90
ME1/3	105
ME2/1	125
ME2/2	134
ME3/1	120
ME3/2	135
ME4/1	123
ME4/2	134

Table 2.1: CSC transverse spatial resolution per station [21].

the others. Table 2.1 summarizes the mean spatial resolution in each CSC station and ring.

2.1.3 The Resistive Plate Chambers

Resistive Plate Chambers (RPCs) [23] 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.

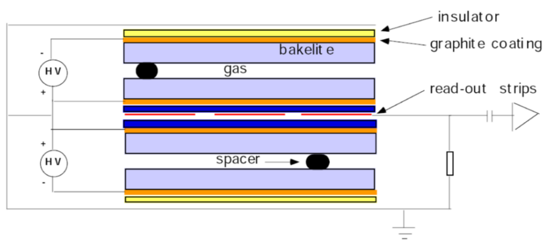


Figure 2.5: Schematic view of a CMS double gap RPC.

A RPC is formed, as shown in Figure 2.5, 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.

Double-gap RPCs operated in avalanche mode have demonstrated the ability to reach an intrinsic time resolution of around 2 ns [24]. This has to be folded in with the additional time uncertainty coming from the time propagation along the strip, which contributes about 2 ns, plus the additional jitter that comes from small channel-by-channel differences in the electronics and cable lengths, again of the order of 1-2 ns. These contributions, when added quadratically, give an overall time resolution of around 3 ns [25]. This is much smaller than the 25 ns timing window of the RPC data acquisition system (DAQ) in CMS.

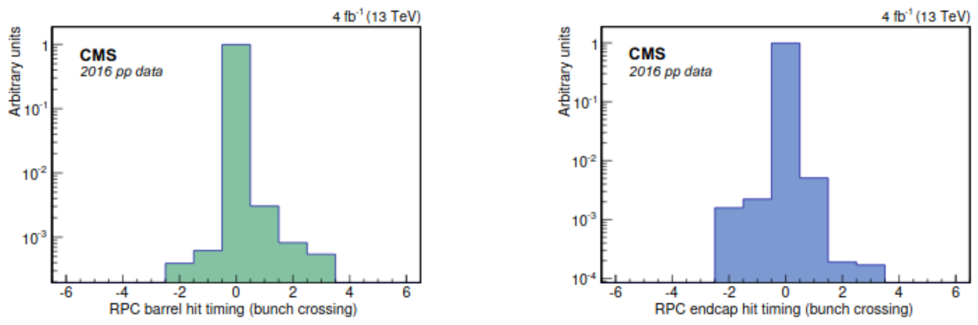


Figure 2.6: The bunch crossing distribution from reconstructed RPC hits in the barrel (left) and in one endcap (right), using the 2016 data [21].

The left-hand side of Figure 2.6 shows the BX distribution of RPC hits associated with global muons in the barrel, while the right-hand side shows the same distribution for one endcap. Each bin corresponds to the 25 ns bunch separation in LHC, and bin 0 is the time of the L1T. In figure 2.6, 0.5% of RPC hits are outside bin 0.

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. Since it is impossible 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. This task is performed by the trigger system, which is the first step of the physics event selection process. The selection criteria is driven by the experiment's physics goals, thus the trigger system must be able to reconstruct physics objects (muons, electrons, gammas, jets, missing energy, etc) with sufficient efficiency and purity to achieve the required rate reduction without compromising the yield of interesting events.

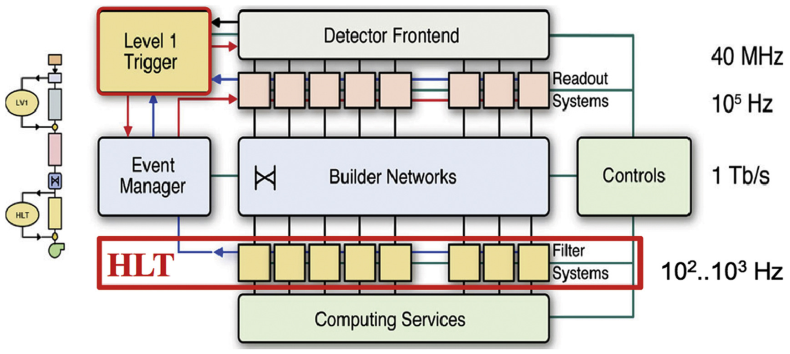


Figure 2.7: Schematic diagram of the CMS Trigger Chain.

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: the Level-1 trigger 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; the High Level trigger (HLT) 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 2.7 a diagram of the CMS trigger chain is pictured, together with the rate of events which characterize each steps of the chain.

2.2.1 The Level-1 Trigger system

The Level-1 Trigger (L1T) [26, 27] 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 (ASICs) and Programmable Lookup Tables (LUTs) 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 2.8. 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.

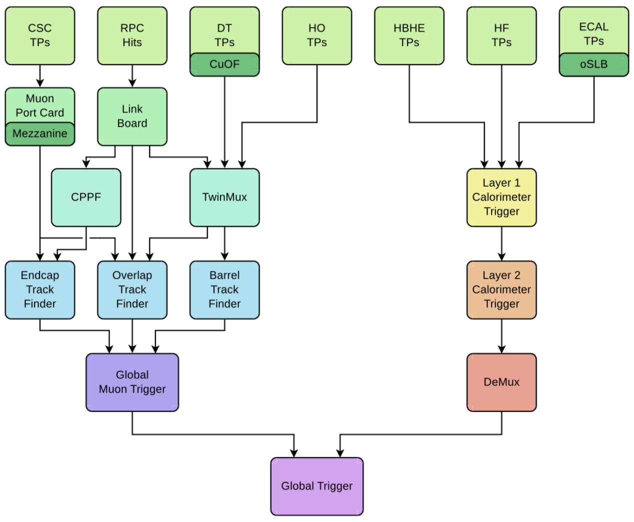


Figure 2.8: Schematic diagram of the CMS Level-1 trigger.

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 2.8, 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 (GMT) 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 [28] 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 DAQ system is shown schematically in Figure 2.7. 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.

2.3 The DT Local Muon Trigger

The DT Local Trigger System [29] (or DT Trigger Primitive Generator DT-TPG) goal is the detection of charged particles crossing the muon barrel chambers, measuring their position and trajectory as well as their BX of origin. It also tags the reconstructed primitives with a quality word based on the number of layers involved in the computation and, eventually, it sends them to later stages of the L1T processing.

In Figure 2.9 the workflow the systems is shown: the signals picked up by the individual wires inside the DTs' superlayers are processed by the Bunch and Track identifiers (BTIs) that perform track fitting and BX assignment, exploiting the mean-timer [30] property, holding for half staggered DT cells characterized by a linear drift. Then, the Track Correlator (TRACO) has to match information

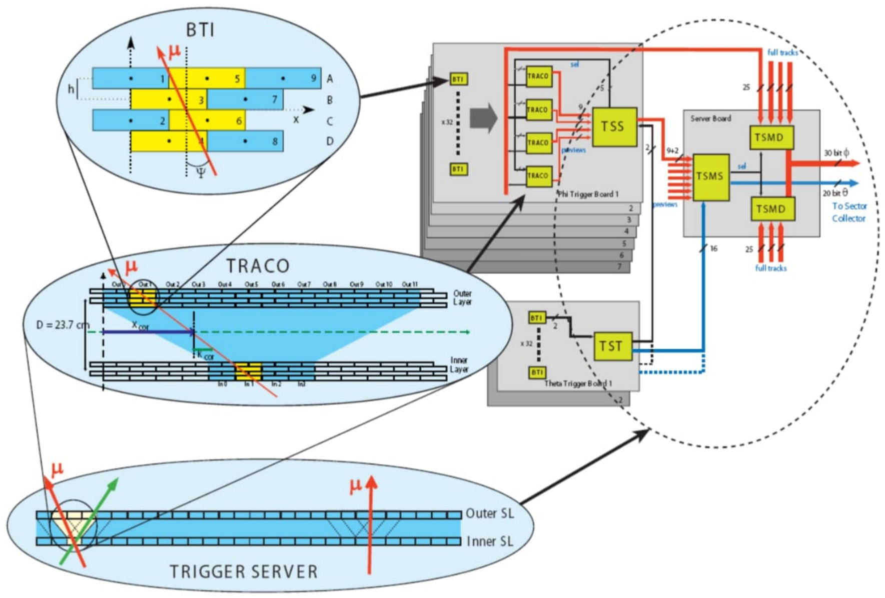


Figure 2.9: Schematic diagram of the CMS DT Local Trigger.

coming from the two ϕ-superlayers of a DT chamber, increasing the lever-arm between the BTI track segments making up the particles' trajectory, and so the accuracy of the parameter measurements. The next step is performed by the Trigger Server (TS), which filters the candidates produced by the TRACO. Finally, the information is sent to the TwinMux card, which merges DT primitives together with the corresponding hits from RPCs. Then, the tracks are transferred to the Trigger processors designed for BMTF and OMTF.

2.3.1 Bunch and Track Identifier

The Bunch and Track Identifier (BTI) [30] can be defined as a hardware track fitting device. It is based on an implementation of the generalized mean-timer method. It was explicitly developed to work on groups of four staggered layers of DTs (superlayers), and to identify tracks which gave a signal in at least three out of the four layers in a fixed time window. Each BTI is connected to 9 cells of the same SL and adjacent BTIs overlap (having 5 common cells) to avoid low efficiency regions [31].

The BTI algorithm relies on the fact that the particle path is a straight line, the wire positions along the path (the measurement points) are equidistant and half-staggered, and the drift velocity is approximately constant in the cell. Therefore, considering the drift times of any three adjacent planes of staggered tubes (e.g. cells in layers A, B and C of Figure 2.10), if the time is a linear function of the space, the quantity

$$T_{MAX} = \frac{T_A + 2T_B + T_C}{2} \tag{2.1}$$

is a constant and this principle holds independently from the track impact point d and angle of incidence ϕ. T_{MAX} is the drift time corresponding to the distance between wires in two staggered layers (2.1 cm). Extending this method to four

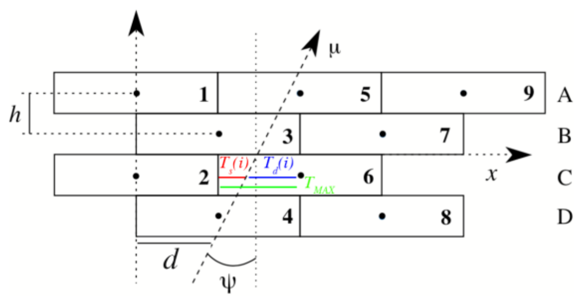


Figure 2.10: Layout of a DT Bunch and Track Identifier.

layers, the track identification is possible even if the drift time of a tube is missing (due to inefficiency) or is altered by the emission of a δ ray. It is also insensitive to uncorrelated single hits and it is therefore well-suited to a high radiation environment.

After a particle's crossing, a signal is induced on the detecting i_{th} wire at a time $T_d(i)$ that depends on the drift velocity v_d as well as the distance between the particle trajectory and the wire. On every BTI clock count, the apparent $T_d(i) = T_{MAX} - T_s(i)$ is computed (since the BTI digitizes the delay time T_s from the wires signal detection with a 80 MHz frequency), and a straight line is reconstructed from all different permutations of pairs of layers. Parameters like the position and the angular parameter $k = h \tan \phi$ (where $h = 13$ mm is the distance between wire planes) are then evaluated for each pair of layers.

Hence at every clock cycle, a whole set of *k-equations* are computed and a BTI trigger is generated if at least three of the six possible combinations are in coincidence. For a single muon traversing the superlayer, the coincidence happens at a fixed latency T_{MAX} after the particle passage, allowing BX identification. In the case of a four hit alignment, an *High Quality Trigger* (HTRG) is raised, while in any other case a *Low Quality Trigger* (LTRG) is issued. However, there might be a possibility that the alignment of four hits at some clock step produces a LTRG just before or after the correct HTRG: a so called *ghost* trigger candidate. The noise reduction of ghosts is obtained issuing the LTRG only if no HTRG is generated before or after the possible LTRG: this mechanism is called *Low Trigger Suppression* (LTS).

2.3.2 The Track Correlator

The TRAck COrrelator [32] task is to find correlations between track segments obtained from BTIs of the two different ϕ superlayers in a station. Each TRACO is connected to 4 BTIs of the inner SL and to 12 BTIs of the outer one (see Figure 2.11). Segment position and angle from the two SLs are compared to verify if they belong to a single track crossing a chamber. In this case, a new angle with better resolution is assigned to the trigger primitive. The total number of TRACOs per chamber depends on the size of the station (usually from 12 to 24 TRACOs are needed). Each TRACO tries to find up to two correlated segments, and forwards their parameters to the Trigger Server. In the θ view, since there is only one detecting SL per station, no TRACO operation is performed, forwarding data directly to the TS.

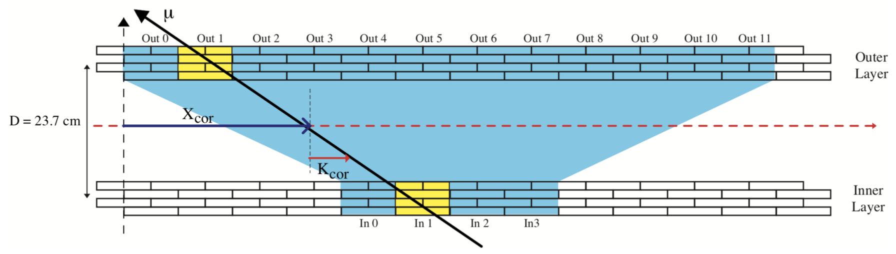


Figure 2.11: TRAck COrrelator geometrical layout

The algorithm of the TRACO starts by selecting, among all candidates in the inner and outer SL independently, the best track segment according to the trigger quality (HTRG or LTRG) and to the track proximity to the radial direction compared to the vertex (i.e its p_t). Then, it computes the k-parameter and the position of a correlated track candidate, using the relations:

$$\begin{cases} k_{COR} = \dfrac{D}{2}\tan\phi = x_{inner} - x_{outer} \\ x_{COR} = \dfrac{x_{inner} + x_{outer}}{2} \end{cases} \tag{2.2}$$

Due to the long arm between the two SLs, the angular resolution of a correlated track is ~ 5 mrad which is significantly improved compared to the BTI resolution, whereas the position resolution is improved by a factor $\sqrt{2}$.

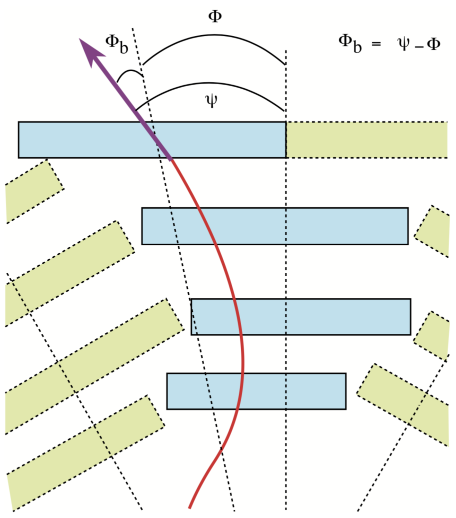


Figure 2.12: Definition of the angles transformed by the TRACO.

Using programmable LUTs, the computed parameters are then converted to the chamber reference system: position is transformed to radial angle ϕ and k-parameter to *bending angle* $\phi_b = \psi - \phi$ (as shown in Figure 2.12). If a correlated track is built, it is forwarded to the TS, for further selection. If the correlation fails, the TRACO forwards an uncorrelated track following a preference list that includes both parent SL (IN/OUT) and the quality bit (H/L) of the two candidate tracks. If no correlation is possible since there is no candidate in one SL, the existing uncorrelated track is still forwarded.

2.3.3 The Trigger Server

The Trigger Server [33] is the last on-board component of the DT Local Trigger logic and has to select the best trigger candidate among the trigger primitives provided by all TRACOs in a muon chamber and sends them to the TwinMux (see Section 2.3.4).

Each muon station is equipped with only one TS and, for this reason, it needs to be extremely robust and redundant. The TS has the following characteristics:

- since interesting physics may have two close-by muons that can hit the same muon chamber, emphasis was put on the efficiency and purity of up to two selected segments in designing the Trigger Server. The selection is based on both the bending angle and the quality of the track segment.
- it has to reject ghosts generated by TRACOs using a configurable ghost rejection algorithm.
- the processing time is independent on the number of TRACOs in the station.
- it has the capability to treat several pile-up events.
- it is equipped with a built-in redundancy since there is only one TS on each chamber, resulting in a bottleneck of the on-chamber trigger.

The TS is composed by two subsystems (see Figure 2.13): one for the transverse view (TSϕ) and the other for the longitudinal view (TSθ). The TSθ has to detect triggers produced by the 64 BTIs which equip the SL in the θ view. This information is sent back to the TRACOs of the ϕ view and can be used as a validation of a trigger in order to reduce backgrounds. The TSϕ is implemented in a two

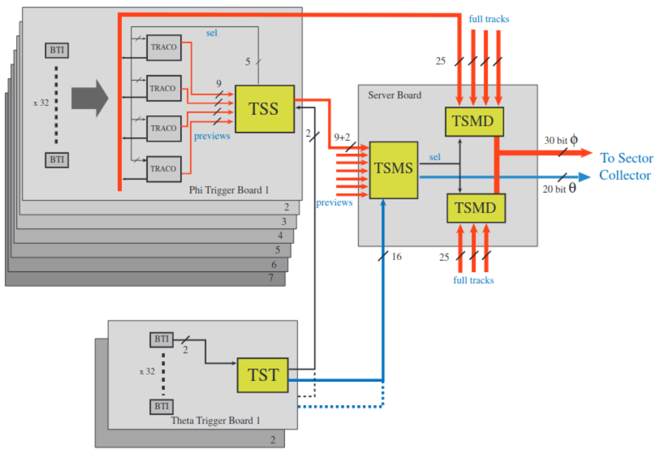


Figure 2.13: Trigger Server architecture.

level structure: the *Track Sorter Slave* (TSS) and the *Track Sorter Master* (TSM).

In this way a parallel approach with two steps of increasing bandwidth is guaranteed. On the other hand, the TSθ is formed by two identical units (TST) which group together data coming from up to 8 BTIs and perform a logical OR on the information received in order to prepare the θ-view trigger for regional processing.

2.3.4 The TwinMux barrel concentrator board

The TwinMux [34] is the adaptive layer for the track finder in the barrel region. It merges and transmit information from DT, RPC and of the Hadronic Calorimeter Outer barrel trigger, unburdening the track finder processors (BMTF and OMTF). It is also responsible for duplicating the trigger primitives in order to reduce connections between the Track Finders, increasing the reliability of the system.

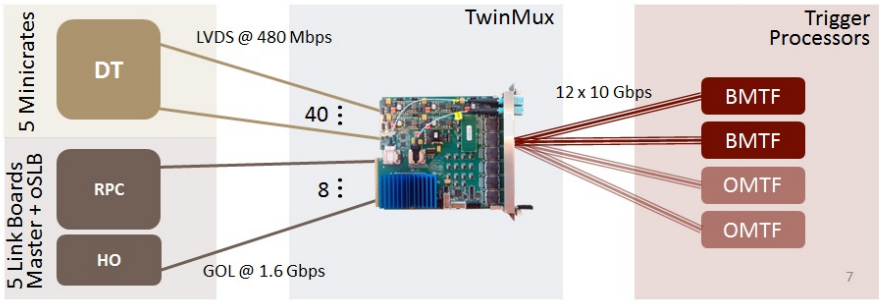


Figure 2.14: TwinMux role in the DT and RPC trigger chain.

The TwinMux is a single slot double-width full-height μTCA board, equipped with 6 front panel SNAP12 receivers and 2 Minipods (one transmitter and one receiver) for high speed data transmission (up to 13 Gbps). Based around a Xilinix Virex-7 FPGA, the TwinMux achieves the merging of several 480 Mb/s trigger links to higher speed serial links and compensates delays to provide BX alignment of the trigger data coming from different inputs.

The TwinMux hosts an algorithm aimed at improving the trigger performance through the simultaneous use of DT and RPC information. This is achieved through two main operations: the calculation of DT+RPC primitives and of RPC-only primitives. Each TwinMux processor receives DT and RPC links from one sector of the barrel muon detector. The data coming from these two detectors contains different information: Trigger Primitives are sent from DTs and they include position, direction, quality and BX information. On the other hand, Hits coming from RPCs include position and BX information only, and they undergo a clustering process, i.e. neighbouring hits are merged and the resulting cluster position is assigned with half-strip resolution and converted into DT coordinates. In case that the same RPC cluster fires in two consecutive BXs, the second one is suppressed.

In MB1 or MB2 station, if there is no DT primitive available, "pseudo primitives" can be built. In fact, MB1 and MB2 are equipped with two RPC chambers, installed on the two opposite sides of the DTs with respect to the beam line. If RPC clusters from these two chambers are close in space, they are combined and position and direction are computed and used to create a primitive which is sent to the BMTF, potentially increasing the number of station layers used to build a BMTF track.

Thus, RPC-only trigger primitives are particularly useful in case of problems affecting the DT detector, ensuring triggers are generated even in case of complete lack of input from a DT chamber. Nevertheless, there is an increase in overall efficiency even in standard, healthy detector conditions, as shown in Fig. 2.15, where the efficiency for DT-only and DT+RPC primitives (red open circles) is complemented with the inclusion of RPC-only primitives as well (blue full circles) [35].

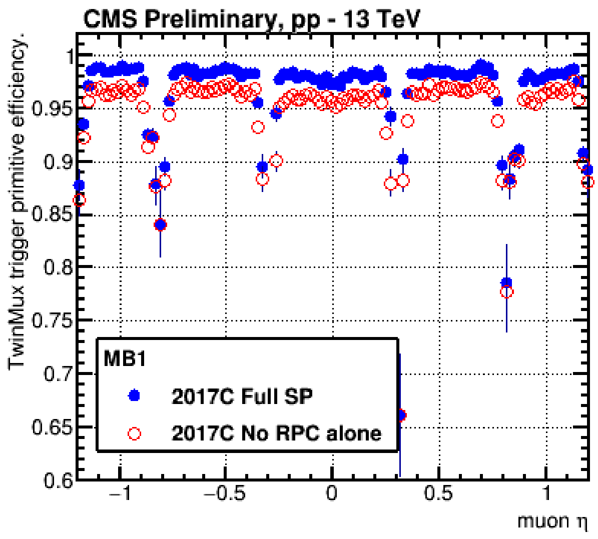


Figure 2.15: Trigger Primitive efficiency, for the MB1 station, versus the reconstructed muon pseudorapidity η. In red, open circles, the efficiency without the introduction of RPC-only primitives. In blue, solid circles, the efficiency with all the primitives types included.

The DT+RPC primitives are obtained following an algorithm which has several steps. First, input data are deserialized and synchronized. RPC clusters close in ϕ to DT Trigger Primitives from the same chamber are searched for in a $\pm$ 1BX time window centered around the DT primitive BX. After that the closest RPC cluster is selected and if the $\Delta\phi$ with respect to the Trigger Primitive is equal or less than 15 mrad, RPC and DT are considered matched and a DT+RPC primitive quality flag is set. Furthermore, if the DT Trigger Primitive was built with less than 8 DT ϕ layers, the Trigger Primitive BX is set to match the RPC cluster BX; if it was built with all 8 DT ϕ layers, its BX is not changed. The impact of this procedure is clear by looking at the different BX assignment distribution for DT-only and DT+RPC primitives in Figure 2.16. Whenever no match with an RPC cluster is found, the original DT Trigger Primitive is sent to the BMTF and the DT+RPC primitive quality flag is not set.

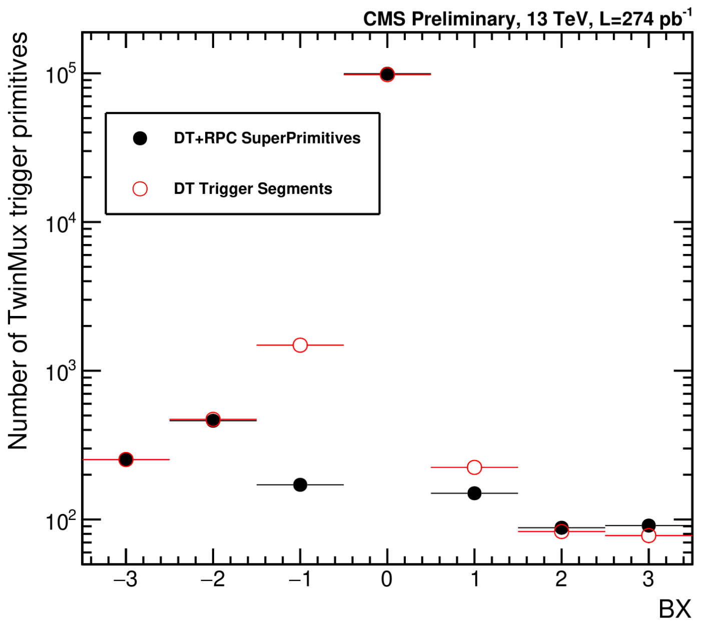


Figure 2.16: Comparison of BX assignment at the output of the TwinMux concentrator board when using DT-only primitives versus DT+RPC primitives.

2.4 The Barrel Muon Track Finder

The Barrel Muon Track Finder (BMTF) [36] receives the muon Super Primitives from the Barrel area of CMS ($|\eta| < 0.85$) and it reconstructs muon tracks and calculate the physical parameters: transverse momentum (p_T), the ϕ and η angles, the quality of the candidate and the track address. The Track Finding algorithm ran over the LHC Run-2 consisted in three stages:

1. The *Extrapolator Unit* combines, using LUTs, the trigger primitives from different DT stations. For selected permutations of stations with primitives, a reference station is chosen, and the primitive from that station is propagated to the other one, called target station. The distance between the propagated trigger primitive and the one from the target station is computed, and, if the two lay within an acceptable window, the latter primitive is retained in the next stage of the track-finding algorithm. The same algorithm also probes the compatibility of primitives from different stations laying in neighbouring sectors.

2. The *Track Assembler Unit* links the segment pairs to the full track and assigns a quality word related to the number and type of stations involved in the reconstruction.

3. The *Assignment Unit* uses LUTs, implemented with Block RAM (BRAM), to assign physical parameters to the track previously built by the algorithm (p_T, ϕ, η).

Due to the limited address space provided by BRAM, the momentum assignment is performed using information from only two stations. In addition, the LUTs are filled assuming that the track originates from the center of the detector. This beam spot constraint improves the momentum resolution, since it effectively adds one more point at the CMS center, exploiting the full bending power of the CMS solenoid.

In Figure 2.17 the entire process of track finding is shown. The algorithm run in parallel for all sectors of one wedge in the same card. Each one of the twelve processing cards delivers the three best muon candidates to the GMT.

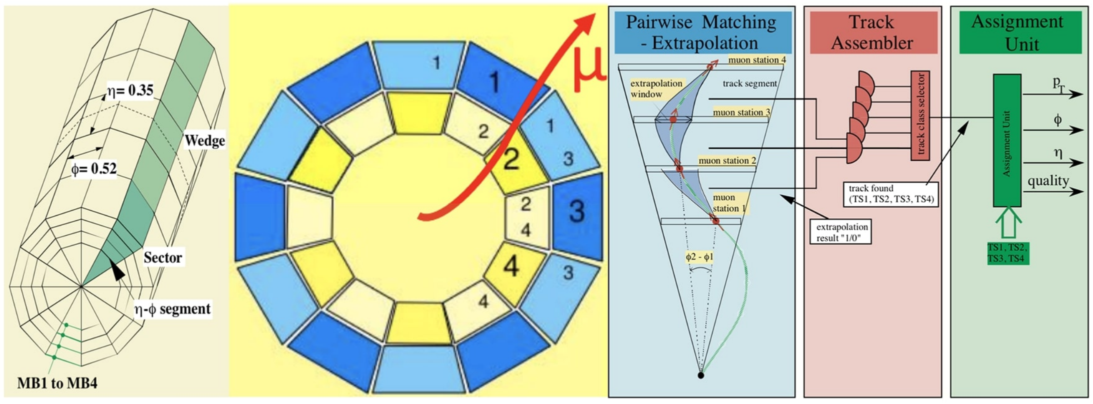


Figure 2.17: Graphical depiction of the steps making up the BMTF algorithm.

2.5 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 [37, 38, 39, 40, 41]. 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 [42] of the L1T system utilizes technological advances to enhance the physics selectivity already at the hardware level of the data acquisition system. 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. Moreover, a longer latency will enable higher-level object reconstruction and identification, as well as the evaluation of complex global event quantities and correlation variables to optimize physics selectivity. The implementation of sophisticated algorithms using particle-flow reconstruction techniques or Machine Learning based approaches are contemplated [42]. In addition, a 40 MHz scouting system harvesting the trigger primitives produced by sub-detectors and the trigger objects produced at various levels of the trigger system is foreseen.

2.5.1 Muon chambers upgrades

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, described in Section 2.3.

In order to better exploit the intrinsic time resolution of the existing RPC system (~ 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 will be installed in the forward region $1.6 < |\eta| < 2.8$. The installation of GEM detectors will allow 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 will 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, GE2/1 and ME0, as shown in red and orange in Figure 2.18.

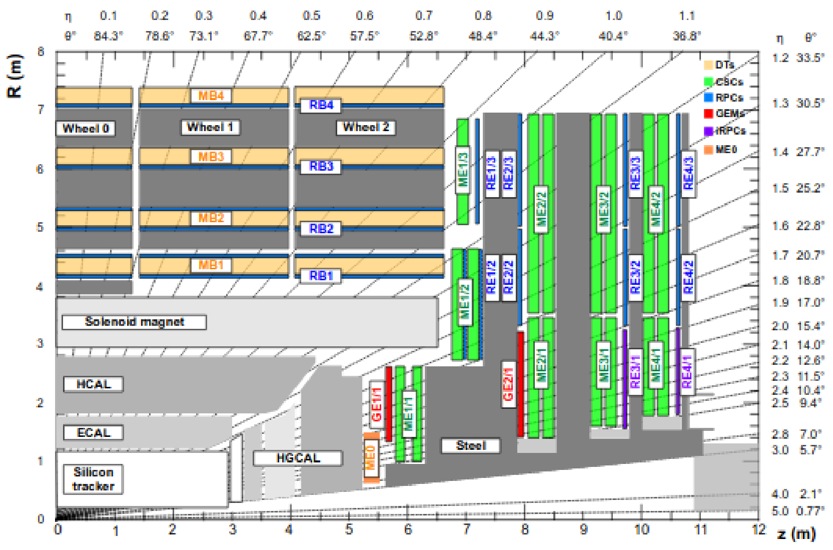


Figure 2.18: Longitudinal view of a quadrant of the CMS Phase-2 muon system. Different colors in the figure refer to different sub-detectors: DT (orange), RPC (light blue), CSC (green), iRPC (purple), GE (red), ME0 (orange).

2.5.2 Phase-2 Muon Barrel Trigger primitives

A new algorithm will be introduced to perform DT TPs generation, called Analytical Method (AM), which assumes that muons follow a straight path inside a chamber and rely on the mean-timer property (see Section 2.3.1). The AM has been designed following a very direct hardware-oriented approach. It operates in three steps, called grouping, fitting and correlation. Starting from a group of 10 nearby cells, distributed across the four layers of a DT SL, the grouping step selects patterns of 3 or 4 fired DT cells compatible with a straight line. From those patterns, the fitting step exploits the mean-timer property to compute unambiguously the BX corresponding to every subset of 3 cells within each pattern. For cases with 4-cells patterns, the BX of each triplet is combined with the others using an arithmetic mean. Track parameters are then computed using exact formulas from χ^2 minimization. Finally, a correlation between the two $r-\phi$ SLs is attempted. A potential match between the primitives of the SLs in the same chamber is looked for, within a window of ± 25 ns. If a match is found, the track parameters are recalculated, either as an arithmetic mean of the ones of each SL (position, time), or as ratio of the difference between the positions of each primitive and the distance between the two SLs (direction). If no match is found, all per-SL primitives are forwarded to the next stage to maintain high efficiency.

After building the DT segments and clustering the RPC hits, the BMTL1 will then merge information from both sub-detectors into combined super-primitives (SP) according to the data flow shown in Figure 2.19. This exploits the redundancy and complementarity of the two sub-systems, combining the DT spatial resolution with the RPC time resolution, thus increasing the trigger primitive performance and making them more robust against effects like detector failures or aging. The baseline SP combination algorithm works as follows. The RPC clusters

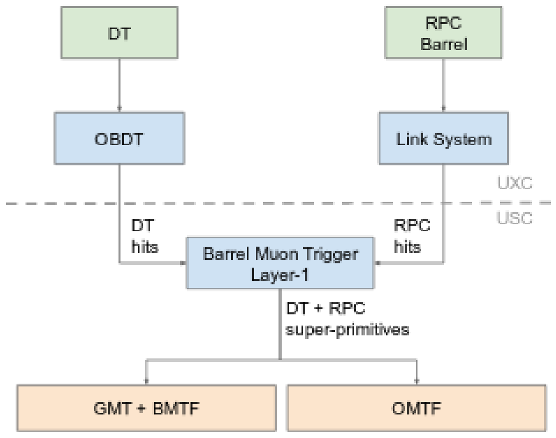


Figure 2.19: Architecture for the Muon Barrel Trigger primitive generation as foreseen in the Phase-2 L1T upgrade. The dashed line represents the separation between the underground experimental cavern (UXC) and the underground service cavern (USC).

are translated into DT coordinate conventions and the DT primitives are matched to RPC TPs based on the azimuth angle, within a three units BX window centered around the DT primitive's BX. For DT TPs that are matched to an RPC cluster, the SP BX and sub-BX timing are set based on RPC information. In the first two stations, the algorithm can also build an RPC-only TP out of hits from the two RPC layers, in chambers where no DT primitive is found. Furthermore, in the whole barrel, RPC clusters not matched to any DT segments nor used to build an RPC-only segment are passed to the track finders.

The time resolution of the muon barrel SPs is shown in Figure 2.20 (left). It is estimated from a MC simulation based on the time coordinate of SPs associated to generated muons with $p_T > 20$ GeV. The time distribution becomes narrower when the information coming from the RPCs is used ($\sigma = 1.9$ ns) with respect to the only-DT primitives ($\sigma = 2.2$ ns). Figure 2.20 (right) shows the efficiency to reconstruct a barrel SP and correctly assign its BX. It is computed with respect to offline reconstructed segments geometrically matched with incoming generated muons and no minimal quality threshold cut is applied to the SPs. In the no-ageing case, the TP efficiency is $\sim 98\%$ for DT-only primitives and increases to above $\sim 99\%$ for combined DT+RPC SPs [1].

The spatial resolution for position and direction of the DT TPs, computed with respect to simulated hits, is presented in the left and right hand-side of Figure 2.21 respectively. The layout of the plots is identical to the one used for Figure 2.20 (right). As an example, MB2 stations show a position resolution of ~ 0.3 mrad, which reflects an improvement of a factor of ~ 6 with respect to the legacy TPs. A similar improvement is observed for the direction resolution, which is ~ 1 mrad

[1]In view of the operation throughout HL-LHC, potential effects due to the ageing or failures of the detectors presently equipping the muon system were considered. A thorough description of such effects goes beyond the scope of this document. Anyhow, more information about the expected ageing of the DT detector can anyhow be found in [43].

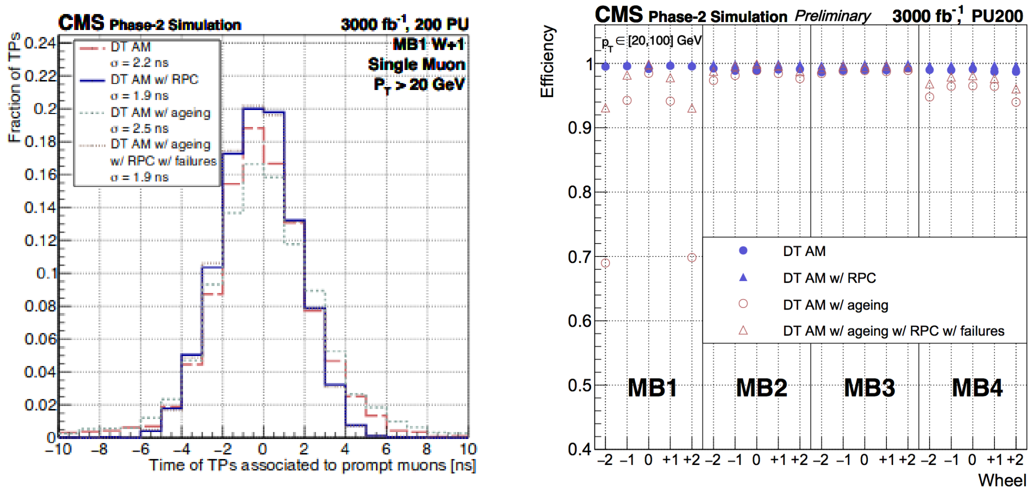


Figure 2.20: Left: Timing distribution of muon TPs from the analytical method (AM) algorithm, generated with and without enabling the combination into DT+RPC SPs. Results are computed with and without including detector ageing and failures. Right: Trigger segment efficiency of muon barrel primitives, measured with the analytical method (AM), with and without enabling the combination of DT+RPC SPs. Results are computed with and without including detector ageing and failures.

for the Phase-2 AM TPs. The same conclusion holds for all stations.

2.5.3 Phase-2 Track Finding with the Kalman Filter

In Phase-2, the DT backend electronics will be upgraded, thus providing better position and time resolution. This allows possible improvements in track reconstruction, hence a better momentum resolution. This can be done by taking advantage of the better DT position resolution and by exploiting information from all stations in the L1T muon reconstruction. This goal cannot be achieved by the LUT approach (see Section 2.4) since the address space required for four stations is 88 bits with the current precision, more than the available space in the LUTs implemented today. The precision is expected to increase even more in Phase-2. Another achievement would be to implement momentum assignment without the beam spot constraint. This is motivated by searches for displaced particles. Since a displaced particle does not originate from the beam spot, the beam spot constraint results in mismeasured p_T and inefficiency of the trigger for such physics signals. To achieve both goals, a new tracking paradigm is implemented exploiting a Kalman Filter (KF) algorithm [42].

Kalman Filtering is a technique very widely used in track reconstruction at hadron colliders. However, to achieve a suitable latency, a special KF implementation has been devised, tuned for the barrel muon trigger and optimized for low latency. A pictorial representation of the KF algorithm is shown in Figure 2.22. To reduce the number of computations, the values of the Kalman gain (i.e. the "factor" used to update the TP parameters at each step of the algorithm) for dif-

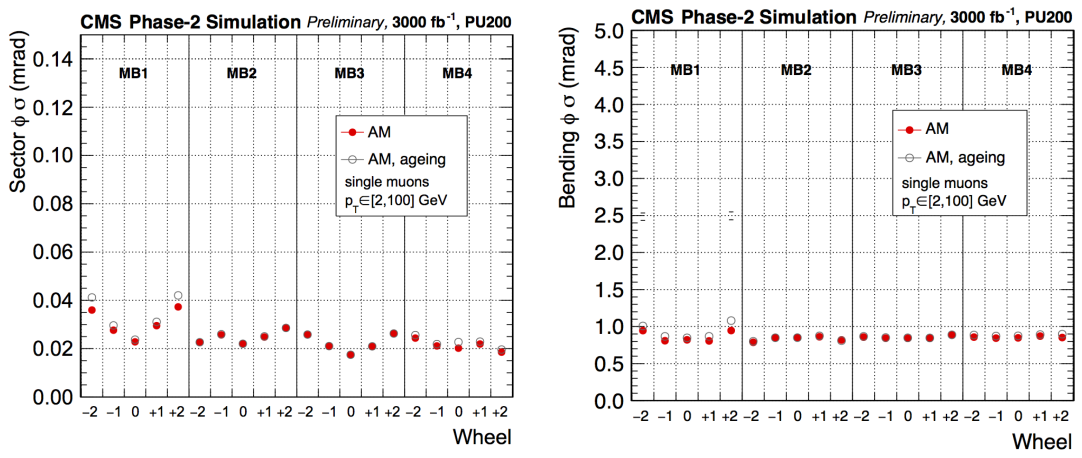


Figure 2.21: Spatial resolution of muon barrel TPs computed with respect to simulated hits in the DT detector. Left: position resolution. Right: direction resolution. Results are computed with and without including ageing for the DT detector.

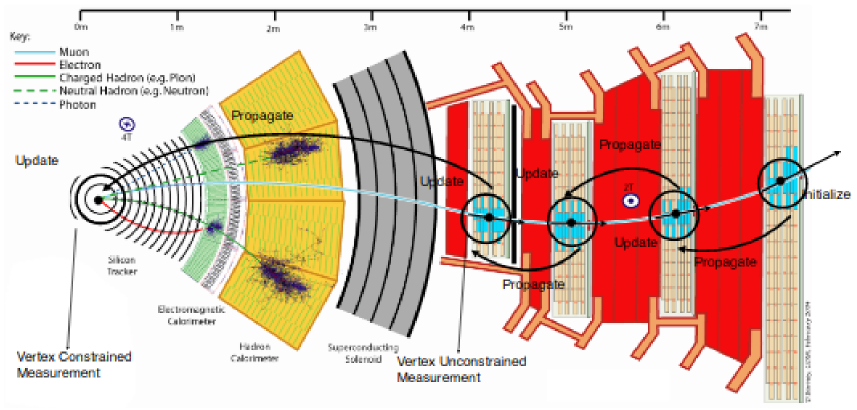

Key:
Muon
Electron
Charged Hadron (e.g. Pion)
Neutral Hadron (e.g. Neutron)
Photon
Update
Propagate
Initialize
Silicon Tracker
Electromagnetic Calorimeter
Hadron Calorimeter
Superconducting Solenoid
Vertex Constrained Measurement
Vertex Unconstrained Measurement
4T
2T

Figure 2.22: Sketch of a muon track traversing the full CMS detector. The Kalman Filter-based track finder is illustrated, starting from the outermost muon station, propagating inwards and updating the track parameters at each station. At the innermost station it provides a vertex-constrained and an unconstrained measurement, allowing for triggering on prompt and displaced muons.

ferent tracks in simulation have been studied; it was found that the gain matrix can be precalculated for different values of curvature (implying different multiple scattering) and different combinations of the stubs used at each station (implying different uncertainty of the already reconstructed track at each station). Therefore, an approximate Kalman Filter was implemented, propagating the track from station to station, and updating it using a precalculated Kalman gain that depends on a 4-bit pattern of the already used primitive and the value of the curvature at the given point of the track.

After reaching the first station, a measurement without the beam spot constraint is stored and then the track is propagated to the center of CMS. While the energy loss was neglected for the station-to-station propagation, for the propagation to the vertex it is taken into account since the material budget of the calorimeters, the magnet and the tracker is significant. After the vertex propagation, the impact parameter of the track to the beam spot is estimated and a Kalman update is performed, requiring that the track should pass through the origin. This final update provides a beamspot constrained measurement. This approximate KF algorithm, named KBMTF, exploits the measurements in all detector stations, provides a vertex unconstrained measurement for displaced muons, and implements a vertex constrained measurement for muons originating from the beamspot, satisfying all requirements for the improved muon trigger.

2.5.4 The Phase-2 Global Muon Trigger

In the Phase-2 muon L1T, the Global Muon Trigger (GMT) receives tracks from the inner-tracker track finder, standalone muon tracks from the regional muon track finders, and all stubs from the muon detectors. Combining information from the muon system with the one of tracks reconstructed in the inner tracker, a reduction of trigger rates, up to a factor of 4 with respect to the Phase-1 BMTF and the KBMTF alone.

The objects reconstructed by the GMT are sent directly to the Phase-2 Global Trigger, where they can be used either as a single-object trigger, or in conjunction with other trigger objects. In addition, GMT objects can also be used to implement the particle-flow algorithm.

Even though the Phase-2 L1T upgrade will introduce reconstruction in the inner-tracker in the L1T, there are cases when the p_T assignment carried out only with information coming from muon chambers still remain critical. One such case, briefly mentioned in Section 2.5.3, concerns displaced muons. This is why a ML Model implemented on an FPGA was studied in this Master thesis: to find alternatives in p_T assignment which are reliable and have low latency but do not strictly rely on combination of information from the muon chambers and the inner-tracker. In Chapter 5, the p_T produced by such a model will be compared with the p_T assigned after the output produced by the existing BMTF (in the *Assignment Unit*). The comparison with the Phase-1 BMTF was pushed by the fact that a full simulation of the Phase-2 L1T has yet to be released to the public.

Chapter 3

Introduction to Machine Learning

The expression "*Machine Learning*" (ML) was first used to describe a particular type of computer algorithms in 1959 by Arthur Samuel [44]. In recent years, 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. E-mail providers have also perfected spam filters with ML, allowing them to classify suspect e-mails as spam more efficiently by looking for more subtle features compared with a "classic" word association algorithm.

ML algorithms, and especially pattern recognition techniques, are also deployed by social networks, for example to identify different people in photos, but also in medicine, e.g. 2D/3D images of human tissues and organs for diagnosis, and beyond.

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.

There are several ways to make an algorithm "learn". The main classes of learning algorithms are *supervised* learning, *unsupervised* learning and $reinforcement$ learning. This work will deal with a specific type of *supervised* learning algorithm, hence in the next Section a more in depth description will be presented. However, for the sake of clarity, a brief explanation about the other types will presented as well.

Unsupervised Learning

In an *unsupervised* learning algorithm, the computer is provided with an input dataset but with no corresponding true output values for underlining a possible structure of data. This algorithm will, therefore, result in a clustering selection of the input data.

Clustering algorithms are used in many web-based application, such as newspaper websites: using thousands of stories as input data, the learning algorithm automatically cluster the information together based on the topic or user preferences, organizing information in a coherent and customizable way.

Unsupervised learning is also used for organizing large computer cluster, deciding which machines tend to work together in order to connect them and increase the overall efficiency. Another application is on social network analysis, identifying between hundreds of friends the closest one, based on private communications or recent social activities.

Reinforcement Learning

Reinforcement learning is the study of learning in a scenario where a learner actively interacts with the environment to achieve a certain goal. This active interaction justifies the terminology of *agent* used to refer to the learner. The achievement of the agent's goal is typically measured by the reward he receives from the environment and which he seeks to maximize.

In recent years, a lot of improvements were observed in this area of research. Most popular examples include DeepMind and the Deep Q learning architecture (dated 2014), which culminated in the famous defeat of the champion of the game of Go by AlphaGo in 2016.

Reinforcement learning can be understood using the concepts of Agents, Actions, Environments, States, and Rewards:

Agent It is the component that take actions, e.g. a video game character navigating its virtual environment, as well as a drone making a delivery.

Action It is 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, 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 It is 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. E.g. in a model in which the Agent would be a real person, the Environment could be the entirety of the physics laws, plus the rules of the society it lives in, and within which all of its *actions* would be processes and the consequences of them determined.

State It is 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 It is the feedback based on which the success or failure of the Agent's choices are 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, 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 is to finish the game with the maximum score, so each additional point obtained throughout the game will affect the Agent's subsequent behaviour (i.e. an Agent might learn that making some combination of Actions will lead to a point to it will try to redo the same actions effectively in case a similar situation will be offered next in the Environment).

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 more and more start-ups are beginning to use this ML type to build products for industrial robotics applications.

3.1 Supervised Learning

This specific type of learning algorithm looks for a function which maps an input to an output based on provided input-output pairs. This is accomplished in the so-called *training* process: namely, a learning phase in which, along with the input features, a target variable is also provided (acting as the "truth"), which contains the information that the machine has to learn during the training process.

In a nutshell, unsupervised learning involves observing several examples of a random vector of features $\mathbf{x}$ and attempting to implicitly or explicitly learn the probability distribution $p(\mathbf{x})$, or some interesting properties of that distribution. On the other hand, supervised learning [45] 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.

There are different kinds of problems that can be solved via an algorithm trained in this way. The one tackled in this work is called *regression* and consists in trying to predict a continuous output value (instead of a discrete output which characterizes a *classification* problem). Visual recognition is another application domain that highly relies on supervised ML algorithms. For instance, a system might need to identify pedestrians on a street in an automotive application for self-driving cars: to do so, its core software is already trained with millions of short videos picturing street scenes, with some of the videos containing no pedestrians at all while others having up to dozens. Since the presence or not of pedestrians, during training, is known a priori, the learning is supervised. In the next section, a more in depth mathematical description will be presented.

3.1.1 How machines "learn"

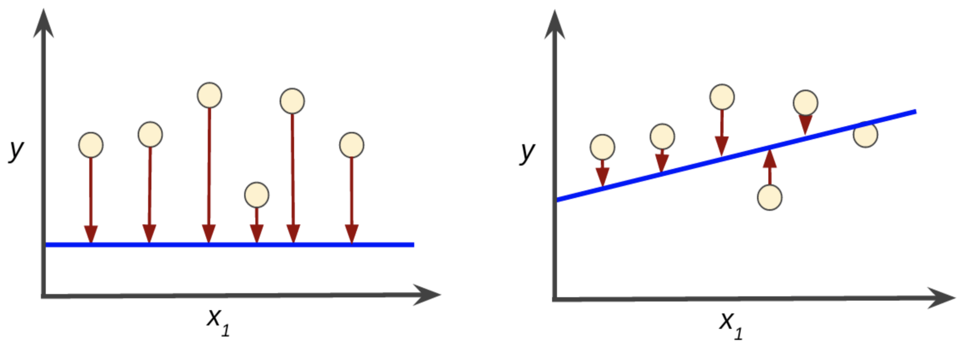
To make the concept of *learning* more concrete, let us consider an example of a simple ML algorithm: linear regression. The goal is to build a system that can take a vector $\mathbf{x} \in \mathbb{R}^n$ as input and predict the value of a scalar $y \in \mathbb{R}$ as its output. The output of linear regression is a linear function of the input. Let $\hat{y}$ be the value that the model predicts, given y as the true label. The output is then defined as

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b \tag{3.1}$$

where $\mathbf{w} \in \mathbb{R}^n$ is a vector of *parameters* and b is a scalar parameter called *bias*.

Parameters are values that control the behaviour of the system. In this case, w_i is the coefficient that is multiplied by feature x_i before summing up the contributions from all the features. $\mathbf{w}$ can be thought of, and it is usually referred to, as a set of *weights* that determine how each feature affects the predictions. If a feature x_i receives a positive weight w_i, then increasing the value of that feature increases the value of the prediction $\hat{y}$. If a feature receives a negative weight, then increasing the value of that feature decreases the value of the prediction. If a feature's weight is large in magnitude, then it has a large effect on the prediction. On the other hand, if a feature's weight is zero, it has no effect on the prediction.

Training a model simply means finding values for all the weights and the bias from labelled examples which result in a good prediction $\hat{y}$. In the case of supervised learning, the algorithm builds a model by examining many examples and attempting to find a model that minimizes the *loss*. The term *loss* is used to describe a sort of penalty for an incorrect prediction: it is a number indicating how bad the model prediction is for a certain example in the training set. A visual representation is shown in Figure 3.1: 3.1a shows a model with predictions which are more far away from the known values with respect to the model in 3.1b, meaning that the loss would be higher in the first case, as visually intuitive by the longer red arrows.



(a) Example of a model with high loss. (b) Example of a model with lower loss.

Figure 3.1: Different regression models for a supervised ML algorithm. Red arrows represent loss while the blue line represents predictions. In (a), an high loss is obtained while in (b) a lower loss is achieved.

In order to describe and aggregate the individual losses associated to predictions from a model, *loss functions* are defined. There are numerous loss functions, each of them more or less suited for different problems tackled by ML algorithms. In case of regression problems, Mean Squared Error (MSE) is frequently used, defined as the sum of all the squared losses $(y - \hat{y})^2$ for individual examples divided by the number of examples, as follows:

$$\text{MSE} = \frac{1}{N} \sum_{(\mathbf{x},y)\in D} (y - \hat{y}(\mathbf{x}))^2 \tag{3.2}$$

where

- $(\mathbf{x}$,y) is an example in which $\mathbf{x}$ is the vector of features that the model uses to make predictions, and y is the example's label (the quantity to predict);
- $\hat{y}(\mathbf{x})$ is a function of the weights and bias;
- D is a dataset containing many labelled examples, i. e. $(\mathbf{x},y)$ pairs.

MSE is not the only possible metric for regression problems: for instance the Mean Absolute Error (MAE) is another loss function, defined as follows:

$$\text{MAE} = \frac{1}{N} \sum_{(\mathbf{x},y)\in D} |y - \hat{y}(\mathbf{x})| \tag{3.3}$$

MAE shows a linear profile which can be useful if the MSE results too strict not allowing a good gradient descent (see 3.1.2), however the discontinuity at 0 can cause errant behaviours for very small loss values.

3.1.2 Training a ML model

The supervised model we are discussing takes one or more features as input and return one prediction as output. Initially, all the weights are randomly set and, at each iteration, the value of the loss function generates new values for w_i. This learning procedure continues until the algorithm discovers the parameters with the lower possible loss. When the loss stops changing (or changes really slowly), the model is said to have *converged*, i.e. it has reached a minimum of the loss function. Therefore, this kind of regression problems will have a convex plot of loss vs. weights. However, calculating the loss function for every value of w_i is an inefficient way of finding the convergence point. This is why most ML algorithms involve optimization of some sort. *Optimization* refers to the task of minimizing some function $f(\mathbf{x})$ by altering $\mathbf{x}$. The function to be minimized is called the *objective* function, other than *loss* function. One of the most common optimization technique is called *gradient descent* (GD) which uses the fact that derivative of a function, in this case the loss function, tells how to change its input in order to reduce its output, down to its minimum, where the derivative will go to zero.

Let us apply this logic to ML. After choosing a starting value for the weights, typically random, the gradient of the loss curve is evaluated. Mathematically speaking, a vector of partial derivatives with respect to the weights, is computed and in case of a negative gradient, the gradient descent algorithm performs the

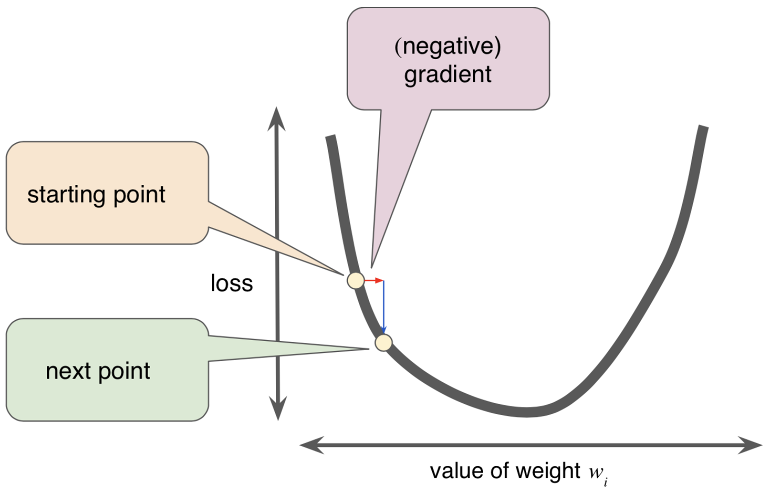


Figure 3.2: Schematic representation of the gradient descent algorithm.

next step, by changing slightly the weights in order to obtain a loss nearer to the minimum of the function.

GD algorithms multiply the gradient by a scalar known as *learning rate* (or *step size*, one of the model's so-called *hyperparameters*) which determines the "size" of the step towards the minimum. Tuning the learning rate is a tricky problem: if a too small learning rate is selected, the learning procedure will take too long; conversely, if it is too large, the next point will exceed the minimum and the convergence may eventually not be reached.

In case of loss function optimization via GD, when large data volumes come into play, other parameters in the strategy become relevant for a fast convergence of the minimisation algorithm, e.g. the so-called *batch size*: it corresponds to the total number of examples that are used to calculate the gradient in a single iteration. By default, the batch matches the entire dataset. However, datasets may often contain millions of entries as well as a huge number of features. In these cases, the evaluation of the gradient can be too time and resource consuming. Smaller batch sizes are therefore suggested: in the *stochastic gradient descent* (SGD), for instance, the batch size is reduced to one single entry per iteration. Given enough iterations, SGD works, but it usually yields noisy results which are too sensible to the single examples given. An intermediate approach between batch GD and SGD is the so called *mini-batch GD*: the original training set is subdivided into smaller batches (usually of the order of 10, but can be much more depending on the size of the dataset) which are chosen randomly when the gradient is evaluated. This allows more parallelization of the entire process, which might be extremely useful in case of heterogeneity of hardware devices available for training. This technique will ultimately provide GD algorithm convergence in a shorter time.

3.1.3 Validation and Testing

A key challenge in ML is "*generalization*", i.e the fact that the algorithm must perform well on new, previously unseen inputs. What separates ML from a general optimization algorithm is that also the *generalization* error, or *test* error, is requested to be as low as possible, other than the loss evaluated with the training data. In order to quantify this error and, in this way, predict the model accuracy with new data, it is common to split the data at hand in three different sets:

Training Set used in the learning algorithm to make the model learn by finding the weights which reduce the loss as much as possible;

Validation Set used to evaluate results coming from the training set each time a parameter is changed, i.e. the learning rate or the addition of an extra feature.

Test Set allows a double-check on the evaluation after the model has passed the validation set.

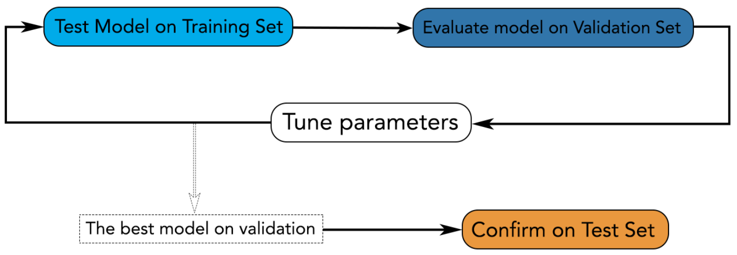

Test Model on Training Set
Evaluate model on Validation Set
Tune parameters
The best model on validation
Confirm on Test Set

Figure 3.3: ML training workflow using a validation and test subset.

In Figure 3.3, an example of workflow is shown: the training set is firstly checked by the validation set, which provides the model's optimal parameters for another cycle of training; then, the best model is tested again using the testing subset to obtain a reliable and generalized model.

There are 4 different main techniques in order to split the dataset into Training and Validation/Test set and measure the accuracy of the model:

1. Training and Test Set Splitting;
2. k-$fold$ Cross-Validation;
3. Leave One Out Cross-Validation;
4. Repeated Random Test-Train Splits.

Training and Test Splitting

This algorithm evaluation technique is very fast and so useful with a slow training model. It is also ideal for large datasets. Splitting a large dataset into large sub-datasets ensures that they are both representative of the underlying problem.

However, this technique can show a high variance. This means that differences in the training and test set can result in meaningful differences in the estimate of

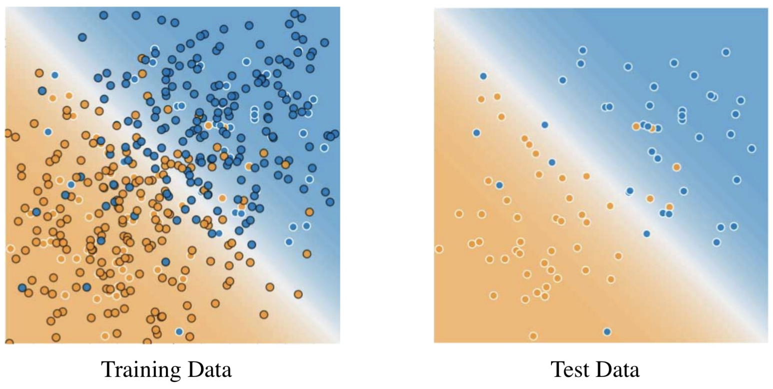


Figure 3.4: Example showing the splitting between training and test datasets. On the left side the training dataset can be seen, while, on the right side, the test data used for verifying the trained model is pictured [46].

accuracy. Furthermore, in order for this split to work, the test has to be representative of the dataset as a whole, i.e. the test set should be a random preview of the future data the model will deal with after deployment.

In Figure 3.4, a simple example is shown where the model is not perfect, since a few predictions are wrong. However, the model does about as well on the test data as it does on the training data.

K-fold Cross-Validation

K-fold Cross-Validation (CV) is another approach that can be used to estimate the performance of a ML algorithm with less variance than a single train-test set split. The dataset is divided into k subsets (called "folds"): for each fold, the kth subset is used as the test set and the other $k-1$ subsets are put together to form the training set (see Figure 3.5). Then, the average error across all k trials is computed, together with the standard deviation.

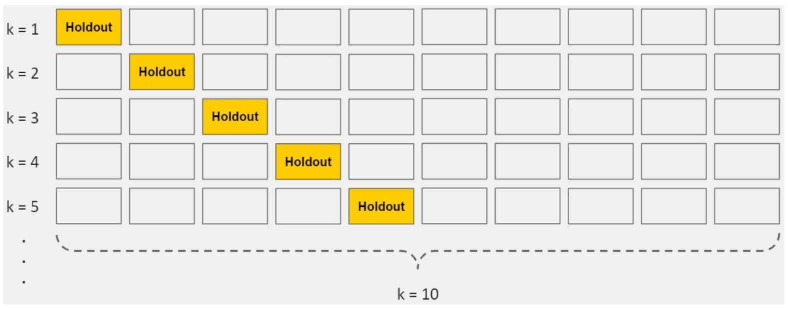


Figure 3.5: Pictorial representation of k-Fold Cross Validation [47], with e.g. $k = 10$.

This technique has the following pros and cons:

Pros: It is less sensitive to how the data splitting is carried out: every data point gets to be in a test set exactly once, and gets to be in training set $k-1$ times. Therefore, the variance of the resulting estimate is reduced as k is increased.

Cons: The disadvantage of this method is that the training algorithm has to be rerun from scratch k times, which means it takes k time as much computation to make an evaluation, thus resulting in evidently longer times to complete the training process.

Leave One Out Cross-Validation

The *Leave One Out CV* is a simplified version of the k-fold one, where the size of the fold is 1 (i.e. $k =$ nb. of entries). The result is a large number of performance measures that can be summarized in an effort to give a more reasonable estimate of the accuracy of your model on unseen data. An evident downside of this procedure is the computational power needed, which is much larger than k-fold CV.

Repeated Random Test-Train Splits

Another variation on k-fold CV is to create a random split of the data like the train/test split described above, but repeating multiple times the process of splitting and evaluation of the algorithm, like a cross-validation. This has the speed of using a simple train/test split and the reduction in variance in the estimated performance associated to k-fold CV. It is also possible to repeat the process as many times as needed to improve the accuracy.

On the other hand, the repetitions in this approach may include much of the same data in the train or the test split from run to run, introducing redundancy into the evaluation.

3.1.4 Examples of Supervised Learning algorithms

When dealing with supervised learning, there are various learning algorithms to choose from. A non exhaustive list is the following:

Support Vector Machines (SVMs): input vectors are mapped to a high-dimension feature space, where a decision surface is constructed, in order to classify the inputs [48];

Decision Trees (DTs): a binary recursive partitioning procedure, where data is subjected to a series of cut based on conditions which are optimized in the learning phase [49];

k-Nearest Neighbour (kNN): data is classified by *clustering* the train set using a user-defined *distance* in the feature space. The probability of belonging to a certain class is evaluated considering the distance of the test from the clusters [49];

Artificial Neural Networks (ANNs): data goes through layers of neurons capable of performing weighted operations on their inputs. ANN architecture is inspired by biological neural connections.

Due to the number of features used to perform the regression and the relative low engineering needed to build an accurate model, ANNs were chosen in this work to be implemented on a Field Programmable Gate Array (see Section 4).

3.2 Artificial Neural Networks

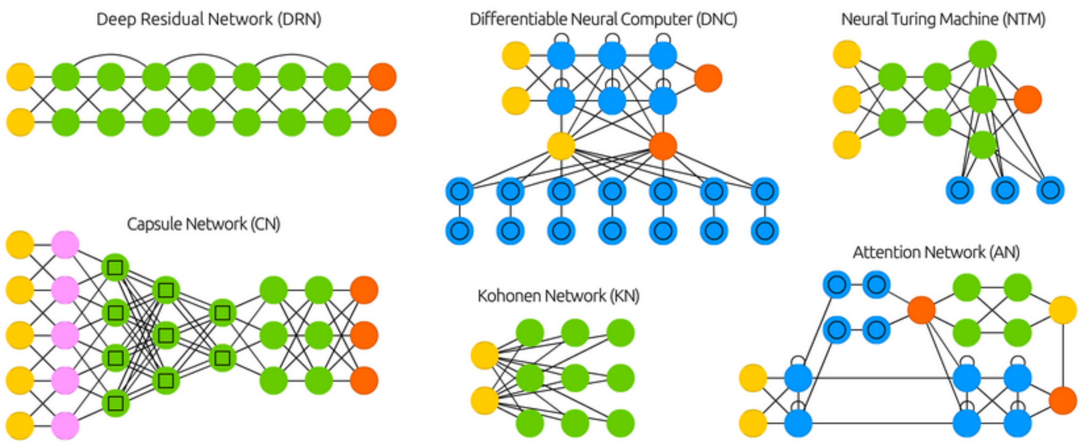
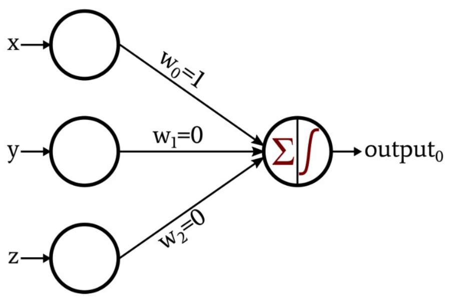
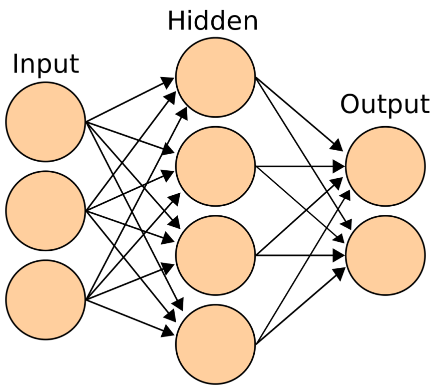

Deep Residual Network (DRN)
Differentiable Neural Computer (DNC)
Neural Turing Machine (NTM)
Capsule Network (CN)
Kohonen Network (KN)
Attention Network (AN)

Figure 3.6: Selection of possible architectures of ANNs [50].

An Artificial Neural Network (ANN) is a network vaguely inspired to the biological neural connections that constitute the human brain, specifically designed to tackle non-linear learning problems. Currently, ANNs, and more specifically Deep Neural Networks (DNN), are one of the main techniques used in ML.



(a) Graphical representation of a Perceptron.



(b) Neural Network basic layout.

Figure 3.7: Simple diagrams representing the layout and functioning principle of Neural Networks.

Among the different architectures of ANN available (see Figure 3.6), the Fully Connected Multilayer Perceptron (MLP) was chosen for this work, due to its relative simple design. MLPs, as the name suggests, are made up of single units called *Perceptrons*. These, as shown in Figure 3.7a, are characterized by [51]:

1. Input Values ($\mathbf{x} = (x, y, z)$);
2. Weights (w_i);
3. Net sum $\sum$;

4. Activation Function $f(\sum)$.

All the input values are multiplied by their weights $(w_i x_i)$. Then, all of these are added together to create the net or weighted sum $(\sum_i w_i x_i)$, which is then given to the *Activation Function* $f(\sum_i w_i x_i)$ which evaluates the perceptron's outputs, adding a non-linearity compared to the simple linear combination alone.

Perceptrons can be stacked together to make a layer of *neurons*, each producing its own outputs. These layers can then be put together to build arbitrarily deep custom networks, by feeding the outputs of a layer to the neurons of the next layer, which will be "hidden" to the user, and resulting in a structure like the one in Figure 3.7b.

Activation functions

As mentioned before, in order to add non-linearities to the model, a specific function is applied to the weighted sum of each perceptron. This functions is usually called *activation function*. Various functions can be selected to fulfill this purpose and a few examples found in literature are:

Sigmoid: a function that converts the weighted sum to a value between 0 and 1:

$$f(x) = \frac{1}{1+e^{-x}} \tag{3.4}$$

Rectified linear unit or ReLU: it usually works better than a smooth function like the sigmoid, being also easier to compute:

$$f(x) = max(0, x) \tag{3.5}$$

Leaky ReLU : a variant of the classic ReLU where the value for inputs smaller than 0 is not simply 0 but:

$$f(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \tag{3.6}$$

with α usually a much smaller number than 1. In this way the number of neurons completely turned off is reduced with respect to ReLU, arguably producing a more adaptive neural network.

3.3 Machine Learning frameworks

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 3.8) there are [52]:

PyTorch [53]: an open source deep learning framework, developed by Facebook, 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 computationwith strong GPU acceleration. With the latest release of PyTorch, the framework provides graph-based execution, distributed learning, mobile deployment and quantization.

TensorFlow [54]: 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.

Theano [55]: a Python library that let the user define, optimize and evaluate mathematical expression, especially ones with multi-dimensional arrays. For problems involving large amounts of data, Theano allows to attain speeds that rival hand-crafted C implementations, and it was demonstrated to also surpass C on a CPU by orders of magnitude via taking advantage of recently released GPU architectures. Architecturally, Theano combines aspects of a computer algebra system with aspects of an optimizing compiler, a combination that is particularly useful for trasks in which complicated mathematical expression need to be evaluated repeatedly and the overall evaluation speed is critical for a project.

Caffe [56]: a deep learning framework developed by Berkeley AI Research (BAIR) and by community contributors. Its expressive architecture encourages application, while models and optimization are defined by configuration and without hard-coding, so it is intended to ease the use of such a framework to a large community of users. The switch between CPU and GPU can be done by setting a single flag. The focus on performance and speed makes Caffe good for research experiments and industry deployment.

The work in this thesis is based on the TensorFlow framework, hence in the next Section 3.3.1 a more accurate description will be given.

3.3.1 Tensorflow

TensorFlow (TF) 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 [58] 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) [59]. Currently, it is one of

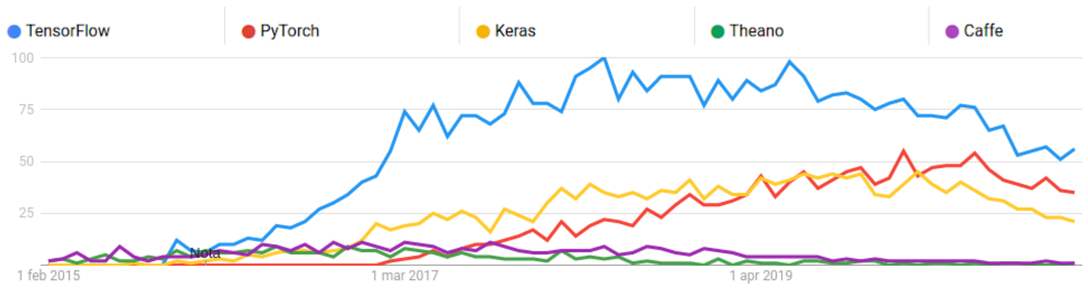


Figure 3.8: Popularity in Google searches of the most popular ML frameworks from January 2010 to February 2021[57]. Numbers represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. A score of 0 means that there was not enough data for this term.

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 application-specific integrated circuits (ASICs) used to accelerate ML workloads [60].

In a not exhaustive list of TensorFlow's features, one could mention [52]:

- TF runs on Windows, Linux and macOS, and also on mobile devices, including both iOS and Android;
- TF provides a Python API which offer flexibility to create all sorts of computations, including any NN architecture one can think of;
- TF 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;
- TF provides several advanced optimization nodes to search for the parameters that minimize a cost function: TF automatically takes care of computing the gradients of the functions one defines, i.e. implements automatic differentiation;
- TF 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 TF, 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 [61] showing how a model that performs regression (see Section 3.1.2) 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.

Each input of data, in TensorFlow, is almost always represented by a tensor, and is often a vector. In supervised training, the output is also a tensor. Here is some data synthesized by adding Gaussian (Normal) noise to points along a line.

```python
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
```

Here the model is defined, together with weights and bias as variables.

```python
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()

# Verify the model works
assert model(3.0).numpy() == 15.0
```

The standard L2 loss can be introduced, also known as MSE (see Section 3.1.1):

```python
# 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 TF, is relatively straightforward:

```python
# 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:

```
# Do the training
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.

Keras

Keras [62] is a deep learning API written in Python, running on top of the machine learning platform TensorFlow: an approachable, highly-productive interface for solving ML problems, with a focus on modern deep learning. However, it is possible to deploy Keras with Theano or TensorFlow 1.0 as back-end. Keras provides essential abstractions and building blocks for developing and shipping ML solutions. It was developed with a focus on enabling fast experimentation, thanks to its simple interface which minimizes the number of user actions required for common use cases and provides clear and actionable feedback upon user error.

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 (as explained in 3.2). 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(train_data, train_label, 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.

3.4 Quantizing Neural Networks

Early efforts in the field of deep learning have focused mostly on the training aspect of neural networks. The success of these efforts has led to widespread deployment of neural networks trained in data centers and on embedded devices, where they are used for inference which in turn emphasized the need to make the inference phase more efficient, especially on hardware that requires low latency, see Chapter 5. Thus, network compression is critical and has become an effective solution to reduce the storage and computation costs for deep NN models.

Quantization, which means conversion of the arithmetic used within the NN from high-precision floating-points to normalized low-precision integers (fixed-point) [63], is an essential step for efficient deployment.

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.

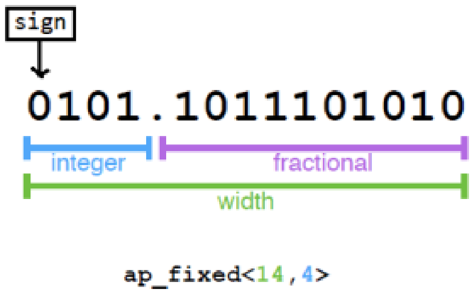


Figure 3.9: Fixed-point number representation. `ap_fixed<`*width*`,`*integer*`>` will be the C type associated to input, output and parameters of the NN model by the hls4ml library in Section 4.2.2.

Fixed-point numbers, instead, consist of two parts, integer and fractional, as shown in Figure 3.9. 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 [64] according to

$$x_Q = \frac{x}{scale} \tag{3.7}$$

where

$$scale = \frac{\max(x) - \min(x)}{2^N} \tag{3.8}$$

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 [65]. 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 [65]): a fixed numerical representation is adopted for the whole model, and the model training is performed enforcing this constraint during weight optimization.

3.4.1 QKeras

To simplify the procedure of quantizing Keras models, the *QKeras* library [66] 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.

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

Listing 3.1: Quantized QKeras layers example.

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 un-quantized Keras layer.

The second code change is to pass approprate quantizers, e.g. `quantized_bits`. In the code snippet 3.1, 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 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:

$$2^{\text{int}-b+1}\text{clip}(\text{round}(x \times 2^{\text{int}-b-1} - 2^{b-1}\, 2^{b-1} - 1) \tag{3.9}$$

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 [67].

The quantizer used for the activation functions in 3.1, `quantized_relu`, is a quantized version of ReLU (see Section 3.2).

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.

An original idea of a neural network for the use case covered in this thesis was indeed implemented on Keras with a Tensorflow backend, improved and modified to best attack the need, migrated whenever necessary to the use of QKeras, and dumped on FPGA resources: this will be discussed in depth in Chapter 5.

Chapter 4

Introduction to FPGAs

In the computer and electronics world, there are two different types of hardware for performing computation: the multipurpose kind (like consumer CPUs) which can be programmed to accomplish a wide range of tasks; and a more specialized kind, such as Application-Specific Integrated Circuits (ASICs), which provides highly optimized resources for quickly perform in critical tasks, but it is permanently configured to only one application via a multimillion dollar design and fabrication effort. Computer software provides the flexibility to change applications and perform a huge number of tasks, but it is orders of magnitude worse than ASIC implementations in terms of performance, silicon area efficiency, and power usage [68].

Field Programmable Gate Arrays (FPGAs) are devices that blend the benefits of both hardware and software. They implement circuits just like hardware, providing huge power, area and performance benefits over software, yet they can be reprogrammed cheaply and easily to implement a wide range of tasks. FPGAs implement computations spatially, simultaneously computing millions of operations in resources distributed across a silicon chip. Such systems can be hundreds of times faster than microprocessor based designs. However, unlike in ASICs, these computations are programmed into the chip, not permanently frozen by the manufacturing process. This means that an FPGA-based system can be programmed and reprogrammed many times. But this flexibility comes at a cost: when producing an ASIC, the main expense is the building of the "master", with a much cheaper production line after development with respect to FPGAs, whose cost per chip is much higher than ASICs. Furthermore, an FPGA programmed to accomplish a specific task will perform worse than an ASIC designed with the same objective in mind.

Figure 4.1 illustrates the internal layout of an FPGA, which is made up of replicated units of digital electronic circuits, called logic blocks, embedded in a general routing structure, hence the *gate* and *array* in the name of this type of devices. The logic blocks contain processing elements for performing simple combinational logic, as well as *flipflops* for implementing sequential logic. Any Boolean combinational function of five or six inputs can be implemented in each logic block. The general routing structure allows arbitrary wiring, so the logical elements can be connected in the desired manner.

Because of this generality and flexibility, an FPGA can implement very complex

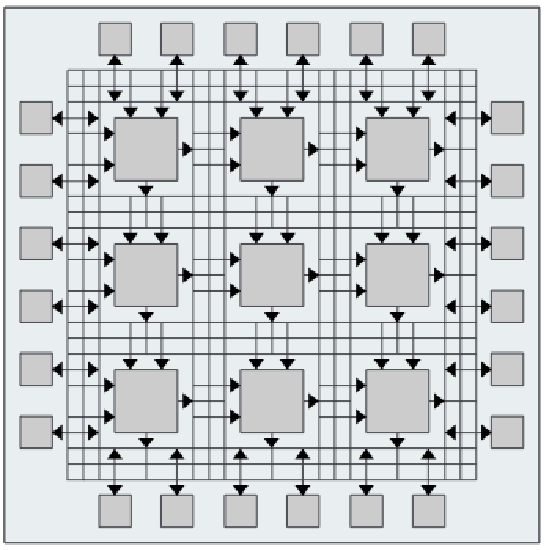


Figure 4.1: An abstract view of and FPGA. Logic cells are embedded in a general routing structure.

circuits. Modern devices can compute functions on the order of millions of basic gates, working with clock frequencies in the hundreds of Megahertz. To boost speed and capacity, in several devices additional ad-hoc elements are embedded into the array, such as large memories, multipliers and even microprocessors. With these predefined, fixed-logic units, which are fabricated into the silicon, FPGAs are capable of implementing complete systems in a single programmable device, by configuring the connections between these units to perform complex algorithms.

4.1 Main logic elements of an FPGA

In very general terms, there are two types of resources in an FPGA, making up the so-called *FPGA fabric*: *logic*, devoted to perform digital operations, and the configurable connections between logic blocks, called *interconnect*. In the next few section, a small selection of logic elements will be described.

4.1.1 Lookup Tables

Any computation can be represented as a Boolean equation and, in turn, these can be expressed as a truth table. From these simple representation, complex structure which can do arithmetic can be built, such as adders and multipliers, as well as decision-making structures that can evaluate conditional statements, such as the classic if-then-else construct. Combining these, elaborate algorithms can be described simply by using truth tables.

From this basic observation of digital logic, the truth table can be seen as the basic computational element of the FPGA. More specifically, one hardware element that can easily implement a truth table is the *Lookup Table* (LUT). From a circuit implementation perspective, a LUT can be formed simply from an N-to-one multiplexer and an N-bit memory. Therefore, using LUTs gives an FPGA the

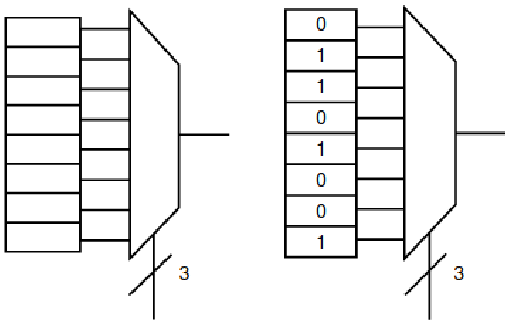


Figure 4.2: A 3-LUT schematic on the left and the corresponding 3-LUT symbol and truth table for a logical XOR on the right.

generality to implement arbitrary digital logic. Figure 4.2 shows a typical N-input LUT that could be find in FPGAs.

The LUT can compute any function of N inputs by simply programming the lookup table with the truth table of the function to be implemented. As shown on the right side of Figure 4.2, if we wanted to implement a 3-input exclusive-or (XOR) function with a 3-input LUT, we would assign values to the lookup table memory such that the pattern of select bits chooses the correct row's "answer". Thus, every "row" would yield a result of 0 except in the four cases when the XOR of the three select lines yields 1.

4.1.2 Flip-Flops

Only LUTs are not sufficient to implement all of the functionality requested from FPGAs. Indeed, implementing digital circuits requires, other than combinatorial logic elements (logic gates), sequential logic ones, in order to deploy algorithms which exploit memory or the concept of "state". To accomplish this need, a simple single-bit storage element is added in the base logic block in the form of a *D flip-flop*.

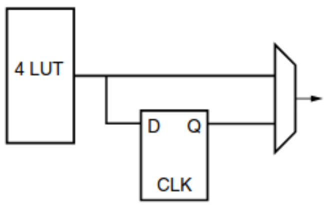


Figure 4.3: A simple LUT logic block with a D flip-flop.

A flip-flop is basically a circuit capable of storing a 0 or a 1. A Delay (or Data) Flip-Flop [69] is a type of flip-flop which delays the transfer of its input to its output. The activation of this type of flip-flop has three external connections: the input, the output and a *clock* input which controls the timing of the transfer of the input value to the output.

With the insertion of the D flip-flop in the logic block, it looks something like Figure 4.3. The output multiplexer on the right selects a result either from the function generated by the lookup table or from the stored bit in the D flip-flop.

The logic blocks of modern FPGAs are made up of multiple LUTs and flip-flops, with multiple programmable connections which are used to reproduce complex functions.

4.1.3 Digital Signal Processors

While it is possible to implement a multiplication with a number of the aforementioned logic blocks, it would cause a large delay penalty or a large logic block footprint. Indeed, these logic blocks are not very efficient at performing multiplications.

Instead of doing it this way, it is possible to delegate this operation to an *ad-hoc* multiplier implemented into the FPGA fabric. Then, instead of inefficiently using simple LUTs to implement a multiplication, the values that need to be multiplied can be routed to actual multipliers implemented in silicon. 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 better.

These units are called Digital Signal Processor (DSP) slices [70]. The DSP slice consists of a multiplier followed by an accumulator, and it can be used to multiply numbers with arbitrary bitwidth. Furthermore, they are usually designed to perform fixed-point arithmetic, which is another reason to quantizing (see Section 3.4) the parameters of a NN model when implementing it on an FPGA.

4.2 Programming an FPGA

FPGA algorithm design is unique with respect to programming a CPU in that independent operations may run fully in parallel, allowing FPGAs to achieve a much higher number of operations per second at a relatively low power cost compared to CPUs and GPUs (Figure 4.4). However, such operations consume dedicated resources onboard the FPGA and cannot be dynamically remapped while running. The challenge in creating an optimal FPGA implementation is to balance FPGA resource usage with achieving the latency and throughput goals of the target algorithm. Key metrics for an FPGA implementation include:

Clock Frequency the frequency of the clock signal used to perform sequential synchronized operations on the FPGA;

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

Resource Usage expressed as the following FPGA resource categories: onboard FPGA memory (BRAM), DSPs, and registers and programmable logic (FFs and LUTs).

The logic and routing elements in an FPGA are controlled by programming interconnections. In these devices, every routing choice and every logic function is controlled by a simple memory bit. With all of its memory bits programmed, by way of a configuration file or *bitstream*, an FPGA can implement the user's

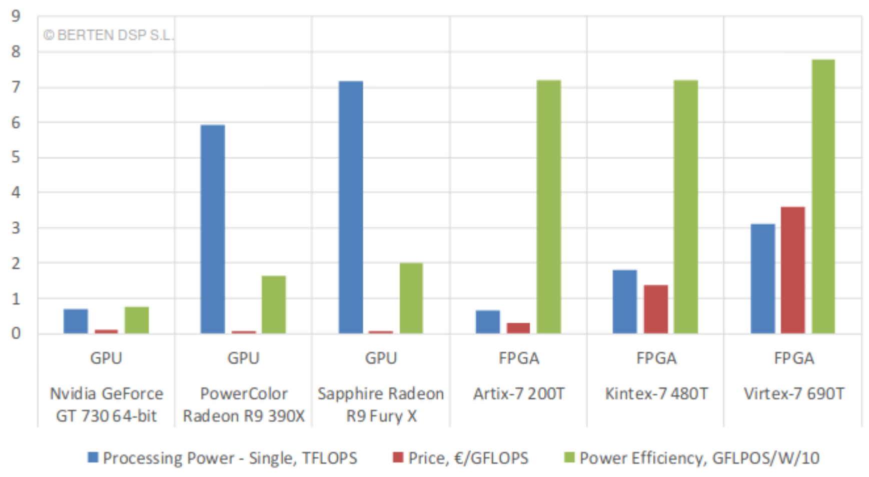


Figure 4.4: GPU vs FPGA performance comparison [71].

desired function. Thus, the configuration can be carried out quickly and without permanent fabrication steps, allowing customization at the user's electronics bench, or even in the final end product. This is why FPGAs are *field programmable*, and why they differ from mask-programmable devices, which have their functionality fixed by masks during fabrication.

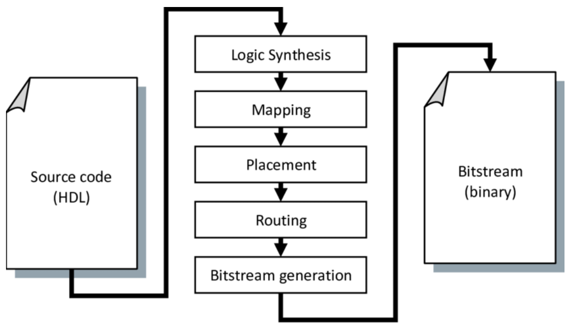


Figure 4.5: A typical FPGA mapping flow.

Customizing FPGA is a process of creating a bitstream to load into the device (see Figure 4.5). Bitsreams can be generated either with programming languages (see Section 4.2.1), from schematics or other proprietary methods. FPGA designers usually start with an application written in a hardware description language (HDL) such as Verilog and VHDL. This abstract design is optimized to fit into the FPGA's available logic through a series of steps: Logic synthesis converts high-level logic constructs and behavioural code into logic gates, followed by technology mapping to separate the gates into groupings that best match the FPGA's logic resources. Next, placement assigns the logic groupings to specific logic blocks and routing determines the interconnect resources what will carry the user's signals. 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/microcontroller downloads the bitstream to the device. There are many known methods for storing the single bits of binary information making up the bitstream. One of the most widely used is volatile static RAM, or SRAM. This method has been made popular because it provides fast and infinite reconfiguration in a well-known technology. Nevertheless, a significant drawback is that SRAM does not maintain its contents without power, which means that at power-up the FPGA is not configured and must be programmed using off-chip logic and storage. Thus, 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 customized electronics device.

The FPGA market is dominated by two main suppliers: Xilinx Inc. [72], acquired by AMD in 2020, and the once called Altera Corporation, acquired by Intel in 2015 [73]. For this work, an evaluation board produced by Xilinx was made available by INFN Bologna. The following tools are offered by the manufacturer to program and make the best use of Xilinx's FPGAs.

Vivado Design Suite

The Vivado Design Suite [74] contains services that support all phases of FPGA designs, starting from design entry, simulation, synthesis, place and route, bitstream generation, debugging, and verification as well as the development of software targeted for these FPGAs. It is an Integrated Development Environment (IDE) that has been built from the ground up to address the productivity bottlenecks in system-level integration and implementation. The edition used in this work was the Vivado HL Design Edition 2018.3. It includes, among other tools:

Vivado High-Level Synthesis compiler (Vivado HLS): a compiler which enables C and C++ programs to be directly targeted into Xilinx devices without the need to manually write in more hardware descriptive languages (like VHDL and Verilog). The concept of HLS will be described in more detail in the next section;

Vivado IP Integrator : it allows to quickly integrate and configure pre-synthesized (but also custom made) blocks (Intellectual Properties or IPs) from the large Xilinx IP library. Its use will be presented in Section 5.4.3;

Vivado Software Development Kit (Vivado SDK): an Integrated Design Environment (IDE) for creating software applications on any of Xilinx's microprocessors embedded into FPGAs.

4.2.1 High Level Synthesis

High-level Synthesis (HLS) [75] is the process of automatic generation of hardware circuit from "behavioral descriptions" contained in a C or C++ program. The target hardware circuit consists of a structural composition of data path, control and memory elements. HLS acts as a bridge between hardware and software domains [76], providing an improvement in productivity for hardware designers who can work at a higher level of abstraction while creating high performance hardware;

and 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 a HLS design methodology allows to develop algorithms at the C-level (in programming languages like C and C++) with typically shorter development time. Moreover, it is more easier to validate functional correctness at this level than with traditional HDLs.

The fundamental tasks in HLS are decomposed into:

Hardware modeling concerns transforming the application described in C or C++, into a synthesizable logic circuit description where operations are placed in order of execution, according to the C application;

Task scheduling schedules operations by assigning these to specific clock cycles or by building a function that determines execution time of each operation at the runtime. In other words, a time schedule for all ordered operations listed in the previous step is created;

Resource allocation and binding determine which resources and their quantity needed to build the final hardware circuit. Binding refers to specific binding of an operation to a resource (such as a functional unit, a memory, or an access to a shared resource).

Control generation and optimization creates a series of control signals which can be used to keep track and control the execution of the final product.

This decomposition of HLS tasks is automatically performed by proprietary software applications to translate sequential languages like C and C++ to a parallel implementation on sequential circuits.

To perform these tasks, HLS compilers synthesize a C code as follows:

1. Top-level function arguments synthesize into I/O ports;

2. C functions synthesize into blocks in the logic circuit. If the C code includes a hierarchy of sub-functions, the final design will include a hierarchy of modules or entities that have a one-to-one correspondence with the original C function hierarchy. All instances of a function use the same block by default.

3. Loops in the C functions are kept *rolled* by default: when loops are rolled, synthesis creates the logic for one iteration of the loop, and the design executes this logic for each iteration of the loop in sequence, managed by a properly generated control logic. Using optimization directives, it is possible to *unroll* loops, which allows all iterations to occur in parallel, by replicating hardware resources.

4. Arrays in the C code synthesize into Block RAM (BRAM) or UltraRAM [1] in the final FPGA design. If the array is on the top-level function interface, HLS implements the array as ports to access a BRAM outside the design.

[1]Block RAM e Ultra Ram are the product names of two different RAM designs built by Xilinx for FPGAs.

HLS creates an optimized implementation based on default behaviour, constraints, and any specified optimization directives. It is possible to use optimization directives to modify and control the default behaviour of the internal logic and I/O ports. This allows to generate variations of the hardware implementation from the same C code.

4.2.2 The hls4ml package

The *hls4ml* package [77] was developed by members of the High Energy Physics (HEP) community to translate ML algorithm, built using frameworks like TensorFlow 3.3.1, into HLS code. In this work hls4ml was used to perform this transformation on a trained NN, defined by its architecture, weights, and biases. A schematic of a typical workflow is illustrated in Figure 4.6.

The part of the workflow illustrated in red indicates the usual software workflow required to design a NN for a specific task 3.3.1. 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. 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 (in this example the model was built using QKeras) [78]:

```python
import hls4ml
from qkeras.utils import load_qmodel

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

Then, a *configuration* has to be created:

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

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`). This flag sets the level of fineness wanted for the parameter tuning. By using `name`, it is possible to configure each layer and activation function individually. While, `type` is used if the developer wants to share the configuration between all layers of the same type. And finally, with `model` a single configuration is used for the entire model.

By modifying the configuration dictionary it is possible to change the arithmetic precision used for weights, biases and results. After the configuration, the model can be converted by specifying, other than the model object and configuration, the output folder and the FPGA model where the project will be implemented:

```python
hls_model = hls4ml.converters.convert_from_keras_model(model,
    hls_config=config, output_dir='HLS_Project', fpga_part='xczu9eg-
    ffvb1156-2-e')
```

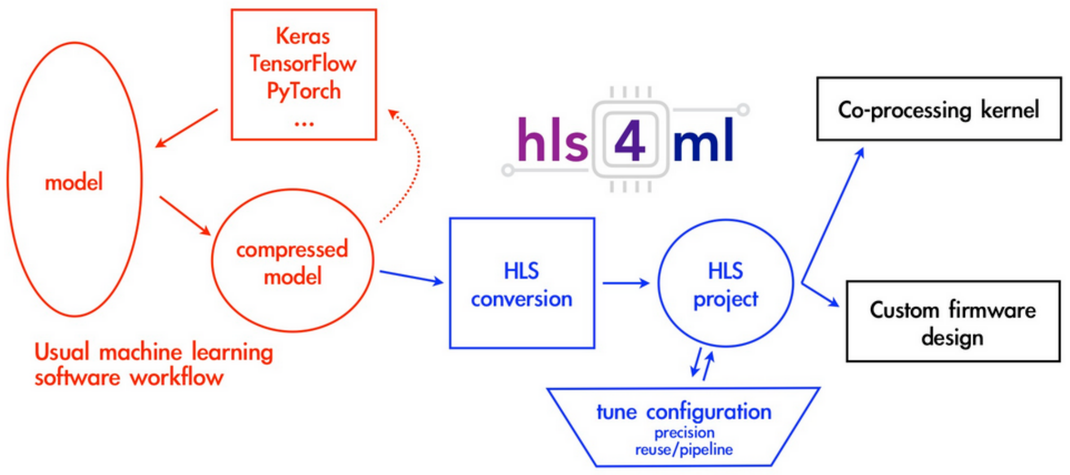


Figure 4.6: A typical workflow to translate a model into an FPGA implementation using hls4ml.

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

It is clear by the couple of lines of code shown, how it is easy to create the HLS project, making it feasible also for people who are not expert in FPGAs or hardware in general. In Section 5.4.2 the configuration used in this work will be presented, together with the steps to follow to go from an HLS project to running the final application on an FPGA.

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 5

Muon momentum assignment using Machine Learning on an FPGA

In Chapter 3, an overview on ML has been presented, together with a simple tutorial on how a NN can be built, trained and later used for predictions. After that, in Chapter 4, FPGAs were presented and the potential in using HLS was discussed.

The main purpose of this thesis concerns the implementation of an ML algorithm on an FPGA, using data coming from local trigger segments (trigger primitives) from the DT chambers (explained in more detail in Chapter 2) at the CMS experiment, in order to perform *muon p_T assignment* predictions. This work is built upon [79], where the possibility of exploiting NNs to perform muon p_T assignment was initially explored.

After a brief description of the physical problem analysed, this chapter will be articulated into four main sections:

1. The first section will describe the muon p_T assignment process for the CMS L1T;

2. The second section will provide an overview of the data used and the preprocessing required in order to make them suitable for a ML model (as described in more detail in [79]);

3. The third section will show the design and optimization of the model, training and test, as well as the predictions for new unseen data;

4. The fourth section will show the steps needed for converting the model built using Python, firstly into an HLS project (as described in general in Section 4.2.2), and then into an application which can be launched on an FPGA;

5. The fifth section will compare the predictions coming from such models with the actual L1T algorithm running in CMS.

All the code written for this thesis and the output data are available in a digital repository [80].

5.1 The muon p_T assignment

Detection of muons is very important in the CMS experiment. Muons are expected to be produced in a large set of physics signatures studied as part of the CMS physics programme: for instance, the *golden* decay channel of the Higgs Boson, sees the production of four muons [81], as shown in Figure 5.1. Since muons can penetrate several metres of iron with little energy loss, unlike most particles, they are not stopped by any layer of the CMS calorimeters. That is why the CMS muon detectors (described in more detail on Chapter 2) are placed far from the interaction point.

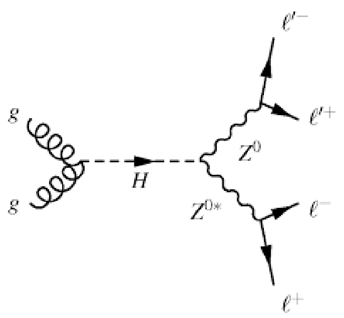


Figure 5.1: Higgs boson's *golden* decay into 4 leptons ($l = e$, μ, τ).

During CMS Phase-1, the trigger system tracked muons through the four stations of the barrel system in two steps. First, individual hits from single chambers are combined by local trigger electronics (see Section 2.3), resulting in track segments (trigger primitives) encoding local position and direction. Then, a Track Finder system combines segments from different stations into muon track candidates. The present Barrel Muon Track Finder (see Section 2.4) estimates the muon transverse momentum in a relatively simple way, using LUTs (see Section 4.1.1) addressed by the difference in the azimuthal position of the trigger primitives from the two innermost stations participating in the track and the ϕ_B of the most internal one.

Improving the transverse momentum measurement performed by the trigger, namely the momentum resolution, is very important to achieve a reduction of trigger rates. In fact, due to the rapidly decreasing shape of the inclusive muon p_T spectrum, even a relatively small reduction of the resolution can provide a significant decrease of the trigger rate at a given p_T threshold, by reducing the number of low momentum muon candidate misidentified as high momentum ones. This becomes particularly pressing in the context of future upgrades of CMS, in view of the High Luminosity LHC (see Section 1.3) upgrade, to avoid using higher momentum thresholds as luminosity increases, with the consequence of losing physics acceptance.

The Phase-1 algorithm for Level-1 muon p_T assignment is based on the use of a fraction of all the available information from the trigger primitives: as a matter of fact, firstly it assumes that the muon comes from the vertex and, secondly, it uses just the ϕ_B (see Figure 2.12) of the most internal station (as long as it is not MB3, due to the magnetic field disposition which places the trajectory's saddle point in this station) and the the ϕ angles of the two most internal stations. A ML approach, especially in the implementation of an Artificial Neural Network, would, instead, make use of all the angle differences, all ϕ_B, the position and direction within a chamber of all trigger primitives, the station/sector/wheel coordinates of the chambers generating the primitives. Furthermore, possible non-linear relations among the input data can be explored through the use of different and deeper network architectures, opening the way for supervised regression algorithms, hence allowing to span a much wider parameter space. By implementing the ML algorithm on an FPGA, this work tries also to search for ways to make the p_T prediction faster, other than more accurate, hence opening the way for designing a track fitting technique at the L1T, possibly achieving better performance than KBMTF included in the Phase-2 CMS Upgrade (see Section 2.5).

In this work, ML predictions are used to provide a momentum measurement limited to the only trigger primitives (at chamber level) associated to muon track candidates built by the present Track Finder system, which will also provide the reference performance in terms of momentum assignment accuracy. In this way it is possible to understand how the new solution proposed in this thesis compares with the current L1T algorithm.

5.2 Data preparation

The first important step prior to model creation is data preparation. In a standard ML problem, this is a delicate phase that deals with aspects such as data cleansing, data scaling (normalization and binning), feature inspection and eventually feature engineering. The data preparation was performed following [79], where it is explained in more details.

The data used was generated out of a simulated sample of muons with different energy and direction crossing the CMS detector and, as anticipated in the previous section, only trigger primitives were used for a more direct comparison with the actual L1T algorithm. Each generated muon passing through the barrel DT chambers results into several trigger primitives. Each muon has an associated number corresponding to the number of chambers crossed by its trajectory. Therefore, inside a plain *csv* file, extracted from a *TTree* in a *ROOT* [82] file (using the Python macro described in [79]), all the necessary information was stored, both the physical information of each simulated muon and the respective primitive hits in the detector, up to the L1T information.

The variables in the *csv* file are the following:

1. **n_Primitive** is an integer value corresponding to the number of primitives contained in that record.

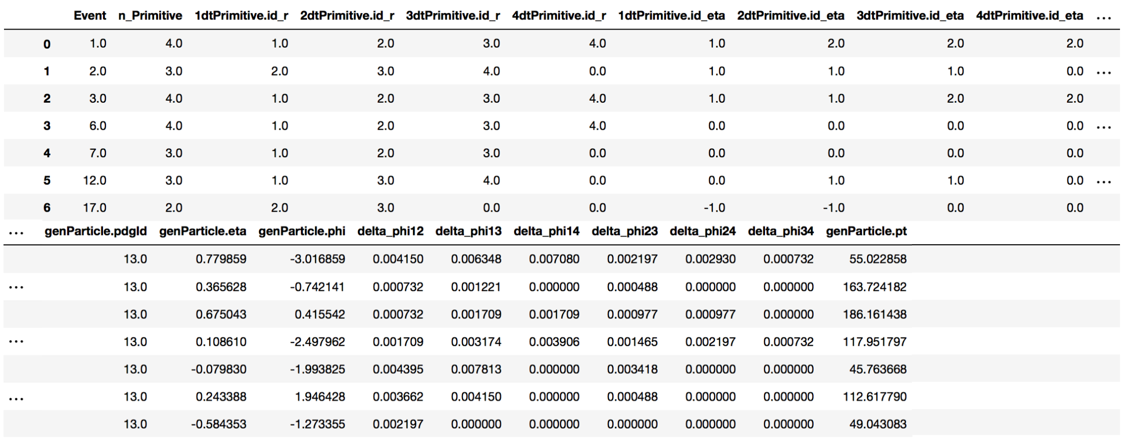
2. *dtPrimitives* contains the variables of muon local segments from the DT chambers (Section 2.1.1):

   dtPrimitive.id_r an integer with the identifier of the trigger primitive station (ranging from 1 to 4);

   dtPrimitive.id_phi an integer with the identifier of the trigger primitive sector (ranging from 1 to 12);

TRACO code	Quality label	Meaning
6	HH	Correlated H+H trigger (outer and inner SL)
5	HL	Correlated H+L or L+H trigger
5	LL	Correlated L+L trigger
4	H_O	Uncorrelated High trigger (outer SL)
3	H_I	Uncorrelated High trigger (inner SL)
2	L_O	Uncorrelated Low trigger (outer SL)
1	L_I	Uncorrelated Low trigger (inner SL)
7	/	Null Track

Table 5.1: TRACO quality code conversion table. See Section 2.3 for terminology clarification.



	Event	n_Primitive	1dtPrimitive.id_r	2dtPrimitive.id_r	3dtPrimitive.id_r	4dtPrimitive.id_r	1dtPrimitive.id_eta	2dtPrimitive.id_eta	3dtPrimitive.id_eta	4dtPrimitive.id_eta	…
0	1.0	4.0	1.0	2.0	3.0	4.0	1.0	2.0	2.0	2.0	
1	2.0	3.0	2.0	3.0	4.0	0.0	1.0	1.0	1.0	0.0	…
2	3.0	4.0	1.0	2.0	3.0	4.0	1.0	1.0	2.0	2.0	
3	6.0	4.0	1.0	2.0	3.0	4.0	0.0	0.0	0.0	0.0	…
4	7.0	3.0	1.0	2.0	3.0	0.0	0.0	0.0	0.0	0.0	
5	12.0	3.0	1.0	3.0	4.0	0.0	0.0	1.0	1.0	0.0	…
6	17.0	2.0	2.0	3.0	0.0	0.0	-1.0	-1.0	0.0	0.0	

…	genParticle.pdgId	genParticle.eta	genParticle.phi	delta_phi12	delta_phi13	delta_phi14	delta_phi23	delta_phi24	delta_phi34	genParticle.pt
	13.0	0.779859	-3.016859	0.004150	0.006348	0.007080	0.002197	0.002930	0.000732	55.022858
…	13.0	0.365628	-0.742141	0.000732	0.001221	0.000000	0.000488	0.000000	0.000000	163.724182
	13.0	0.675043	0.415542	0.000732	0.001709	0.001709	0.000977	0.000977	0.000000	186.161438
…	13.0	0.108610	-2.497962	0.001709	0.003174	0.003906	0.001465	0.002197	0.000732	117.951797
	13.0	-0.079830	-1.993825	0.004395	0.007813	0.000000	0.003418	0.000000	0.000000	45.763668
…	13.0	0.243388	1.946428	0.003662	0.004150	0.000000	0.000488	0.000000	0.000000	112.617790
	13.0	-0.584353	-1.273355	0.002197	0.000000	0.000000	0.000000	0.000000	0.000000	49.043083

Figure 5.2: Data extract from the plain *csv* input file containing the input data.

dtPrimitive.id_eta an integer with the identifier of the trigger primitive barrel wheel (ranging from -2 to 2);

dtPrimitive.phiB the bending angle of the trigger primitive (see Figure 2.12 for reference);

dtPrimitive.quality an integer with a value representing the quality of the hit (a map of such values for the quality of the primitive is shown in Table 5.1).

3. *l1Muons* contains the variables of muons exiting the L1T (after the correct BX assignment and the BMTF, described in Section 2.4):

l1Muon.eta and **l1Muon.phi**: the value of the η and ϕ angles of the muon track estimated by the L1T;

l1Muon.charge : an integer representing the charge of the muon (-1 or +1);

l1Muon.quality : an integer with a value of quality given by the L1T depending on the type of Track Finder involved (in this case the BMTF). Ranges from 0 to 15, an higher value implies, in general, the use of more trigger primitives and/or the use of primitives from the innermost stations.

l1Muon.pt : the estimation of the p_T made by the L1T system. This variable is quite important as it is used to compare the performance of the p_T assignment by the analyzed model.

Both *dtPrimitives* and *l1Muons* features are repeated 4 times for each entry in order to describe these quantities for the trigger primitive reconstructed in the 4 stations MB1, MB2, MB3 and MB4.

4. *genParticles* contains the variables of the generated muons. These are the "true" physical values since the original interacting muon is generated and not coming from measured data:

   genParticle.status an integer flag assuming different values depending on the generation process. All particles have a status flag equal to 1 being the ones propagated from the generator to the GEANT [83] simulation;

   genParticle.pdgId : Particle Data Group code of the generated particle (-13 for muons and +13 for anti-muons);

   genParticle.eta and **genParticle.phi**: "real" value of the η and ϕ angles of the muon;

   genParticle.pt : transverse momentum of the generated muon. This variable will be used as the target from the ML algorithm, i.e. the quantity the model has to learn.

5. **delta_phiNM** (with $NM = 12, 13, 14, 23, 24, 34$): these columns are filled with the difference, in module, of the ϕ angle, expressed in global detector coordinate, of the different trigger primitives linked to the same muon. Physically, the p_T of a particle, crossing the detector, is inversely proportional to the bending angle caused by the effects of the strong magnetic field produced by the solenoid magnet. Therefore, there is a *delta_phi* for each combination of station pairs (MB1-MB2, MB1-MB3, etc...). In the eventuality of no primitive in a certain section, a placeholder (0) is inserted.

In Figure 5.2, a snapshot of the data stored in a plain *csv* file is shown.

5.3 Choosing the model

NNs were chosen as the ML algorithm analyzed in this work due to their flexibility with respect to the alternatives (as described in detail in [79]): instead of following a set of predefined instruction to solve a problem, NNs process information by a large number of interconnected processing elements working in parallel (as shown in Section 3.2); the resulting model is also deployable in other machines and can be tested with new datasets with little to no modifications required.

The following sections will provide a detailed description of the process needed to create such networks, using a Python macro with QKeras framework environment (described in Section 3.4.1).

Input dataset and feature selection

For this specific analysis, the entire datased used contains about 300000 generated muons with a range in p_T from 3 to 200 GeV/c, distributed uniformly. Feature selection is the first step in the creation of the NN model: from the list presented in Section 5.2, only the variables concerning the trigger primitives were effectively used as input feature:

- **n_Primitive** being the number of primitives used for the track reconstruction related to the accuracy of its p_T assignment;
- **dtPrimitive.id_r**, **dtPrimitive.id_eta**, **dtPrimitive.id_phi** for the location of the primitive in the detector;
- **dtPrimitive.phiB** and **delta_phi** (one for each combination of stations) for their correlation to the muon p_T;
- **dtPrimitive.quality** for the correlation between the number of hits used for the construction of the primitive, which has an impact on the resolution of ϕ_B.

As already stated in the previous section, each *dtPrimitive* variable refers to one trigger primitive only; each muon, however, has more than one TP (*n_Primitive* indicate the exact number of primitives for a certain muon record, up to 4). For this reason, every variable has an additional integer number at the beginning of their name indicating the primitive number. For instance, **dtPrimitive.id_r** in the list above actually means: **1dtPrimitive.id_r**, **2dtPrimitive.id_r**, **3dtPrimitive.id_r** and **4dtPrimitive.id_r**, for the other variables as well.

Target variable

The target variable is **genParticle.pt**, which contains the p_T of the simulated muon, distributed from 3 to 200 GeV/c and normalized between 0 and 1 since, in a NN environment (but this logic can be extended to all ML algorithms), the target variable is supposed to be normalized in order to be learned more easily.

5.3.1 Neural Network architecture

In [79], a fully connected NN (built using Keras, see Section 3.3.1) was studied with 2 hidden layers of 1000 and 50 nodes respectively. However, in order to find alternatives which reduce the size of the NN and are more accurate, two different NN architectures have been studied in this thesis:

1. a Fully Connected MLP with 3 hidden layers containing 80, 50 and 35 nodes respectively (from here onwards it will be called S model);
2. a Fully Connected MLP with 5 hidden layers made up of 60, 50, 30, 40, 15 nodes each (D model).

Both were implemented using QKeras as follows (the description of the functions used in this snippet can be found in Section 3.3):

```python
# First model (shallow)
model = Sequential()
model.add(QDense(80,  input_shape=(27,),
                      kernel_quantizer=quantized_bits(16,1),
                      bias_quantizer=quantized_bits(16,1),
                      kernel_initializer='random_normal'))
model.add(QActivation(activation=quantized_relu(16,1),
                name='relu1'))
model.add(QDense(50,kernel_quantizer=quantized_bits(16,1),
                      bias_quantizer=quantized_bits(16,1)))
model.add(QActivation(activation=quantized_relu(16,1),
                        name='relu2'))
model.add(QDense(35,kernel_quantizer=quantized_bits(16,1),
                      bias_quantizer=quantized_bits(16,1)))
model.add(QActivation(activation=quantized_relu(16,1),
                        name='relu3'))
model.add(QDense(1,kernel_quantizer=quantized_bits(16,1)))
model.add(QActivation(activation=quantized_relu(16,1),
                        name='relu4'))
# Second model (deep)
model2 = Sequential()
model2.add(QDense(60,  input_shape=(27,),
                      kernel_quantizer=quantized_bits(16,1),
                      bias_quantizer=quantized_bits(16,1),
                      kernel_initializer='random_normal'))
model2.add(QActivation(activation=quantized_relu(16,1),
                        name='relu1'))
model2.add(QDense(50,kernel_quantizer=quantized_bits(16,1),
                      bias_quantizer=quantized_bits(16,1)))
model2.add(QActivation(activation=quantized_relu(16,1),
                        name='relu2'))
model2.add(QDense(30,kernel_quantizer=quantized_bits(16,1),
                      bias_quantizer=quantized_bits(16,1)))
model2.add(QActivation(activation=quantized_relu(16,1),
                        name='relu3'))
model2.add(QDense(40,kernel_quantizer=quantized_bits(16,1),
                      bias_quantizer=quantized_bits(16,1)))
model2.add(QActivation(activation=quantized_relu(16,1),
                        name='relu4'))
model2.add(QDense(15,kernel_quantizer=quantized_bits(16,1),
                      bias_quantizer=quantized_bits(16,1)))
model2.add(QActivation(activation=quantized_relu(16,1),
                        name='relu5'))
model2.add(QDense(1,kernel_quantizer=quantized_bits(16,1)))
model2.add(QActivation(activation=quantized_relu(16,1),
                        name='relu6'))
```

Listing 5.1: Python script to build two NN models using the QKeras library.

In both cases the first hidden layers presents a gaussian random number as kernel initializer (for the initial randomized weight assignment). Between layers, the activation functions (ReLUs, see Section 3.2) are defined. All the object initialized here are quantized, as coming from the QKeras library, with a bitwidth of 16 bits of which 1 is used to represent the integer part of the value (another bit is used for the sign).

5.3.2 Neural Network optimization via weight pruning

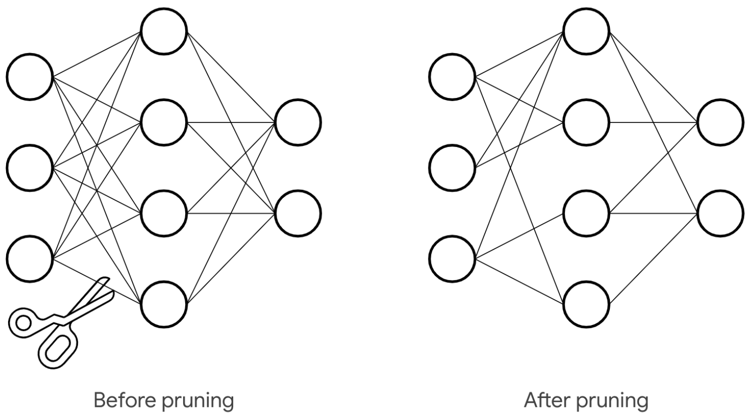


Figure 5.3: Graphical representation of the weight pruning optimization.

When building a NN model, the final hardware platform where the inference computation will be run, has to be considered. In order to minimize the resource utilization when a NN is targetted to FPGA, further optimization was needed with respect to [79], aimed at reducing the number of parameters and operations involved in the computation, by removing connections, and thus parameters, in between NN layers. This process is commonly called *weight pruning*, i.e. the elimination of unnecessary values in the weight tensor, by practically setting the NN parameters' values to zero, which will be translated into "cut" connections between nodes of the NN during the synthesis of the HLS design (see Section 5.4.2). The pruning is done during the training process to allow the NN to adapt to the changes. The TensorFlow Sparsity Pruning (TFSP) API [84] was selected 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 5.4).

At the end of the training procedure, the tensors corresponding to the pruned layers will contain zeros according to the final sparsity target for the layer. Looking at Table 5.2 (which concerns the second model built in the previous section), it is clear how this technique reduces the number of weights different from zero, hence the number of operations performed when the NN is used. This causes a reduction of the number of DSPs (see Section 4.1.3) scheduled to be used by Vivado HLS during the synthesis of the NN (see Section 5.4.2).

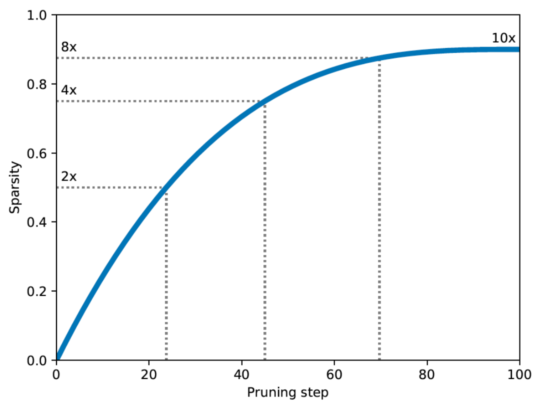


Figure 5.4: Example of sparsity ramp-up function with a schedule to start pruning from step 0 until step 100, and a final target sparsity of 90%. [84]

	Weights		Biases		Total	
	Complete	Pruned	Complete	Pruned	Complete	Pruned
1st layer	1620	409	60	60	1680	469
2nd layer	3000	750	50	50	3050	800
3rd layer	1500	375	30	29	1530	404
4th layer	1200	300	40	40	1240	340
5th layer	600	150	15	15	615	165
Output layer	15	4	1	1	16	5

Table 5.2: Table showing the results of the pruning technique for the D model. There is a complete list of parameters for every layer before and after pruning. In both cases the model was built using QKeras.

5.3.3 Training the Neural Network

The following lines of code were used to start the NNs training with weight pruning enabled:

```
pruning_params = {"pruning_schedule" :
                        pruning_schedule.ConstantSparsity(0.75,
                                                begin_step=2000,
                                                frequency=100)}
model = prune.prune_low_magnitude(model, **pruning_params)
adam = Adam(lr=0.001)
model.compile(loss='mean_squared_error', optimizer=adam)
model.fit(X_training, y_training,
            batch_size=size,
            epochs=epochs,
            verbose=1,
            validation_data=(X_validation, y_validation),
            callbacks=[history, pruning_callbacks.UpdatePruningStep()])

model = strip_pruning(model)
```

The `pruning_params` dictionary contains the pruning schedule as an object from the TFSP library, initialized providing: the final sparsity target, the initial step when starting pruning and after how many steps perform the procedure again. The model is then wrapped in the `prune.prune_low_magnitude` container, allowing the recovery of the original QKeras model by simply calling the `strip_pruning()` function after the training is completed.

By calling the `model.compile()` method, the loss function and the optimizer (described in Section 3.2 and Section 3.1.2 respectively) are set. The *Adaptive Momentum estimation* (Adam) [85] was used as optimizer: it is a method for efficient stochastic optimization that only requires first-order gradients with little memory requirement. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments (mean and variance) of the gradients.

	Training RMSE	Validation RMSE
S model	0.102409	0.105033
D model	0.109454	0.112008

Table 5.3: Training and validation scores of both NN models studied in this thesis.

The actual training was initiated with the `model.fit()` method with the following arguments:

X_training and **y_training** training data and "true" values used for training

batch_size number of entries used to perform the gradient descent at once. The training iteration ends when all dataset is used, one batch at a time;

epochs number of times the entire training set is used to train the model;

verbose verbosity of the output;

validation_data validation set (see Section 3.1.3)

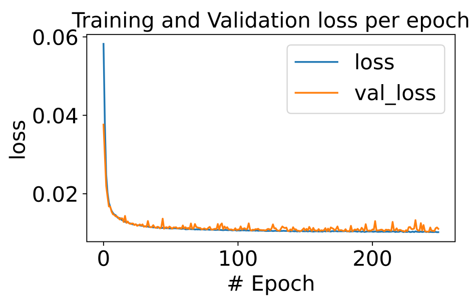
callbacks functions called during the training process to keep track of the algorithm and apply the pruning procedure.

In order to validate the models, a train/test split of 80/20% was performed. Using 200 epochs for the first model and 130 for the second, and a batch size of 300 records for both, Table 5.3 shows the results for both the training and validation phases. Figures 5.5a and 5.5b show the loss value trend over the iterations (epochs) during training and validation: for each iteration, the minimization of loss is noticeable, ending up at an approximately constant value.

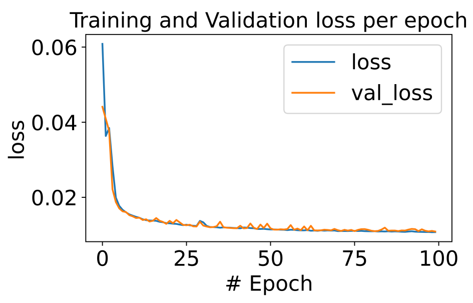
5.3.4 Neural Network model performance on CPU

The final step for the model creation is testing such models using a separated dataset. The QKeras models trained in the previous section were saved into a HDF5 file [86], containing:

- The architecture of the model, in order to be reproduced when it is needed to be used with other datasets;



(a) S model loss values.



(b) D model loss values.

Figure 5.5: These plots show the trend of the loss function for each iteration (epoch) of both pruned models under study. The blue line indicates the training phase while the orange line the validation phase. Notice that, for each epoch, the loss function is always minimized.

- The weights of the model;
- The training configuration (loss, optimizer);
- The state of the optimizer, allowing to resume training exactly where it was left.

Then, using another *csv* file with an independent set of simulated muons (about 15k muons), created and processed in the same way as the training sample, it was possible to load the models and predict the muon p_T by adding the following lines of code

```
model = load_qmodel("model.h5")
predictions = model.predict(X_test)
```

The results of the prediction were compared to the p_T estimated by the p_T assignment unit of the BMTF (see Section 2.4). The analysis code performed this steps:

1. Loading of the model;
2. For each generated muon, a $dR = \sqrt{d\phi^2 + d\eta^2}$ matching with the L1T tracks was carried out: in this way only muons with corresponding Level-1 tracks were considered, factoring the Level-1 reconstruction efficiency out of the study;
3. If dR were below a certain threshold, the track was selected and histograms were filled with that candidate.

The input data for the analysis contained about 14000 muon entries. Two different types of plots were produced as output:

- p_T resolution histograms;
- Efficiency curves computed as function of the generated muon p_T ("turn-on curves").

$\mathbf{p_T}$ resolution histograms

The p_T resolution histograms, for each generated muon, are filled using the following relation:

$$\frac{\Delta p_T}{p_T} = \frac{p_{T_{est}} - p_{T_{gen}}}{p_{T_{gen}}} \tag{5.1}$$

where $p_{T_{est}}$ is the estimation of the transverse momentum, given by the model prediction or the L1T p_T assignment unit, and $p_{T_{gen}}$ is the generated transverse momentum. 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 will get. In the case under study, the distributions' mean will vary in a small range of values, and this is why this effect was not taken in consideration in the following remarks about the results.

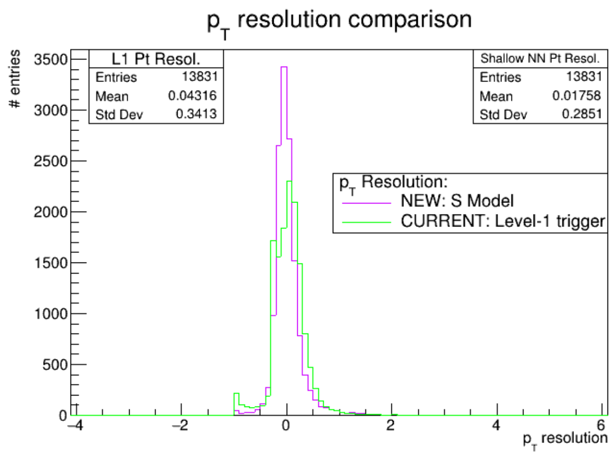


Figure 5.6: Transverse momentum resolution histograms computed for the smaller ML-based model (violet) and L1T (green) based momentum assignment, computed for muons generated in the 3 - 200 GeV p_T range.

Figure 5.6 and 5.7 show the p_T resolution for the entire range analysed (from 3 to 200 GeV/c) for both produced models. The green line indicates the resolution of the L1T system while the violet (for the S model) and the blue (for the D model) lines indicate the resolutions of the predictions made by the NN models. In particular, it is possible to notice a less broad distribution for the ML resolution, resulting in a overall improvement (yet small) with respect to the L1T system. Another noticeable detail, from the L1T curve, is the small peak corresponding to the value -1: this happens when the p_T assigned by the L1T is significantly underestimated with respect to the generated muon p_T. The ML momentum assignment is less prone to large p_T underestimation. As will be discussed in the next section, this difference has implication in the efficiency of triggering on muons when a minimal p_T cut is applied on the L1T track.

In general, an additional feature visible in all the p_T resolution plots is the scale: the peaks of the ML-based distributions are closer to zero if compared with

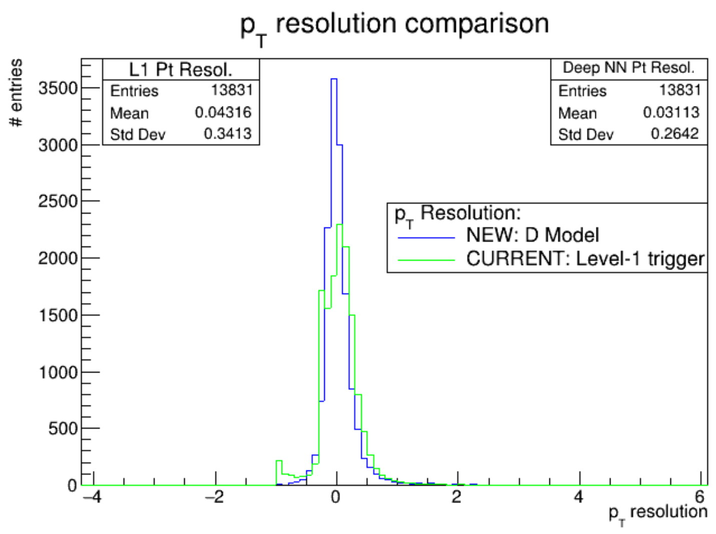
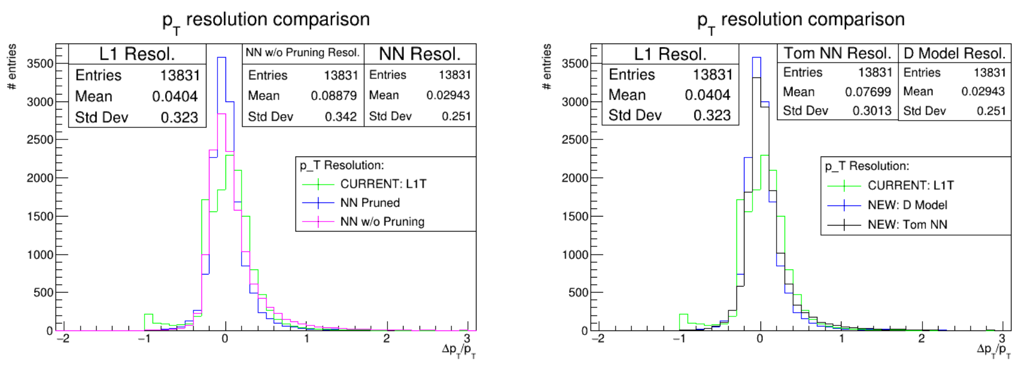


Figure 5.7: Transverse momentum resolution histograms computed for the deeper ML-based (blue) and L1T (green) based momentum assignment, computed for muons generated in the 3 - 200 GeV p_T range.

the L1T ones. This is due to a different calibration of the p_T scale in the two cases. The L1T p_T assignment is in fact calibrated as the point for which the efficiency turn-on curve (described in the next section) of muons with fixed p_T reaches the 90% value. On the other hand, the ML attempts to estimate the generated muon p_T and is therefore characterized by a smaller scale bias.



(a) *D* Model with v. w/o weight pruning. (b) *D* model v. model analyzed in [79].

Figure 5.8: Transverse momentum resolution histograms computed for the deeper ML-based (blue) and L1T based momentum assignment, computed for muons generated in the 3 - 200 GeV p_T range, and compared with D model without the weight pruning optimization (see Section 5.3.2) (magenta) on the left, and the resolution obtained in [79] (black) on the right.

In Figure 5.8a, the p_T resolution for the D model without the weight pruning optimization (see Section 5.3.2) is shown for reference. The results appear not affected badly by this technique, instead the distribution appear slightly more biased towards higher values with respect to the D model that will be used from here onwards. Figure 5.8b depicts the comparison between the model described in [79] and the D model, showing an overall increase in both peak position and standard deviation of the p_T distribution.

The increase in performance after the pruning process could be explained by the fact that pruning "instability" (i.e. the test accuracy immediately following a pruning step) is positively correlated with the final level of generalization attained by the pruned model [87]. However, when dealing with ML fine-tuning and optimization, there is no defined path towards better results. Indeed, the process could be described as a continuous search for an optimal trade-off between better performance and more optimization among various metrics (speed, accuracy, resource usage, etc.), trying different alternatives at the same time which could sacrifice some aspects in order to improve others. More studies ought to be done in order to understand better the dependencies at play in the "generalization-stability tradeoff" and make a clearer path towards the optimal working point of a given ML algorithm.

Efficiency turn-ons

The efficiency turn-ons, as the word states, represent an efficiency ϵ defined as follows:

$$\epsilon = \frac{\text{number of muons that passes a cut at a defined } p_T \text{ threshold}}{\text{total number of muons}} \tag{5.2}$$

After the definition of a minimal p_T threshold cut, the efficiency numerator is filled if the p_T value from the ML or the Level-1 p_T assignment is above threshold. Then the ratio with the denominator is performed. As outlined above, both numerator and denominator are filled only in case a geometrical match between a L1T muon track and a generated muon is found.

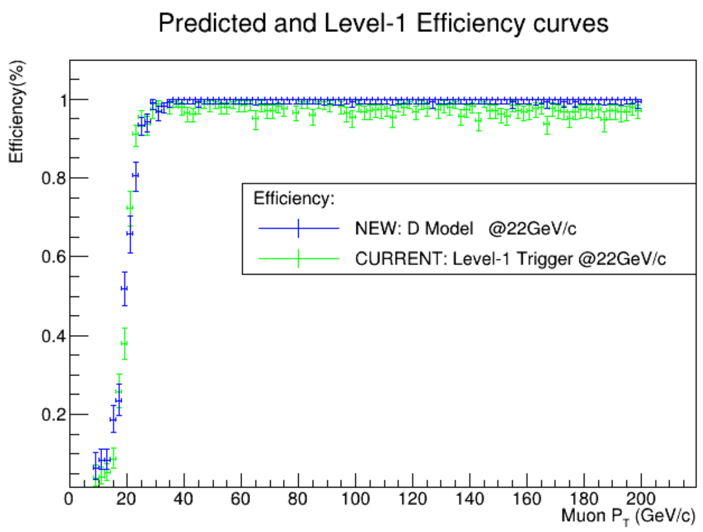


Figure 5.9: Efficiency turn-on given a p_T threshold cut of 22 GeV. The blue dots show the efficiency for the ML based momentum assignment related to the D model, while the green dots show the efficiency for the Level-1 trigger p_T assignment.

In Figure 5.9 a turn-on efficiency curve is shown for the D model. A threshold of 22 GeV/c was selected, as it was the threshold of the lowest p_T-cut unprescaled single muon triggers over Run-2. Such a momentum cut, in fact, has nearly full acceptance for signatures of events at the electroweak scale ($Z \to \mu^+\mu^-$, $W \to \mu^-\bar{\nu_\mu}$) and above.

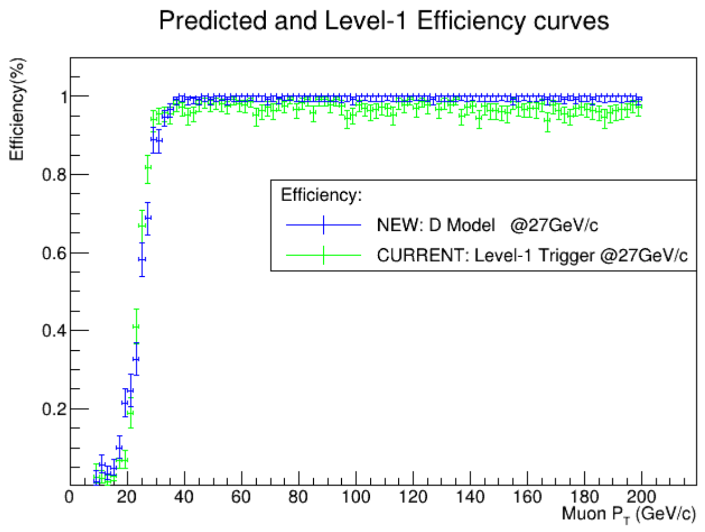


Figure 5.10: Efficiency turn-on given a p_T threshold cut of 27 GeV. The blue dots show the efficiency for the ML based momentum assignment related to the D model, while the green dots show the efficiency for the Level-1 trigger p_T assignment.

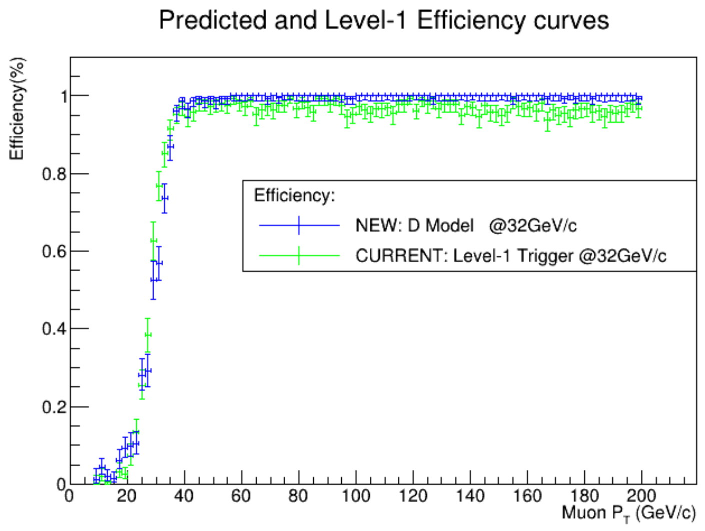


Figure 5.11: Efficiency turn-on given a p_T threshold cut of 32 GeV. The blue dots show the efficiency for the ML based momentum assignment related to the D model, while the green dots show the efficiency for the Level-1 trigger p_T assignment.

Figures 5.10 and 5.11 show the same curves with different cuts, respectively at 27

and 32 GeV/c.
As for the p_T resolution plots, turn-ons have a different calibration for ML and L1T. However, this difference has been accounted for and the resulting plots are calibrated.

From these plots, a very high efficiency in the plateau region of the ML turn-on curve can be observed: the lower efficiency of the Level-1 curve in the high-p_T region is mainly caused by the underestimation of the muon p_T in some events, which is visible as the small peak at -1 in the p_T resolution plots. The ML model, instead, is not affected by this issue and the efficiency at high values of p_T is nearly 100%. On the other hand, Figures 5.9 and 5.10 show a small increase of efficiency in the 10 - 20 GeV/c region which is comparable with a worse resolution in the low p_T range. This is not optimal since it is located below the threshold, thus resulting in a higher fraction of muons with low momentum misidentified as high momentum, and consequently in an increase of overall rate.

5.4 Implementation of a Neural Network on a Field Programmable Gate Array

This section will present the main subject of the work described in this thesis: the implementation on a FPGA of the p_T inference performed by a NN using data coming from local trigger segments (trigger primitives) from the DT chambers (explained in more detail in Chapter 2) at the CMS experiment. The aim was to find an alternative to the operating muon p_T *assignment* algorithm installed in the L1T with a faster and more accurate technique, thus capable of sustaining the increase in luminosity which will characterize the upgrades of the HL-LHC project. Before going into details about the steps required to build an application deployable on an FPGA, a brief description of the software and hardware used will be presented.

5.4.1 Hardware characteristics

The target hardware consisted in a Xilinx ZCU102 Evaluation Board [88] featuring the Zynq Ultrascale+ XCZU9EG-2FFVB1156E Multiprocessor System on a Chip (MPSoC) with a quad-core Arm Cortex-A53, dual-core Cortex-R5F real-time processors, and a Mali-400 MP2 graphics processing unit based on Xilinx's 16nm FinFET+ programmable logic fabric. The ZCU102, shown in Figure 5.12, supports all major peripherals and interfaces, enabling development for a wide range of applications. The ZCU102 MPSoC houses 600k logic cells (flip-flops and LUTs, see Section 4.1) and 2520 DSP slices (Section 4.1.3) working with an internal memory of 32.1 Mb. This board was chosen because of the faster development the use of a microprocessor ensures, as only the firmware concerning the NN had to be designed, leaving the rest (e.g. I/O interface) manageable via software.

Modern MPSoCs [89], such as the Xilinx Ultrascale+ device in exam in this thesis, combine heterogeneous computing with the high performance of FPGAs. MPSoCs have been traditionally used in networking, automotive, communications, signal processing, and multimedia among other applications.

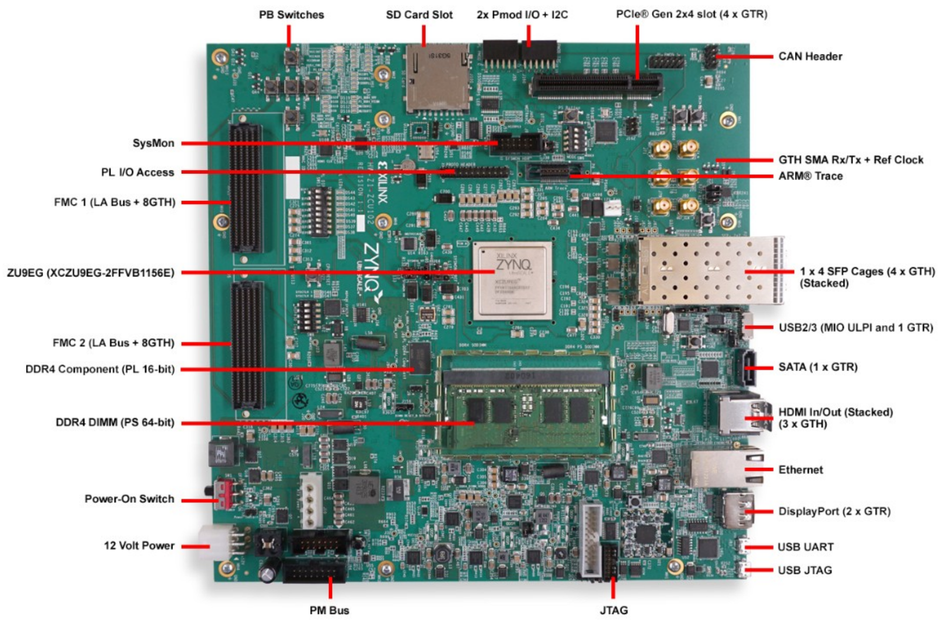


Figure 5.12: The Xilinx ZCU102 Evaluation Board.

In all MPSoC devices from Zynq Ultrascale family, the system is subdivided in two main parts (Figure 5.13):

Processing System (PS) which contains all the microprocessors, in particular the quad core ARM Cortex-A53 that implements the ARM v8-A 64-bit instruction set with frequency up to 1.5 GHz, and whose cores are accessed using the Xilinx Software Development Kit (how it is done is described in more details in Section 5.4.4);

Programmable Logic (PL) which contains the re-configurable logic (FPGA in Section 4).

Vivado and its Software Development Kit (SDK) can establish a JTAG [90] connection to the MPSoC through the Digilent SMT2.5 USB-to-JTAG module with off-module microUSB connector. This link was used for the programming of the FPGA and first steps of debugging.

On the other hand, to communicate and so retrieve the output of the computation carried out in the PL and handled by the PS, the Universal Asynchronous Receiver-Transmitter (UART) [91] interface was used. Indeed, the CP2108 quad USB-UART on the ZCU102 board provides four UART connections through a single micro-B USB connector. UART is basically a protocol that involve transmitting and receiving serial data. It was used to establish a communication interface on the microprocessor linked to the standard I/O of a console.

Inside the device, multiple interconnection options between PL and PS are available, making high-speed data transfer possible, suitable for the most demanding applications.

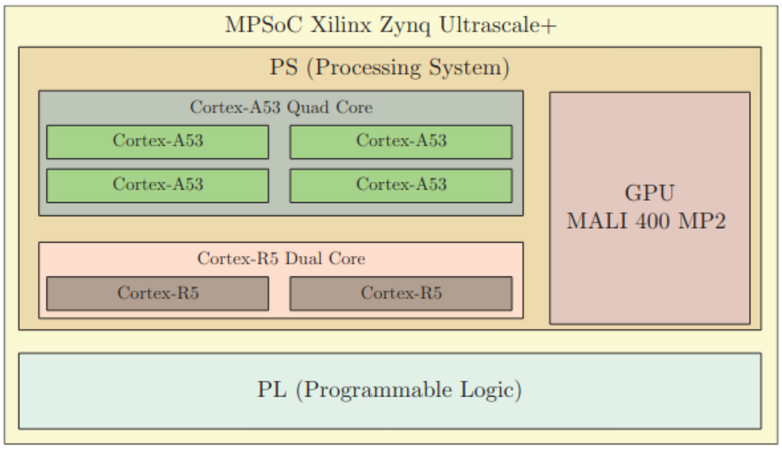


Figure 5.13: Functional block diagram of the Xilinx Zynq Ultrascale+ MPSoC.

5.4.2 From an HLS project to an IP core

Once the target hardware had been defined, the thing to do was transform both trained models shown in Section 5.3 into two distinct HLS projects using the functions from the hls4ml library listed in Section 4.2.2. The projects were then opened with Vivado HLS to modify the input/output interface from the one created automatically by hls4ml:

```
#pragma HLS INTERFACE ap_vld port=q_dense_input,layer_out
```

to AXI4-Lite interfaces:

```
#pragma HLS INTERFACE s_axilite port=q_dense_input bundle=input
#pragma HLS INTERFACE s_axilite port=layer_out bundle=output
#pragma HLS INTERFACE s_axilite port=return bundle=input
```

This was done in order to allow the PS to communicate easily with the PL through the MPSoC IP in the Vivado IP integrator (see Section 5.4.3). Then, the NN I/O consisted in reading and writing registers in the PS on-board memory, handled by the software.

The Advanced eXtensible Interface (AXI4) [92] is a family of buses defined as part of the fourth generation of the ARM Advanced Microcontroller Bus Architecture (AMBA) standard. AXI4-Lite is a subset of AXI which has a simpler interface than the full AXI4 but lacks burst access capability, i.e. for every "word" of 32 bits exchanged between PS and PL, a new handshake procedure must be performed. This obviously makes the I/O operation slower than using a single burst of data. However, in order to focus on the NNs implementation without producing firmware dedicated to handle the I/O, this approach was chosen, with the intention of producing a faster and more efficient I/O interface for the models' IP cores in the future. An entire write cycle for a single entry, dealing with all the 27 features making up the test set, took approximately 500 ns, as shown in Figure 5.14.

The next step was to synthesize the two HLS projects. After synthesizing the project, performance and utilization estimates can be analyzed in the *post-synthesis report*, which contains information on the following performance metrics:

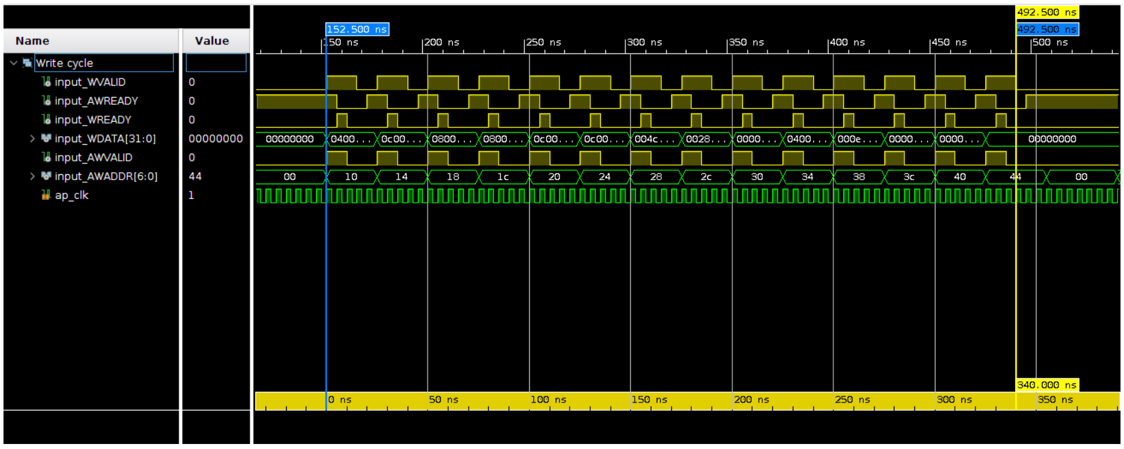


Figure 5.14: Simulation of a write cycle of the 27 features in input for a single entry through a AXI4-Lite interface between the Processing System to the Programmable Logic.

Area Amount of hardware resources required to implement the design based on the resources available in the FPGA, including look-up tables (LUT), registers, block RAMs, and DSPs;

Latency Number of clock cycles required for the function to compute all output values;

Initiation interval (II): Number of clock cycles before the function can accept new input data;

Loop iteration latency : Number of clock cycles it takes to complete one iteration of the loop;

Loop initiation interval : Number of clock cycles before the next iteration of the loop starts to process data. For instance, a maximal pipelined design correspond to II=1;

Loop latency : Number of cycles to execute all iterations of the loop.

These reports for both S and D models are shown in Figure 5.16a and 5.16b respectively. In general, when implementing NNs into FPGAs, the main problem which can be faced is the lack of DSP slices. The models chosen for analysis have been proven to be suitable for implementation in the target hardware available for this work. From this point onward, only the D model will be presented, due to the almost equal resource consumption, but better performance on the test set shown by the D model. Before going further, a simulation of the model behaviour on the FPGA was carried out, yealding the promising results depicted in Figure 5.15, which shows a comparable behaviour with the NN inference on software (more in detail analysis will be described for the implemented NN in Section 5.5).

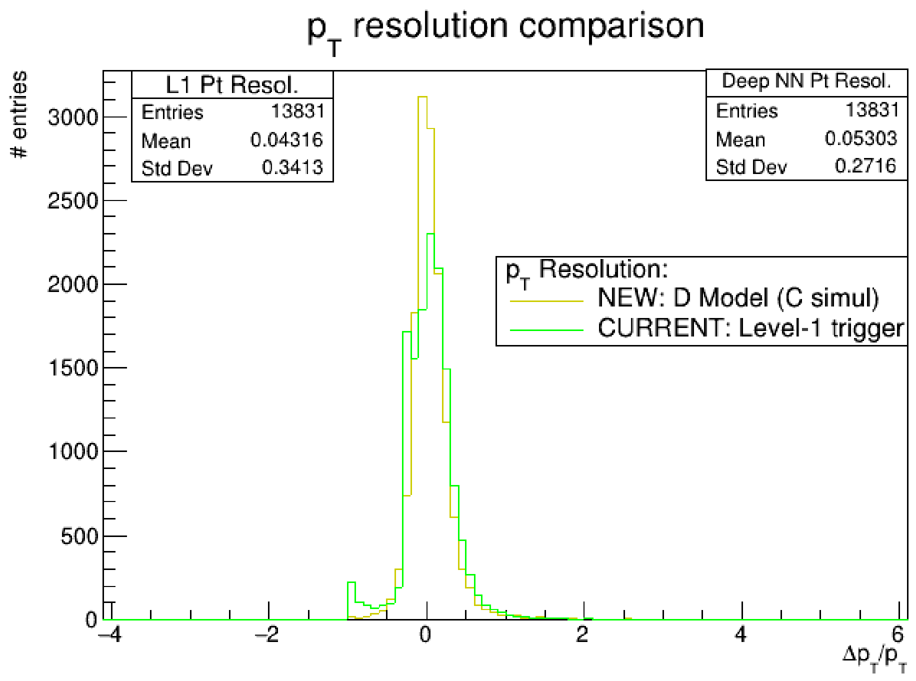
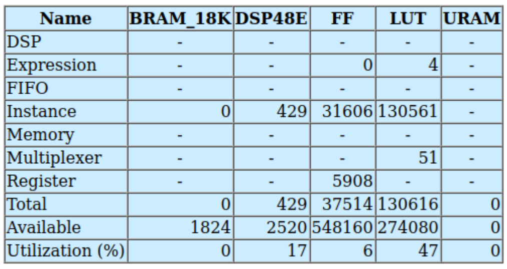
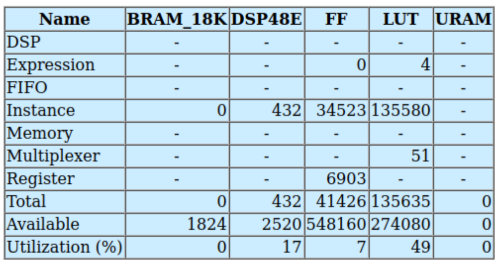


Figure 5.15: Transverse momentum resolution histograms computed for the deeper ML-based simulated on the FPGA (dark yellow) and L1T based momentum assignment, computed for muons generated in the 3 - 200 GeV p_T range.



Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	4	-
FIFO	-	-	-	-	-
Instance	0	429	31606	130561	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	51	-
Register	-	-	5908	-	-
Total	0	429	37514	130616	0
Available	1824	2520	548160	274080	0
Utilization (%)	0	17	6	47	0

(a) *S* model.



Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	4	-
FIFO	-	-	-	-	-
Instance	0	432	34523	135580	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	51	-
Register	-	-	6903	-	-
Total	0	432	41426	135635	0
Available	1824	2520	548160	274080	0
Utilization (%)	0	17	7	49	0

(b) *D* model.

Figure 5.16: Tables containing the resource footprints of both models in terms of BRAM, DSPs, FFs, LUTs and URAM available and used.

As described more in detail in the following section, in order to implement the chosen NN model in a complete design, its project was then exported via the Vivado IP Packager. It enables to create a reusable Intellectual Property module that can be added to the Vivado IP Catalog, which is accessible from the Vivado IP Integrator.

5.4.3 Implementing a Block Design

Intellectual property (IP) cores [74] provide an easy mechanism for incorporating complex logic in the design (i.e. implemented in fabric logic), from high-speed gigahertz transceivers to digital signal processors as well as soft microprocessors to an embedded ARM SoC.

Vivado IDE provides the IP integrator (IPI) with graphic connectivity canvas to select peripheral IPs, configure the hardware settings, and stitch together the IP blocks to create the digital system. This is done in workspaces called Block Designs (BDs).

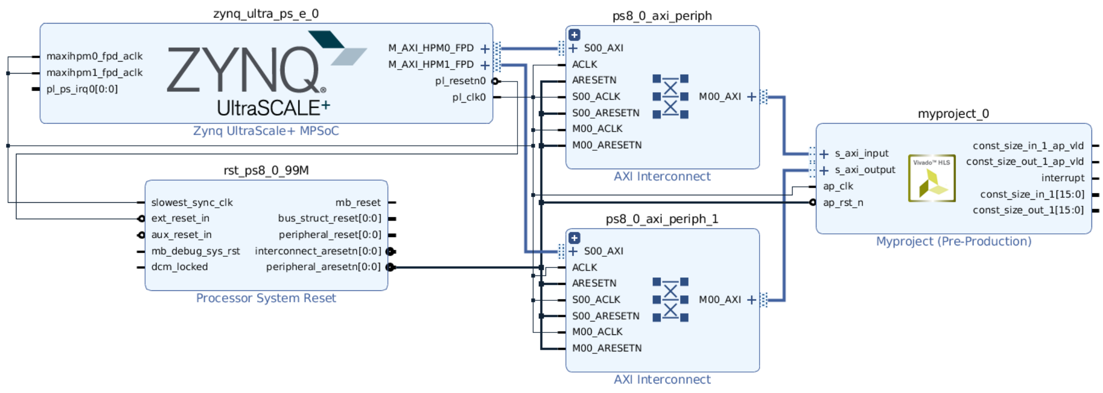


Figure 5.17: Block Design showing the IP core containing the NN connected to the PS, through IPs handling the I/O interface of the implemented NNs.

In the case under study, the model was represented inside a Block Design by the IP core produced by Vivado HLS. On the other hand, the PS was represented by the *Zynq Ultrascale+ MPSoC* IP. In this way, through the addition of IPs handling the AXI4-Lite interfaces, it was possible to connect the application with the ARM processor, which will be used to communicate with the PL. The final diagram representing the entire design is shown in Figure 5.17.

Once the design was ready, an HDL wrapper was created in order to synthesize and implement the project. Finally, the bitstream could be generated, which would be copied onto the FPGA in order to program it (for a description on how FPGA are programmed, see Section 4.2).

5.4.4 Inference on the FPGA using Vivado SDK

At this stage the *D* model have been successfully translated in bitstreams. The computation on the FPGA have been performed through Vivado SDK, calling functions in a C++ code, compiled on a host PC that drive the execution into the PL. These C++ functions were defined in a library generated by the Export functionality of Vivado, which contained, among other tools, methods to: set and get the input of the NN, get its output, control and check the execution in the FPGA.

To make the floating point data in the test set with the same format as the NN inputs, the *ap_fixed.h* library from the *include* directory of Vivado HLS was imported. By defining the input as objects of the `ap_fixed<16,6>` class, the input was cast to fixed point numbers of 16 bits. Because the function for setting the input data requested 14 words of 32 bits, instead of 27 numbers 16 bits long, the inputs were concatenated two by two and put together in a `struct` to be written in the register's address read by the NN as input data.

Here is a snippet of commented code showing how to perform a single inference on the FPGA:

```c
//Outside the main function
#include "xmyproject.h" // This includes the project library generated by Vivado
static XMyproject z; //Declaration of the instance handler passed by reference to all other functions

//Inside the main function

XMyproject_Q_dense_10_input_v datatest; //Input data. The type is defined in the project library as a struct containing 14 32-bits "words"
u32 res_hw; //Result

if(XMyproject_Initialize(&z,NN_DEVICE_ID)!= XST_SUCCESS)return XST_FAILURE; //Initializing the target and handling in case of failure

XMyproject_DisableAutoRestart(&z); //Option to ensure a single read/write cycle

if (XMyproject_IsReady(&z))
     xil_printf("\nHLS peripheral is ready. Starting...\n");
else {
     xil_printf("\n!!! HLS peripheral is not ready! Exiting...\n");
     return XST_FAILURE;
}

XMyproject_Set_q_dense_10_input_V(&z,datatest); //Setting the input
XMyproject_Start(&z); //Starting the computation

do {
     res_hw= XMyproject_Get_layer13_out_0_V(&z); //Polling to retrieve the result
} while (!XMyproject_IsReady(&z));

res_hw= XMyproject_Get_layer13_out_0_V(&z);
```

The `XMyproject` functions were generated by Vivado HLS when the IP core was exported, and they handle the NN I/O, as well as the execution of the inference.

The results were expressed as 17 bits fixed point numbers with 2 bits for sign and integer representation.

This structure was enclosed in a loop, allowing the inference for all 14284 entries in the test dataset. The output was then retrieved via the UART interface, connected to a serial port on the host PC, using a Python script which also converted the integer representation back into floating point numbers for further analysis.

5.5 Results: comparisons between Hardware, Software and Level-1 Trigger

A main advantage of switching to a FPGA assigning the p_T is the reduction of time needed for single predictions. This was evaluated by counting the number of clocks pulses between the input of a pattern and the production of the related output. The model took approximately 74 clock cycles (corresponding to ~ 0.368 μs) for each candidate on the FPGA, in contrast with the times obtained with a consumer CPU of 37.29 ms for a single prediction, and 0.184 s for each entry in a dataset, obtained using QKeras' `predict()` function. It is, however, comparable with the ~ 9 BXs (~ 0.225 μs) expected to be needed by the KBMTF (see Section 2.5) to reconstruct a track.

The data coming from the FPGA was then analyzed, producing the same type of plots presented in Section 5.3.4. However, this time they will show how the p_T assignment changes from the algorithm included in the L1T used today, to the NN running on a consumer grade CPU as a halfway step, and finally to the inference carried out by the FPGA.

5.5.1 Resolution histograms

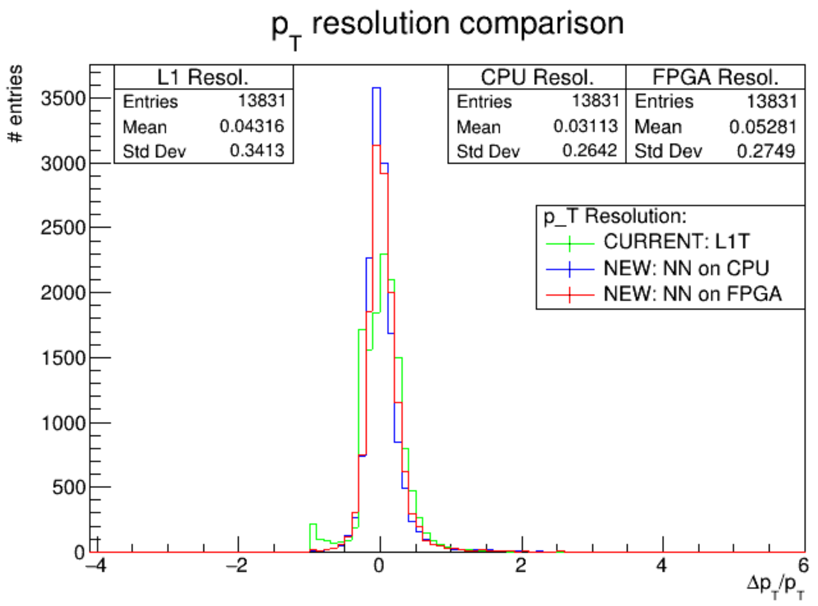


Figure 5.18: Transverse momentum resolution histograms computed for the deeper ML-based model running on a consumer CPU (blue), running on an FPGA (red) and L1T based momentum assignment (green), computed for muons generated in the 3 - 200 GeV p_T range.

In Figure 5.18 the resolution histogram for the D model is shown. In this overall picture, it is clear that the model infer momenta with a resolution which is narrower when the computation is carried out on a CPU. When the assignment is performed on an FPGA, slight worse results are produced with respect to a traditional CPU, with a small bias towards higher values of resolution. However, the FPGA distribution still has a smaller standard deviation with respect to the one associated to the assignment performed by the L1T.

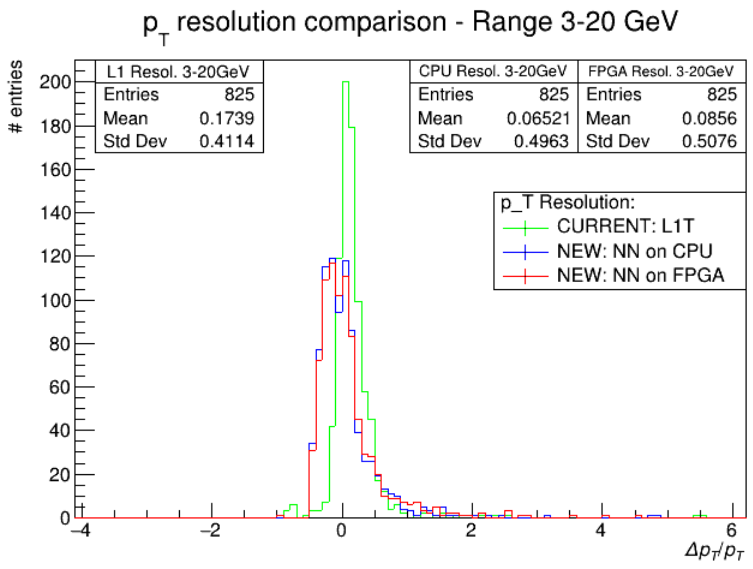


Figure 5.19: Transverse momentum resolution histograms computed for the deeper ML-based model running on a consumer CPU (blue), running on an FPGA (red) and L1T based momentum assignment (green), computed for muons generated in the 3 - 20 GeV p_T range.

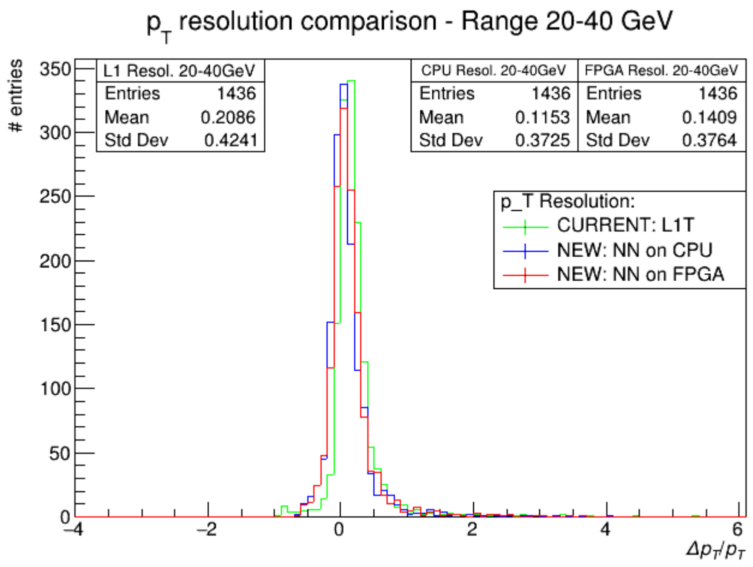


Figure 5.20: Transverse momentum resolution histograms computed for the deeper ML-based model running on a consumer CPU (blue), running on an FPGA (red) and L1T based momentum assignment (green), computed for muons generated in the 20 - 40 GeV p_T range.

Delving into portions of the p_T spectrum, Figure 5.19 displays a struggle by the NN in performing good p_T assignment when compared to the L1T, in the range 3

- 20 GeV/c. However, by looking at Figure 5.20 (20 - 40 GeV/c) it is clear how the NN starts catching up in accuracy, once the p_T starts getting larger. This explains the overall better precision of the ML-based algorithm in p_T assignment.

The passage from software inference to FPGA has caused a small decrease in the advantages achieved by using a ML-based algorithm to assign p_T to muons, as demonstrated by the slightly wider resolutions' distributions. This could be the effect of the loss in precision the input features undergo, due to the conversion to fixed-point representation needed to perform computations efficiently in an FPGA.

Nevertheless, the hardware approach still appears compatible or, in case of higher momenta, even better than the L1T based momentum assignment.

5.5.2 Efficiency turn-ons

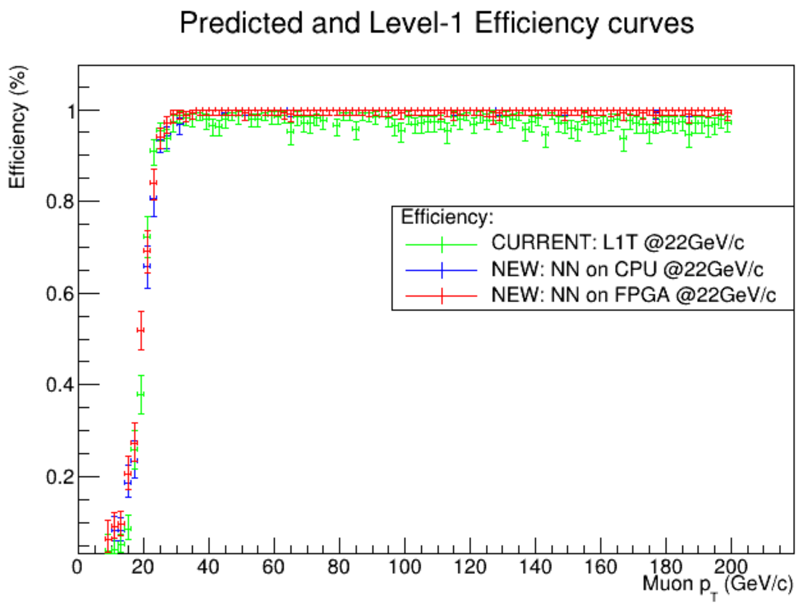


Figure 5.21: Efficiency turn-on given a p_T threshold cut of 22 GeV/c. The blue dots show the efficiency for the ML based momentum assignment related to the D model running on a consumer CPU, the red dots show the results for the ML model implemented on the FPGA, while the green dots show the efficiency for the Level-1 trigger p_T assignment.

As shown in the previous section, the accuracy of the model under study in the range 3 - 20 GeV/c results worse than the L1T. This, as explained in 5.3.4, can cause problems when cuts in momentum are applied, especially when the threshold is in the low part of the p_T spectrum. Figure 5.21 depicts the turn-on curves for the D model when a threshold cut at 22 GeV/c is applied. A small difference between software and hardware based computation was achieved. The same behaviour can be observed with higher thresholds, as portrayed in Figure 5.10 for a 27 GeV/c threshold cut and Figure 5.11 for a 32 GeV/c threshold cut. In all of these turn-on curves, the almost 100% efficiency in the high p_T territory is also shown to be conserved in the implementation of the NN on the FPGA.

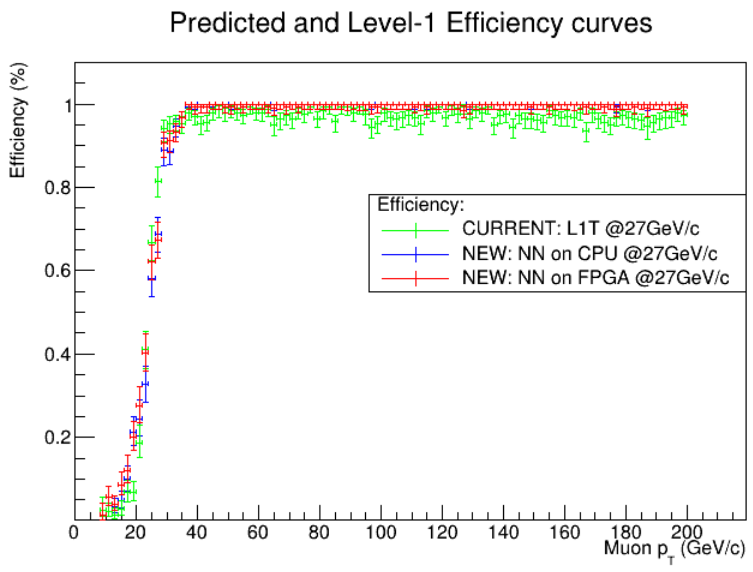


Figure 5.22: Efficiency turn-on given a p_T threshold cut of 27 GeV/c. The blue dots show the efficiency for the ML based momentum assignment related to the D model running on a consumer CPU, the red dots show the results for the ML model implemented on the FPGA, while the green dots show the efficiency for the Level-1 trigger p_T assignment.

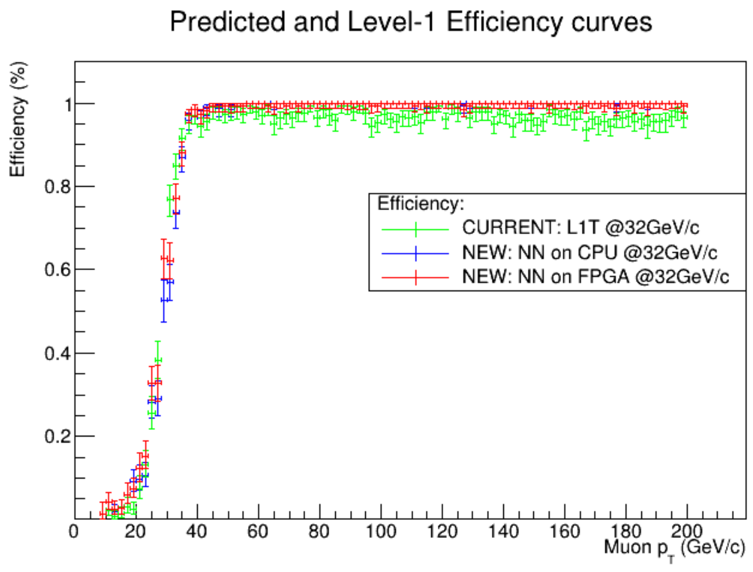


Figure 5.23: Efficiency turn-on given a p_T threshold cut of 32 GeV/c. The blue dots show the efficiency for the ML based momentum assignment related to the D model running on a consumer CPU, the red dots show the results for the ML model implemented on the FPGA, while the green dots show the efficiency for the Level-1 trigger p_T assignment.

5.6 Summary and next Steps

The aim for this thesis was, starting from the work described in [79], to produce an alternative method for assigning p_T to muons crossing the barrel region of the CMS muon chambers, in preparation for the environment of the High-Luminosity phase of the LHC (HL-LHC), when the instantaneous luminosity of the Large Hadron Collider at CERN is expected to increase up to $7.5 \cdot 10^{34} cm^{-2} s^{-1}$.

The first thing to do was to prepare the data available for analysis. This task, carried out following the procedure described in [79], produced a final *csv* file containing about 300000 entries for training and validation of the ML models with 27 features each, describing the position and direction of Monte Carlo generated muons with a range in p_T from 3 to 200 GeV/c.

Using this dataset, two NN were built: both had an input layer with 27 features, although the first model (S model) contained 3 hidden layers with 80, 50 and 35 neural units respectively; while the second model (D model) had 5 hidden layers made up of 60, 50, 30, 40 and 15 nodes. All layers, in both models, used the *Rectified Linear Unit* as activation function. In order to create models suited for hardware implementation, two optimizing techniques were performed: *weight pruning* which reduced the number of multiplication needed when the NNs are used for inference, by setting, during the training procedure, equal to 0 the weights with the lowest values, up to 75% of them, basically cutting the less "important" connections. The second way of optimizing the NNs was to build them using the $QKeras$ Python library. This made all the parameters inside the models *quantized*, i.e. represented as normalized fixed-point numbers with defined bitwidth. This strategy was motivated by the better performance of FPGAs in dealing with fixed-point numbers and their smaller size with respect to the 64-bit floating numbers used by all computing carried out on PCs, including ML frameworks like TensorFlow.

Both models were tested in software, yielding the results shown in Figure 5.7 and 5.6, proving a better overall p_T resolution compared to the current L1T. However, in both cases, the accuracy in the low range of momentum (0-20 GeV/c) saw a decrease in quality, as depicted also in the turn-on curve in Figure 5.9, where a surplus in efficiency below threshold is noticeable. Being the main objective of this thesis the completion of an implementation workflow leading to NN inference on FPGA, the D model was chosen for running on hardware, even though the momentum resolution does not reach the accuracy of the current L1T at low p_T, hoping to build more fine-tuned NNs in the future. One alternative to explore, in order to obtain a better performing model, would be to use, as target of inference, the ratio between the charge q of the particle and its p_T. Indeed, the transverse momentum is inversely proportional to the curvature of muon's trajectory (see Figure 2.12 to understand the link between the curvature and the variables in the dataset) inside the magnetic field, and so, by using its inverse, misassigned low p_T would gain more weight in the training process, making small variations in p_T, big variations in track curvature. Moreover, various parameters concerning quantization could be explored: a different bitwidth for the parameters of each layer, new activation functions (which are still in development at the time of writing) and an energy consumption optimization [65].

Having decided which model to implement, the next step was to produce a

HLS project using the hls4ml Python library. This tool allows to create, from a model saved on disk, a High-Level Synthesis project, i.e. a "behavioral description" contained in a C++ program, which can be synhtesized into an hardware circuit description and packed into an IP core, to be used in the Vivado IP Integrator to produce the bitstream needed to program the FPGA. The NN IP core was connected to the board MPSoC via the AXI4-Lite interface, which allowed an easy implementation, without the need of creating a dedicated I/O firmware. This will be overcome to perform I/O operation "on-the-fly", allowing to insert the FPGA implemented NN in a real trigger scenario.

In Figure 5.16b the resources needed for implementing the D model, as obtained from the post-synthesis report of the HLS project, are shown. From there the low resource cost achieved using the optimization techniques can be appreciated. Even though these values are surely improvable, this is another example of why the work done in this thesis can be considered a good "proof of concept": by using a small set of optimization routines, a NN was made implementable in a FPGA, without an exaggerating amount of hardware resources needed.

After the creation of the bitstream, it was time to program the FPGA and run the inference computation. This was controlled from software, via the Vivado SDK tool, and produced overall comparable resolutions with respect to the L1T system (Figure 5.18), with slightly better results at higher p_T values (Figure 5.20), but worse assignments in the 3 - 20 GeV/c range (Figure 5.19). The steepness of the turn-on curves indicates, in general, a reasonable agreement between the traditional L1T system and the ML-based new solution. Anyhow, in the plateau region of the ML turn-on curve an efficiency very close to 100% is observed, significantly better than in the present L1T. On the other hand, compared to the present L1T, the proposed algorithm shows a higher efficiency in the low p_T region (e.g. 10 - 15 GeV, for a 22 GeV/c threhold cut) of the turn-on (Figure 5.21). This feature, which would definitely imply a high rate for the newly proposed solution, must be tackled in the next development stages.

The time needed for inference was also computed by counting the number of clocks pulses between the input of a pattern and the production of the related output. The D model took approximately ~ 0.368 μs for each candidate on the FPGA, in contrast with the times obtained with a consumer CPU of 37.29 ms for a single prediction, and 0.184 s for each entry in a dataset. It is, however, comparable with the ~ 9 BXs (~ 0.225 μs) expected to be needed by the KBMTF (see Section 2.5) to reconstruct a track.

Conclusions

In this thesis, a new strategy for data processing based on Machine Learning (ML) was analyzed, in preparation for the high instantaneous luminosity environment of the High-Luminosity phase of the LHC (HL-LHC).

By implementing such ML-based models onto Field Programmable Gate Arrays (FPGAs), an alternative to the Level-1 Trigger track reconstruction algorithms developed for the Phase-2 Upgrade of the Compact Muon Solenoid (CMS) was developed and tested. FPGAs promise smaller latency with a relatively small loss in accuracy with respect to traditional inference algorithm running on software.

It was possible to build two Neural Network (NN) models able to predict the transverse momentum p_T of simulated muons crossing the Barrel region of the CMS muon chambers, and compare the results with the p_T assigned by the current CMS Level 1 Barrel Muon Track Finder (BMTF) trigger system.

The training dataset contained about 300000 muons uniform in p_T within a range from 3 to 200 GeV/c. The two NN architectures built for this analysis differs in the number of layers: one more shallow with 3 hidden layers, called S model, and the other deeper with 5 hidden layers, called D model.

In order to create models suited for hardware implementation, two optimizing techniques were performed: *weight pruning*, which reduced the number of multiplication needed when the NNs are used for inference, by setting, during the training procedure, equal to 0 the weights with the lowest values, basically deleting the terms with negligible contributions. The second way of optimizing the NNs was to make all the parameters inside the models *quantized*, i.e. represented as normalized fixed-point numbers with defined bitwidth. For this reason, the NNs were built using the *QKeras* Python package. This strategy was motivated by the better performance of FPGAs in dealing with fixed-point numbers and their smaller size with respect to the 64-bit floating numbers used by all computations carried out on CPUs, including ML frameworks like TensorFlow.

The two NNs were translated into two distinct *HLS* project, using the *hls4ml* Python package, which then were synthesized into two *Intellectual Property* (IP) cores. The two models outputs were compared with the ones obtained from software, showing a satisfactory agreement. After that, the D model was chosen to complete the workflow towards the implementation on hardware, due to better results at the same resource cost with respect to the S model. The D model's IP core was then inserted into a *Block Design* for the Vivado IP integrator and connected to the MPSoC IP core of the Xilinx ZCU102 Evaluation Board via Axi4-Lite interfaces. Following synthesis and implementation (i.e. place and route of the design) steps, the *bitstream* was produced in order to program the FPGA.

The inference for the chosen model produced overall comparable resolutions with respect to the L1T system, with slightly better results at higher p_T values, but with worse assignments in the 3 - 20 GeV/c range. These results are reflected in the turn-on curves, which indicate a reasonable performance for the ML-based new solution. In the plateau region of the ML turn-on curve an efficiency very close to 100% is observed, better than in the present L1T. Thus, investigating the behaviour of the ML p_T assignment for very high-energy muons where a reduction in efficiency due to large p_T misestimation is observed for the existing L1T, could be an interesting continuation of this work. On the other hand, compared to the present L1T, the proposed algorithm shows a higher efficiency in the low p_T region (e.g. 10 - 15 GeV/c, for a 22 GeV/c cut) of the turn-on. This feature, which would definitely imply a high rate for the newly proposed solution, must be tackled in the next development stages.

Regarding the time needed for the p_T assignment, the FPGA showed an inference time of the order of $\sim$ 70 clock cycles, corresponding to $\sim$ 14 BXs, which is comparable with the latency of the current CMS trigger system, demonstrating the promises of faster inference times related to the use of FPGAs.

This work represents a first stepping stone towards more refined and fine-tuned NNs, implemented in a more optimized way onto FPGAs. There is space for improvements in every phase of the workflow, from finding more suitable bitwidths for the weights, to a more evolved I/O interface used to send and retrieve data from the NN, in order to process data in a pipelined way and to reduce even more the time needed for computation.

The work presented in this thesis has been accepted as an oral presentation at the forthcoming "International Symposium on Grids & Clouds 2021" (ISGC 2021) in Taipei (Taiwan), 22-26 March 2021, in the conference track on "Converging High Performance infrastructures: Supercomputers, clouds, accelerators".

Bibliography

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

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

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

[4] Public CMS Luminosity Information. URL: https://twiki.cern.ch/twiki/bin/view/CMSPublic/LumiPublicResults.

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

[6] The CMS Experiment. URL: https://home.cern/science/experiments/cms.

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

[8] The ALICE Experiment. URL: http://home.web.cern.ch/about/experiments/alice.

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

[10] The ATLAS Experiment. URL: http://home.web.cern.ch/about/experiments/atlas.

[11] The LHCb Collaboration. "The LHCb Detector at the LHC". In: *Journal of Instrumentation et al.* 3.08 (Aug. 2008), S08005. DOI: 10.1088/1748-0221/3/08/s08005.

[12] The LHCb Experiment. URL: http://home.web.cern.ch/about/experiments/lhcb.

[13] The LHCf Collaboration. "The LHCf detector at the CERN Large Hadron Collider". In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08006. DOI: 10.1088/1748-0221/3/08/s08006.

[14] The LHCf Experiment. URL: https://home.cern/science/experiments/lhcf.

[15] The TOTEM Collaboration. In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08007. DOI: 10.1088/1748-0221/3/08/s08007.

[16] The TOTEM Experiment. URL: https://home.cern/science/experiments/totem.

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

[18] The HL-LHC Project. URL: https://hilumilhc.web.cern.ch/content/hl-lhc-project.

[19] P. Campana, M. Klute, and P.S. Wells. "Physics Goals and Experimental Challenges of the Proton–Proton High-Luminosity Operation of the LHC". In: *Annual Review of Nuclear and Particle Science* 66.1 (2016), pp. 273–295. DOI: 10.1146/annurev-nucl-102115-044812.

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

[21] The CMS Collaboration. "Performance of the CMS muon detector and muon reconstruction with proton-proton collisions at $\sqrt{s} = 13$ TeV". In: *Journal of Instrumentation* 13.CMS-MUO-16-001 (Apr. 2018), P06015. DOI: 10.1088/1748-0221/13/06/P06015.

[22] Jay Hauser. "Cathode strip chambers for the CMS endcap muon system". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 384.1 (1996), pp. 207–210. ISSN: 0168-9002. DOI: 10.1016/S0168-9002(96)00905-9.

[23] 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.

[24] M. Abbrescia et al. "Beam test results on double-gap resistive plate chambers proposed for CMS experiment". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 414.2 (1998), pp. 135–148. ISSN: 0168-9002. DOI: 10.1016/S0168-9002(98)00571-3.

[25] The CMS Collaboration. "The performance of the CMS muon detector in proton-proton collisions at $\sqrt{s}=$ 7 TeV at the LHC". In: *Journal of Instrumentation* 8.11 (Nov. 2013), P11002–P11002. DOI: 10.1088/1748-0221/8/11/p11002.

[26] A Tapper and Darin Acosta. *CMS Technical Design Report for the Level-1 Trigger Upgrade*. Tech. rep. CERN-LHCC-2013-011. June 2013. URL: https://cds.cern.ch/record/1556311.

[27] The CMS Collaboration. "Performance of the CMS Level-1 trigger in proton-proton collisions at $\sqrt{s} = 13$ TeV". In: *Journal of Instrumentation* 15.CMS-TRG-17-001-003. 10 (June 2020), P10017. DOI: 10.1088/1748-0221/15/10/P10017.

[28] Sergio Cittolin, Attila Rácz, and Paris Sphicas. *CMS The TriDAS Project: Technical Design Report, Volume 2: Data Acquisition and High-Level Trigger. CMS trigger and data-acquisition project.* Technical Design Report CMS. Geneva: CERN, 2002. URL: http://cds.cern.ch/record/578006.

[29] Pierluigi Paolucci. "The CMS Muon System". In: *Astroparticle, Particle and Space Physics, Detectors and Medical Physics Applications*, pp. 605–615. DOI: 10.1142/9789812773678_0096.

[30] Lorenzo Castellani et al. *Performance in Magnetic Field of the Bunch and Track Identifier Prototype for the Muon Barrel Trigger: Results of the 2000 Test Beam.* Tech. rep. CMS-NOTE-2001-051. Geneva: CERN, Nov. 2001. URL: http://cds.cern.ch/record/687381.

[31] Luigi Guiducci. "Design and Test of the Off-Detector Electronics for the CMS Barrel Muon Trigger". PhD thesis. Università degli Studi di Bologna, 2006.

[32] Roberto Martinelli, A J Ponte-Sancho, and Pierluigi Zotto. *Design of the Track Correlator for the DTBX Trigger.* Tech. rep. CMS-NOTE-1999-007. Geneva: CERN, Feb. 1999. URL: http://cds.cern.ch/record/687083.

[33] The CMS Collaboration. *CMS TriDAS project: Technical Design Report, Volume 1: The Trigger Systems.* Technical design report. CMS. URL: https://cds.cern.ch/record/706847.

[34] Marco Lorusso. "Combined use of drift tubes and resistive plate chamber information in the CMS muon barrel trigger". Bachelor's Thesis. 2017. URL: http://amslaurea.unibo.it/16943/.

[35] The CMS Collaboration. "Performance of the CMS TwinMux Algorithm in late 2016 pp collision runs". In: (Dec. 2016). URL: https://cds.cern.ch/record/2239285.

[36] Janos Ero et al. "The CMS Level-1 Trigger Barrel Track Finder". In: *Journal of Instrumentation* 11.03 (Mar. 2016), pp. C03038–C03038. DOI: 10.1088/1748-0221/11/03/c03038.

[37] The CMS Collaboration. *A MIP Timing Detector for the CMS Phase-2 Upgrade.* Tech. rep. CERN-LHCC-2019-003. Geneva: CERN, Mar. 2019. URL: https://cds.cern.ch/record/2667167.

[38] The CMS Collaboration. *The Phase-2 Upgrade of the CMS Endcap Calorimeter.* Tech. rep. CERN-LHCC-2017-023. Geneva: CERN, Nov. 2017. URL: https://cds.cern.ch/record/2293646.

[39] The CMS Collaboration. *The Phase-2 Upgrade of the CMS Muon Detectors.* Tech. rep. CERN-LHCC-2017-012. This is the final version, approved by the LHCC. Geneva: CERN, Sept. 2017. URL: https://cds.cern.ch/record/2283189.

[40] The CMS Collaboration. *The Phase-2 Upgrade of the CMS Barrel Calorimeters*. Tech. rep. CERN-LHCC-2017-011. This is the final version, approved by the LHCC. Geneva: CERN, Sept. 2017. URL: https://cds.cern.ch/record/2283187.

[41] The CMS Collaboration. *The Phase-2 Upgrade of the CMS Tracker*. Tech. rep. CERN-LHCC-2017-009. Geneva: CERN, June 2017. URL: https://cds.cern.ch/record/2272264.

[42] The CMS Collaboration. *The Phase-2 Upgrade of the CMS Level-1 Trigger*. Tech. rep. CERN-LHCC-2020-0041. Geneva: CERN, Apr. 2020. URL: http://cds.cern.ch/record/2714892.

[43] The CMS Collaboration. "Results of DT Longevity Studies". In: (June 2019). URL: https://cds.cern.ch/record/2682229.

[44] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM Journal of Research and Development* 3.3 (1959), pp. 210–229. DOI: 10.1147/rd.33.0210.

[45] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[46] Machine Learning Crash Course - Google Developers. URL: https://developers.google.com/machine-learning/crash-course.

[47] Overfitting in Machine Learning: What It Is and How to Prevent It. URL: https://elitedatascience.com/overfitting-in-machine-learning.

[48] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks". In: *Machine Learning*. 1995, pp. 273–297.

[49] Xindong Wu et al. "Top 10 algorithms in data mining". In: *Knowledge and Information Systems* 14 (Dec. 2007). DOI: 10.1007/s10115-007-0114-2.

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

[51] What is a perceptron? URL: https://towardsdatascience.com/what-is-a-perceptron-210a50190c3b.

[52] Luca Giommi. "Prototype of machine learning ?as a service? for CMS physics in signal vs background discrimination". PhD thesis. URL: http://amslaurea.unibo.it/15803/.

[53] PyTorch Website. URL: https://pytorch.org/.

[54] TensorFlow Website. URL: https://www.tensorflow.org/.

[55] Theano Documentation. URL: https://theano-pymc.readthedocs.io.

[56] Caffe Website. URL: http://caffe.berkeleyvision.org/.

[57] Google Trends. URL: https://trends.google.com/trends.

[58] Google Brain Team. URL: https://research.google.com/teams/brain.

[59] TensorFlow Documentation. URL: https://www.tensorflow.org.

[60] Cloud Tensor Processing Units (TPUs). URL: https://cloud.google.com/tpu/docs/tpus.

[61] Basic training loops. URL: https://www.tensorflow.org/guide/basic_training_loops.

[62] Keras Documentation. URL: https://keras.io/.

[63] Adam Taylor. "The basics of FPGA mathematics". In: *Xilinx Xcell Journal* 80 (2012).

[64] Eldad Meller et al. "Same, Same But Different: Recovering Neural Network Quantization Error Through Weight Factorization". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. PMLR, June 2019, pp. 4486–4495. URL: http://proceedings.mlr.press/v97/meller19a.html.

[65] Claudionor N. Coelho Jr. et al. "Automatic deep heterogeneous quantization of Deep Neural Networks for ultra low-area, low-latency inference on the edge at particle colliders". In: (June 2020). arXiv: 2006.10159 [physics.ins-det].

[66] QKeras Github Repository. URL: https://github.com/google/qkeras.

[67] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. ISSN: 1476-4687. DOI: 10.1038/s41586-020-2649-2.

[68] André DeHon Scott Hauck. *Reconfigurable computing: the theory and practice of FPGA-based computation*. Systems on Silicon. Morgan Kaufmann, 2007. ISBN: 9780123705228.

[69] Barrie Hayes-Gill John Crowe. *Introduction to Digital Electronics*. Essential Electronics Series. Butterworth-Heinemann, 1998. ISBN: 9780340645703.

[70] *UltraScale Architecture DSP Slice*. Xilinx, Inc. 2008.

[71] *GPU vs FPGA Performance Comparison*. Berten Digital Signal Processing. 2016.

[72] Xilinx Incorporated Website. URL: https://www.xilinx.com/.

[73] Intel FPGAs and Programmable Devices Website. URL: https://www.intel.com/content/www/us/en/products/programmable.

[74] Sanjay Churiwala. *Designing with Xilinx® FPGAs: Using Vivado*. 1st ed. Springer International Publishing, 2017. ISBN: 9783319424385.

[75] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuits*. June 2008.

[76] *Vivado Design Suite User Guide - High-Level Synthesis*. Xilinx Inc. 2020.

[77] J. Duarte et al. "Fast inference of deep neural networks in FPGAs for particle physics". In: *Journal of Instrumentation* 13.07 (July 2018), P07027. DOI: 10.1088/1748-0221/13/07/p07027.

[78] hls4ml Documentation. URL: https://fastmachinelearning.org/hls4ml.

[79] Tommaso Diotalevi. "CMS level-1 trigger muon momentum assignment with machine learning". MA thesis. Alma Mater Studiorum - Università di Bologna. URL: http://amslaurea.unibo.it/16326/.

[80] Marco Lorusso's GitHub project repository. URL: https://github.com/DrWatt/MuonTriggerML.

[81] A. Denner et al. "Standard model Higgs-boson branching ratios with uncertainties". In: *The European Physical Journal C* 71.9 (Sept. 2011). ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-011-1753-8.

[82] R Brun, F Rademakers, and S Panacek. "ROOT, an object oriented data analysis framework". In: (2000). URL: http://cds.cern.ch/record/491486.

[83] GEANT4 website. URL: https://geant4.web.cern.ch/support.

[84] TensorFlow Model Optimization Toolkit — Pruning API. URL: https://blog.tensorflow.org/2019/05/tf-model-optimization-toolkit-pruning-API.html.

[85] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. arXiv: 1412.6980 [cs.LG].

[86] The HDF Group. URL: https://portal.hdfgroup.org/display/support.

[87] Brian R. Bartoldson et al. "The Generalization-Stability Tradeoff In Neural Network Pruning". In: (2020). arXiv: 1906.03728 [cs.LG].

[88] Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. URL: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[89] Jose A. Belloch et al. "Evaluating the computational performance of the Xilinx Ultrascale+ EG Heterogeneous MPSoC". In: *The Journal of Supercomputing* 77 (2021), pp. 2124–2137. DOI: 10.1007/s11227-020-03342-7.

[90] The JTAG Connection. URL: https://semiengineering.com/the-jtag-connection.

[91] Eric Peña and Mary Grace Legaspi. "UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter". In: *AnalogDialogue* 54 (Dec. 2020). URL: https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.

[92] Introduction to AXI4-Lite. URL: https://www.realdigital.org/doc/a9fee931f7a172423e1ba73f66ca4081.