POLITECNICO DI TORINO

Master of Science in Nanotechnologies for ICTs

Master's Degree Thesis

Implementation of Deep Neural Networks for the Level 1 Trigger system of the future High-Granularity Calorimeter (HGCAL)

Thesis advisor:

Prof. Dr. Guido MASERA

Scientific Supervisor:

Dr. Jean-Baptiste SAUVAN

Scientific Co-Supervisors:

Dr. Frédéric MAGNIETTE

Dr. Florian BEAUDETTE

Candidate

Elena FERRO

October 2020

Acknowledgements

ACKNOWLEDGMENTS

I would like to express my gratefulness to all my supervisors who made this project possible. Their kindness and willingness have allowed me to spend the last six months in a stimulating and constructive environment that helped me grow personally and professionally. I will never thank enough Jean-Baptiste for having always been there and for having taught me so much during this experience despite the particular situation.
I would like to express my gratitude to Andrea, who has always been by my side during this journey, giving me the strength and courage to move forward and reach my goals.
I thank my family, my rock, for allowing me to undertake this unforgettable experience.
My mother Anna Laura, my best friend, for her ambition and her ability to overcome her fears, letting me undertake the experience she always dreamed.
My father Antonio, for conveying to me his passions and showing me that from nothing you can reach so much, the important thing is to believe in yourself and commit to achieve the goal.
My brother Luca, for always being my accomplice and friend, for our trips to Turin, Milan, Zurich which made this experience even more special.
I thank Giulia, Maria and Alessandra for always being by my side both in the happy moments and in the more complicated ones, and for sharing this experience with me, even if not physically.
Giulia, whom with her courage and genuineness pushed me towards this adventure, motivating me to always give my best.
Maria, whom with her rationality and determination has always pushed me to face obstacles and fears.
Alessandra, whom since high school has been a role model for my studies, and has accompanied me - in her own way - in this experience, knowing that I can always count on her.
I would also like to thank my nanotech friends (Alessandro, Giovanni, Gabriele, André and Alberto) who have made this difficult journey around the world a little less complicated and enriched it with good memories. Particularly, I would like to thank Alessandro, whom with his enviable positivity and kindness has made the last months of this experience a little lighter.

Summary

The CMS detector (Compact Muon Solenoid) is a general-purpose particle detector recording the output of proton-proton collisions at the CERN Large Hadron Collider (LHC), located in the Geneva area. It has been recording collision data since 2010 and in 2012, it has lead to the discovery of the Higgs boson, together with the ATLAS detector [1, 2]. In 2026 the LHC will enter a new phase, the high-luminosity phase (HL-LHC). For the HL-LHC, the end-cap calorimeters of the CMS detector will be replaced by a new radiation-resistant and highly granular calorimeter (HGCAL) [3]. The high granularity of this detector poses severe challenges on the trigger system of the experiment, a two-stage online system which selects collision events of interest to be recorded. These stages are called Level-1 trigger (L1T) and High Level Trigger (HLT). The first stage is the one of interest in this study, it is based on custom FPGA boards and takes as input data coming from the detector at an event rate of 40 MHz. The purpose of this stage is to decide whether to keep an event or not, since not all the data can be sent out of the detector, and eventually send the retained data to the HLT within $12\mu s$. This last constraint is dictated by a trade-off between (1) the area required by the buffers, that are used to keep the data of the events until the decision is taken and (2) the time within this decision is taken, that must be sufficient to process the L1T data. In fact, the events cannot be kept too long in the buffer because they accumulate very fast (every 25 ns) and the more events we want to keep at the same time, the more buffers area we need. This larger area implies more power consumption. As a consequence, it is necessary to limit the size of the buffers as much as possible, which also implies faster decision of the L1T.
More precisely, the first stage (L1T) is composed by different subsystems, one of which is called Trigger Primitive Generator (TPG). The TPG is in turn divided into two stages [4]:

- **Stage 1**, set of FPGAs that repack, synchronize and calibrate the data outgoing from the detector;
- **Stage 2**, set of FPGAs that process the output of the stage 1 by implementing a 3D cluster algorithm.

The TPG needs to process the data and send them to the next subsystem within $5\mu s$, where about $2\mu s$ are available for each of its stages to process the data, and $1\mu s$ is kept for the transfer of the data among the stages. The purpose of the HGCAL trigger subsystem is to reconstruct clusters of energy deposited in the detector and to classify these reconstructed clusters as belonging to one of the two fundamental classes: Electromagnetic (EM) or Hadronic (HAD). This binary classification task needs to be done with a tight latency constraint of a few microseconds dictated by the event rate. Only simple hand-made clustering algorithms (corresponding to the **Stage 2** of the TPG) combined with classifiers based on cluster shapes, such as Boosted Decision Trees (BDT), were developed so far in order to comply with this constraint. This thesis is focused on a new area of development that is now emerging, which consists in using Deep Neural Network (DNN) models to perform this classification task. Two different methodologies are introduced in this project to improve the traditional method to perform the classification. The first one consists in using the Multilayer Perceptron (MLP) instead of the traditional classifier based on the shapes of the cluster. The second one is based on a Convolutional Neural Network (CNN) to analyze raw data directly coming from TPG's **Stage 1** output by performing a simple projection to generate an image. This operation allows to avoid the need for additional hand-made clustering algorithms. This last idea is particularly promising because the possibility to prevent any pre-processing is expected to speed up the process. The approach of using DNN models is interesting because High Level Synthesis tools for Neural Network (NN) models have been recently developed, facilitating the implementation of deep learning models on FPGAs. This allows the creation of dedicated hardware, reducing the time needed to perform a classification task and hopefully reaching real-time efficiency. Therefore, the goal of the thesis is mainly to develop CNN models for particle classification and to evaluate them in terms of performances, logic usage and latency on a target FPGA. In Chapter 1, the main tools used during all the work are introduced. Particularly, it is described the sub-structure of interest of the CMS detector with the aim of understanding how the data are generated and then, how this data can be used to create the two dataset to be employed. The first one coincides with the characteristics of the clusters that are extracted after the **stage 2** of the TPG; the second one corresponds to images obtained via a projection procedure performed with the energy values of the particles that are detected and processed up to the **stage 1** of the TPG. Once clarified how the dataset are generated, a general description of the Artificial Neural Network (ANN) models is presented. The ANNs are parametric models whose performances strongly depend on the number and types of their hyper-parameters. The optimal hyper-parameters are selected while keeping others fixed, i.e. selecting the architecture, during the training process and thereafter verified during the validation process. Since these two processes are by themselves time requiring, due to the size of the dataset at hand in particle physics, and considering the fact that this whole procedure needs to be repeated for many architectures, the computational burden requires an optimization to reduce the time needed for selecting the ANN to be used. In order to deal with this problem, a tool called Integrated Neural Network Automatic Trainer and Evaluator has been developed by the Machine Learning group at the LLR laboratory. This tool is described and used during all the thesis to perform the whole ensemble of training processes, which have been run on GPUs to speed up the training time. Once the NN model is trained and optimized from a software perspective, it is necessary to convert the code into an Hardware Description Language (HDL). For this purpose, two tools called High Level Synthesis for Machine Learning (hls4ml) and Vivado High Level Synthesis (Vivado HLS) are detailed and employed. The first one allows the conversion of the Python model into a C++ code, while the second one enables the conversion of the C++ code into an HDL code. In Chapter 2, it is shown that the traditional algorithm (**Stage 1** of the TPG) along with the MLP, is an efficient way of performing the classification. In fact, this approach allows to reach an accuracy for the classification of EM and HAD particles of 99.56%, suggesting that the use of the MLP to perform this task can be optimal to substitute the traditional BDT classifier. Furthermore, by comparing the performances achieved with those of the more challenging CNN approach it is verified its effective applicability. In fact,

in this last case where images at the output of the **Stage 1** of the TPG are analyzed and no 3D clustering algorithm or classifier based on the shapes of cluster are used, the accuracy is shown to be 99.88%. It is worth mentioning that this result is obtained without any kind of network's optimization. However, considering that the NNs are parametric models whose performances strongly depend on the number and type of their parameters, the second step performed is the optimization of these hyper-parameters to try to improve the CNN performances. In this context, Chapter 3 is devoted to design and apply a method for the optimization of the hyper-parameters characterizing the CNN. This includes hand-tuning and more advanced techniques such as the Bayesian Optimization (BO) for the optimization of the Python model performance. A significant attention is given to the BO technique and to the process that characterizes it, called Gaussian Process, since it is the technique used to optimize the size of the network. This is done because when the aim is not only to optimize the network to have good performances, but also to implement it on FPGA, it is important to have a network with reasonable dimensions to provide a low resources usage that can match the constraints implied by the limited number of resources available on the target FPGA. The BO allows a general study of the hyper-parameters space reducing significantly the computation time. For this reason, it is useful to know if the performances strongly degrade with the size of the network. Once having discovered that the best CNN model for the classification of EM and HAD particles is characterized by (1) too large images size (corresponding to the ideal size of images on the output of the detector: 21 x 21 x 3), (2) a huge number of filters and neurons to be implemented using Vivado HLS, and that in this specific case, the performances of the network do not change significantly (validation loss: 0.032±0.006) for a different number of filters and neurons, we moved to a structure that is more suitable - in terms of images size and number of filters and neurons - for the implementation on FPGA with the current available tools. This network is characterized by the parameters shown in *Table 1*. Once the structure of the network is defined, Chapter 4 is focused on the

Table 1: Structure of the selected CNN after the optimization of its hyper-parameters.

	Input Layer	Convolutional	Convolutional	Flatten	Dense	Dense
Size	$9 \times 9 \times 3$	5	15		8	1
Kernel		3×3	3×3			
Activation		ReLu	ReLu		ReLu	Sigmoid
Max Pooling		2×2	2×2			
Dropout		0.2	0.2		0.2	
Loss			Binary Cross-Entropy			
Optimizer			Adam			

implementation of this network on the target FPGA (xcku15p-ffve1760-2-e). In particular, both the hls4ml and Vivado HLS are used to convert the Python code into a HDL. Once the network is implemented, it is possible to proceed with the synthesis step using a tool called Vivado that allows to have information about the real number of resources and latency that are necessary to implement the network on FPGA. These results are shown in the first row of *Table 2*. *Table 2* demonstrates that the number of resources required for the implementation is smaller than the total number of resources available for each type. In the framework of the latency only, two different values are shown. This is because in the implementation of the pooling layer using hls4ml, the parallelization of its operation is still not supported. The pooling layer is indeed implemented by serializing the operations and this implies a very small number of resources usage and a huge latency (of the order of 3500 clock cycles) that is not affordable in this classification problem. For this reason, in *Table 2* the parameter *Theoretical Latency* is used. It takes into account the pooling layer as if it was implemented with a parallelization of its operations as it is done for all the other layers. More precisely, we considered its latency equal to about 20 clock cycles, that corresponds to the worst case latency for layers of similar size. By doing this observation, we suggested that it is reasonable to assume that it may be possible to reach a latency of around 445 clock cycles once the parallelization procedure for the pooling layer will be implemented. Performing the synthesis using 5ns and observing that the constraint is met because the circuit requires a clock period of 3.99ns, it is possible to conclude that the latency that can be reached by this circuit is $2.22\mu s$. This is a very promising number considering the constraint imposed by the rate of the events collisions, that requires a latency of the order of few microseconds. During this implementation we discovered that the number of bits that needs to be used to convert the floating point precision (used by Python for the weights, the biases and the predictions) to the fixed point precision (understandable by Vivado HLS) is 13. This means that using this technique, it is almost impossible to reduce significantly the resources usage when implementing the network on FPGA, since it is impossible to reduce the associated number of bits to a number smaller than 13. For this reason, in Chapter 5 a new type of network called Quantized Neural Network is considered. These models allow to perform the quantization of the network before the training by associating a certain number of bits for the weights and the biases of the quantized layer. This approach allows to understand that by quantizing just one layer such as the first dense layer and by using 3 bits, it is possible to reduce the resources usage while keeping the performances and the latency (see *Table 2*).

Table 2: Resource and latency usage when using the structure of the selected CNN and when using the quantized dense layer instead of the first dense layer. The latency is expressed in terms of number of clock cycle.

	BRAM	DSP	FF	LUT	Total Latency	Theoretical Latency
Standard NN	245	772	128617	164407	3959	445
Ternary	191	687	125254	129997	3958	445
Available	1968	1968	1045440	522720		

This result is the starting point that suggests either to explore more this approach and that by quantizing the entire model we could be able to reduce more the resources. However, we observed that the performances of the network, when quantizing the entire model, are kept unchanged (99.76%) only when a number of bits at least equal to 4 is used for the quantization of the convolutional layers. This type of quantization is still not supported for the conversion into an HDL code. Consequently, we were not able to implement it, however it could be a starting point for further studies.

Table of Contents

Acronyms

ANN Artificial Neural Network
AUC Area Under Curve

BO Bayesian Optimization
BRAM Block Random-Access Memory

CMS Compact Muon Solenoid
CNN Convolutional Neural Network

DSP Digital Signal Processing

EM Electromagnetic

FF Flip Flop
FNN Fully connected Neural Network
FPGA Field Programmable Gate Array

GS Grid Search
GP Gaussian Process
GPU Graphics Processing Unit

HAD Hadronic
HDL Hardware Description Language
HGCAL High-Granularity Calorimeter
HLS High Level Synthesis
hls4ml High Level Synthesis for Machine Learning

INNATE Integrated Neural Network Automatic Trainer and Evaluator

LHC Large Hadron Collider
LUT Look Up Table

MLP Multilayer Perceptron

NN Neural Network

ROC Receiver Operating Characteristic
RTL Register Transfer Level

TPG Trigger Primitive Generator

WP Working Point

Chapter 1

Introduction

This chapter provides the main knowledge needed to understand the project. *Section 1.1* describes the most important parts of the Compact Muon Solenoid (CMS) detector's structure. The knowledge of their task and of their functioning is enough to understand how, starting from the particles collision, the data that will be used during all the thesis are produced.
These data are structured in different ways and carry information on the interactions of particles with the detector (called showers). The goal of this project is the real-time classification of these representations of a particle detection into two classes: Electromagnetic (EM) particles and Hadronic (HAD) particles. The classification of the particles, produced during the collisions and characterized by a very wide range of energies, into EM and HAD is of crucial importance in particle physics in order to isolate the events that generate EM particles, which occur more rarely, from those that generate HAD particles and to allow their study. In particular at Large Hadron Collider (LHC), where the CMS detector is positioned, the events are dominated by hadronic jets, which are collimated hadrons produced by the shower of quarks and gluons. EM showers, produced by electrons and photons, need to be separated because of their production in physics processes of interest, which appear at a lower rate. For example some decays of the Higgs boson can produce electrons or photons, and so it is of interest to isolate events that generate these EM particles. *Section 1.2* describes how the data on the output of the detector are used to produce the datasets that will be used as the input in the classification problem. In order to explain how these data can be classified, *Section 1.3* introduces a computational model called Artificial Neural Network (ANN). This ANN must be trained and optimized in order to find the proper parameters of the model that will allow to have a classification with high accuracy and then can be implemented on existing hardware (FPGA) in order to perform a real time classification. *Section 1.4* describes the main tools used during all the study to perform the steps related to the ANN. Starting from INNATE, a framework for training neural networks, I will then talk about hls4ml, a software developed for converting a Keras model into a C++ code. Finally I will move to Vivado HLS that provides the mapping of the C++ code into an hardware description language.

1.1 Compact Muon Solenoid (CMS)

The Large Hadron Collider (LHC) is the largest particle accelerator in the world [5], it is a part of the accelerator complex at CERN. Its main purpose is to try to answer to some of the fundamental physics questions such as: the nature of mass, the dimensionality of space, the unification of the fundamental forces, the particle nature of dark matter, and the fine-tuning of the Standard Model [6]. Four main detectors are operating at the LHC: CMS, ATLAS, ALICE and LHCb. The first two, namely the 'Compact Muon Solenoid' (CMS) [7] and the 'A Toroidal Large hadron collider ApparatuS' (ATLAS) [8], are the two biggest general-purpose detectors. They use different technical solutions and different design systems but they share an analogous capability of performing particle experiments involving a large variability of physical phenomena. In particular, the Laboratoire Leprince-Ringuet (LLR) contributes to the CMS experiment and, in what follows, I will describe the corresponding detector.
The CMS detector presents a cylindrical shape with concentric layers (see *Fig. 1.1*) [9]. It is located in one of the points of the LHC's ring where the protons (or heavy ions) collide generating energy that is spread in all directions in the form of particles. Its purpose is to detect any new physical phenomenon by analyzing 3D pictures corresponding to the output of the proton-proton collisions, appearing with a frequency of 40 MHz.

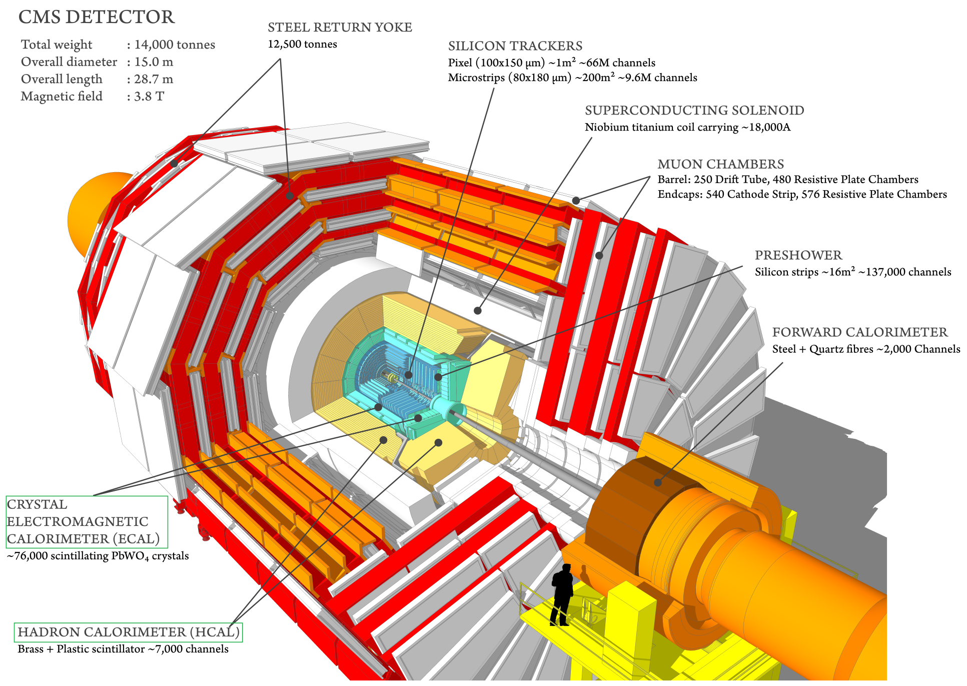

CMS DETECTOR
Total weight : 14,000 tonnes
Overall diameter : 15.0 m
Overall length : 28.7 m
Magnetic field : 3.8 T
STEEL RETURN YOKE
12,500 tonnes
SILICON TRACKERS
Pixel (100x150 μm) ~1m² ~66M channels
Microstrips (80x180 μm) ~200m² ~9.6M channels
SUPERCONDUCTING SOLENOID
Niobium titanium coil carrying ~18,000A
MUON CHAMBERS
Barrel: 250 Drift Tube, 480 Resistive Plate Chambers
Endcaps: 540 Cathode Strip, 576 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

Figure 1.1: Structure of the CMS detector [9]. The descriptions of the Electromagnetic Calorimeter and of the Hadron Calorimeter are squared in green.

The CMS components of interest for this project are the Electromagnetic Calorimeter (ECAL) and the Hadron Calorimeter (HCAL), that are able to collect the information about the energy of the particles produced during the proton-proton collisions. The ECAL is the first encountered by the particles and, by stopping mainly electrons and photons, measures their energies. The HCAL, instead, stops and principally measures the energies of protons, neutrons, pions and kaons (hadrons) that are not stopped by the ECAL. The ECAL is a sampling calorimeter made

of layers of an absorber material. The HCAL is still a sampling calorimeter, but made of layers of either absorber and scintillating materials. The particles mainly interact in the absorber and create secondary particles or more generally a cascade of particles called 'shower', whereas the scintillator emits light when a particle passes through it.
These two calorimeters were designed for an integrated luminosity[1] of 500 fb^{-1}, but with the upgrade program, whose purpose is to substitute the LHC with the High-Luminosity LHC (HL-LHC) after 2025, the integrated luminosity is expected to increase up to 3000 fb^{-1} and the two existing calorimeters are expected to experience performance degradation. For this reason, the CMS collaboration proposed to replace the endcap calorimeters (ECAL and HCAL) with the High Granularity Calorimeter (HGCAL). The future HGCAL, shown in *Fig. 1.2*, is characterized by 28 longitudinal layers in the electromagnetic part (electromagnetic calorimeter, CE-E) and 24 layers in the hadronic part (hadron calorimeter, CE-H).

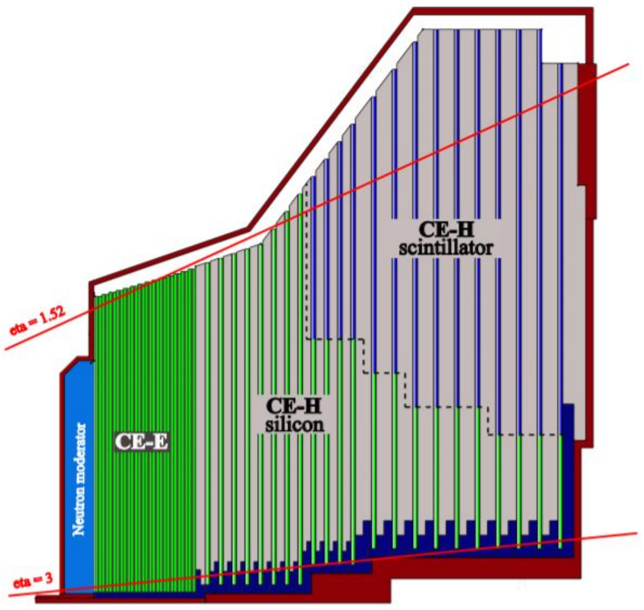


Figure 1.2: Structure of the cross-section of the HGCAL [4].

The CE-E and the CE-H present a very high granularity and are made of silicon cells of area around 1 cm^2 or 0.5 cm^2, except for a small part of the hadron collider that consists of plastic scintillator tiles that are read out by silicon photo-multipliers (SiPM). The charges generated in the SiPM or deposited in the Silicon sensors after the collision are measured by the Front-End (FE) electronics (located on the detector) that, at the end of the measurement, digitizes and transmits data to the Back-End (BE) electronics (located 100m away from the detector). In particular, the Silicon pads sensors and the SiPM are connected to the FE ASIC called HGCROC (for high granularity calorimeter readout chip), which performs the measurement of the charges extracted from the HGCAL at a frequency of 40 MHz. Its entire structure can be seen in *Fig. 1.3*.

[1]Measure of the emitted electromagnetic power. fb^{-1} is called inverse femtobarn and it is used to measure the number of particle collision events in a defined cross-section [10, 11].

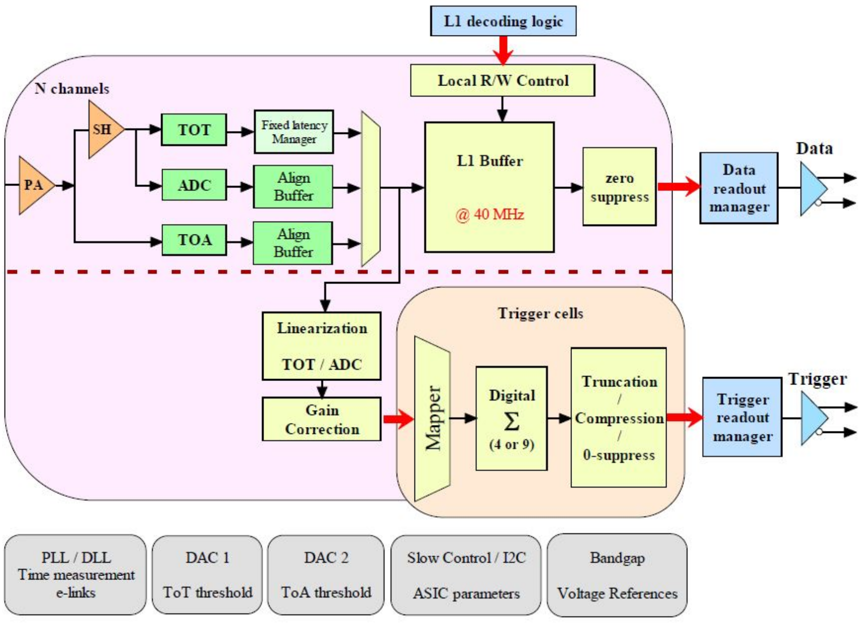


Figure 1.3: Structure of the HGCROC [3].

As it is possible to observe in *Fig. 1.3*, the general idea behind HGCROC working principle is that after the first measurement, where the data from either the Silicon sensors and from the SiPM are digitized by the Analog to Digital Converter (ADC) and the Time Over Threshold (ToT); these data are linearised to the same Least Significant Bit (LSB) in order to use them to generate trigger sums. Depending on the granularity of the sensor a certain number of cells are summed up in order to create the trigger cells (TC) (see *Fig. 1.4*). Henceforth, these data are sent to the concentrator ASIC. This trigger system, called Level 1 Trigger, is crucial because it is impossible to register data from all the cells at the collision rate of 40 MHz (each 25 ns). Besides, one of the purposes of the concentrator ASIC is to select the trigger sums of interest and to send them to the BE electronics.

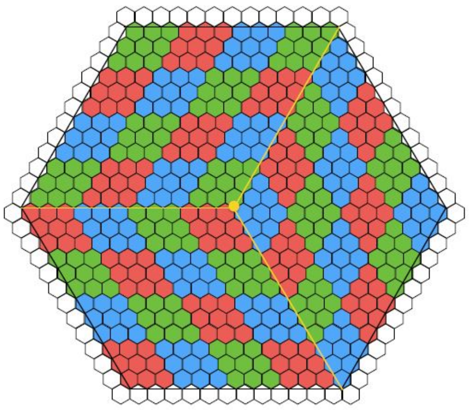


Figure 1.4: Structure of the hexagonal module where the cells are aggregated in order to create the trigger cells. The trigger cells are represented in different colors [3].

The HGCROC is mounted on an hexaboard (hexagonal module), which provides the pads connection to the motherboard (the next board in the system) where the concentrator is located. The goals of the motherboard are to recombine and serialize the data in the concentrator ASIC as well as to distribute the fast control signals, e.g. the clock and everything necessary to configure the FE electronics.
Once the data are digitized and transmitted by the FE electronics, they are sent to the BE electronics. For what concerns the BE-electronics instead, the interesting element in the framework of this study is the Trigger primitive generator (TPG). The TPG receives as input the selected trigger cells and gives as outputs a 3D cluster (a 3D aggregates of trigger cells) expressed in η and Φ (where η is the pseudorapidity while Φ is an angle that describes the position of the particle around the z-axis, the meaning of η is introduced in *Section 1.2*), which are reconstructed starting from the selected trigger cells. These outputs are obtained at the end of two stages [4]:

- **Stage 1** consists of a set of FPGAs whose purpose is to repack, synchronize and calibrate the data outcoming from the detector. It is connected to the second stage in a time multiplexing fashion;
- **Stage 2** consists of a set of FPGAs that processes the output of stage 1 and implements the 3D clustering algorithms. This stage is connected to the Central Level 1 Trigger (L1T).

In particular, the TPG will analyze the data coming from the two endcaps of both CE-E and CE-H separately but with exactly the same hardware. The general structure of TPG is shown in *Fig. 1.5 [4].*

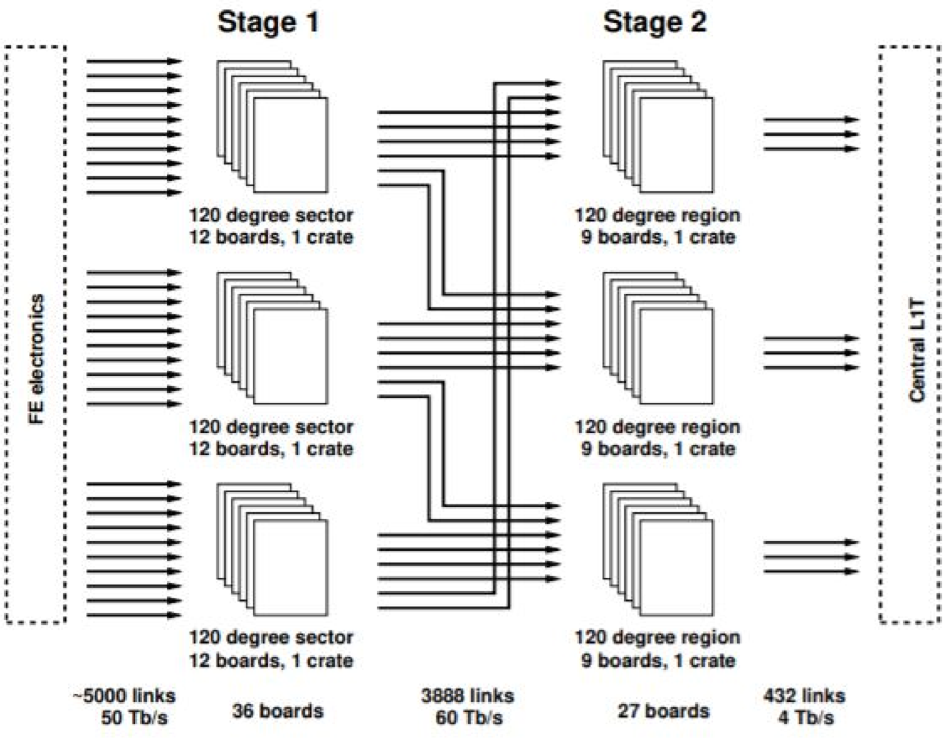


Figure 1.5: Structure of the TPG for one endcap [4].

In the first stage, the data from either the CE-E and the CE-H are processed by the boards. More specifically, the data extracted from the trigger cells are prepared in a format that is suitable for the operation that will be performed in the stage 2. The outputs of the first stage, corresponding to the inputs of the second stage, are the trigger cells data in a different format.
In the second stage the clustering algorithm is divided into two steps, called seeding and clustering. In the process of seeding the cluster, the energies of the trigger cells are first projected and summed into histogram bins in the plane $\frac{r}{z} - \Phi$ ([4]), where z is the beam axis while r is the distance from the beam. Afterwards, by selecting the local maxima above a certain threshold, namely E_{seed}, the seeds are defined. Once the seeding is completed, during the clustering process the trigger cells will be associated to one seed according to the distances among them which must fall within a specific range. It is possible that one trigger cell can be associated to more than one seed and, in this case, the closest one will be selected. At the end of this clustering process on the output of the second stage there will be 3D clusters of trigger cells.
The outputs of the stage 2 of the TPG structure are then given as inputs to the Central Level 1 Trigger and, therefore, they represent the current data that are analyzed and classified at the output of the detector by using a classifier based on the cluster shapes, such as the Boosted Decision Tree. These outputs of the stage 2 correspond to characteristics of the showers and will be used to generate the dataset described in *Section 1.2.1* while the outputs of the stage 1 can be projected on a virtual plane to generate virtual images of the particles coming from the collisions and will be used to generate the dataset described in *Section 1.2.2.* Due to the high rate of collision, i.e. every 25 ns, and the fact that each collision generates many particle showers with a very high variability of energies, on the output of the TPG structure we will have a very huge amount of data that need to be classified in a very short time, of the order of few microseconds, to perform a real time classification. More specifically, the time devoted to the HGCAL TPG for processing the input data, including the classification, and deliver the output data is about $5\mu s$, implying around $2\mu s$ for each stage to perform the associated operations and $1\mu s$ for the transfer of the data between stages. This constraint comes from the fact that the whole system (Level 1 Trigger), of which HGCAL TPG is a constitutive element, can be viewed as a very long and complex pipeline, whose latency ($12\mu s$) is the result of a trade-off between either data acquisition storage capability and processing time. In fact, data acquisition storage is performed by a buffer implemented on the FE-ASIC, which collects the events coming with the aforementioned event rate of 40 MHz and whose area increases as the number of events to be kept at the same time increases. The increase in the area of this buffer implies larger silicon area on the ASIC and larger power consumption and for these reasons it should be limited as much as possible. Conversely, Level 1 Trigger (L1 Trigger) needs to distinguish whether an event is meaningful to be kept and sent towards the subsequent step (High Level Trigger) or not in the shortest amount of time as possible to process all the data associated to the event under analysis. The outcome of the trade-off is assessed to be about $12\mu s$, that is considered sufficient to process the L1 Trigger data and make the buffer in the FE-ASICs manageable.

1.2 Datasets

Once the origin of the data is understood, we can move to the description of the dataset that will be used during this project. Two different types of dataset, containing the features that characterize the particles in two different representations, will be considered. When each particle arrives on the calorimeters, it starts to deposit its energy and for each Trigger Cell (TC) of each layer we know the information about the energy deposited. At this point two alternatives arise:

- Dataset 1: it is possible to take the information from the TCs, which is associated to the energy, and use a simulation code to reconstruct clusters of energy. If we have energy deposited in close-by trigger cells they will be clustered together and summed to form the cluster (aggregates of TC which ideally corresponds to one particle shower). The idea is to reconstruct the particles shower in one object. Then, starting from this cluster and the TC inside it, it is possible to compute the cluster shapes, like the cluster's depth, its first and last layers and many other characteristics obtained by processing the raw data. All these quantities define the dataset.
- Dataset 2: it is also possible to directly get a projection of the information of the energy contained in the TCs on a virtual plane to generate a virtual image.

The Dataset 1 corresponds to the standard way of reconstructing the trigger events, and this is the current output of the HGCAL system (described in *Sec. 1.1*). There are other ways to reconstruct the trigger events, which are now under study. In particular instead of using this algorithm (and so instead of using the stage 2) for the reconstruction, it is possible to directly use the Convolutional Neural Networks (CNN). This is the reason why the second dataset, called Dataset 2, is taken into account. This will allow to avoid the generation of the clusters starting from TCs and to directly analyze the images on the output of the detector. The simplicity of the second dataset compared to the first and the fact that both of them are created from the same raw information are the reasons why the CNN and images are considered in this project.

1.2.1 Dataset 1

In order to understand the problem we are working with, namely the classification of particles into electromagnetic particles (EM) and hadronics particles (HAD), an optimal starting point can be considering a dataset of measurements obtained via the current way the data are processed. This dataset consists of 84952 samples, corresponding to the different particles, and 56 columns, corresponding to the characteristics. The most important parameters and those that are in general used to describe these particles are contained in the first 17 columns, while the last columns contain information of the energy deposited in some of the layers of both the CE-E and the CE-H. The first 17 columns are summarized and explained in the following:

- **Pseudo rapidity (η)**: it is evaluated with respect to the z-axis and it is a transformation of the angle that the particles, generated by the collisions, create with respect to the z-axis, also called θ angle [12]:

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

For example if $\theta = 90^\circ$ then $\eta = 0$, or if $\theta = 0^\circ$ then $\eta = \infty$. More precisely, it tells which is the position of the particle with respect to the beam axis. Its domain is between $-\infty$ and $+\infty$, however, in the dataset at hand, it assumes only values between -3 and -1.5 and between +1.5 and +3 because the HGCAL only covers these regions of the detector. The plus and minus signs are used in order to distinguish the two sides of the detector. As it is possible to observe from *Fig. 1.6a*, the η distribution for EM and HAD particles is very similar;

- **Shower length**: As said in the previous section, the calorimeters are characterized by 52 layers where the energy of the particles can be deposited. With reference to the EM particles, it is expected that they mainly deposit their energy in the Electromagnetic collider (CE-E), i.e. before the first 28 layers, while, considering the HAD particles, they tend to deposit on the Hadron collider (CE-H), i.e. after the 28^{th} layer characterizing the CE-E. The shower length is defined in terms of these layers and it is the length between the first and the last layer where the energy is deposited (the distribution of this parameter can be observed in *Fig. 1.6b*). *Fig. 1.6b* is an example where it is possible to observe that the distribution of the two classes of particles can be overlapped. This means that there are HAD particles that can start to deposit some of their energy already in the CE-E and there could be some electronics noise generated by EM particles in the CE-H. Even if these events are much less likely to happen, they imply a complication in the classification problem because sometimes EM and HAD particles can show similar properties. It is also important to notice that the energy is not always deposited starting from the first layer and on consecutive layers, there can be empty layers between two deposited ones.

- **Core shower length**: it is the maximum number of consecutive layers where there is deposited energy. The Core shower length distribution for EM and HAD particles is shown in *Fig. 1.6c*;

- **First layer**: the first layer where the energy starts to be deposited;

- **Max layer**: it is the layer that presents the highest value of deposited energy. The Max layer distribution for EM and HAD particles is shown in *Fig. 1.6d*;

- **Sigma z-axis (szz)**: it is the spread of the cluster along the z-direction. More precisely, it is the weighted Root Mean Square (RMS) of the width of the cluster along the z-direction:

$$\sigma = \frac{\sum\limits_{TC \in cluster} E(z - <z>)^2}{E_{tot}}$$

  where TC are the trigger cells, E is the energy, z corresponds to the position of the TC along the z-axis and $<z>$ is the mean value of z. The RMS is computed in order to assign more weight to the trigger cells that are characterized by more energy;

- **see tot**: it is the spread of the cluster along η, it is similar to the parameter sigma z-axis;

- **spp tot**: it is the spread of the cluster along Φ, it is similar to the parameter sigma z-axis;

- **srr tot**: it is the spread of the cluster, expressed in centimeter, along a parameter that is the distance from the points where there is a deposited energy and the z-axis. It is related to the parameter η because they always express the distance from the beam axis: η is related to the angle that the particle creates with respect to the z-axis, while srr tot is the distance expressed in centimeter.

- **srr mean**: it is similar to srr tot but the distance for each layer is evaluated independently.The sum is not computed over the trigger cells in the cluster ($TC \in cluster$), but over the trigger cells in the layer ($TC \in layer$) and the operation is repeated for each layer. The mean of all the srr values is computed in the end;

- **Ratio h over e (hoe)**: it is the ratio between the sum of the energies deposited in the CE-H and the sum of the energies deposited in the CE-E;

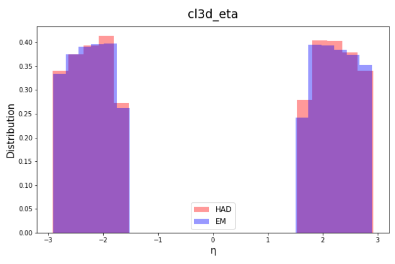
- **Mean z**: it is the barycenter of the cluster:

$$\frac{\sum_{TC} E(z)}{E_{tot}}$$

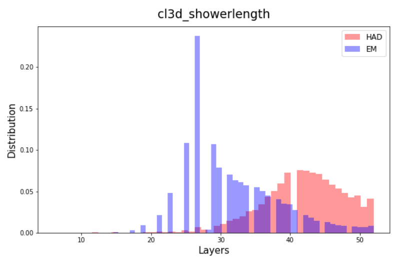
- **Layer 10, Layer 50, Layer 90**: layer where there is 10%, 50%, 90% of the total deposited energy. The layer 10 distribution for EM and HAD particles is shown in *Fig. 1.6e*;

- **Number of Trigger Cells (ntc 67, ntc 90)**: it counts how many trigger cells inside the cluster contain 67% or 90% of the total energy of the cluster. More specifically, it tells the compactness of the cluster. The ntc 67 distribution for EM and HAD particles is shown in *Fig. 1.6f.*

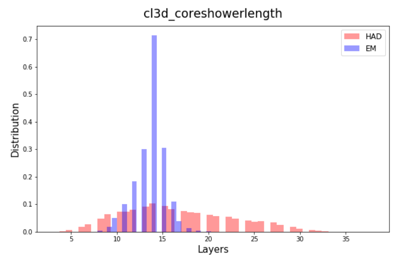
It is important to notice that if on one hand, the EM particles show very stable properties and due to this, during different collision they tend to deposit almost on the same layer; on the other hand, the HAD particles are characterized by very high properties variability and they tend to deposit on different layers every time a different collision is considered. This is the reason why the HAD distribution is more spread than the EM distribution.
This difference between EM and HAD properties is related to the different natures of the interactions between EM showers and HAD showers. Some of the mentioned variables are strongly correlated, and some of them could probably be removed without impacting the performance during this study.



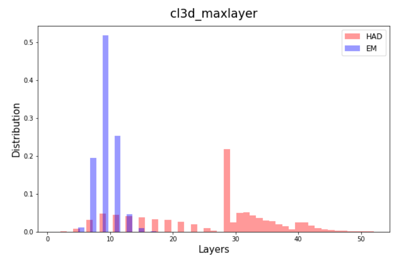
(a) *Pseudorapidity.*



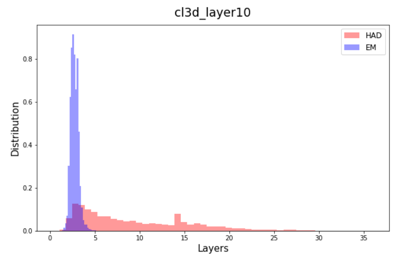
(b) *Shower length.*



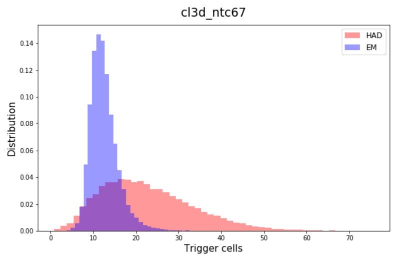
(c) *Core Shower length.*



(d) *Max Layer.*



(e) *Layer 10.*



(f) *Ntc 67.*

Figure 1.6: EM and HAD particles distribution for different parameters.

1.2.2 Dataset 2

This dataset is a little bit different with respect to the previous one. It contains some variables associated to the particles (called genpart exeta, genpart exphi, genpart energy, seedx and seedy) and in addition it also contains virtual images representing projections of EM and HAD particles energies. The genpart parameters are the true information from the generated particles. They are useful to calculate for instance the efficiency of the classifier as a function of these quantities, but they are not used for the training. In fact, this true information can be accessed only in simulated data, in real data we only have access to the reconstructed information.
More specifically, exeta and exphi are the eta (η) and phi (Φ) of the generated particle extrapolated to the front face of the detector. Hence, it is basically the η/Φ position of the generated particle at the first layer of the HGCAL. Genpart energy is the energy of the generated particle. Seedx and seedy are the $\frac{x}{z}$, $\frac{y}{z}$ positions of the image center (the seed) in the detector.
For what concerns the images, each image is obtained by projecting the trigger cells, containing information on the energy deposited by the particles, on a virtual plane positioned on the front

of the detector, at z_0 (see *Fig. 1.7*), by using the following transformation:

$$\begin{cases} \hat{x} & \leftarrow \frac{x}{z} \cdot z_0 \\ \hat{y} & \leftarrow \frac{y}{z} \cdot z_0 \end{cases}$$

where x,y,z are the coordinates of the trigger cells in the CE-E and CE-H, and so they represent the position where the information about the energy of the particle is stored. These new coordinates, $\hat{x}, \hat{y}$, correspond to the position on the virtual plane where the information about the energy contained in the TCs will be positioned. More precisely, this transformation is repeated for each TCs in the different layers and the corresponding energy is added in the virtual plane depending on the new coordinates. At the end, when all the TCs have been considered, more than one TC can be associated to one pixel of the images and the total energy in the pixel will correspond to the sum of the energy of the associated TCs. It means that each pixel value of the virtual images corresponds to a value of the energy and it will be characterized by an area of 1 $cm \times 1$ cm on the virtual plane. It is important to mention that one virtual image is not an image covering the full detector, it is a small region centered around a shower.

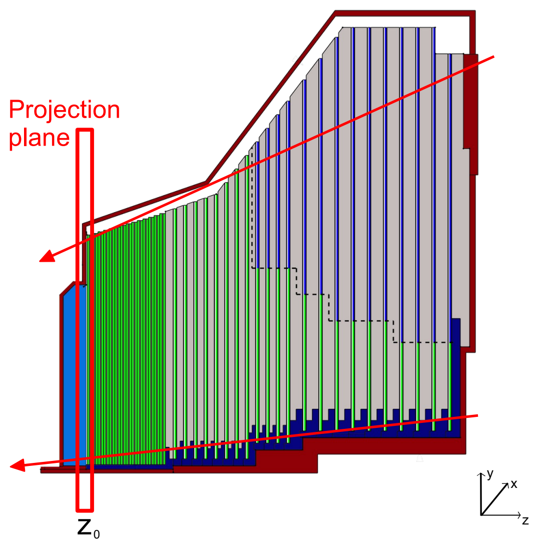
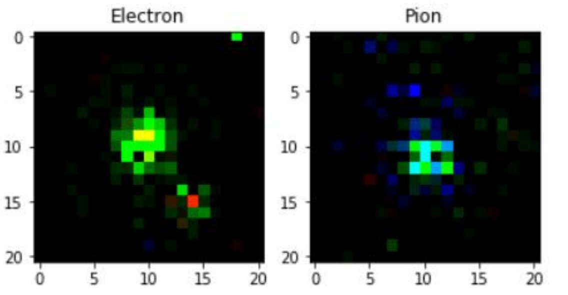


Figure 1.7: Projection of the trigger cells in the CE-E and CE-H on a virtual plane located in the middle of the detector.

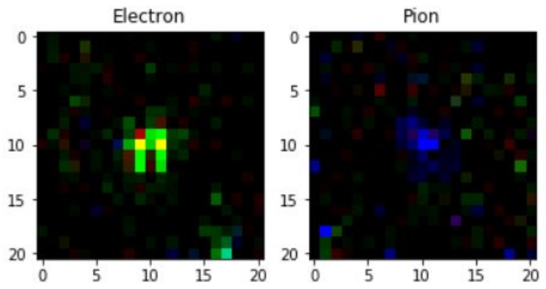
The virtual images can be visualized as RGB images, also if they do not use the RGB encoding. In fact, the virtual images are 3D images where one dimension is associated to the colors. The single color depends on the properties of the particles:

- Red: is associated to particles that come from the CE-E region of the calorimeter (from layer 1 to layer 5);
- Green: is associated to particles that come from the CE-E region of the calorimeter (from layer 6 to layer 28);
- Blue: is associated to particles that come from the CE-H region of the calorimeter (from layer 29).

An example of how these images appear, when interpreted as RGB images, is shown in *Fig. 1.8*.



(a) *Image 1 of EM and HAD particles.*



(b) *Image 2 of EM and HAD particles.*

Figure 1.8: Images of EM and HAD particles as they appear on the output of the detector.

In this case, the dataset is characterized by 64957 color images of dimensions $21 \times 21 \times 3$ for the train and 16240 color images of dimensions $21 \times 21 \times 3$ for the test [2]. In the next sections I will refer to these images as the images of EM and HAD particles even if they correspond to projection of their energies.

1.3 Artificial Neural Networks for a classification problem

In the previous sections we have discussed different methodologies with which the information, associated to the particles and extracted from the detector, can be processed in order to perform the detection of the particles. In particular, we discussed two different representations of these information, i.e. properties related to the energy and images, which have allowed to introduce two datasets. We also mentioned that, in the scientific community, it is of great interest to perform a classification of these particles into EM or HAD particles. This problem can be seen as a binary classification problem, due to the presence of only two classes to whom the particles may belong: EM or HAD.
In recent years ANNs have proved to be widely effective in solving classification problems in a wide range of fields [13] [14]. For example the most common application is the handwritten recognition [15], but their importance is much wider. They are used in cancer classification [16], to classify the heartbeats [17] and also in physics in the context of the LHC [18]. It is also proven that the ANNs are very useful and perform well with problems that are high dimensional, for this reason they can deal very well in a context, such as the LHC, in which the experiments are complex and high dimensional [6].
Neural networks can be used for two different types of learning tasks, that are unsupervised learning and supervised learning. In the former the network does not require a target function for the training of the network, like in the auto-encoders, while in the latter the network does. In this project I will focus on the supervised learning task.
At this point it comes spontaneously to ask what is an ANN and how it works. ANNs are

[2] $21 \times 21 \times 3$ is the ideal size for the images on the output of the detector.

biologically inspired parametric models. They are structured to mimic the memory function and the architecture of the human brain and the dramatic increase of interest in the study of such models has led to the ideation of many types of Neural Network (NN) architectures. The simplest structures of ANNs are characterized by units, also called artificial neurons, that mimic the behaviour of biological neurons, and by connections between them that allow to reproduce the synaptic behaviour of the human brain. These connections are characterized by numbers, also called weights, that describe the contribution of the previous artificial neuron to the next one.
To understand why the ANNs are ideal to deal with classification problems, we can start by considering the simplest possible form of an ANN: the Perceptron. We will then describe the architecture obtainable via the assembly of such a building block and, finally, we will illustrate a more complex architecture whose functioning is different from the previous ones.

1.3.1 Perceptron

The perceptron is characterized by the input layer and one neuron for the output layer (see *Fig. 1.9*).

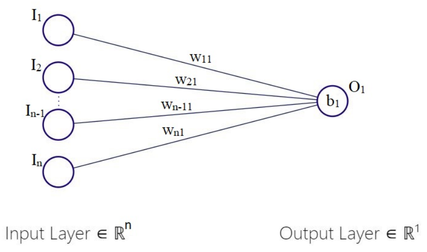


Figure 1.9: Structure of the simplest ANN architecture, the perceptron. It consists of an input layer of a size that depends on the number of inputs and an output layer with only one neuron.

This structure alone already allows to solve simple classification problems, such as some binary classification problems (linearly separable). It is able to learn the weights of the model after a certain number of trials, during a process called training, and at the end is able to learn a linear function that allows to perform the classification. The parametric formula of this structure can be easily written as:

$$O_1 = \sigma(b_1 + \sum_{i=1}^{n} I_i \cdot w_{i1})$$

where b_1 is called bias and it is a parameter of the model that is learnt during the training process together with the weights, n is the number of inputs, w_{i1} is the weight associated to the connection between the i-th input and the output and σ is a function called activation function. This last parameter is a parameter that adds to the output of the perceptron a non-linearity. Trying to give a geometrical interpretation to the quantities introduced so far, let us consider a problem which is linearly separable. This means that the n-dimensional input data belong to two different classes populating sub-spaces separable using an hyper-plane. Such an hyper-plane belongs to a family of planes, uniquely identified via a n-dimensional vector, orthogonal to all of them. This vector is nothing else than the one that it is possible to define using the weights.

The bias, on the other hand, allows to choose the wanted hyper-plane in this family by imposing the output of a perceptron, having a linear activation function, to be equal to zero:

$$E_1 = b_1 + \sum_{i=1}^{n} I_i \cdot w_{i1} = 0. \tag{1.1}$$

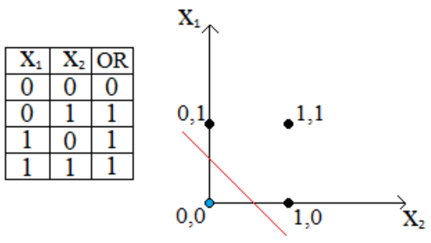
The hyper-plane defined by the weights divides the hyper-space into two sub-spaces: while for any input vector laying on one side of it E_1 will have a negative value, all of the vectors laying on the other side will be characterized by a positive value of E_1. At this point, if the non-linearity is not applied the output of the perceptron is just a linear application that gives negative numbers for inputs belonging to one class and positive numbers for inputs belonging to the other one. In order to transform these positive and negative numbers to 1 and 0 respectively, it is possible to apply a Heavyside-like non-linearity, such as a sigmoid, to the output. This sigmoid will give 0 when the output of the perceptron is sufficiently negative, 1 when the output of the perceptron is sufficiently positive and an intermediate value when the input is ambiguous. In this way the solution of the classification problem is completed and we do not need to transform positive and negative values into 1 or 0.
For example, every logic function can be thought of as a binary classification problem because knowing the inputs the goal is to determine if the result of the logic function belongs to the 'true' class or to the 'false' class. If we want to define the OR function, that is a binary classifier in which, depending on the values of its input the output will be 1 or 0, it is possible to use the perceptron to generate a linear function that is able to determine if the output associated to each combination of the inputs is 1 or 0 depending on which region of the plane the points associated to the inputs are with respect to this line (see *Fig. 1.10*). As mentioned before, this is possible only if the activation function is applied to the output. This can be written as follows:

$$x_1 OR x_2 = \begin{cases} 1 & \text{if } x_1 + x_2 > 0.5 \\ 0 & \text{else} \end{cases} = H(x_1 + x_2 - 0.5) \tag{1.2}$$

where x_1 and x_2 are the inputs of the OR function and H is a non-linear function (Heaviside function) that is 0 when its argument is negative and 1 when the argument is positive. Due to the fact that in this specific case the sum of the inputs is only 2,1 or 0 the Heaviside function can be well approximated by the sigmoid function

$$\frac{1}{1 + e^{x_1 + x_2 - 0.5}} = \frac{1}{1 + e^{w_1 x_1 + w_2 x_2 + b}}$$

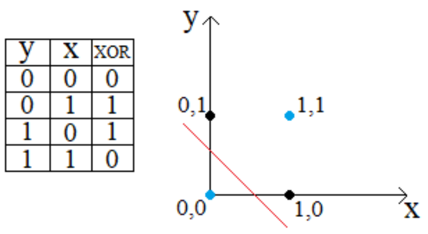
that corresponds exactly to the parametric formula of the perceptron ($\sigma(w_1 x_1 + w_2 x_2 + b)$).



X_1	X_2	OR
0	0	0
0	1	1
1	0	1
1	1	1

Figure 1.10: On the left side of the image, the truth table of the OR function is represented, while on the right side, there is a graphical representation of the OR function. The black points represent the output as 1; the light blue point represents the output as 0. The linear function learnt by the perceptron to perform the classification, is highlighted in red.

As it is possible to observe from *Fig. 1.10*, the red line represents the linear function learnt by the perceptron. This means that all the combinations of the inputs that give a point in the same region of the black points in the space with respect to this line will be classified as belonging to the class 1 and the output of the perceptron will be 1. While the combination of inputs for which the points are in the same region of the light blue point will be classified as belonging to the class 0 and the output of the perceptron will become 0.
However, this structure is not able to solve all the problems. For instance, if we want to define the XOR function, that apparently is very similar to the OR function, the perceptron is no more able to classify the points determined by the inputs [19]. This is due to the fact that in this case one linear function is not enough to perform the classification because in one region of the space with respect to the line there will always be points associated to both class 1 and class 0 (see *Fig. 1.11*).



y	x	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Figure 1.11: On the left side of the image, the truth table of the XOR function is represented, while on the right side, there is a graphical representation of the XOR function. The black points represent the output as 1; the light blue points represent the output as 0. An example of the linear functions that the perceptron can learn to perform the classification, is highlighted in red.

The fact that some functions, such as the XOR, could not be solved using the simplest perceptron is the main reason why for several years the interest in the study of ANNs had dropped. However, it was then demonstrated that every regression problem can be solved using the ANNs and in particular, when the problem becomes too complicated, it is possible to put together more perceptrons, generating the so called Multilayer Perceptron (MLP).

1.3.2 Multilayer Perceptron (MLP)

The Multilayer Perceptron (MLP) can be used, as stated by K. Hornik, et Al. [20] in 1989, as a universal function approximator. In particular, the MLP can approximate the probability function for a particle to belong to one of the two classes: EM or HAD. Each MLP consists of an input layer, an output layer and in between them there can be one or several hidden layers which perform most of the computation. An example of a MLP structure is shown in *Fig. 1.12.*

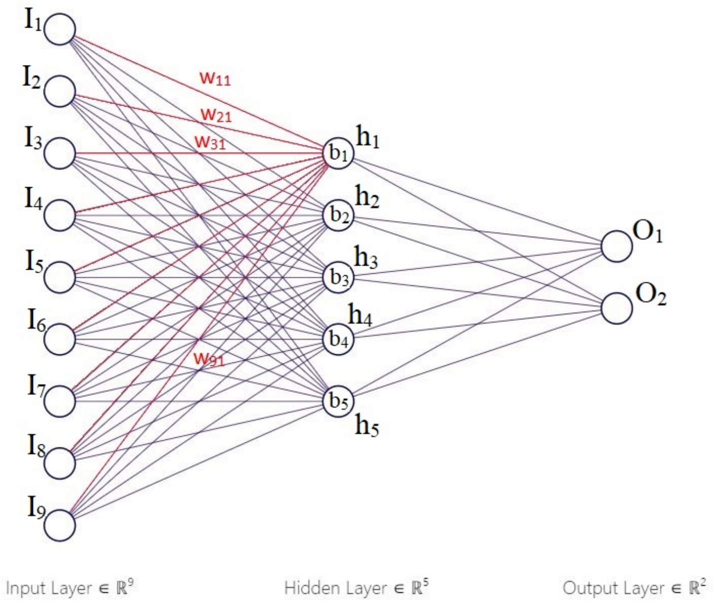


Figure 1.12: General structure of a MLP, with the input layer, the output layer and one hidden layer. In red are highlighted the contribution of the neurons of the first layer to the first neuron in the hidden layer.

This architecture is able to process input data and to classify them by learning the previously mentioned parameters, the weights and the biases, using some already known results, known as targets. In order to understand the learning process that consists in the comparison of the state of the NN before receiving the data and after processing the data, let us consider a binary classification problem and a MLP with only the input layer, the output layer and one hidden layer.
As stated before, each neuron in a given layer is connected to each neuron in the next layer by a connection characterized by a weight. These weights at the beginning of the process are randomly associated to each connection and are some of the parameters that the MLP will learn during the process. The neurons in the hidden layers and in the output layer are characterized not only by the weights that connect them to the neurons in the next layer but also by another numerical value called bias that is added to the output of the specific neurons and will be learnt by the MLP during the process. To be more explicit, if we are interested for example in the evaluation of the input of the first neuron in the hidden layer, following the notation used in *Fig. 1.12*, it is possible to write:

$$h_1 = \sum_{i=1}^{n} I_i \cdot w_{i1} + b_1$$

w_{i1} is the value associated to the connection between the i-th neuron in the input layer and the first neuron in the hidden layer; n is the total number of inputs; I_i is the value of the input of the i-th neuron in the input layer; b_1 is the bias associated to the first neuron in the hidden layer; and h_1 is the output of the first neuron in the hidden layer.
Moreover, as seen for the OR function, in order to take into consideration the non-linearity of the problem, it is necessary to modify a little the output of each neuron, and as a consequence the input of the next neuron, by applying the *activation function* (σ) to the output. It means that the correct way to write, for example, the output of the first neuron in the hidden layer will be:

$$h_1 = \sigma^h(\sum_{i=1}^{n} w_{i1} \cdot I_i + b_1^h)$$

Analogous formulas can be written for the neurons in each layer in the NN. The parametric formula for the MLP, with one hidden layer, becomes:

$$O_k = \sigma^o\left(\sum_{i=1}^{n_h} w_{i,k}^o \cdot \sigma^h\left(\sum_{j=1}^{n} w_{i,j}^h I_i + b_j^h\right) + b_k^o\right) \tag{1.3}$$

n is the number of neurons in the input layer; n_h is the number of neurons in the hidden layer; the apex 'o' indicates parameters associated to the output layer while the apex 'h' indicates parameters associated to the hidden layer; and k is the label identifying a neuron in the output layer. This implies that, given a set of input values, it is possible to use this set of equations and to propagate any input data through the network generating the so called *forward propagation*. Thanks to this operation and depending on the values assumed by the two neurons in the output layer, the output is determined. These values on the output layer correspond to probabilities that allow to discriminate which is the output of the network between the two classes. What could happen is that the MLP, after the forward operation, associates the highest value to the first neuron and so the predicted class by the network will be the one associated to the first neuron in the output layer.
However, at the beginning of the process, this prediction will not be the correct one. In fact, during a process, called *training process*, a certain number of inputs is given to the MLP and for each input, the output of the network is compared with the expected result (the target output) in order to evaluate the error, that is generally assessed by using a function called *loss function*. This loss function is a function of the parameters of the NN, that are learnt during the training process, such as the weights and the biases, and that are used to determine the output of each neuron. The evaluation of this error allows to establish what is the distance between the output of the network and the expected result. This information is vital in the learning process because it is propagated backward through the network. In fact, during this process, the weights and the biases associated to each neuron will be corrected by using this error in order to improve the accuracy of the network in the classification. The process in which the error on the output is propagated back through the network is called *back propagation*.
The forward propagation together with the back propagation are repeated, for different inputs, to minimize the error on the output until it gets very small and the network is able to discriminate in a correct way the different classes, generating the learning process.
Once the training is performed, the parameters such as the weights and the biases are determined and fixed. In order to determine the accuracy of the trained network in the classification task, it is possible to perform a second process called *Validation* or *Test*. In this process a new dataset,

with the same characteristics as the one used as input during the training process, is given as the input of the network. During this process the previously evaluated weights and biases are used to determine the output of the network for each input and each time the output is compared with the target in order to evaluate the accuracy of the NN. This accuracy corresponds to a percentage value that states how many times the network gives the correct output over the total number of sample in the dataset.
In order to understand how this MLP will be used in the next sections let us consider our classification problem. The input of the MLP will be the dataset described in *Section 1.2.1* that means we have an input size of 56, where 56 corresponds to the number of the characteristics of the cluster in input. Each characteristic of the cluster becomes the input of one of the neurons in the input layer, meaning that the size of the input layer will be equal to the number of characteristics or properties of the considered input object, while for the output there are two possibilities. Either there will be an output per each class and the network will have to learn to maximize the correct output while minimizing the other, or there will be only one output containing simultaneously the whole information. In the last case it is possible to consider, for example, that the output corresponds to the probability to belong to the EM class, $P(EM)$, and the probability to belong to the HAD class will be equal to $1-P(EM)$. The MLP will learn how to classify the input clusters into one of these two classes.

After having discussed a simple architecture such as the MLP, also called Fully Connected Neural Network (FNN), it is possible to study a different kind of architecture that is more suitable for studying the classification of images: the Convolutional Neural Network (CNN).

1.3.3 Convolutional Neural Network (CNN)

The Convolutional Neural Network (CNN), as stated by Grace W. Lindsay [21] and by Kunihiko Fukushima [22], is inspired by specific operations that are performed in the brain of cats. In particular, in the primary visual cortex of cats by two main cells called simple cells and complex cells. In this way the neural network is able to perform pattern recognition as the one of the human being [22].
More specifically, at the level of the simple cells, the simplest features of the image, such as regions of light and shadow or edges, are recognized. Then, the complex cells take the modified image and are able to identify more complex features such as the translation in the space and allow the translation-invariance characteristic when observing different images [21]. These functionings were respectively mimicked introducing the convolution operation and the pooling operation, [22], that will be now discussed. For these reasons this kind of network is not as simple as the MLP even if the general idea of how it works is exactly the same.
The CNN involves operations called convolutions that will allow to identify the features in the input image and to position them into a features map. These operations are determined by the presence of a parameter that appears in the so called convolutional layers and that is called *filter*. A filter is a matrix of a certain size, the most common is 3×3, whose purpose is to act on the initial image and to perform operations on it. In particular in these convolution operations each element of the filter is multiplied only with the corresponding element of the image matrix and then all these numbers are summed in order to generate the output of the layer as shown in *Fig. 1.13*. In this picture for example, during the first step, the filter represented by a green square

is acting on the elements of the matrix delimited by the first 3 rows and the first 3 columns, corresponding to the size of the filter, and each element of this sub-matrix is multiplied by the corresponding element in the filter. All these values are finally summed to generate the first element of the new matrix.
In practice if a_{ij} is the element of the original matrix corresponding to the i-th row and the j-th column, and f_{ij} is the element of the filter corresponding to the i-th row and the j-th column, in order to explicit how the first element of the new matrix is evaluated it is possible to write:

$$a_{11} \cdot f_{11} + a_{12} \cdot f_{12} + a_{13} \cdot f_{13} + ... + a_{32} \cdot f_{32} + a_{33} \cdot f_{33}$$

Then the filter is moved by 1 or more strides, on the right (as done in the second step in *Fig. 1.13*) and also downwards (as done in the third step in *Fig. 1.13*) until all the elements in the original image are covered and the new matrix is obtained. This means that in the CNN there are not weights and biases as in the case of MLP and the parameters that will be learnt during the training process are the elements of the filters matrices.

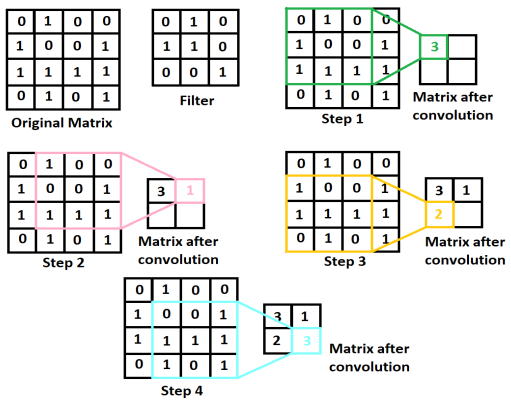


Figure 1.13: Steps of a convolution operation performed on a matrix called Original Matrix using a Filter of size $3 \cdot 3$.

Also in this case the convolutional layer is followed by the application of the non-linearity, meaning that after this layer there will always be an activation function. In addition, compared to the layer of the simplest ANN architectures, after the convolutional layer it is typically defined the so called *pooling* layer, in general characterized by a matrix of size 2×2. There are different types of pooling layers, such as the average pooling and the max pooling, but in this project only the max pooling will be considered. The max pooling operation consists in taking the maximum value of the selected region in the features map, see *Fig. 1.14*.

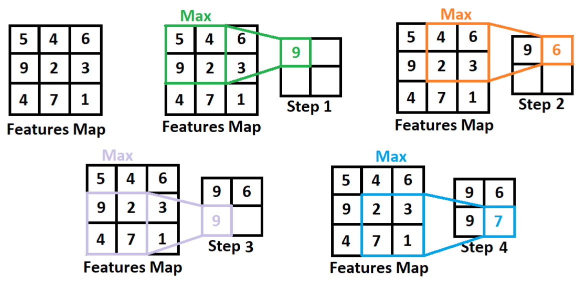


Figure 1.14: Steps of a pooling operation performed on a matrix called Features Map using a pool size of 2×2.

The pooling layers are mainly meant to reduce the scale at which the image is observed in order to search for higher level features. It takes, in the case of 2×2 size, 4 elements of the original matrix and by selecting the maximum one it converts these elements into one element only in the new matrix, compressing the size of the image. The convolution is translational equivariant, [21], since the same filter is applied all over the image, and a pattern in one region of the image will be identified in the same way as in any other region of the image. What the pooling layer also adds is a local translational invariance, in the sense that if a feature is shifted by one pixel it will not make a difference for a 2x2 pooling [23].

1.4 Tools

After having described the structure of the ANNs, the purpose is to describe the main tools and libraries that will be used for the optimization and for the hardware implementation of the networks. In fact, we are not only interested in having a network capable of solving efficiently the classification problem but also to create dedicated hardware encoding the chosen network to classify a certain detection in the quickest possible way. For what concerns the problem of optimizing the network, particularly when considering techniques such as the Bayesian optimization based on the Gaussian process (both will be explained in *Chapter 3*), this will require extremely time demanding simulations. This implies the need for efficient environments to perform such calculations. On the other hand, there is the need for implementing the classifier on hardware. Due to the fact that an FPGA is subjected to restrictions in terms of components, the network cannot be arbitrarily big. Therefore, the complexity of the object to be implemented on hardware has led to the creation of tools for the automatic implementation, so that the user can be freed from the hard task of writing directly a NN model on an Hardware Description Language (HDL) and can focus more on the optimization step, in order to implement the optimal network for a given FPGA. In this section I will first analyze a tool that is used for all the training processes in order to speed them up. Then, once the neural network model is selected, it is necessary to convert the Python model into an HDL in order to implement it on existing hardware. In order to do that two different steps will be considered. The first one is a conversion from the Python model into an High Level Synthesis (HLS) code (such as C++). The second one is a translation from the HLS code into an HDL.

1.4.1 Integrated Neural Network Automatic Trainer and Evaluator

During the training and the optimization of an ANN, using the Keras library, there are some hyper-parameters that require a very long time in order to be optimized. This is related to the size of the hyper-parameters space that in some cases becomes very huge. In order to solve this issue and to reduce the timing required by this kind of optimization, it is possible to perform the training on a GPU cluster. In this project this is done with the help of a tool that has been developed in the course of the last year by the Machine Learning group at the LLR Laboratory. This tool is called Integrated Neural Network Automatic Trainer and Evaluator (INNATE) and the idea behind it is to execute some training of Neural networks on a GPU cluster without the need of being an expert in systems, clusters and GPUs. The only things needed are the knowledge of the theory behind NNs and the definition of the task to be performed since the technicalities are automated in the tool. The schematic functioning of the software is represented in the following picture:

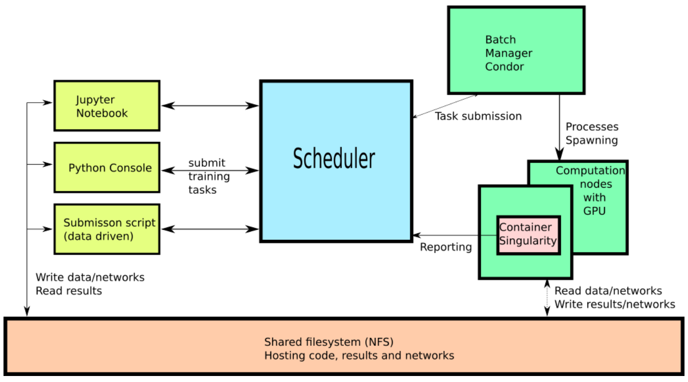


Figure 1.15: INNATE architecture [24].

On the left side of the image are mentioned all the possible user interfaces, namely all of the possible ways in which the source code associated to the training or the analysis to be performed on the NN architecture can be presented to the tool:

- Jupyter Notebook;
- Python Console;
- Python script.

Once the code is submitted to the INNATE pipeline, the scheduler is devoted to handle the code to interface it to the machines dedicated to the running process. The scheduler is on a specific machine and it has access to a GPU cluster, called Accelerated Computing for Physics (*ACP*). Once the scheduler receives the training requests from the user interface, namely the source code that it has to compile, it distributes them on a cluster by using a system called batch manager Condor, [25], that is able to find the computational resources in the cluster and allows to deploy them. The LLR-INNATE machine runs the training into a Singularity container to avoid problems of configuration [26]. In fact, the Singularity container is a container that contains all the libraries or software that allow to make all the computations and allows their reproducibility. While the machine is running, it constantly reports the improvements to the

scheduler that in turns prints the information at the level of the interface used by the user. This allows the user to have some control on the process. When the task is completed the results need to be organized so that they can be properly exploited by the user, some of them are given online using the scheduler interface, others, the biggest ones like the trained Neural Network, are too big to be transmitted in this way. For this reason there is a *Shared Filesystem* (NFS) [3], directly accessible by the user via the user interface, that is used for reading the data and writing the results [27].
This kind of architecture is very useful not only because it allows to train the NNs in parallel on GPUs, speeding up the training, but also because of the possibility to perform the so called *asynchronous running*, where, depending on the number of available GPUs, it is possible to run one NN per GPU, leaving the kernel free to do any other things during this computation. This can be particularly useful when performing training of NNs requiring a computational burden of several hours.
INNATE will be used during all the processes in order to run the simulation for the NN architectures comparison and the optimization. This is fundamental in this study due to the prolonged computation time of these operations that could require multiple days.

1.4.2 HLS4ML: High Level Synthesis for Machine Learning

Once the idea of how the NN is optimized is clear, we can move to the description of a tool that allows to perform the first step that goes from the software implementation of the NN to the hardware implementation of the NN, which is what we want to achieve. This step consists in the conversion of a Python model into a High Level Synthesis (HLS) code that can be easily converted into a Hardware Description Language.
There is a high variability of tools that allow to perform this conversion and they are of fundamental importance because they grant the possibility to reduce the time required for writing a HLS code and as a consequence they save time that can be used for the development.
During this study the High Level Synthesis for Machine Learning (hls4ml) tool will be utilized. The hls4ml is a Neural network translation library for High Level Synthesis and was developed by the collaboration of the Fermi National Accelerator Laboratory, the Massachusetts Institute of Technology, HawkEye360 (Herndon), CERN and the University of Illinois at Chicago. Its goal, as reported in the original paper by Javier Duarte et al. [28], is to convert a traditional Keras, PyTorch or TensorFlow model into a High Level synthesis code. This means that hls4ml is able to convert a Python model into a C++ code. The extended hls4ml workflow is reported in *Fig. 1.16*.
In particular, the conversion from a traditional Keras, PyTorch or TensorFlow model is performed by taking as an input the files that contain the biases, the weights and the architecture of the neural network model. The architecture is given in a *json* format while the weights and the biases in the *hdf5* format. It can also take as input the traditional inputs used to validate the Keras, PyTorch or TensorFlow model that can be used in the next step, when the implementation on FPGA in its final form will be analyzed. In particular they will become the inputs of a file that allows to verify the correctness of the converted HDL code by generating a file containing the

[3]NFS is a standard protocol implemented on Unix machine and it is configured to be available across all cluster machines, scheduler machine and user machine.

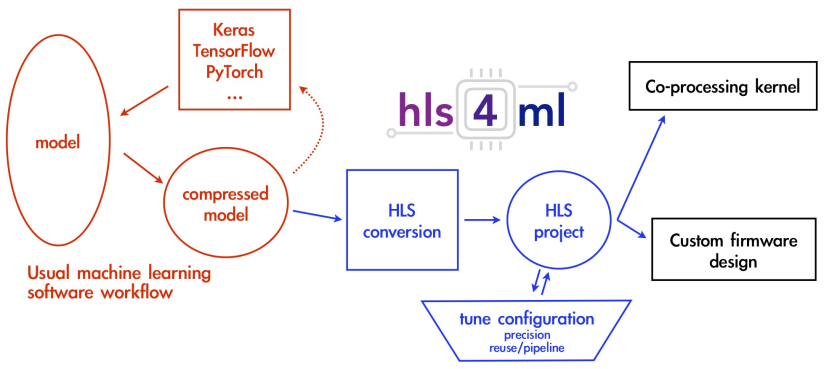


Figure 1.16: hls4ml workflow for the translation of a Keras/PyTorch model into a FPGA implementation, the orange part represents the operations performed before using hls4ml and are the typical machine learning steps, while the blue part represents the hls4ml sub-workflow [28].

output that can be compared with the target. This aspect will be analyzed in *Section 1.4.3*. The output of hls4ml is a High Level Synthesis project that can be used in Vivado HLS that converts it into an Hardware description Language (HDL) such as VHDL or Verilog, as explained in the next section (*Section 1.4.3*).
The general idea of the project is to implement on FPGA a Neural Network that requires a very small time, of the order of μs, to compute the output given the associated input. This tool has been created for optimizing the way an ANN is implemented on FPGA. In particular, these parametric models involve well defined operations: multiplications, additions and a pre-computed activation functions. All of these operations require a very high number of resources on FPGA and for this reason, in addition to the built-in techniques present in hls4ml for minimizing the number of resources, some precautions when characterizing the network can help to further reduce both the hardware needed and the latency[4]. One of these techniques, implemented by hls4ml, allows to reduce the precision of the calculations and is called Quantization [28]. More specifically, with this technique the floating point precision with which inputs, weights and biases are described in a Keras model are converted into a fixed point precision reducing the calculation precision. This procedure is implemented in hls4ml by specifying in the command line for the conversion the type:

$$ap\_fixed < a,b >$$

where b stays for the number of bits used to convert the integer part and a corresponds to the total number of bits. It is important to specify that b is a signed number and for this reason one of its bits is used to represent the sign. Another possible technique is called Parallelization [28], it is used to decide how much the operations are to be parallelized to have faster or slower inference vs. FPGA resources. This can be controlled by using a specific parameter called Reuse Factor (RF), the higher it is the more the same resources are used and the less we parallelize. In order to explain this concept, in *Fig. 1.17* is illustrated a simple example in which are shown the number of resources and the latency needed to perform a simple operation,

[4]The latency is the total time required by the hardware, employed for the implementation of the model, to perform all the operation that are necessary to process an input and to give the associated output.

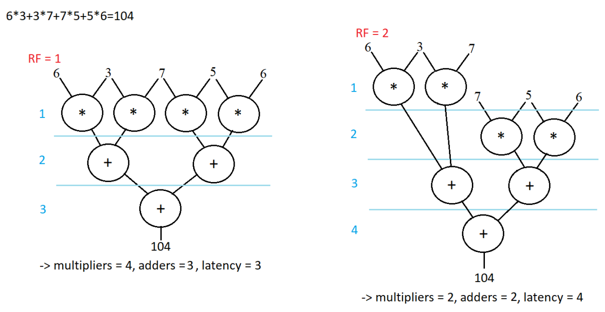

6*3+3*7+7*5+5*6=104
RF = 1
-> multipliers = 4, adders =3 , latency = 3
RF = 2
-> multipliers = 2, adders = 2, latency = 4

Figure 1.17: Graphical representation of the effect of the variation of the RF on the number of resources (multipliers are represented with the symbol '$*$' while the adders with the symbol '$+$') and latency (highlighted in light blue on the left of each graph). On the left a RF equal to 1 is used, while on the right RF is equal to 2.

$6 \cdot 3 + 3 \cdot 7 + 7 \cdot 5 + 5 \cdot 6$, when the RF changes. On the left of the image the RF is equal to 1, and the number of resources required to perform the operation are: 4 multipliers, 3 adders and the total latency is 3. More specifically, when the RF is 1 it means that each resource used for one operation cannot be used in a different time for a different operation. If we increase the RF from 1 to 2, it is possible to use the available resources at two different times and as a consequence to reduce the number of resources. For instance, as it is done on the right side of the same image, the two multipliers used at a time 1 to perform two operations ($6 \cdot 3$ and $3 \cdot 7$) can be used at time 2 to perform other two operations ($7 \cdot 5$ and $5 \cdot 6$) reducing the total number of multipliers to 2. However, in this case the latency required to perform all the multiplications is at least 2 and it is higher than the case in which RF is equal to 1 where all the operations are performed in parallel (latency equal to 1). This means that when the RF is modified it is important to take into consideration its relationship with the latency and not only with the resources. In fact, if the RF increases, the number of resources usage becomes smaller but the time required for the computation increases.
These two techniques will be very important in this project because we are going through the exploration of machine learning algorithms in low-latency and real-time processing.
Another technique, that can be mentioned, requires some operations that are not directly implemented on hls4ml but need to be performed during the training of the model. In this technique, called Compression [28] of NN, the redundant synapsis and neurons are removed while the performances are kept unchanged. One way of doing this is to train the NN with L1 regularization, to sort the weights and to prune weights falling below a certain percentile and re-train [29] (see *Fig.1.18*).

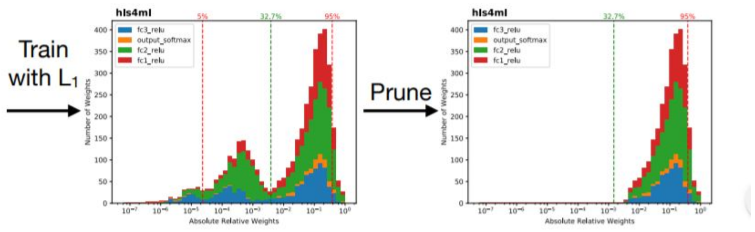


Figure 1.18: Compression process to efficiently use the resources in FPGA [29].

Moreover, it is important to specify that there are different approaches that can be followed when implementing a neural network on FPGA. One could be to find an optimal Neural network architecture in terms of performances and then, in order to reduce the complexity of the network, it is possible to remove the products of the coefficients that are characterized by very small values or to compress the network once the network is already trained and optimized. Another approach could be to consider the reduction of the complexity before the implementation of the network on FPGA and so during the training and the optimization process.
In this project the considered approach is the second one in which the network is optimized both in terms of performances and in terms of its size in order to obtain an optimal network that can be implemented on FPGA with the minimum number of resources. Moreover, once this optimization is done other techniques such as the quantization of the network using the type

$$ap\_fixed < a,b >$$

or the parallelization will be considered. In addition it will be also explored the quantization of the network at the beginning and the training of the network already quantized to reduce the number of bits for the weights and for the biases since the training process. This can be done, using QKeras, thanks to the possibility to fix the number of bits for each layer before the training (this will be explained in details in *Chapter 5*).

1.4.3 Vivado HLS

After having considered how the optimization is done and how the conversion from a Python model into a HLS code is done, the attention can be moved towards the conversion from a HLS code into a HDL.
Vivado High-Level Synthesis (Vivado HLS) is a tool that is able to implement a HLS-code, such as a C/C++ or System C code, into an optimized Register Transfer Level (RTL[5]) microarchitecture. The optimization is in terms of latency, power and throughput and the HLS specifications are directly synthesized into VHDL or Verilog RTL. The main flow of Vivado HLS is [30]:

- Use C, C++ or System C to design the source code and the test bench code;
- Compile the source code (C simulation);
- Translate the C code into an optimized RTL code (synthesis);
- Use the original testbench to perform the RTL validation (co-simulation);
- If everything is correct, export to an IP core that consists in a description of the circuit that can be implemented on FPGA.

More specifically, Vivado HLS takes as input the HLS-code and extracts a datapath and a control unit based code (that are the elements that characterize the Finite State Machine [6]) in

[5]A Register Transfer Level is an high level description of an electronic circuit.

[6]A Finite State Machine is an abstract model that is used in Digital Electronics to describe a system with a finite number of states. It is divided into a *Datapath* and a *Control Unit* that work in parallel. The *Datapath* is the part where the data are evaluated. It includes the Arithmetic and Logic Units (ALU) and the registers, while the *Control Unit* controls the operations done in the *Datapath*.

order to generate the hardware. In addition it is also possible to give to the tool some directives in order to control the way it synthesizes the different C/C++ constructs.
The extraction of a datapath and a control unit is done at a ’top level’, which means that only functions or loops will be associated to a state in the Finite State Machine. Then, the key attributes of the C/C++ code, such as functions, loops, arrays, types etc, are synthesized by Vivado HLS using some specific default implementations for each of them. For example, it associates an RTL block to a function, it rolls the loops and uses the same hardware resources for each loop iteration, it assigns memories to the arrays, etc.. [30]
Instead, the mapping from a C/C++ code into an hardware design is performed by Vivado HLS using the so called *scheduling* and *binding* processes. The *scheduling* determines the time structure of the code, at which clock cycle each operation is performed. While the *binding* aims to determine the number of hardware components used to cover all the operations. More specifically, there can be for example one resource of a specific type that can be used for more than one operation.
At the end of the process, Vivado HLS generates the synthesis report that contains information about the latency, Initiation Interval (II), area and others.
The latency corresponds to the time, in terms of clock cycles, between the moment when the logic receives an input and propagates the output.
The II, instead, is a parameter that is important in Vivado HLS and corresponds to the time required by the logic to take an input and accept a new one, still in terms of clock cycles [30].
Finally, the area corresponds to how many resources are needed on FPGA to implement the converted HLS code. The resources available on FPGA are:

- Block RAM (BRAM), that corresponds to the memory block available,
- Digital Signal Processing (DSP), resources used for multiplications;
- Flip Flop (FF), a sequential element able to store one bit of data;
- Look up table (LUT), a table that determines the output given a certain input.

Chapter 2

Hand-optimization of the hyper-parameters

The purpose of this chapter is to explore different neural network architectures in order to find which is the best one for the particles classification problem. First of all, in *Section 2.1*, I will describe some useful concepts, such as the loss function, the Area Under the Curve (AUC) and the Working Point (WP), that will be at the basis of the different approaches, presented in the next sections for selecting the best NN hyper-parameters, such as the batch size, the activation function, the optimizer, the regularizer and the number of filters and neurons, for this specific classification problem. Then, I will explore two different Neural Network architectures that are the Fully connected Neural Network (FNN) and the Convolutional Neural Network (CNN) in terms of their hyper-parameters. The first one, detailed in *Section 2.2*, allows to classify particles depending on their characteristics, described in *Section 1.2.1*, and for this reason it allows to better understand the properties of the specific particles under study. This architecture is used as a case study to learn more about the objects we are working with and to get acquainted with the world of Neural Networks in the LHC context, starting with simple architectures.
The second one, detailed in *Section 2.3*, allows to discriminate images of EM and HAD particles. Even if it is the ANN of greatest interest because it works better with images, due to its higher complexity compared with the FNN, it will be considered only when it is clearer why these hyper-parameters need to be optimized and, as a consequence, after a simpler study performed with a simpler network.
Finally, this last architecture will be compared in terms of performances when using different hyper-parameters in order to select the best ones in the LHC context.

2.1 Fundamental concepts

Some figures of merit, such as the loss function, the AUC and the WP of the classifier are necessary for understanding the hyper-parameters optimization performed for both the FNN and the CNN, respectively in *Section 2.2.1* and *Section 2.3.1*. As already said, the loss function ($\mathscr{L}$) allows for an evaluation of the difference between the target and the output of the Neural Network. To any fixed architecture it is associated a multidimensional parameters space, whose

variables are the weights to be trained and where the loss function is a scalar field. Each of these architectures needs to be trained, which means that the purpose is to find, in the specific parameter space associated to the architecture under analysis, the best possible ensemble of parameters, yielding the absolute minimum $\mathscr{L}^*$ of $\mathscr{L}$. As a consequence, if one labels a trained architecture with the value of $\mathscr{L}^*$ reached at the end of the training process, it is possible to compare different architectures via a scalar ($\mathscr{L}^*$), carrying by itself the information about how close will be a prediction of the model to the true value.
Before introducing the remaining concepts, it is worth recalling that our purpose is to determine the best possible classifier for EM and HAD particles, easily mapped in a binary classifier. When handling with this kind of structures, it is important to keep in mind that, after the training, by giving as an input the information associated to a certain particle one obtains as an output the estimation, made by the model, of the probability of the particle to belong to class 1 (EM particles). Consequently, this value carries the information about the probability of belonging to class 0 (HAD particles) as well. In this context, four quantities are commonly used, and they are usually presented in the form of a *Confusion Matrix* (CM):

$$CM = \begin{bmatrix} tn & fp \\ fn & tp \end{bmatrix} \tag{2.1}$$

These entries are:

- tp is the true positive number, i.e. how many times the NN gives 1, as output, when the correct value is 1;
- fp is the false positive number, i.e. how many times the NN gives 1, as output, when the correct value is 0 instead;
- tn is the true negative number, i.e. how many times the NN gives 0, as output, when the correct value is 0;
- fn is the false negative number, i.e. how many times the NN gives 0, as output, when the correct value is 1 instead;

These quantities are better being expressed as probabilities (or rates) by normalizing them along the columns. This yields the confusion matrix written in terms of rates:

$$CM_r = \begin{bmatrix} tn_r & fp_r \\ fn_r & tp_r \end{bmatrix} \tag{2.2}$$

where

$$\begin{cases} tn_r + fp_r = 1 \\ fn_r + tp_r = 1 \end{cases} \tag{2.3}$$

This having been said, these probabilities can be used for determining the quality of the trained architecture via a general concept, which is the ROC curve (Receiver Operating Characteristic [31]), and some of its characteristics, which are the AUC and the WP. The ROC curve has the tp_r, also called sensitivity, and the fp_r, where $1 - fp_r$ is also known as specificity, along its axis. This kind of graph allows to understand the ability of the binary classifier to discriminate between the two selected classes. One of the figures of merit that can be directly extracted from

the ROC curve and that gives useful information is the AUC, the area under the ROC curve, that gives a number between 0 and 1 corresponding to the probability that, applying this specific model to an element belonging to the class 1, the value on the output is higher than the output obtained when applying the same model to an element belonging to the class 0 [31].
For what concerns the WP, it is better to consider the ROC curve expressed as tp_r along the x-axis and $1 - fp_r$ along the y-axis. By choosing a given signal efficiency (e.g. 99%), we choose a point on the ROC curve, which corresponds to a working point of the classifier. This point is the background rejection at 99% of signal efficiency and will be indicated as WP_99 during the project. The idea is that, in this specific application, in order to have a good classifier, it should be characterized by a very high sensitivity and a very high specificity, that means very high value of the AUC and WP_99. When we say very high value, due to the fact that we are working with probabilities, we mean a value that is very close to one, such as 0.99. This value is subjective, it depends on which accuracy we want the model to achieve and it is fixed arbitrarily.

2.2 Fully Connected Neural Network (FNN)

Once the main concepts useful to evaluate the performances of the network have been introduced, it is possible to move to the description of a real application of how these concepts are used when analysing a simple ANN architecture, such as the FNN. This model is used to analyse the output of the stage 2 of the TPG by substituting the traditional classifier based on the shapes of the clusters. The structure of the FNN is exactly the structure of the MLP described in *Section 1.3*, where each neuron of one layer is connected to all the neurons in the next layer. This kind of structure has been used at the beginning of the project in order to understand the properties of the particles and to become confident with the neural network architectures. It also allows to observe how changing one hyper-parameter, the optimizer (described later in this section), that will be one of the hyper-parameters also present in the CNN, impacts the performances of the network. Another information of interest this analysis addresses is the relevance of some properties of the EM and HAD particles on the performances of the neural network. By verifying the goodness of the classification at different values of the generated particles position, it is possible to check the stability of the predictions and if the network fails for any of these values.

2.2.1 Preliminary analysis using a Fully Connected Neural Network

The Fully connected Neural Network (FNN) is used to have a better visualization of the data that characterize the dataset and how the data can influence the network performances.
Firstly, it is possible to consider a simple FNN (see *Fig. 2.1*) characterized by 55 inputs, 3 dense layers with 100, 55, 1 neurons respectively, the ReLu[1] as activation function except for the last layer that is characterized by a Sigmoid and the binary cross entropy [2] as a loss. At this

[1]is the Rectified Linear Unit and it is expressed as: $f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$

[2]$L = -\frac{1}{N}\sum_{m=1}^{M}[y_m \times log(h_\theta(x_m)) + (1 - y_m) \times log(1 - h_\theta(x_m))]$ where M is the number of training examples, y_m is the target label for training example m, x_m is the input for training example m, h_θ is the output of the neural

step, the ReLu is taken into account because it is the the most used activation function and in general works well for a wide range of problems [33].

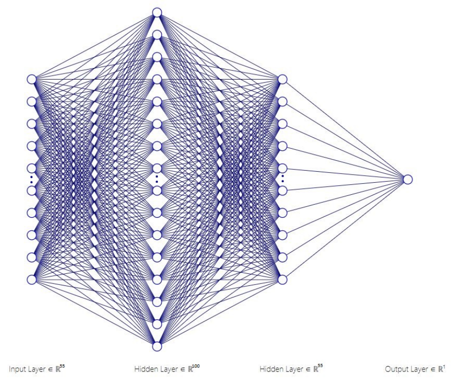

Input Layer $\in \mathbb{R}^{55}$
Hidden Layer $\in \mathbb{R}^{100}$
Hidden Layer $\in \mathbb{R}^{55}$
Output Layer $\in \mathbb{R}^{1}$

Figure 2.1: Structure of the Fully Connected Neural Network.

By fixing these hyper-parameters of the FNN, it is possible to run multiple networks with the same architecture but with different optimizers, which are methods used to explore the parameters of the model space with the aim of reducing the loss, in order to see which is the optimizer that gives a neural network with optimal performances for this specific binary classification problem. The different optimizers can be Adam, Adadelta, Adamax, Adagrad, RMSprop and SGD (see *Appendix A* for more details on their algorithms).
In order to understand which network gives the best results it is possible to evaluate some of the already mentioned figures of merit as the ROC curve and the background efficiency at 99% of the signal efficiency (that corresponds to 1-WP_99). As already said, the ROC curve is the curve that represents the value of the false positive rate (fp_r) as a function of the true positive rate (tp_r). These two rates correspond, in particle physics, to the background efficiency and to the signal efficiency respectively. From this curve, it is possible to directly extract the information about the background efficiency at 99% of the signal efficiency (fp_r_99) and the AUC for each Neural Network:

Table 2.1: Comparison of neural networks optimized with different optimizers.

Optimizers	fp_r_99	AUC
Adam	0.0298	0.9956
Adadelta	0.0876	0.9900
Adamax	0.0339	0.9950
Adagrad	0.0401	0.9942
RMSprop	0.0318	0.9949
SGD	0.0497	0.9911

As it is possible to observe from *Table 2.1* and from *Fig. 2.2*, the results in terms of AUC and in

network model [32].

terms of background efficiency are different for all the cases and in particular it is possible to observe significant variation when considering, for example Adam and Adadelta. In these two specific cases, it is possible to observe that Adam gives an AUC that is higher and a fp_r_99 that is smaller than the Adadelta case. As a matter of facts, by observing all the possible optimizers we will select the one that gives highest AUC and smallest fp_r_99. In this case, we should select Adam.

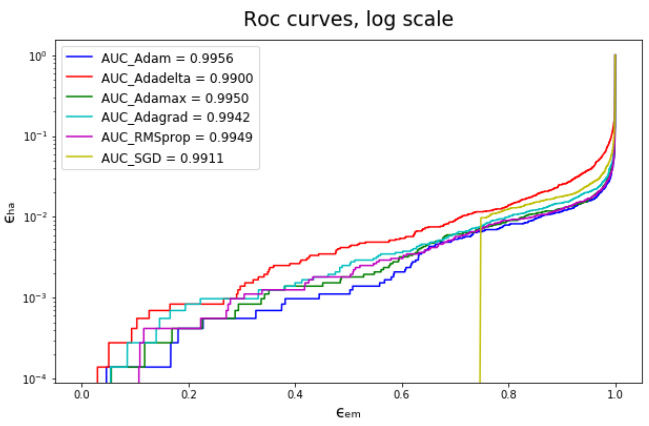
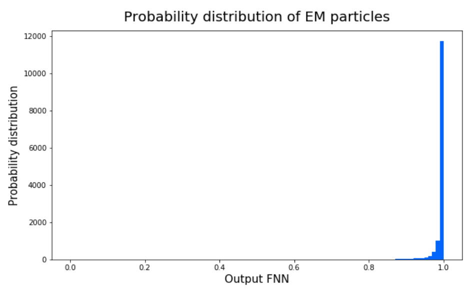


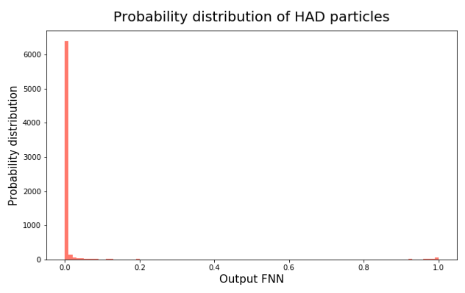
Figure 2.2: Comparison of the ROC curves for NNs with different optimizers, in logarithmic scale. ε_{em} corresponds to the signal efficiency, while ε_{ha} corresponds to the background efficiency (fp_r_99).

This is just a simple comparison of FNNs where the only hyper-parameter to be optimized was the optimizer, while all the others were already fixed. However, when we want to study a specific problem and we want to create an ANN that is the ideal one to solve it, it is necessary to find the whole set of optimal hyper-parameters of this network.
These hyper-parameters can be divided into two classes. The first amounts for all of those parameters necessary to fix the architecture, such as the *activation functions*, the *number of neurons* in each dense layer and the *size* and the *number of the filters* in each convolutional layer. The second class of parameters to be fixed is the numerical value of the filters to be applied and of the weights in the dense layers. Once the structure of the architecture is fixed, the goal of finding the numerical values of the last class can be reached by introducing a target function that is defined in the space of the hyper-parameters and whose minimum is reached for the optimal values of these hyper-parameters. After this function is selected, there are different approaches to explore the hyper-parameters subspace associated to the second class and to reach this minimum, but the philosophy behind them is the same. The idea is to evaluate the gradient and go in an opposite direction with respect to it in the hyper-parameter space. The amplitude of each movement is dictated by a parameter called learning rate in the process defined by the so called *optimizers*. Since the evaluation is computationally expensive, the way the optimizers are implemented is by dividing the dataset into sub-datasets whose dimensionality is called *Batch size*. The movement is thus performed considering just the cheaper partial evaluation of the gradient rather than the more expensive full evaluation.
Due to the fact that our goal is the classification of particle images into EM (corresponding to the class 1) and HAD (corresponding to the class 0), this kind of optimization, where all these

mentioned hyper-parameters are explored, is what we will perform for the CNN in the next section.
It is worth noticing that the process of optimization is not an exact method and there is not a specific path to follow. It depends on who is doing the optimization to choose a road and follow it, for example choose a parameter to start with, fix it and move on to the next selected parameter.
In addition to this simple comparison, in order to verify the goodness of the model with the Adam as optimizer, the output of the NN is represented in terms of probability distributions (see *Fig. 2.3*). As expected for the EM particles the probability distribution is around 1 while for the HAD particles it is around 0, meaning that the network is performing the classification well. However, there are some cases, even if in smaller quantity, where the network is giving the opposite results compared to what expected. What we suppose is that, by changing other hyper-parameters of the network, these errors will be corrected.



(a) *Probability distribution of predictions of EM particles.*



(b) *Probability distribution of predictions of HAD particles.*

Figure 2.3: Probability distribution of the FNN predictions of EM and HAD particles.

In particles physics, it is common to check efficiencies as a function of the pseudorapidity (η) because the detector is not uniform, and the energy flow of the collisions is also not uniform. Therefore, in general the performance are not constant and depends on where we look in the detector (vs. eta). Consequently, it is interesting to consider one parameter of the dataset described in *Section 1.2.1*, η, that defines the position of the EM and HAD particles with respect to the beam axis and that could affect the performance of the network. In order to see how the signal efficiency[3] of the NN varies for different η, it is necessary to divide the validation dataset into different sub-datasets depending on the absolute value of the pseudorapidity ($|\eta|$), where the absolute value is added because η could be positive or negative depending on which side of the detector the particle arrives. More precisely, the dataset was divided into 14 sub-datasets with an interval of η's value of 0.1, starting from $|\eta| = 1.5$ until $|\eta| = 2.9$ (it means that, for example, the first sub-dataset is characterized by $1.5 \leq |\eta| < 1.6$).
In this way some sub-datasets can become very small and do not have a sufficient number of samples (at least 100 samples) to correctly test the network and for this reason they can be

[3]The signal efficiency for a NN is the ratio between the number of times the NN gives the correct output (1) when the particle to be classified belong to the class 1 over the total number of particles belonging to the class 1 in the dataset.

removed. This is what happened for $1.5 \leq |\eta| < 1.6$, because $|\eta| = 1.5$ is the minimum value for $|\eta|$ and a very small number of particles presents this characteristic.
Once divided the validation dataset, it is possible to evaluate the signal efficiency of the networks, i.e. the number of times the output of the network is higher than a certain threshold when the target is an EM particle (class 1) over the total number of predictions made by the NN for the EM particles. The threshold can be selected, for example, by using the interpolation method. As a matter of fact, it consists in the evaluation of the ROC curve of the trained network with the complete dataset, based on the evaluation of tp_r and fp_r at different value of a threshold. Then from this curve the threshold at which the tp_r is equal to 0.99 is selected as a criterion to check the efficiency of the network. The result, shown in *Fig. 2.4*, allows to see that the efficiency is almost constant for different values of η and around 0.9898 ± 0.0069 and that there is not a significant dependence of the performances of the network on the position of the particle to be recognized.

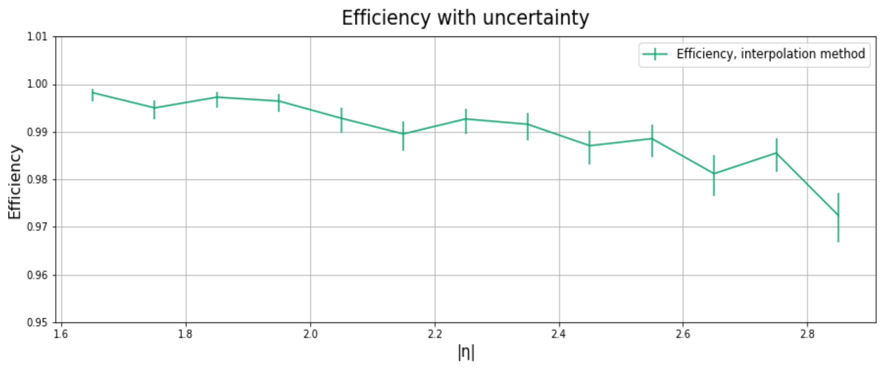


Figure 2.4: Signal efficiency as a function of the pseudorapidity (η).

This analysis focused on the FNN, and allows to see that substituting the traditional classifier based on the cluster shapes with a deep neural network model is promising for the classification of EM and HAD particles on the output of the TPG structure in the CMS detector. This is because this new approach allows to achieve performance of the order of 0.99, that are optimal being near to the ideal case (1).

2.3 Analysis of the Convolutional Neural Network (CNN)

Once a very simple study of the hyper-parameters of the network and of its performances is done, it is possible to move towards the description of the analysis of a more sophisticated structure of the ANNs, the convolutional neural network. As already mentioned in *Section 1.3*, this architecture is that of highest interest in our study because it is capable of performing well with images and, as stated in *Section 1.2*, it is a possible solution to reduce the complexity in the analysis of the EM and HAD particles on the output of the CMS detector.

2.3.1 Hyperparameters optimization

The first thing that can be considered in this study is a small comparison of the performance of the FNN with the performance of a CNN without any kind of optimization, just to observe if they have similar performances when the data to be analyzed contain the same raw information, yet in a different representation. In order to do that, it is possible to choose some non-optimized parameters for the CNN. The considered structure (shown in *Fig. 2.5*) was:

- Input layer of size $21 \times 21 \times 3$, corresponding to the size of the images;
- A 2D convolutional layer, with 16 filters, a 3×3 kernel, a ReLu as activation function, a pooling layer with size 2×2, a dropout layer with a fraction of unit's layer to drop out equal to 0.2;
- A 2D convolutional layer, with 32 filters, a 3×3 kernel, and ReLu as activation function, a pooling layer with size 2×2, a dropout layer with a fraction of unit's layer to drop out equal to 0.2;
- A Flatten layer;
- A dense layer, with 127 neurons and a ReLu as activation function, a dropout layer with a fraction of unit's layer to drop out equal to 0.2;
- A dense layer, with 2 neurons as output and a softmax as activation function.

The dropout layers are used as a regularizer method, for example to avoid overfitting, i.e. to avoid a learning 'by hearth' of the network. Without regularization, the network would indeed correctly classify during training only, but no longer during the validation step.

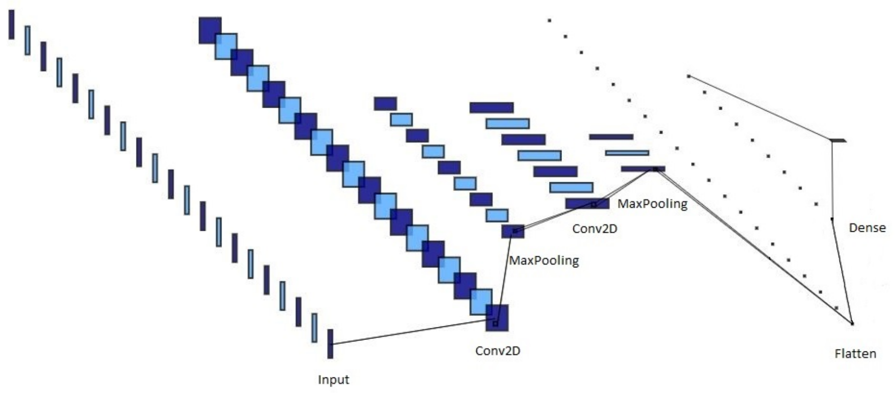
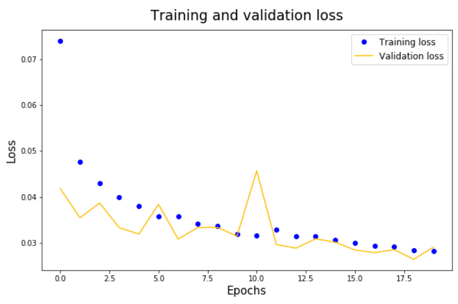
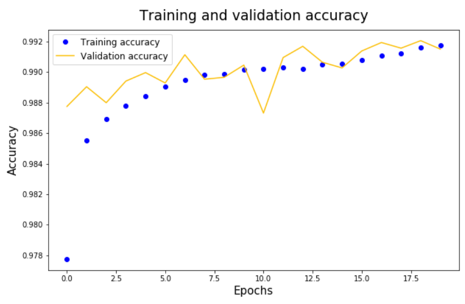


Figure 2.5: Structure of the Convolutional Neural Network.

Finally, in order to compile and train this model we also selected the optimizer and the loss, as done for the FNN, to be Adam with a learning rate equal to 0.001 and the categorical cross entropy respectively. This categorical cross entropy is just a generalization of the binary cross entropy that works well even when the problem is not a binary classification, i.e. there are more than two classes. The accuracy and the loss for both the training and the validation sets are shown in *Fig. 2.6.* It is important to specify that the model was trained for 20 epochs.



(a) *Loss as a function of the number of epochs.*



(b) *Accuracy as a function of the number of epochs.*

Figure 2.6: Loss and Accuracy as a function of the number of epochs. The blue curves are associated to the training while the yellow ones to the validation.

At this point, it is possible to evaluate the same figures of merit considered when dealing with the FNN: fp_r_99, AUC (*Table 2.2* and *Fig. 2.7*) and the signal efficiency as a function of η (*Fig. 2.8*).

Table 2.2: Comparison between the FNN and the CNN.

Neural Networks	fp_r_99	AUC
FNN	0.0298	0.9956
CNN	0.0125	0.9988

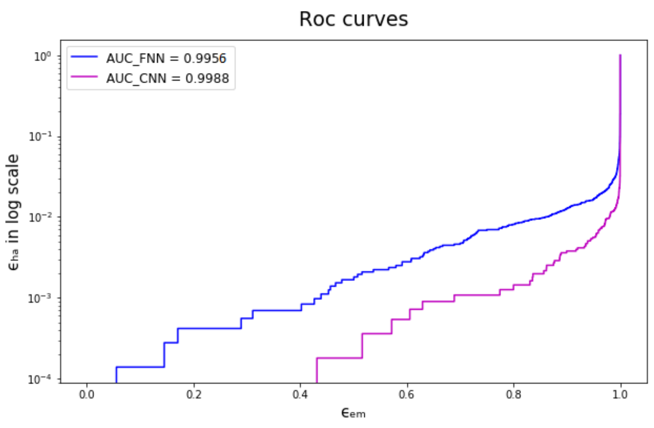


Figure 2.7: ROC curve for the FNN, in blue, and for the CNN, in magenta.

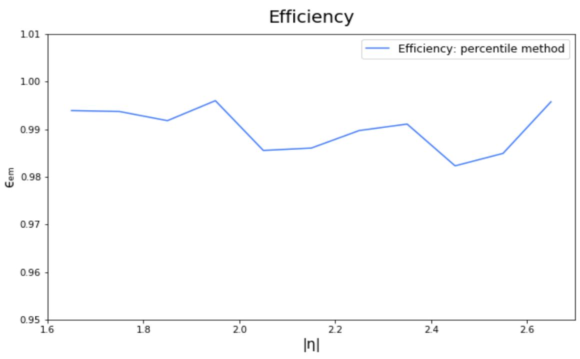


Figure 2.8: Signal efficiency of the CNN as a function of different values of the pseudorapidity ($|\eta|$).

As it is possible to observe from *Fig. 2.7* and from *Table 2.2*, the AUC for the CNN is better than the one obtained with the FNN because it reaches a higher value. *Fig. 2.8*, instead, shows that the signal efficiency does not change for particles at different position in the detector for the CNN as well. It is always around 0.99 and the spread is less significant than that obtained with FNN: 0.9900 ± 0.0045. These results suggest that the CNN architecture presents optimal performances. This aspect, along with the possibility to analyse the images instead of the characteristic of the particles, thus avoiding the reconstruction of the clusters, makes the CNN very promising as a replacement of the current method (cluster algorithm plus classifier based on cluster shapes) used to analyse the output of the CMS detector. In addition, it seems more promising than just replacing the classifier, such as the Boosted Decision Trees, with the FNN, because it allows to remove the stage 2 of the TPG, speeding up the process.
Considering the fact that the CNN can be useful to map the information contained in the images into a vector, by exploiting the flatten operation, it is also possible to insert additional features FNN-like as additional nodes at the level of the flattening operation for this network. This can be done to see if the efficiency of the network will improve by adding the aforementioned features. For this purpose, two parameters that correspond to the position of the center of the image in the detector and are contained in the dataset 2 can be added to the flatten layer, e.g. seedx and seedy. The goal is to give the information about the position of the shower, which cannot be extracted from the images (since the images given to the network are very small portions of the detector). The compared performances of the network in terms of signal efficiency vs. $|\eta|$ with and without these additional parameters are plotted in *Fig. 2.9.*
Observing the previous image, the variations in terms of efficiency do not give any improvement when seedx and seedy are added as inputs. For this reason all the following consideration will be done by keeping as input only the images.

As mentioned before, the CNN allows to analyze images. On the image a certain number of filters is applied in order to extract features from it. Once this operation is performed, the new images obtained by applying each filter on the image to be classified, can be mapped to a vector in a space of higher dimensionality through the flatten operation. This flatten layer will correspond to features of the images. Finally, the last layer will give as output the probability for the particle to belong to each class. As already said, in order to have a good classification it is necessary to find the optimal parameters of the network. Later on in the thesis some useful

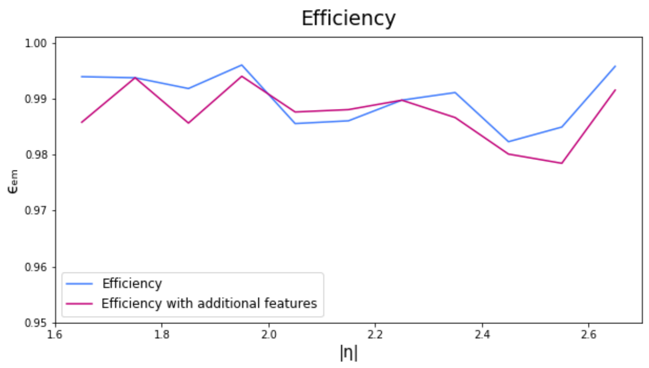
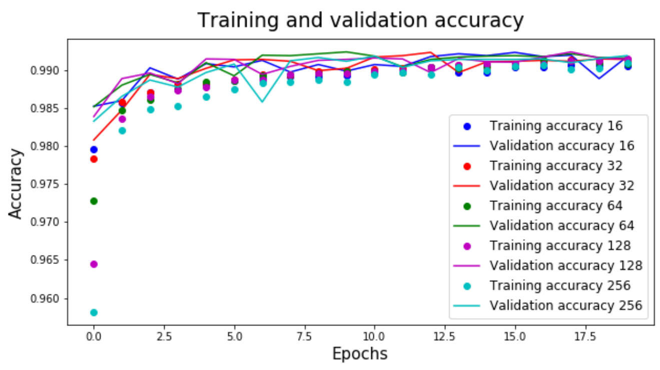


Figure 2.9: Signal efficiency of the CNN as a function of different values of the pseudorapidity ($|\eta|$). In blue the efficiency without the additional features and in magenta the efficiency with the two additional features.

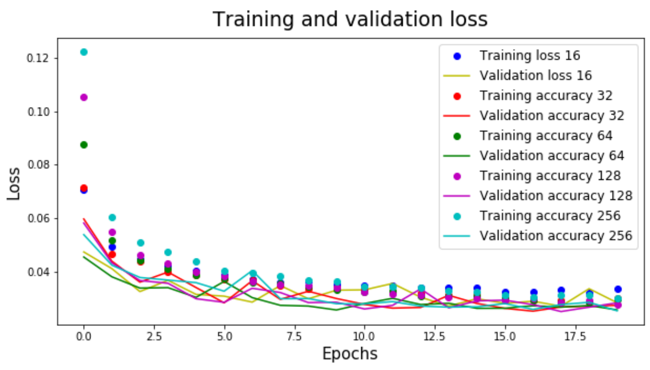
tuning techniques will be considered in order to explore different hyper-parameters of the CNN. Nonetheless, before doing that it can be useful to reduce the parameters to be selected by performing the so called 'Hand Tuning' of the easiest parameters to be set: Batch size, Activation Function, Optimizer. It will also be explored the possibility of using or not the dropout or a regularizer to avoid the overfitting during the training process. The role of each of these parameters will be explained in more details in the next paragraphs.

Batch Size

The batch size corresponds to the number of samples that are analyzed in one iteration of the evaluation of the gradient during the process of minimization of the loss. For example, in this case the number of samples is 64957, this means that if 32 is selected as a batch size, 2029 iterations are required to complete one epoch of the training process.
In order to understand which is the best Batch Size for this specific application, five neural networks with the architecture described in *Section 2.3* but with different Batch Size are considered and the accuracy and the loss as a function of the number of epochs for both training and validation sets are compared. The selected values for the Batch Size are: 16, 32, 64, 128, 256, where all of them are powers of two because it is the best choice that can be done when implementing neural networks on GPU, since the GPUs present better performance with numbers that are powers of two [34]. Once all the networks are trained and tested with the same train and test datasets, it is possible to compare their performances by representing the value of the accuracy and of the loss as a function of the number of epochs for both training and validation sets, as shown in *Fig. 2.10.* The best batch size, or in other words the one that allows to achieve the fastest decay of the loss and the fastest growth of the accuracy, is 32. The choice is justified because the number of epochs required to reach a small loss and an high accuracy is smaller with a batch size of 32. For this reason, from now on, this parameter will be fixed to 32 for all the next studies.



(a) *Accuracy as a function of the number of epochs for both training and validation sets for different batch sizes.*



(b) *Loss as a function of the number of epochs for both training and validation sets for different batch sizes.*

Figure 2.10: Training and validation accuracy and loss.

Activation Function

In order to select which is the best activation function for the first two convolutional layers and for the first dense layer, it is possible to train and test different neural networks that present always the same architecture, but with different activation functions. In this case the problem is characterized by an image that can be classified as belonging to one of the two classes: EM and HAD particles. However, these particles sometimes present properties that are very similar and there can be an overlap in the distribution of their characteristics or the images can be characterized by the same color. This suggests that these images of the particles cannot be linearly separated. As a consequence, in order to guarantee the classification, it is possible to take into consideration the non linearity by using non linear activation functions. The considered non-linear activation functions are:

- **ReLu**: is the Rectified Linear Unit and it is expressed as:

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

- **Tanh**: it is the hyperbolic tangent, in general it is very useful when the network needs to learn negative values because it is defined between -1 and 1.

- **SELU**: Scaled Exponential Linear Unit [35] that is expressed as:
$$SELU(x) = \lambda \begin{cases} x, & \text{if } x > 0 \\ \alpha e^x - \alpha, & \text{if } x \leq 0 \end{cases}$$
where $\alpha \simeq 1.67326$ and $\lambda = 1.05070$ [36];
- **Elu**: Exponential Linear Unit [37], it is expressed as:
$$ELU(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha e^x - \alpha, & \text{if } x < 0 \end{cases}$$
where α is selected between 0.1 and 0.3.
- **Sigmoid**: it is a logistic function that can be used in the problem where the output is a probability because it assumes values between 0 and 1 [36]:
$$\Phi(x) = \frac{1}{1+e^{-x}}$$

After having trained and tested the networks characterized by these activation functions, it is possible to compare them. In this case, the most effective way to compare them is not observing the loss and accuracy as a function of the number of epochs, but evaluating one of the most useful figures of merit for this classification problem for each of these networks. For instance, the background rejection at 99% of the signal efficiency (WP_99), the AUC, the background rejection at 95% of the signal efficiency (WP_95) or the validation loss at the last epoch, in order to have only one value to compare. Among all these useful figures of merit, we chose WP_99. In order to visualize the result, the value of the WP_99 for different activation functions is shown in *Fig. 2.11.*

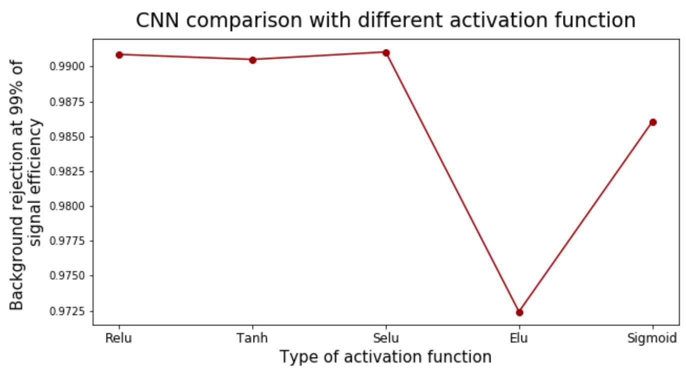


Figure 2.11: Background rejection at 99% of the signal efficiency for different activation functions.

As it is possible to observe, the results obtained with ReLu, Tanh and SELU are very similar and of the order of 0.9925 while the other two activation functions give worse performances. Among the three best results, the ReLu seems to be the best. In addition, it is also the most widely used in the machine learning community due to its numerous qualities compared with the others activations [33]. For these reasons, it is selected to be the activation function for the 1st and 2nd convolutional layers and for the first dense layer.

Selection of optimizer

Finally the last hyper-parameter that can be determined by a simple hand tuning is the optimizer. When considering a deep learning problem, there are different optimizers that can be used in order to improve the method and to reach the minimum faster. One optimizer can be the Gradient Descent (GD), where each step of updating parameter should be evaluated on the whole dataset. For this reason, each computation is very long especially if the dataset is very big. In this case it would be better to move towards different optimizers that, instead of taking the whole data, take batches of data reducing the computation time that is the case of the Stochastic Gradient Descent (SGD). In the SGD, there is still a fixed step size to move towards the global minimum of the function, but in this case the evaluated gradient is no more the exact gradient, but rather an approximation that comes from the consideration of one batch of data points instead of the whole set of points.
One of the disadvantages of the SGD is the fixed learning rate, i.e. the fixed step size during the achievement of the minimum of the function. It can bring to a lot of back and forward around the minimum, without allowing to reach it. In order to avoid this kind of oscillation, other optimizers that present an adaptive learning rate can be taken into consideration. One of them is called Adam and it is characterized by a learning rate that starts with a 'high' value (for example 1 or 0.1), in order to not slow down the process, and then it becomes smaller when the minimum becomes closer, in order to avoid the jumping around it.
When adaptive learning methods, such as Adam, are not used as optimizers, it can be useful to do something that allows to modify the learning rate. It is indeed always better to have a large step-size at the beginning to guarantee a fast algorithm, but the learning rate should be further reduced in order to reach the global minimum without causing overshoot (see *Fig. 2.12*). There are some techniques that can do that, such as *Annealing* or *Cosine Annealing*. The last one is used because there can be some holes in the function that may prevent from reaching the global minimum that should be minimized (see Fig. 2.13)

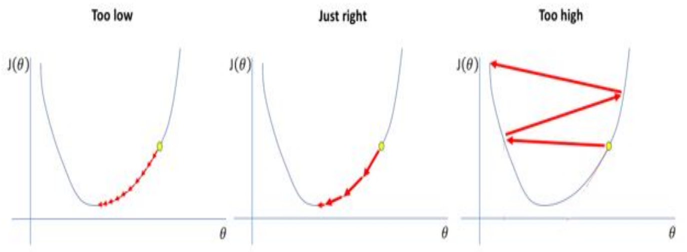
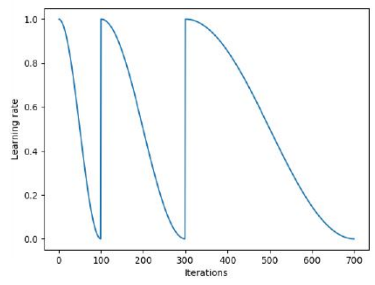
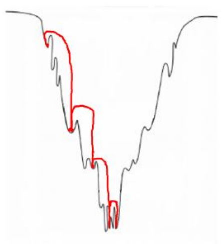


Figure 2.12: Behaviour of the evaluation of global minimum with different learning rates [38].

This kind of optimization can be used, for example, when Adam fails. It can then be safe to go back to SGD and use this function in order to speed it up and avoid being trapped in local minima. It means that convergence could become very fast.
After these considerations, it is possible to consider neural networks with the same architecture as the one described in *Section 2.3* with fixed batch size (32) and activation function (ReLu for the convolutional layers and for the first dense layer), but in this case characterized by different optimizers: Adam, Adagrad, SGD, RMSprop, Adamax, Adadelta. In order to compare them



(a) *Cosine Annealing function used to reduce the learning rate in an adaptive way.*



(b) *Effect of the Cosine Annealing function in the evaluation of the global minimum of the objective function.*

Figure 2.13: Cosine Annealing function used to reduce the Learning Rate in an adaptive way and improve the NN training [39].

the background rejection at 99% of signal efficiency (WP_99) is observed for each network (see *Fig. 2.14*) and the highest value of WP_99 is computed for Adam and Adamax. Due to the fact that Adam is able to perform well with an extensive number of non-convex problems in the machine learning field [40], we decided to select it as the optimizer for our case of study.

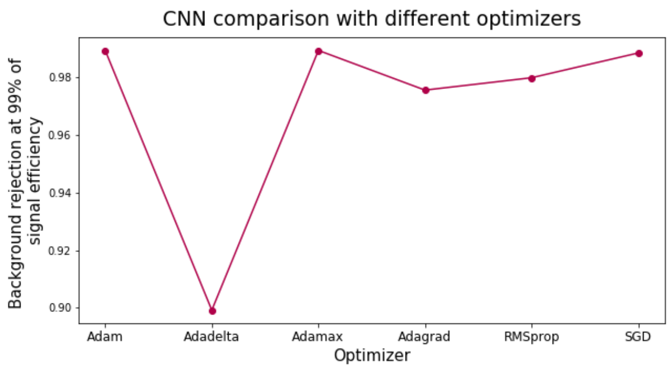


Figure 2.14: Background rejection at 99% of the signal efficiency for different optimizers.

Another important aspect to be considered when optimizing a NN is related to the overfitting problem. For this reason some regularization techniques can be considered for optimization such as the Dropout [41] and the use of Regularizers [42].

Dropout

As stated in the original paper by Nitish Srivastava et al [41], the dropout technique allows to reduce the overfitting problem. During the training of the network, for each new input data, some of the neurons are randomly kept in a 'non-active' mode and, due to this statistical behaviour, for each new data the deactivated neurons during the training are different. This method, which affects training time only, allows to simulate a training with different neural network architectures while, during the validation, it allows to consider the complete neural network architecture where the trained weights are re-scaled by the dropout factor.

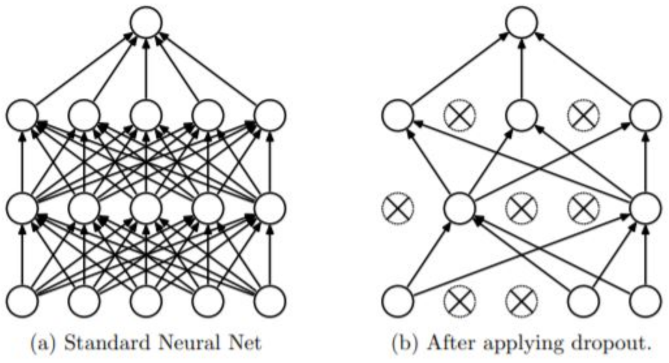


Figure 2.15: Simple NN structure with two hidden layers on the left and a structure of the NN when applying the Dropout during training on the right [41].

This is only the general idea of how this method works. In fact, when considering a CNN the meaning of the dropout is not exactly the same, because the concepts of neurons is substituted with the concept of filters. In the case of CNN, where the convolution can be easily described as simply matrix multiplications, the dropout technique replaces some of the elements in the matrix multiplication with zeros [43]. It is important to notice that this technique, in particular for CNN, is not always a good solution and can lead to worse results. For this reason in the following I will compare the performances, in terms of loss, for three NNs: one without the dropout technique, one with a dropout applied after each convolutional and dense layers, and the last one with the dropout applied only after the dense layer (see *Fig. 2.16*).

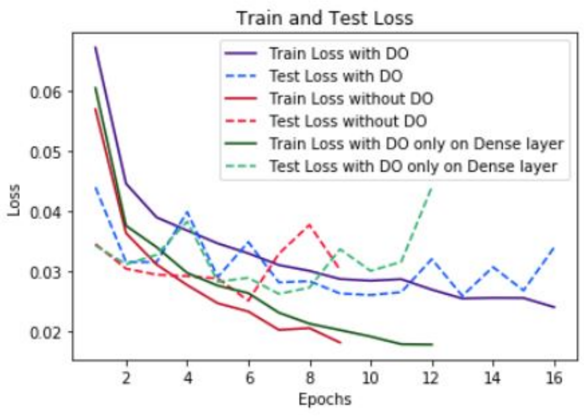


Figure 2.16: Comparison NNs with and without dropout technique.

As it is possible to observe from *Fig. 2.16*, either the red curves, representing the training and validation loss when no dropout technique is applied, and the green curves, representing the case in which the dropout is applied only on the dense layer, stop after a very small number of epochs because the validation loss starts very soon to increase instead of decreasing [4]. On the other hand, the blue curves, representing the NN trained and validated with the dropout technique, stop at a higher number of epochs meaning that there is less over-fitting than the red or green curves. Due to the less over-fitting reached by the curves that represent the model with the dropout technique, I decided to keep it after the convolutional layers and after the first dense layer for all the following analysis.

Regularizer

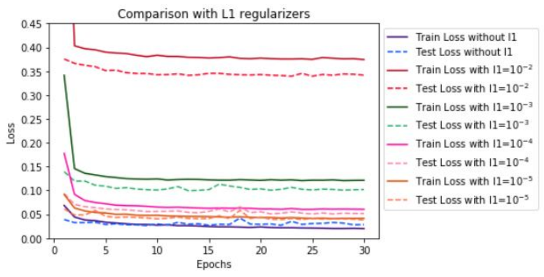
The purpose of the regularizers is similar to the one of the dropout layers. They can be added to the layers of the network in order to reduce the overfitting of the model. In particular the regularizers allow to introduce penalties in the weights of the network. This is possible by modifying a little bit the loss function, i.e. by adding to it the regularizer.
There are three main types of regularizers called: L1, L2, and a combination of L1 and L2. They are defined as follows:

- L1 regularization [44]:
  $$\lambda \left\| w \right\|_1$$
  λ is the regularization parameter and it will be optimized in order to improve the performances; and w are the weights of the network or the values of the filter of the network. This regularizer allows to reduce some weights to zero and to simplify the model.

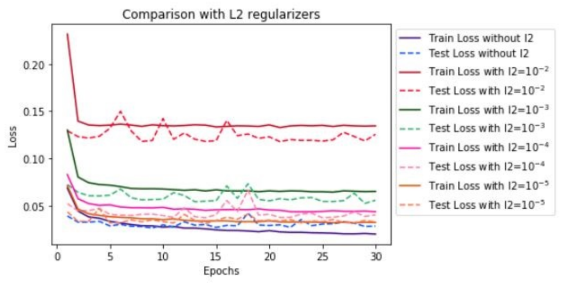
- L2 regularization [44]:
  $$\lambda \left\| w \right\|_2^2$$
  λ is again the regularization parameter and it will be optimized in order to improve the performances; w are the weights of the network or the values of the filters of the network. This regularizer forces some values of the weights of the network to assume very small values, which are near but not exactly equal to zero as in the case of L1.

In order to understand if this type of regularization is necessary during the training, we applied the three different types of regularizer to our model and we compared the results for different values of λ for each of them (see *Fig. 2.17*).
Due to the fact that when a regularizer is applied on the layer the values of the loss are modified by this additional factor, the plots in *Fig. 2.17* are not meaningful to understand which model gives the best performances. In order to have a more significant comparison the values of the AUC and of the WP_99 for all the different networks were evaluated and the results are summarized in *Table 2.3* for L1, in *Table 2.4* for L2 and in *Table 2.5* for L1_L2 .

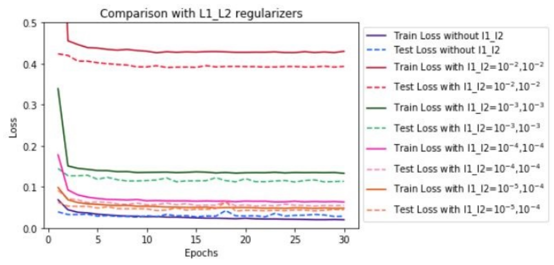
[4]In order to stop the training of the network when the model starts to become a bad model, we used the early stopping criterion, where the training is stopped once the validation loss becomes higher than the training loss for at least three consecutive times.



(a) *Comparison using L1 as regularizer.*



(b) *Comparison using L2 as regularizer.*



(c) *Comparison using the combination of L1 and L2 as regularizer.*

Figure 2.17: Comparison of the performance of the network in terms of train and validation loss without the regularizer (blue curves) and with regularizer with different values of λ: 10^{-2} in red,10^{-3} in green,10^{-4} in pink,10^{-5} in orange.

Table 2.3: Comparison of the AUC, the WP_99 and the loss when using L1 as regularizer and different values of λ.

	no regularizer	L1($1e^{-2}$)	L1($1e^{-3}$)	L1($1e^{-4}$)	L1($1e^{-5}$)
AUC	0.9988	0.9894	0.99688	0.9989	0.9992
WP_99	0.9893	0.9228	0.9788	0.9877	0.9915
Loss	0.0283	0.3417	0.1019	0.0517	0.0383

Table 2.4: Comparison of the AUC, the WP_99 and the loss when using L2 as regularizer and different values of λ.

	no regularizer	L2($1e^{-2}$)	L2($1e^{-3}$)	L2($1e^{-4}$)	L2($1e^{-5}$)
AUC	0.9988	0.9957	0.9986	0.9990	0.9992
WP_99	0.9893	0.9726	0.9846	0.9895	0.9918
Loss	0.0283	0.1255	0.0558	0.0403	0.0328

Table 2.5: Comparison of the AUC, the WP_99 and the loss when using L1_L2 as regularizer and different values of λ.

	no regularizer	L1($1e^{-2}$) L2($1e^{-2}$)	L1($1e^{-3}$) L2($1e^{-3}$)	L1($1e^{-4}$) L2($1e^{-4}$)	L1($1e^{-5}$) L2($1e^{-4}$)
AUC	0.9988	0.9883	0.9963	0.9989	0.9989
WP_99	0.9893	0.9152	0.9748	0.9864	0.9886
Loss	0.0283	0.3934	0.01136	0.0551	0.0442

As it is possible to observe from *Fig. 2.17* and from *Tables 2.3, 2.4, 2.5* when a not very small value of λ, such as 10^{-2}, is used, the performance of the network becomes worse of three orders of magnitude (10^{-3}) for the AUC and of the order of 10^{-2} for the WP_99. This suggests to try to decrease the value of λ in order to reach similar performances. However, when the value of λ decreases the performances of the networks are similar to the case where there is no regularizer, but the behaviour of both the validation and the training loss (shown in *Fig. 2.17*) becomes very similar to the behaviour of the train and validation loss when no regularizer is used (blue curves). This result suggests to avoid using any regularizer in this specific study and go ahead with the model without this kind of regularization.

2.4 Summary

In this chapter we have analyzed the hand-tuning of the hyper-parameters, first for a very simple structure such as the FNN in order to observe the effect of the variation of the hyper-parameters on the performance of the network in the EM and HAD classification problem. Then we analyzed a more complex neural network, that is the one of interest in this kind of classification problem, because it allows to classify images of the particle showers.
More specifically, in the case of the CNN we considered all the possible hyper-parameters that can be optimized by hand such as: the batch size, the activation function, the optimizer. We also tried to add some regularizations to our model, in particular the dropout and the regularizer.
At the end of the hand-tuning of the hyper-parameters of the CNN, the structure that we have is characterized by:

- An input layer of size $21 \times 21 \times 3$;
- A 2D convolutional layer, with 16 filters, a 3×3 kernel, a ReLu as activation function, a pooling layer with size 2×2, a drop out layer with a fraction of unit's layer to dropout equal to 0.2;
- A 2D convolutional layer, with 32 filters, a 3×3 kernel, and ReLu as activation function, a pooling layer with size 2×2, a dropout layer with a fraction of unit's layer to drop out equal to 0.2;
- A Flatten layer;
- A dense layer, with 127 neurons and a ReLu as activation function, a dropout layer with a fraction of unit's layer to drop out equal to 0.2;

- A dense layer as the output layer, with 2 neurons, and a Softmax as activation function.

with a batch size = 32, the presence of the dropout layers and without any L1, L2 regularizer. At this point, using the hyper-parameters selected by the hand-tuning, the only hyper-parameter that can be still tuned is the network's size.
For the selection of this parameter, a more sophisticated analysis needs to be done because it requires a lot of computational time. This analysis is the optimization of the number of filters in the convolutional layers and of the number of neurons in the first dense layer. The latter optimization will be of fundamental importance because the number of filters and neurons are the main responsible of the size of the neural network architecture when implementing it on FPGA. For this reason, we are really interested in finding the optimal value of these parameters that allows to have both great performances and a small size when implementing the network on FPGA, in order to reduce the overall resources usage.

Chapter 3

Bayesian Optimization

At this point, the hyper-parameters of the CNN, that can be explored by a simple hand optimization, are fixed and it is necessary to focus on the size[1] of the network. In order to fix the size of the selected architecture (see *Section 2.3*) three parameters need to be determined:

- The number of filters in the first convolutional layer (sl1);
- The number of filters in the second convolutional layer (sl2);
- The number of neurons in the dense layer (sl3).

This means that the goal is to find the best combination of these three parameters that gives the best network both in terms of performances and in terms of resources usage on FPGA once implemented the network on it. Due to the high space, generated by all the possible values of these three parameters, that needs to be explored in order to determine their best combination, the hand optimization is not very suitable because it requires very long time to explore all the possible combinations of the points in the space. For example, if sl1, sl2 and sl3 can assume values between 1 and 128, all the possible combinations that need to be explored will be $128 \times 128 \times 128$. This means that different optimization methods need to be considered in order to find the optimal parameters for the size of the network.
For this purpose, this chapter will provide the idea of how the optimization of these types of hyper-parameters can be done. *Section 3.1* introduces some of the useful concepts and methods to perform the optimization of the hyper-parameters, such as the number of filters and neurons, that require an high dimensional space where the ideal values can be searched and to understand the reasons why these parameters need to be optimized. *Section 3.2* provides the explanation of the process behind one of the methods, called Bayesian optimization, that will be used to find the number of filters and neurons for the CNN. Then *Section 3.3*, will describe in more details the Bayesian Optimization and will present the methods used in order to determine two functions useful to perform this kind of optimization, such as the target function and the kernel function. Finally, *Section 3.4* will show the results obtained when performing the Bayesian Optimization to find the best parameters that will determine the size of the network.

[1]In an ANN with a fixed architecture characterized by convolutional and dense layers, with size of the network we refer to the number of filters and neurons of these layers.

3.1 Fundamental concepts

When the size of the network is not known, it is necessary to use some techniques to find which is the best neural network topology. This can be done by using a target function assuming a different value for each trained topology and defined in such a way that its value is indicative of the goodness of the size of the network. In particular, the better the topology, the higher the value of such a function or, alternatively, the lower the overall error committed when using this architecture. Therefore, the definition of such a function allows to map the problem into a maximization problem since the global maximum of the target function will correspond to the best architecture. In this study it is interesting to optimize the topology because it will affect the hardware implementation, e.g. the resources usage. In general, it is of interest also because such a maximization guarantees to have a good model where it is possible to reach the so called bias-variance trade-off [39]. This is of interest whenever the relationship between the given data and a function depending on them, as in this case the images of the EM and HAD particles and the probability to belong to one of the two classes, can be represented by a parametric model, such as a neural network. In fact, depending on the number of parameters introduced in this neural network model there are two kinds of error, depending on the complexity of the model, that can appear [45]:

- **Bias error** or **underfitting**. Its presence indicates that the assumptions done on the properties of the parametric model would not allow the correct estimation of the target function. In the specific case of topology selection, this corresponds to the fact that the model is not complex enough and cannot correctly represent the data. As a matter of facts, the more complex the parametric model is, the more the peculiarities of the function to be estimated will be easy to approximate and, consequently, the lower the bias error will be.
- **Variance error** or **overfitting**. Its presence is indicative of the fact that the complexity reached by the network is too high. The more complicated the model, the higher the variance, because the model starts to encode fluctuations that are too much data dependent.

This means that as the model complexity increases the bias error becomes smaller but the variance error increases while, if the model is too simple the variance error becomes smaller but the bias error increases. Therefore, there is a point in the space of the variables that defines the size of the network (such as sl1, sl2 and sl3) at which the global error, i.e. the sum of these two, is optimal, when the model is neither too complex nor too simple. It is worth to be noticed that finding this point is not the final answer to the problem. This would solve the problem for what concerns the software side of the optimization problem. Nonetheless, the focus of the thesis is not only to find the network that optimally solves the classification problem, but also to obtain the network which is optimally implementable on hardware. This additional trade-off to be considered requires a deepened study of the behavior of the target function in terms of the topology, in such a way that one can understand how it is possible to change the parameters of the network to obtain the best architecture both in terms of classification problem and in terms of resources usage.

So far we have discussed that a maximization problem needs to be solved in order to find the optimal topology of the network. Nevertheless, in neural network optimization there is no way to know the target analytically as a function of the topology and, most importantly, it is not guaranteed that such a functional relationship is derivable. In fact, each of the points in the space of the topologies is a neural network training and, due to the partial randomicity of the result of a training process, neither continuity is certain. Additionally, it is very expensive to evaluate the target at each of these points. Therefore, even a numerical maximization after the evaluation of the target on the whole domain is out of discussion due to computational limitations. This implies the need for alternative methods for exploring the space of topologies, namely the *Zero-Order Optimizations*. These optimization techniques have been designed to optimize non-convex objective functions and do not require the evaluation of the gradients [46]. There are different techniques that belong to this optimization class and can be used to explore the space of the parameters that define the size of the network (sl1, sl2 and sl3), for example: Grid search, Random Search, Covariance Matrix Adaption Evolution Strategy (CMA-ES) and Bayesian Optimization (BO).

The Grid search (GS) is a technique where all the possible combinations of all the hyper-parameters in a selected range are analysed. In particular, it is based on a for loop where at each iteration a target neural network is built, trained and then the evaluation is performed to have some results, such as the validation loss, the AUC, the WP_99, to compare the trained network with the other NNs. This kind of techniques has a time complexity that increases very quickly with the number of hyper-parameters.
Alternatively, it is also possible to use the Random search (RS) that is almost the same technique as the grid search, but in this case the hyper-parameters are randomly chosen and not all the possible combinations are taken into consideration, reducing the computational time [47].

Another possible technique is the CMA-ES that is a sort of genetic algorithm that starts with a population of points (which in our case would correspond to sl1,sl2 and sl3) and at every generation keeps the best half of the points, then it evaluates the correlation of these points and regenerates a new generation of points following the found correlation (see *Fig. 3.1* [48]). This algorithm converges very fast. The main problem is the computational cost, because it is a function of the square of the dimension of the points at each generation ($O(dim^2)$).

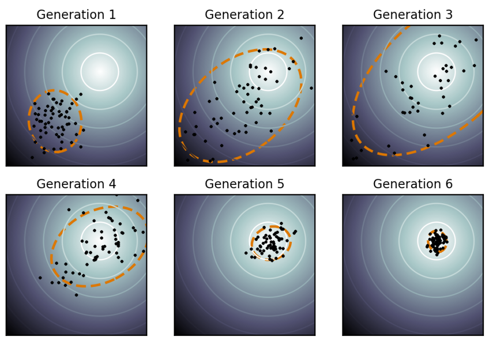


Figure 3.1: Steps of a CMA-ES algorithm [48].

Every time the computation is very expensive, it is possible to use the *Data-driven sampling*. In particular, these techniques start by sampling some data, computing a model that represents the data and then, instead of sampling every time all the data, they use this model to evaluate the best next sampling point and so on. There are different algorithms to perform Data-driven sampling and the best one is the *Bayesian Optimization*.

The Bayesian Optimization (BO) allows to estimate the target function, defined in the space of the topologies, by estimating its mean and its accuracy interval, and to define a second function, called acquisition function, that is used as a support during each iteration of the optimization. At each iteration, the acquisition function is used to define, from a probabilistic point of view, which is the best region of the space, namely the new best combination of the topology parameters (sl1, sl2 and sl3), to be explored in order to find a point belonging to the target function that could have a value that is higher than the previous known points. In order to include this additional information in the estimation of the target function, which corresponds to a new value of the target function for the new best combination of the topology parameters, a method called Gaussian Process (GP) [49] is used. The main assumption at the basis of this method is that the vector formed using the values of the target function can be described by means of a multivariate gaussian distribution. Moreover, this distribution, or more precisely its mean and covariance matrix, is assumed to be parametrized by the topology parameters (sl1, sl2 and sl3). The mean and the covariance matrix of the multivariate gaussian distribution, through which the target function is modeled, are updated at each iteration, changing the approximation thanks to the additional information that is introduced.
Ideally, the BO iteration procedure will converge to the maximum. This is to say that, after a sufficient number of iterations is performed, it is expected that the acquisition function will suggest to repeatedly sample the same combination of the topology parameters, not changing anymore the approximation of the target function. Since this is just an ideal condition and the convergence to the exact maximum is not guaranteed in a reasonable number of steps, it is better to fix the desired number of iteration to stop the optimization. In any case, the final topology identified by the BO will belong to a region, in the space of the hyper-parameters, where the target will be higher than the other explored regions. Once the process of optimization is completed, it is possible to check for which combination of the topology parameters the maximum of the found target function is achieved and this combination will define the best topology of the network.

More precisely, the Bayesian Optimization [50] is based on the Bayesian Inference, a statistical technique that allows to infer the model from the data[2]: starting from the data and a prior, that is a prior idea of what the model is, it is possible to update the prior with the data and find the so called posterior distribution. The evaluation of this posterior distribution is based on the Bayes theorem:

$$p(model|data) = \frac{p(data|model) \cdot p(model)}{p(data)}$$

In general the model that is used to represent the data is a Gaussian Process that allows to find the parameters of the multivariate gaussian distribution that model the function of the data (target function) that can be characterized or not by a stochastic component. Using this Gaussian Process, it is possible to find a representation of this target function in terms of a

[2]The data in this kind of model and in the case of a neural network where we are optimizing the size of the network, correspond either to the different combinations of the topology parameters for which we know the associated value of the target function, such as sl1, sl2 and sl3, and to the associated values of this target function.

mean function and a variance/covariance function. This variance/covariance function gives the variability around the mean and it depends on the points of the target function (see *Section 3.2*). So the idea is to use the Bayesian inference to calculate which is the best representation of our data starting from a prior information of what the data are. In general, for the prior, it is assumed that the mean is zero while for what concerns the covariance an hypothesis is used: the covariance depends on the distance between points, the farther the points are, the lower is the covariance, in a non trivial way to be approximated and it will be described by a function called *kernel* that needs to be defined. After the evaluation of the mean and the covariance, the goal at each iteration is to search the best point to define the new combination of the parameters (sl1, sl2 and sl3) to be considered, in particular we need to find where the mean plus the standard deviation allows to reach a better point than our maximum. In order to find this better point an *acquisition function* is used. Different choices are possible for this function, but for this specific study the one called Upper Confidence Bound (UCB) will be considered. The UCB is characterized by the mean plus the standard deviation:

$$A(x) = \mu(x) + k\sigma(x)$$

μ is the mean; σ is the standard deviation; x corresponds to the parameters that fix the size of the network (sl1, sl2, sl3); and k is an hyper-parameter of the optimization process that needs to be determined. It could be useful to specify that the UCB is equivalent to the Lower Confidence Bound (LCB) where the sign of the mean is minus ($-\mu(x)$).
Then, at each iteration the network is trained for the new values of the hyper-parameters, a fit of the data with the additional values of the target function for the new hyper-parameters is performed using the GP, the acquisition function is evaluated at each point in the space and the maximum of the acquisition function will determine the new point to sample in the space.
The value of the hyper-parameter k gives a trade-off between exploration and exploitation. Exploration means that some regions of the function are explored randomly and the higher is k the higher is the exploration of the space. Exploitation means that the points in the space that are already found out are explored more, and the lower is k the higher is the exploitation. The optimization of the k parameter is in general done by hand.

The main limitation of the Bayesian Optimization is the Curse of Dimensionality [51]: as the dimensionality of the space, where the data are defined (in this case the topology space), increases, the volume of the space increases and as a consequence the data become sparse. It means that an increasing number of data are required to get a proper fit with a method that relies on the distance-based similarity between points. Consequently, this kind of optimization is limited in dimension and performs well up to a dimension of the space that is 20-30. However, in this specific problem the dimensionality is only 3, because only sl1, sl2 and sl3 need to be determined at this point, and as a consequence it is possible to use this method for the optimization without being affected by its limitation.

3.2 Gaussian Process

Having introduced in a general way different methods useful to optimize the parameters associated to the size of the network (number of filters, sl1 and sl2, and number of neurons, sl3) and having identified what might be the most suitable one when the computation time is

very expensive, called Bayesian Optimization, it is important to understand how the Gaussian process works, since it is at the basis of the Bayesian Optimization principle.
The Gaussian Process (GP) is a stochastic process involving random variables with a Gaussian distribution. The general idea ([52]) is to consider an independent variable (x) ranging in a domain ($\mathscr{D}$) of our function, and a random variable that is the function of interest (y = f(x)). At the first step a subset of the domain $\{X_1 : x_i \in \mathscr{D}, i = 1,..,n_1\}$[3] whose corresponding Y ($Y \in Y_1$) are know, is selected. They are the known parameters that are used to select the prior for the model. The main assumption is that the random variable (Y) follows a Multivariate Normal distribution ($\mathscr{N}(\mu(X), \Sigma(X))$), where $\mu(X)$ is the mean; $\Sigma(X)$ is the covariance matrix; and $\sigma(x)$ is the standard deviation that is related to the spread of the points from the mean. In particular, in the GP, the covariance plays a very important role because it characterizes the entire process. In fact, it tells us how much the values of the function at two different points of its domain are correlated. This is a structural information, essential for an optimal approximation using such a multivariate gaussian description. The covariance matrix is in general not known. It needs to be imposed from the outside in such a way that it satisfies the requirements of a gaussian covariance matrix (positive definiteness and symmetry). In particular, it is forced to be parametrized by the x-values by means of a function called kernel ($K(x_a, x_b)$) that can be of different functional forms. The most used kernel is the RBF kernel, the exponentiated quadratic covariance function:

$$k(x_a, x_b) = e^{-\frac{1}{2\ell^2}\|x_a - x_b\|^2} \tag{3.1}$$

where ℓ is the length-scale parameter and it is a positive number ($\ell > 0$). Starting from this consideration, it is possible to consider new samples from $\mathscr{D}$ and evaluate the so called posterior probability of Y_2. This means that a new sample $\{X_2 : x_i \in \mathscr{D}, i = 1,..,n_2$ where n_2 is the number of new samples considered in the domain $\mathscr{D}\}$ is selected and the corresponding values of the y variable (called for simplicity Y_2) can be predicted. In fact, the posterior probability distribution of Y_2 is the conditional probability of Y_2 given X_1, Y_1, X_2. More specifically, this parameter gives information about the possible interval of values that can be assumed by each variable $y \in Y_2$ knowing the parameters X_1, Y_1, X_2. In the following, I will derive the update equations, already present in the literature [49, 52], at the basis of the regression in the Gaussian Process. As said before, we consider the multivariate random variable $Y \in \mathbb{R}^N$, that contains both Y_1 and Y_2 as

$$Y = \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} \tag{3.2}$$

Its joint distribution can be written as:

$$p(Y_1, Y_2 | X_1, X_2) = \frac{1}{\sqrt{2\pi|\Sigma|}} e^{-\frac{1}{2}(Y-\mu)^T A (Y-\mu)} \tag{3.3}$$

where the inverse of the covariance matrix, $A = \Sigma^{-1}$, has been introduced. The purpose is to find the probability distribution from which to sample Y_2 given the information on Y_1, X_1 and the points in which the function needs to be estimated. In order to determine such a distribution

[3] n_1 corresponds to the number of elements, in the subset of the domain X_1, that we want to consider as a prior for our model.

the Bayes theorem must be used:

$$\begin{aligned} p(Y_2|Y_1,X_1,X_2) &= \frac{p(Y_2,\, Y_1,\, X_1,\, X_2)}{p(Y_1,\, X_1,\, X_2)} = \\ &= \frac{p(Y_2,Y_1|X_1,\, X_2)}{p(Y_1|X_1,\, X_2)} = \\ &= \frac{p(Y_2,Y_1|X_1,\, X_2)}{\int dY_2 p(Y_2,Y_1|X_1,\, X_2)} \end{aligned}$$

The probability distribution to be found has been written in terms of the known probability distribution function and, as a consequence, the task is to explicitly perform the computations. The first thing to be done is to work with the argument of the exponential:

$$\begin{aligned} (Y-\mu)^T A(Y-\mu) &= \begin{bmatrix}(Y_1-\mu_1)^T & (Y_2-\mu_2)^T\end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} Y_1-\mu_1 \\ Y_2-\mu_2 \end{bmatrix} = \\ &= \tilde{Y_1}^T A_{11}\tilde{Y_1} + \tilde{Y_1}^T A_{12}\tilde{Y_2} + \tilde{Y_2}^T A_{21}\tilde{Y_1} + \tilde{Y_2}^T A_{22}\tilde{Y_2} \end{aligned}$$

where the variable $\tilde{Y_i} \doteq Y_i - \mu_i$ has been introduced. In order to proceed it is necessary to express the Y_2 dependent part of this term as a quadratic form of Y_2, possibly introducing a term $\gamma(Y_1)$ to complete the square. First the general expression of the quadratic form, characterized by two parameters to be fixed $\tilde{\mu_2}(Y_1)$ and $\tilde{A}(Y_1)$, is introduced. Then by forcing it to be equal to the last expression, once completed the square, the explicit values of the parameters are inferred.

$$\begin{aligned} (\tilde{Y_2}-\tilde{\mu_2})^T\tilde{A}(\tilde{Y_2}-\tilde{\mu_2}) &= \tilde{Y_1}^T A_{12}\tilde{Y_2} + \tilde{Y_2}^T A_{21}\tilde{Y_1} + \tilde{Y_2}^T A_{22}\tilde{Y_2} + \gamma \\ \tilde{Y_2}^T\tilde{A}\tilde{Y_2} - \tilde{Y_2}^T\tilde{A}\tilde{\mu_2} - \tilde{\mu_2}^T\tilde{A}\tilde{Y_2} + \tilde{\mu_2}^T\tilde{A}\tilde{\mu_2} &= \tilde{Y_1}^T A_{12}\tilde{Y_2} + \tilde{Y_2}^T A_{21}\tilde{Y_1} + \tilde{Y_2}^T A_{22}\tilde{Y_2} + \gamma \end{aligned}$$

At this point it is possible to directly obtain that $\tilde{A} = A_{22}$ whereas $\tilde{\mu_2}$ is obtained by solving:

$$\begin{aligned} A_{22}\tilde{\mu_2} &= -A_{21}\tilde{Y_1} \\ \tilde{\mu_2} &= -A_{22}^{-1}A_{21}\tilde{Y_1} \end{aligned}$$

The knowledge of the parameters allows to obtain the value of γ by solving:

$$\gamma = \tilde{\mu_2}^T\tilde{A}\tilde{\mu_2} = \tilde{Y_1}^T A_{12}A_{22}^{-1}A_{21}\tilde{Y_1}$$

Finally, the desired distribution can be computed in terms of the known parameters and of the A matrix:

$$\begin{aligned} p(Y_2|Y_1,\, X_1,\, X_2) &= \frac{e^{-\frac{1}{2}\tilde{Y_1}A_{11}\tilde{Y_1}+\frac{1}{2}\gamma}e^{-\frac{1}{2}(\tilde{Y_2}-\tilde{\mu_2})^T A_{22}(\tilde{Y_2}-\tilde{\mu_2})}}{e^{-\frac{1}{2}\tilde{Y_1}A_{11}\tilde{Y_1}+\frac{1}{2}\gamma}\int d\tilde{Y_2}e^{-\frac{1}{2}(\tilde{Y_2}-\tilde{\mu_2})^T A_{22}(\tilde{Y_2}-\tilde{\mu_2})}} = \\ &= \sqrt{\frac{|A_{22}|}{2\pi}}e^{-\frac{1}{2}(\tilde{Y_2}-\tilde{\mu_2})^T A_{22}(\tilde{Y_2}-\tilde{\mu_2})} = \\ &= \sqrt{\frac{|A_{22}|}{2\pi}}e^{-\frac{1}{2}(Y_2-\mu_2-\tilde{\mu_2})^T A_{22}(Y_2-\mu_2-\tilde{\mu_2})} \end{aligned}$$

As expected the marginal distribution of a Gaussian is itself a Gaussian with the mean and the covariance matrix to be explicitly written in terms of the original covariance matrix. A_{22} corresponds indeed to $(\Sigma^{-1})_{22}$ and its connection to the sub matrices forming Σ is not trivial. In particular, the expression of the sub-matrices forming A in terms of the sub-matrices forming Σ can be obtained by imposing the product of the two matrices to be equal to the identity:

$$\begin{aligned} \mathbb{1} &= \Sigma\Sigma^{-1} = \Sigma A = \\ &= \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \\ &= \begin{bmatrix} \Sigma_{11}A_{11} + \Sigma_{12}A_{21} & \Sigma_{11}A_{12} + \Sigma_{12}A_{22} \\ \Sigma_{21}A_{11} + \Sigma_{22}A_{21} & \Sigma_{21}A_{21} + \Sigma_{22}A_{22} \end{bmatrix} \end{aligned}$$

This leads to the linear set of equations:

$$\begin{cases} \Sigma_{11}A_{11} + \Sigma_{12}A_{21} = 1 \\ \Sigma_{11}A_{12} + \Sigma_{12}A_{22} = 0 \\ \Sigma_{21}A_{11} + \Sigma_{22}A_{21} = 0 \\ \Sigma_{21}A_{21} + \Sigma_{22}A_{22} = 1 \end{cases} \tag{3.4}$$

By solving this system it is possible to obtain:

$$\begin{cases} A_{11} = \Sigma_{11}^{-1} - \Sigma_{21}^{-1}\Sigma_{22}\Sigma_{12}^{-1} \\ A_{21} = \Sigma_{12}^{-1} - \Sigma_{22}^{-1}\Sigma_{21}\Sigma_{11}^{-1} \\ A_{12} = \Sigma_{11}^{-1}\Sigma_{12}(\Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12} - \Sigma_{22}) \\ \boxed{A_{22}^{-1} = \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}} \end{cases} \tag{3.5}$$

The last term corresponds to the covariance matrix of the desired Gaussian distribution. The missing terms for completing its characterization is the mean:

$$\begin{aligned} \mu_{2|1} &= \mu_2 + \tilde{\mu_2} = \mu_2 - A_{22}^{-1}A_{21}\tilde{Y_1} = \\ &= \mu_2 - A_{22}^{-1}A_{21}(Y_1 - \mu_1) \end{aligned}$$

In order to evaluate the value of $A_{22}^{-1}A_{21}$ one can use the second equation of the system 3.4:

$$\begin{aligned} \Sigma_{11}A_{12} &= -\Sigma_{12}A_{22} \\ A_{12} &= -\Sigma_{11}^{-1}\Sigma_{12}A_{22} \\ A_{12}A_{22}^{-1} &= -\Sigma_{11}^{-1}\Sigma_{12} \\ (A_{12}A_{22}^{-1})^T &= -(\Sigma_{11}^{-1}\Sigma_{12})^T \\ A_{22}^{-1}A_{21} &= -(\Sigma_{11}^{-1}\Sigma_{12})^T \end{aligned}$$

That yields:

$$p(Y_2|Y_1, X_1, X_2) = \mathcal{N}\left(\mu_{2|1} = \mu_2 + (\Sigma_{11}^{-1}\Sigma_{12})^T(Y_1 - \mu_1);\ \Sigma_{2|1} = \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}\right) \tag{3.6}$$

where μ_1 and μ_2 correspond to the prior information on the mean value of the function in the considered points. In our case, they are both set to 0. On the other hand, all of the sub-matrices can be computed from the kernel as

$$\Sigma_{ij} = K(X_i, X_j)$$

The distribution is then fully determined and the Y_2 points can be sampled from it. This sampling procedure can be performed all at once or, as in the case of the Bayesian Optimization, it is possible to design an iterative procedure for progressively updating the knowledge on a desired function based on the previsions done with the knowledge at hand.
From a practical point of view, the product $\Sigma_{11}^{-1}\Sigma_{12}$ is the most computationally expensive part and the optimized way of implementing it, as it is done in the scikit-learn [53] implementation, is by means of the Cholesky decomposition ([54]) of the Σ_{11} sub-matrix. In fact, being it positive definite and symmetric, it can be written as $\Sigma_{11} = LL^T$ and the matrix $\chi = \Sigma_{11}^{-1}\Sigma_{12}$, corresponding to the solution of the linear problem $\Sigma_{11}\chi = \Sigma_{12}$ becomes $LL^T\chi = \Sigma_{12}$. L is a lower triangular matrix and the solution of a linear system having the matrix of the coefficients of this form is easily solvable. Therefore, one first finds the solution of the linear system

$$L\xi = \Sigma_{21}$$

and then the χ matrix is obtained by solving

$$L^T\chi = \xi$$

The main advantage of the Gaussian Process is that it can be used not only to learn a function that is well defined, as for example a sinusoidal function, but also to learn functions that are rather characterized by noise. This aspect of the GP is very useful in order to find the best number of filters in the first and second convolutional layers and the number of neurons in the dense layer, because the function that is used as a target in the BO, and so in the GP, can be the loss function or the AUC or the WP_99 that are not well defined functions of these parameters. In particular the GP parameter that plays a significant role in determining the presence of noise rather than a correlation is the covariance matrix and so the kernel. What can be done is to consider the kernel as a sum of two contribution, for example:

- The RBF kernel that will approximate the correlated part;
- The White kernel, that can be described as a diagonal matrix having the noise level along the diagonal, will take into consideration the uncorrelated part and so the noise that characterized the points.

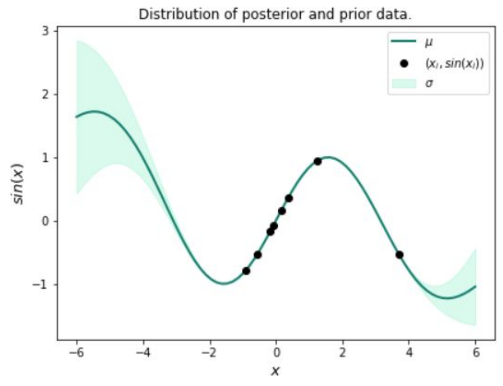
The RBF kernel is characterized by two parameters, the first one is the length scale, that correspond to the parameter ℓ in the previous formula (*Eq. 3.1*), and a coefficient (α) multiplying it that describes the importance, and so the weight, of the RBF term for the specific function it is trying to learn. For what concerns the white kernel, it can be described by one term, i.e. the noise level (β), that states how much the function is dominated by the noise. More generally the formula of the full kernel can be written as:

$$k(x_a, x_b) = \alpha e^{-\frac{|x_a - x_b|^2}{2\ell^2}} + \beta\delta(x_a - x_b)$$

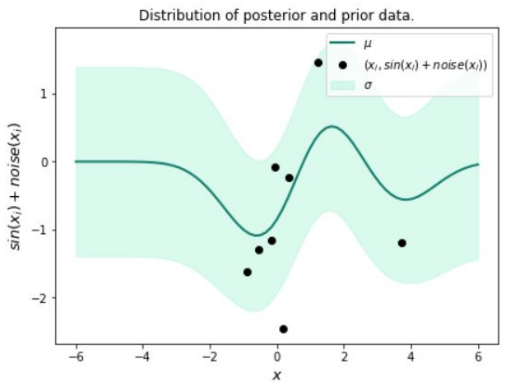
The mentioned parameters are optimized with a gradient descent in the scikit-learn implementation itself that selects which are the best ones to approximate the model.
In order to clarify this concept, it is possible to consider two simple examples where the GP, with the kernel equal to the previous formula, is used in order to approximate first a sinusoidal function, $sin(x)$, where the x values are sampled by a uniform distribution. Then a GP with the same characteristic as the first one is used to approximate a sinusoidal function where a random noise, sampled from a normal distribution with $\mu = 0$ and $\sigma = 1$, is summed for each value of x, namely $sin(x) + noise(x)$.
At this point the parameters of the kernel selected by using the gradient descent, after the fit of the GP with the prior data, are:

$$\alpha = 3.20, \;\; \ell = 2.52, \;\; \beta = 10^{-10} \quad for\, sin(x)$$
$$\alpha = 0.94, \;\; \ell = 1, \;\; \beta = 0.998 \quad for \;\; sin(x) + noise(x)$$

As it is possible to observe from this result, the value of the noise is very small in the first case and very high in the second case, meaning that in the second case the model contains a significant noise component, also if the contribution of the sine is still visible. The result of the GP with the two function is shown in *Fig. 3.2a* and in *Fig. 3.2b* respectively.



(a) *Prior and posterior distribution for the sinusoidal function. The GP is modelling the sinusoidal function because there is not noise.*



(b) *Prior and posterior distribution for the sinusoidal function with additional noise. The GP is modelling a significant noise component.*

Figure 3.2: Prior and posterior distribution obtained by using the GP for two functions. In black are represented the prior distributions of the known points, while in green and light green are represented, respectively, the mean and the standard deviation predicted by the GP for all the other values of the function at different values of x with respect to the prior points.

3.3 Practical hyper-parameters tuning

In this section I will focus on the use of two of the previously described hyper-parameters tuning techniques, the Grid Search (GS) and the Bayesian Optimization (BO), and on the comparison of their results. As already said, in order to perform the BO, it is necessary to find a target function that gives significant results for the neural network comparison, i.e. a function whose value changes with the number of filters and neurons and it is not constant, as well as the kernel

to determine the correlation between the points of this function.
It is worth mentioning that when the parameters of a model, in this case of the Bayesian optimization, need to be determined, it is important to define a procedure for this selection. As well as what we did for the hand-tuning optimization of the hyper-parameters of the neural network (in *Section 2.2.1*), there is not a unique way of performing such an optimization. In this case we decided to first use the default kernel employed in the implementation of the BO process available on scikit-learn ([53, 55]), i.e. the so called Matern(ν=2.5) kernel[4], in order to check which is the best target function. Independently from the type of kernel used, at the end of the process it is possible to obtain a rough estimate of the target function. This estimation is enough to understand which target function could be more significant for the optimization of the network with respect to the topology parameters. Once the target function will be fixed the kernel will be modified to observe if there are other kernel functions that give a better approximation in this specific case. Consequently, the first thing that will be considered is the target function that will be used with 3D BO in order to determine which is the best neural network in terms of performances. In order to select the proper target function, a first 1D BO and a 3D GS will be performed and then the results will be compared. The 1D BO will be performed considering a 1D topology space determined by the different values assumed by the number of filters in the first convolutional layer (sl1), while the 3D GS will be done considering different combinations of the number of filters in the first and second convolutional layers (sl1 and sl2) and of the number of neurons in the dense layer (sl3).
As already said, there are some parameters that are meaningful in particle physics like the AUC and the Background rejection at 99% or 95% of signal efficiency (WP_99 and WP_95 respectively). For this reason, all of them will be taken into consideration along with the validation loss at the last epoch, where, in the case of BO, the minus sign is added to the loss since the BO is implemented on scikit-learn to solve maximization problems. The selection of the target function is very important for the BO problem, either because it expects a function with a mean $\mu = 0$ and so the selection of the target will influence the performance, but also for the selection of the kernel that will describe the correlations between the points of that function. Once the target function will be defined, we will move towards the selection of the kernel, comparing different possible kernels and combinations between them.

3.3.1 Grid search and Bayesian optimization with loss as a target

The first function that can be used as a target to compare different models is the validation loss function $\mathscr{L}$. This target is used first in a GS optimization where each model is characterized by different values of *sl1*, *sl2*, *sl3* that are selected from the union of two different spaces: the first one corresponds to all the possible triplets that can be formed using $sl1 \in \{8,12,16,18,24,32,64\}$, $sl2 \in \{8,12,16,32\}$ and $sl3 \in \{8,12,16,24,32,64,128\}$ while the second one is generated in the same way, using $sl1 \in \{8,16,32\}$, $sl2 \in \{24,64,128\}$, $sl3 \in \{8,12,16,24,32,64,128\}$. After the GS the comparison of the results for different combination of sl1, sl2 and sl3 in terms of loss function is shown in *Fig. 3.3.*

[4]The Matern kernel is defined as: $\frac{1}{\Gamma(\nu)2^{\nu-1}}(\frac{\sqrt{2\nu}}{l}d(x_i,x_j))^{\nu}K_{\nu}(\frac{\sqrt{2\nu}}{l}(d(x_i,x_j)))$, where d is the Euclidean distance, ν allows to specify the smoothness of the function, K_ν is the modified Bessel function, Γ is the gamma function (see [49]).

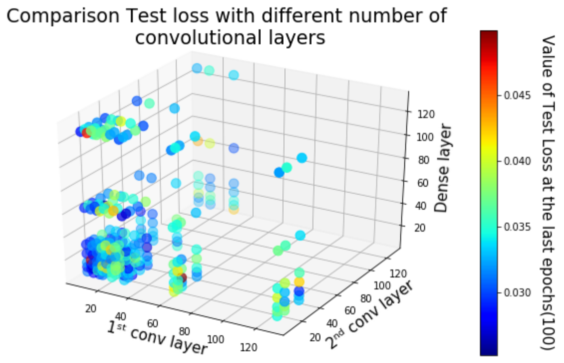


Figure 3.3: 3D representation of the results of the GS method for different numbers of filters in the 1st and 2nd convolutional layers and the number of neurons in the dense layer. The different colors represent the value of the loss function after the test of the model. The maximum in dark red, and the minimum in dark blue.

To better visualize the behaviour of the loss function for different values of *sl1*, *sl2*, *sl3*, it is possible to plot the value of the loss for each different combination of that parameters as a function, for example, of the number of filters in the 1st convolutional layer (see *Fig.3.4*).

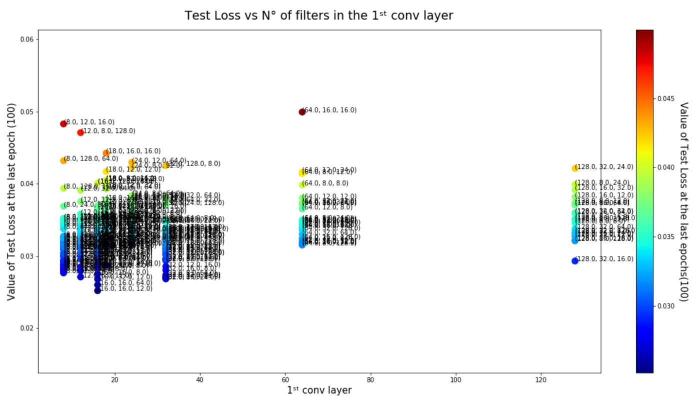
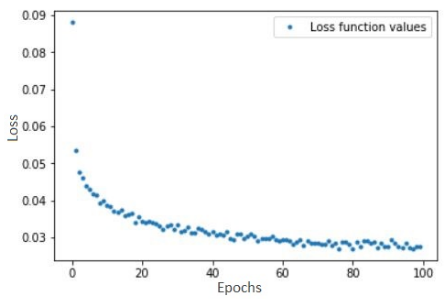
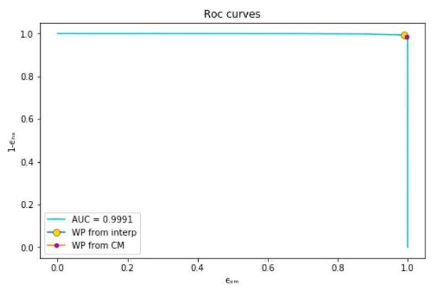


Figure 3.4: 2D plot of the loss as a function of the number of filters in the 1st convolutional layer.

Among all of these networks, the one minimizing the loss function is characterized by the hyperparameters *sl1* = 16, *sl2* = 16 and *sl3* = 12.
It is worth mentioning that the found minimum is not guaranteed to be always found as the absolute minimum. This is due to the presence of an intrinsic randomness of the loss. For this reason, the BO should be better since it is the only method described so far that is able to take into consideration this aspect. Before proceeding, it is important to check the goodness of the performances of the model by evaluating the loss as a function of the number of epochs, the confusion matrix and the ROC curve together with its useful parameters (AUC, WP). The results of the loss function and the ROC curve are shown in *Fig. 3.5a* and *Fig. 3.5b* respectively.



(a) *Loss function vs. the number of epochs.*



(b) *ROC curve. The yellow WP is evaluated by performing the interpolation of the ROC curve (expressed in the form 1-fp_r vs. tp_r) at the abscissa $tp_r = 0.99$, while the magenta WP is evaluated from the confusion matrix that contains the information about fp_r by performing 1-fp_r.*

Figure 3.5: Loss function and ROC curve of the model that gave the minimum value of the validation loss.

These results show that the final value of the loss at the end of the test of the model is **0.0252**, while the WP is (0.99, 0.9920) and the AUC is 0.9991.
Finally the model is characterized by the following CM and CM$_r$:

$$CM = \begin{bmatrix} 5424 & 82 \\ 25 & 10709 \end{bmatrix} \qquad CM_r = \begin{bmatrix} 0.9851 & 0.0149 \\ 0.0023 & 0.9977 \end{bmatrix} \tag{3.7}$$

Checking the value along the diagonal of the CM_r it is possible to understand that the model is able to recognize with a very high accuracy the EM particles (element on the second row and second column of the matrix) and the HAD particles (element on the first row and the first column of the matrix). The global accuracy of this model is 0.99.
Now it is possible to perform the Bayesian Optimization[5] using as a target $-\mathscr{L}$, since its maximization corresponds to the minimization of the loss function. In order to understand the behaviour of the BO it is better to reduce the parameter space from 3 to 1 in order to have a 2D representation of the BO. For this purpose two parameters need to be set, for example: *sl2* = 12 and *sl3* = 16. It means that, once fixed the hyper-parameter of the BO (k=2), only *sl1* is a parameter to be optimized, and the one-dimensional space in which the BO method can choose *sl1* is between 1 and 128. After twelve iterations the result of the Bayesian optimization, shown in *Fig. 3.6*, shows that the target function improves up to 25 filters and once the number of filters becomes higher than that, it starts to become worse (higher in absolute value). This could be very interesting for our purpose because the implementation of the NN on FPGA requires a number of layers, filters and neurons that should be as small as possible.

[5]All the Bayesian Optimization are performed using the algorithm implemented by the scikit-learn library [53].

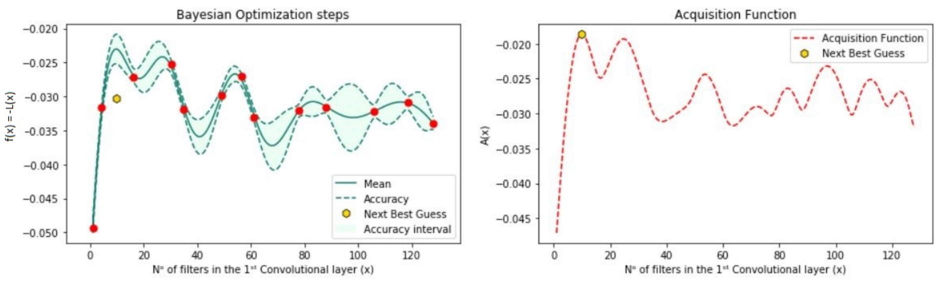


Figure 3.6: Results of Bayesian Optimization using Matern($\nu = 2.5$), for a space between 1 and 128, after 12 iterations. The y axis is negative because the BO has only the max function implemented and so to have the minimum value the loss is multiplied by a minus sign.

In order to confirm the obtained result it is possible to increase the one-dimensional space in which the BO can select the *sl1* parameter from 128 to 256, it means that the new space is between 1 and 256. As it is possible to see from *Fig. 3.7* , the loss becomes worse when the number of filters in the first convolutional layer increases, validating the previous result. These results suggest to consider a small number of filters for the first convolutional layer rather than an high one.

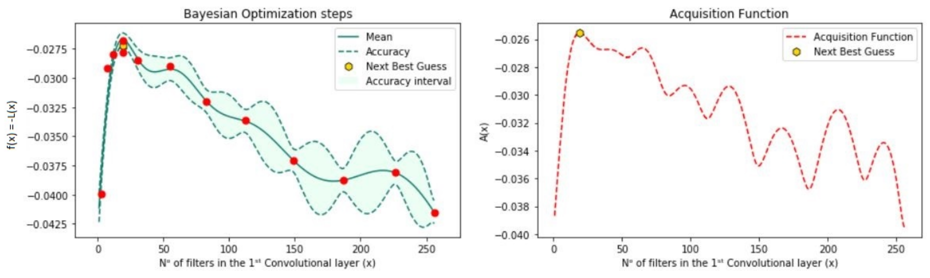


Figure 3.7: Results of Bayesian Optimization using Matern($\nu = 2.5$), for a space between 1 and 256, after 12 iterations. The y axis is negative because the BO has only the max function implemented and so to have the minimum value the loss is multiplied by a minus sign.

In order to compare the results of the BO with the 3D grid search it is possible to consider only the points of the 3D GS that are characterized by sl2 equals to 12 and sl3 to 16 (to have the same values used in the BO) and to observe the corresponding values of the validation loss (see *Fig. 3.8*). As it is possible to see from *Fig. 3.7*, the behaviour of the BO is very similar to what is expected from the grid search (see *Fig. 3.8*).
With the GS method it is also possible to see, from *Fig. 3.4*, that for a small number of filters in the first convolutional layer there are very bad and very good points including the point at which the loss is minimal, that is more isolated from the others due to the very small value of the loss function. Considering the BO, it is possible to extract similar information but with a very small number of iterations, due to the fact that the BO does not explore all the points in the space, but uses the information previously evaluated to select the next best guess and reach at the end a very similar result, in terms of optimized target ($\mathscr{L}$ in this case).
As a conclusion, it is possible that the two methods will find a different best result, because of the different selection of the hyper-parameters from the space, where the differences in

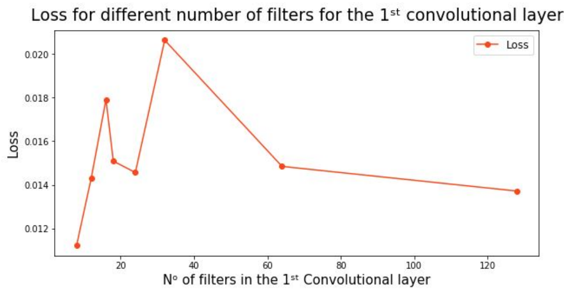


Figure 3.8: Representation of the loss, evaluated with the GS method, as a function of different numbers of filters in the 1^{st} convolutional layer. For this plot only the evaluated values of the loss for the models that have sl2=12, sl3=16 are considered.

terms of optimized target (loss) are very small. In this specific example, the best value of the $\mathscr{L}$ during the test, according to the Bayesian optimization, is obtained with *sl1* = 19, and the corresponding value of the loss is **0.0268**. As it is possible to see, the results in terms of loss, for the two methods, are very similar and the difference is of the order of 10^{-3}. Conversely, the combinations of the number of filters and neurons for the best point are different, (*sl1* = 16, *sl2* = 16, *sl3* = 12) with GS and (*sl1* = 19, *sl2* = 12, *sl3* = 16) with the BO, also due to the restriction applied in the Bayesian optimization (*sl2* = 12 and *sl3* = 16). As a consequence, it is possible to use the Bayesian optimization, instead of the GS, in order to reduce the time of computation but reaching a model that will be characterized by a value of the loss that is almost the same as the one that we can find with the GS in more time. Once these considerations are done, it is possible to move towards different types of target such as AUC, WP_99 and WP_95 in order to see which is the most suitable one for this classification problem.

3.3.2 Grid search and Bayesian optimization with AUC as a Target

The same considerations done for the GS and BO with minus the loss $-\mathscr{L}$ as a target can be done by considering the AUC as a target. Considering the same models as those trained for the GS with the loss as a target, it is possible to select which of them reaches the highest value of AUC, i.e. the best model. The results of the GS obtained with AUC as a target are shown in *Fig. 3.9.*
In this case the maximum value of AUC is shared by two different architectures:

- *sl1*=12, *sl2*=16, *sl3*=8;
- *sl1*=12, *sl2*=32, *sl3*=24.

and it is equal to **0.9992**. In this case, in order to implement the BO with a target equal to the AUC it is necessary to change the mean and the spread of the AUC to find a proper function for the GP. This is due to the fact that the value of AUC is always near 1 and the Gaussian process requires a function with a mean almost near 0 and a spread of 1 to properly work.
For this reason, in order to find this function it is possible to extract approximate rescaling factors by observing the results obtained with the GS, focusing only on the models that have *sl2*=12, *sl3*=16, just to have the same models used in the BO with the loss function as a target

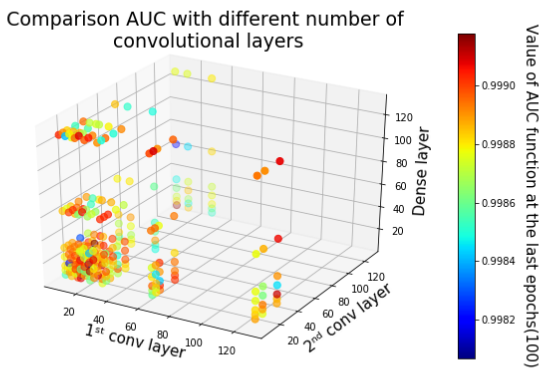


Figure 3.9: 3D representation of the results of the GS method for different number of filters in the 1st and 2nd convolutional layers and of neurons in the dense layers. The different colors represent the value of the AUC. The maximum in dark red and the minimum in dark blue.

and see which model is the best one in terms of AUC. Given the aforementioned models, the behaviour of the AUC as a function of the first convolutional layer (*Fig. 3.10*) allows to find a good mean and a good spread (standard deviation) that will be used to adjust the AUC for the BO.

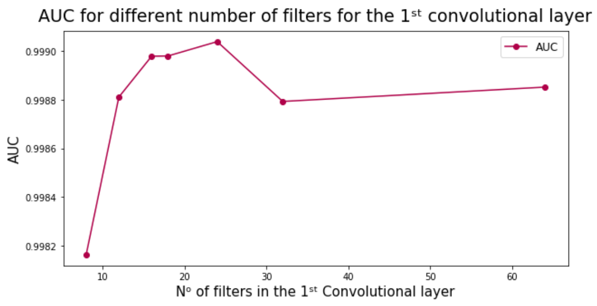


Figure 3.10: Representation of the AUC, evaluated with the GS method, as a function of different numbers of filters in the 1st convolutional layer. For this plot only the evaluated values of AUC for the models that have *sl2*=12, *sl3*=16 are considered.

The value of the mean is 0.9988 while the value of the spread is 0.00028, and so the AUC in the BO is renormalized in the following way:

$$AUC_{corr} = \frac{AUC - 0.999}{8 \cdot 10^{-4}}$$

In words, the AUC is shifted by the mean and rescaled by a factor obtained by multiplying the approximate spread of the AUC by 3. In fact, the Gaussian process, at the heart of the BO, is based on the choice of the next guess in within 3 times the standard deviation from the average value.

Considering now AUC_{corr} as the target for the BO it is possible to analyse its behaviour as a function of the number of filters in the 1st convolutional layer (*sl1*) by fixing *sl2*=12 and *sl3*=16. Also in this case it is possible to start by observing the evolution of the BO when the hyper-parameter k is equal to 2 and the space in which it selects *sl1* is between 1 and 128 (see *Fig. 3.11*).

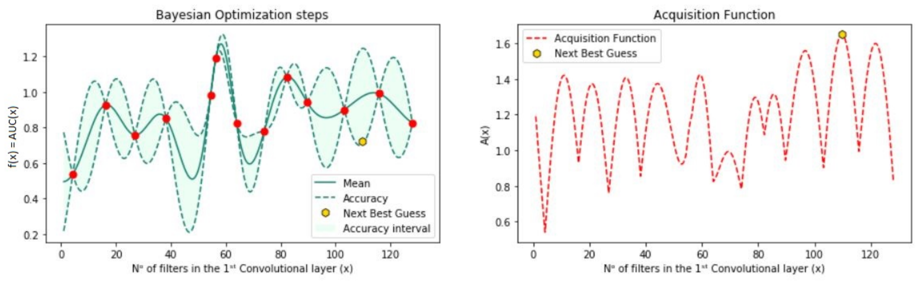


Figure 3.11: Results of Bayesian optimization using Matern($\nu = 2.5$) and k=2, for a space between 1 and 128, after 12 iterations.

As it is possible to see from this picture, there are a lot of fluctuations for different values of *sl1* and there is not a well defined trend of AUC when the number of filters in the 1st convolutional layer increases. In this case, it is possible to increase the *sl1* space in order to see if for very large values of *sl1* there is a significant variation of AUC. For this reason it is possible to perform the BO always keeping the hyper-parameter k equal to 2, but selecting the space where *sl1* is chosen in between 1 and 256 (see *Fig. 3.12*).

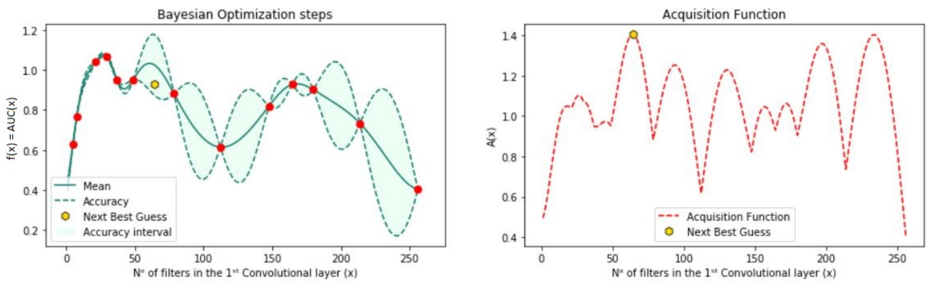
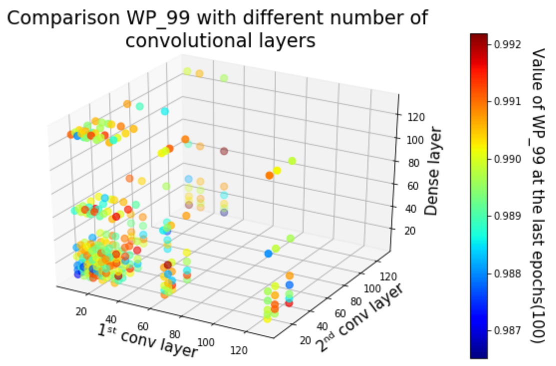


Figure 3.12: Results of Bayesian optimization using Matern($\nu = 2.5$) and k=2, for a space between 1 and 256, after 12 iterations.

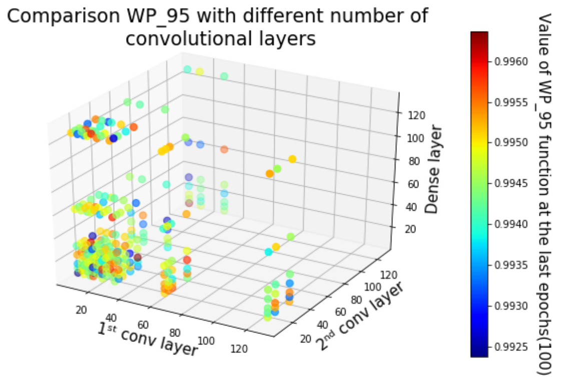
As it is possible to observe from *Fig. 3.12*, after 170 filters in the 1st convolutional layer the value of AUC starts to become smaller and the best value of AUC is obtained for *sl1*=30 and the corresponding value of AUC is **0.9991**. Also in this case it is difficult to directly compare the results from GS and those from BO, because the best point found with GS has *sl2* = 16 or 32 and, respectively, *sl3*=8 or 24 cannot be found by means of the BO when *sl2* is fixed to 12 and *sl3* is fixed to 16. Instead, it is possible to compare only a specific subset of the results of GS, those shown in *Fig. 3.10*, with the behaviour seen with BO, in *Fig. 3.12*, in a restricted space between 1 and 60. As it is possible to see from the two figures, it seems that around *sl1* $\simeq$ 25 there is the maximum value of AUC and then the characteristic presents a local minimum. In conclusion it seems that also comparing the AUC value allows to prove the effectiveness of GS and BO methods and the coherence of their results. Now that we also observed the behaviour of the BO when using AUC as target, we can move to the last two possible targets, such as WP_99 and WP_95.

3.3.3 Grid search and Bayesian optimization with WP_99 or WP_95 as a target

It is also possible to consider, instead of the $\mathscr{L}$ and the AUC, the Background rejection at 99% or 95% of signal efficiency as a target for the GS and BO. Starting with the GS and still considering the already trained models used for the $\mathscr{L}$ and for the AUC, it is possible to represent the results of the GS in terms of WP_99 (see *Fig. 3.13a*) and in terms of WP_95 (see *Fig. 3.13b*).



(a) *3D representation of the results of the grid search methods for WP_99.*



(b) *3D representation of the results of the grid search methods for WP_95*

Figure 3.13: 3D representation of the results, in terms of WP_99 and of WP_95, of the GS method for different number of filters in the 1st and 2nd convolutional layers and of neurons in the dense layer. The different colors represent the value of the WP_99, in the figure on the top and of WP_95 in the figure on the bottom. The maximum in dark red and the minimum in dark blue.

When considering WP_99 as a target, different models, according to the GS, give the same best value and they are characterized by:

- *sl1* = 8, *sl2*= 128, *sl3* = 8;

- *sl1* = 32, *sl2* = 128, *sl3* = 64;
- *sl1* = 12, *sl2* = 16, *sl3* = 8.

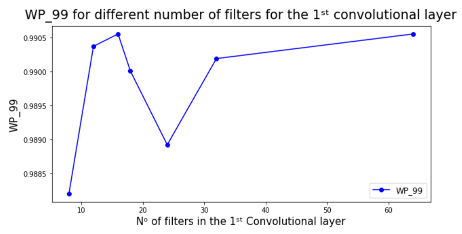
and the corresponding value of WP_99 is **0.9922**.
While for WP_95 the best model is characterized by: *sl1* = 32, *sl2* = 32, *sl3* = 16 and the corresponding value of WP_95 is **0.9964**.
In order to perform the BO, even in this case it is necessary to adjust the target by using a certain mean and a certain spread. In order to evaluate these parameters, it is possible to use the results of the GS, but selecting only the models that are characterized by *sl2*=12 and *sl3*=16 in order to keep the same network configuration used for the BO with $\mathscr{L}$ and AUC as target. In this way, the value of WP_99 as a function of the number of filters in the 1^{st} convolutional layer (see *Fig. 3.14a*) and the value of WP_95 as a function of the number of filters in the 1^{st} convolutional layer (see *Fig. 3.14b*) allow to evaluate proper approximate means and standard deviations for the BO models that are:

- for WP_99 the approximate mean is 0.9898 and the approximate standard deviation is 0.00085. And the redefined target for the BO is

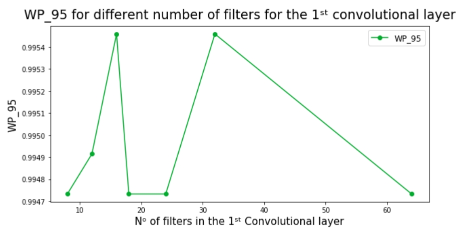
$$WP\_99_{red} = \frac{WP\_99 - 0.9898}{2.55 \cdot 10^{-3}}$$

- for WP_95 the approximate mean is 0.9950 and the approximate standard deviation is 0.00032. And the correct target for the BO is

$$WP\_95_{corr} = \frac{WP\_95 - 0.9950}{9.60 \cdot 10^{-4}}$$



(a) *WP_99 vs. the number of filters in the* 1^{st} *convolutional layer.*



(b) *WP_95 vs. the number of filters in the* 1^{st} *convolutional layer.*

Figure 3.14: Representation of the WP_99 and WP_95 as a function of the number of filters in the 1^{st} convolutional layer. For this plot only the evaluated values of WP_99 and WP_95 for the models that have *sl2*=12, *sl3*=16 are considered.

Now, considering WP_99 as the target, the hyper-parameter k equal to 2, *sl2*=12, *sl3*=16 and *sl1* in a range between 1 and 128, the results of the BO are shown in *Fig. 3.15*.

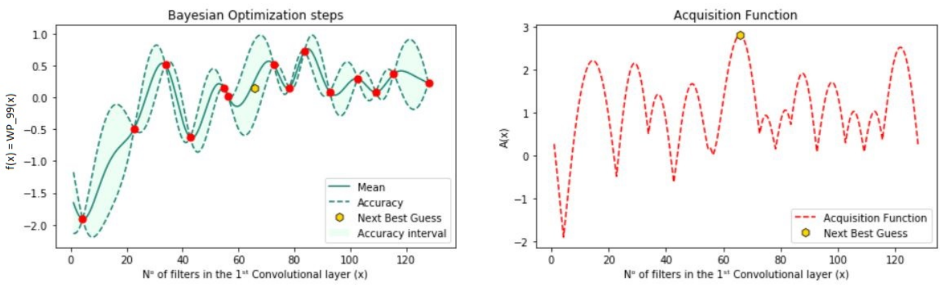


Figure 3.15: Results of Bayesian optimization using Matern($\nu = 2.5$), for a space between 1 and 128, after 13 iterations.

In this result, it seems that the value of WP_99 becomes better as the number of filters in the first convolutional layer increases, even if there are some oscillations. It is also possible to increase the space where the value of *sl1* is chosen from 128 to 256 to see if WP_99 will continue to keep the same behaviour. For this reason, retaining the same value of k as before and considering the space in between 1 and 256 a new BO is performed and the result is shown in *Fig. 3.16*.

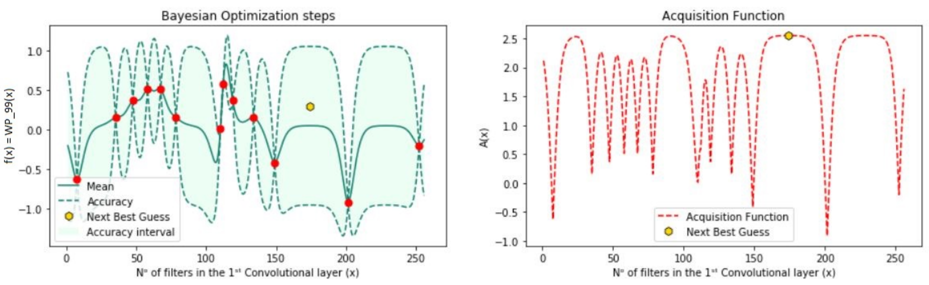


Figure 3.16: Results of Bayesian optimization using Matern($\nu = 2.5$), for a space between 1 and 256, after 12 iterations.

In this case it seems that the result of the BO is not very significant because there are a lot of fluctuations of the parameter WP_99. The best value of WP_99 is found for *sl1* = 112 and the corresponding value of WP_99 is **0.9913**.
Instead, considering WP_95 as the target, the hyper-parameter k equal to 2, *sl2*=12, *sl3*=16 and *sl1* in a range between 1 and 256, the results of the BO are shown in *Fig.3.17*.

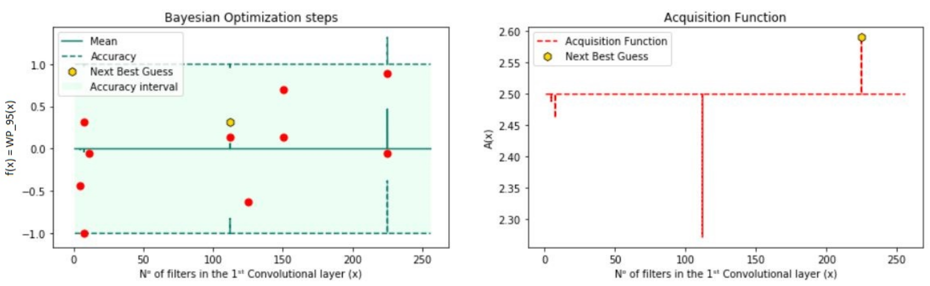


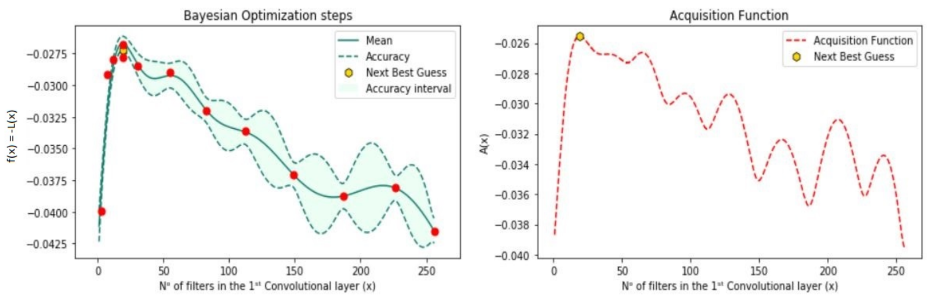
Figure 3.17: Results of Bayesian Optimization using Matern($\nu = 2.5$), for a space between 1 and 256, after 12 iterations.

Observing the behaviour of the WP_95 shown in *Fig. 3.17*, it is possible to observe that the BO is not very useful if the target is WP_95, since it seems that all the points are interpreted as noise and that there is not a dependence on the number of filters that can allow to understand which combination is the best one.

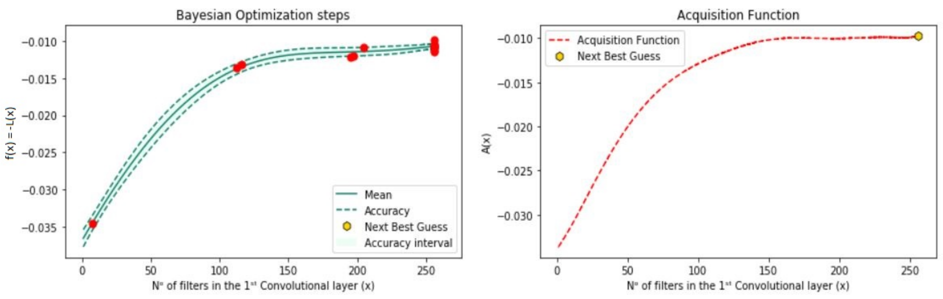
After having observed the results for different targets, such as: the validation loss, the AUC, the WP_99 and the WP_95, what it is possible to say is that when using the WP_99 or WP_95 or the AUC as a target, it is not simple to identify a characteristic behaviour of these functions because they seem to not have a meaningful variation when the number of filters or neurons change and, as a consequence, the best target function that can be used in the BO, with reference to this specific classification problem, is the *validation loss*. This consideration can be done because the loss as a function of the number of filters and neurons presents a non-steady behaviour. In particular, it starts to become worse (i.e. to decrease in absolute value) when the number of filters becomes too big. This is the reason why we decided to select the validation loss as the target function and also to reduce the target space in a range between 1 and 128 (instead of considering a range between 1 and 256) for the next steps. Finally, we also decided to fix the value of the hyper-parameter of the BO, k, to 4 in order to increase the exploration of the space. Now that the target function, the target space and k are selected they will be fixed and it will be possible to explore different kernel functions in order to see if better approximations of the selected target function can be reached by considering kernels different from the Matern (ν=2.5) kernel used until now.

3.3.4 Selection of the number of epochs

Before going ahead with the selection of the kernel for the BO, it can be useful to introduce a criterion for the selection of the number of epochs during the training in order to avoid the over-fitting (also called over-training) of the network, that we observed during the comparison of the validation loss and the train loss in the selection of the target function. The over-training problem can be visualized by comparing the training loss and the validation loss, as it is done in *Fig. 3.18*, where it is possible to observe the behavior of the training and validation loss as a function of the number of filters in the first convolutional layer. In particular what can be observed is that the training loss and the validation loss start having an opposite behaviour after a certain point and the value of the validation loss becomes higher than the value of the train loss. This is unexpected in a good model because when the validation loss is much higher than the training loss, it is possible that one of the two loss functions is increasing instead of decreasing. It means that the network is not modeling the data in a correct way and that the network gives randomly corrected outputs during the training which will not be correct during the validation. This suggests that for a high number of filters (high *sl1*) the networks are characterized by an over-training.
In order to better visualize this problem, it is useful to compare two models obtained in the BO with the validation loss as a target that behaves in an opposite way. The two selected models are: the one that gives the minimum validation loss that is characterized by *sl1*=19, *sl2*=12, *sl3*=16 (Best model) and one of the model that gives one of the highest values of the validation loss (Worst model), for example, *sl1*=256, *sl2*=12, *sl3*=16.
For both the models the training and validation loss as a function of the number of epochs are represented in *Fig. 3.19*.

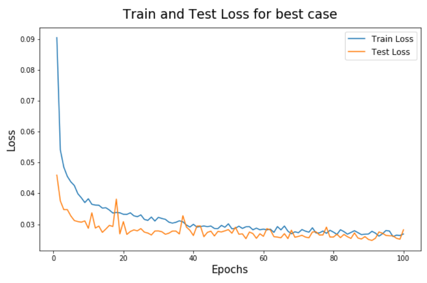


(a) *Result of BO with Test Loss as a target.*

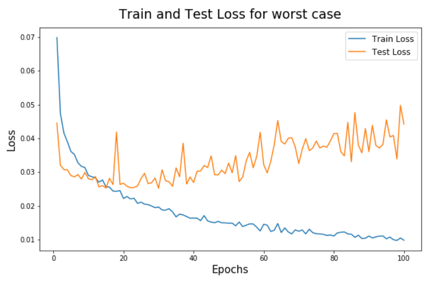


(b) *Result of BO with Train Loss as a target.*

Figure 3.18: Comparison results BO with Test and Train loss as a target.



(a) *Loss of the best model.*



(b) *Loss of the worst model.*

Figure 3.19: Comparison of the loss as a function of epochs for the best and worst model found with BO.

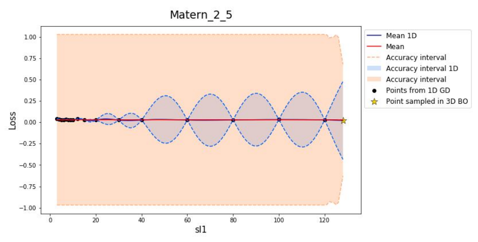
This result shows that in the worst model the validation loss, after almost 20 epochs, starts increasing instead of decreasing and becomes higher than the train loss. This means that there is an over-fitting and it could be the reason why at large value of *sl1* the two BO results show an opposite behaviour when using validation and training loss as target. On the contrary, in the best model it seems that there is still a small over-fitting, but after almost 90 epochs. It is perhaps better to reduce the number of epochs in the training process from 100 to 20 to avoid the over-fitting in both cases. This stopping criterion will be used in *Section 3.3.5*, where we will still be doing a preliminary analysis for the selection of the kernel that allows the best approximation of the target function in the BO, with the aim of avoiding the over-fitting.
In a second moment, during the real evaluation of the best combination of sl1, sl2 and sl3 in

Section 3.4, in order to finalize the model and to obtain a more general stopping criterion, an improved and more adaptable criterion will be considered. Instead of stopping the training for each model at 20 epochs, the past history of the validation loss across the epochs is taken into consideration in order to determine whenever it starts to increase instead of decreasing. Moreover, the presence of several fluctuations can influence the stopping criterion. For this reason, instead of directly checking the value of the validation loss at each epoch, an average of the validation loss across the previous $\mathtt{n} = 5$ epochs is evaluated at each new epoch and the training is stopped whenever this average is higher than that evaluated at the previous epochs for at least $\mathtt{k} = 3$ consecutive times.

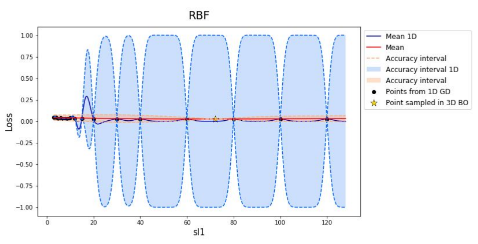
3.3.5 Kernel selection

Once the target function is determined and a stopping criterion is selected for the training process, it is possible to focus on the selection of the kernel that can be used as a covariance matrix in the optimization of the hyper-parameters of the CNN, such as the number of filters and neurons. As already mentioned, the selection of the kernel is of fundamental importance because it describes the correlation between the points of the target function in the optimization process and it needs to be selected at the beginning of the optimization by hand.
In order to do that, different possible kernel functions and their combinations will be considered: Matern(ν=2.5), RBF, k·RBF + W, RBF +k·W, Matern(ν=2.5) + Matern(ν=1), Matern(ν=2.5) + k·Matern(ν=1), k·RBF·k·RBF + W, k·RBF +k·RBF, k·RBF + k·Matern(ν=2.5), Matern(ν=2.5) + W, Matern(ν=2.5) + Matern(ν=1) + W, Matern(ν=2.5) +k· Matern(ν=1) + W, k·RBF + k·RBF+ W; where k is the weight associated to the considered kernel and it corresponds to the parameter of the kernel that is optimized in the scikit-learn implementation of the GP; the RBF and the White kernel (W) are described in *Section 3.2*; and the Matern kernel is a generalization of the RBF kernel, [49], where it is possible to specify the smoothness of the function (ν=1 means once differentiable, ν=2.5 twice differentiable, ν=0.5 the kernel is identical to the exponential kernel and ν=∞ the kernel corresponds to the RBF).
In particular, the first thing to consider is a 3D BO for each kernel, where the variables to be determined are always sl1, sl2 and sl3, in order to have a prediction of the mean and the covariance in the 3D space for each possible kernel.
Then, due to the fact that the BO generates an estimate of the target function, in order to understand the goodness of the model it is necessary to observe how close is the selected kernel to the descriptions of the real correlations. Therefore, each model approximation needs to be compared with something that is known to be a good result. In order to do that, it is possible to perform, for each kernel, a GS along sl1 by fixing sl2 and sl3 to the values corresponding to the best point obtained in the BO with the same kernel. In fact, such a grid search can be used to obtain a more detailed estimation of the target function along with the considered slice with which the BO estimation can be checked against. Each GS is performed for 17 different values of sl1, and in particular the selection of these values depends on the number of parameters of the network. It was reasonable to take more values closer together when the number of filters is small and then take larger range: 3,4,5,6,7,8,9,10,12,15,20,30,40,60,80,100,120. This is done because, when comparing neural networks, it is expected to have more variation in terms of performances when the parameters of these networks change more. This appears for example when the number of filters is doubled rather than changed of a small amount, i.e. it is expected to have more variation in terms of performances when two networks are characterized by 3 and

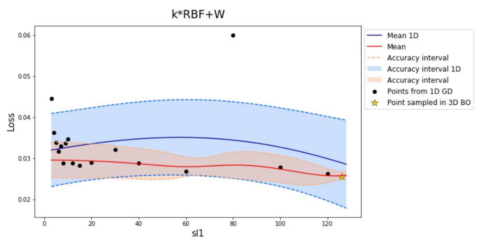
6 filters rather than by 30 and 40. This is because any ANN is a parametric method and the way it learns a function strongly depends on the number of parameters in it. Any parameter can be thought of as a feature, that the network can extract from the data and used to classify the image under analysis. Consequently, changing the number of features extractable by doubling them is more informative than changing them by a small amount each time.
In this way, there will be a reasonable number of points after each GS where it is possible to perform the GP to determine either the empirical mean and the covariance. In this way, there will be something that can be compared with the corresponding estimates obtained via the 3D BO once it will be projected in 1D by taking one slice along sl2 and sl3 equals to those for which the best point in the 3D BO is obtained. This means that the sl2 and sl3 of either the GS and the 1D slice of the corresponding 3D BO are equal and a significant comparison can be made, see *Fig. 3.20* and *Fig. 3.21.*
What is expected from these figures is that a good kernel will allow to obtain the predictions made by the 3D BO, when considering one slice along sl2 and sl3, that are similar to the predictions made on the GS points by the GP. As a consequence, it is possible to consider as good a kernel that allows to have a similar behaviour of the two curves, the orange one and the blue one with the corresponding accuracy intervals. Nonetheless, there is not a unique way to quantify how similar two curves are and, in the following, two criteria will be considered. Firstly, it will just be analyzed the distance in between the two average values. Then, the information on the accuracy interval will be taken under consideration and a number will be associated to the overlap of the two accuracy intervals to define the distance existing between the two approximations. These are two ways of defining a distance between the approximations. In any case, the smaller is the distance, the better will be the kernel used for the approximation. What it is possible to intuitively conclude from these plots is that the best results are obtained with the weighted RBF kernel plus the white kernel, by the product of the weighted RBF plus the white kernel or by the sum of the weighted RBFs plus the white kernel (*Fig. 3.20c*, *Fig. 3.21a* and *Fig. 3.21g* respectively), where the product of the weighted RBF is strictly related to the power of two of the weighted RBF. This can be intuitively said because these three kernels are those for which there is a more significant overlap between the expected result from the GS (blue curves and intervals) and the corresponding estimation made by the BO (orange curves and intervals). In addition, these plots are not characterized by mean curves that exactly follow the points. The latter condition would mean that they would be modelling the noise and not the correct relations between the points, as in the case of *Fig. 3.21e.* Moreover, in order to have a quantitative and more precise comparison between these kernels it is possible to perform a second analysis.



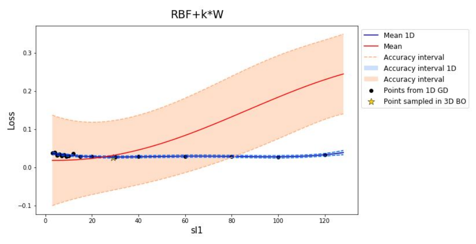
(a) *Predictions when using the Matern kernel with $\nu = 2.5$*



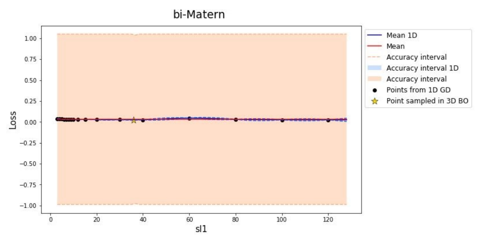
(b) *Predictions when using the RBF kernel.*



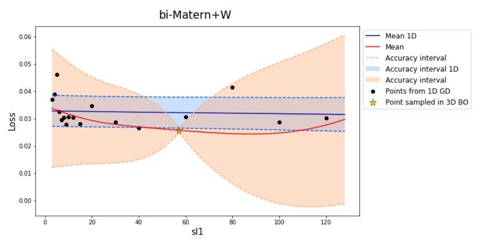
(c) *Predictions when using the weighted RBF kernel plus the White kernel.*



(d) *Predictions when using the RBF kernel plus the weighted white kernel.*

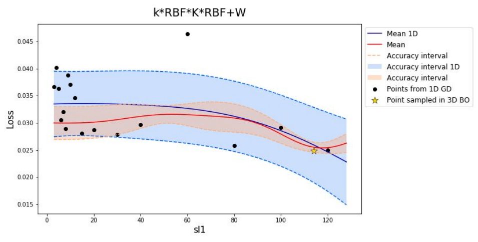


(e) *Predictions when using the Matern kernel with $\nu = 2.5$ plus Matern kernel with $\nu = 1$.*

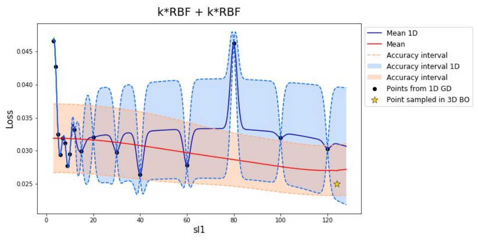


(f) *Predictions when using the Matern kernel with $\nu = 2.5$ plus the Matern kernel with $\nu = 1$ plus the white kernel.*

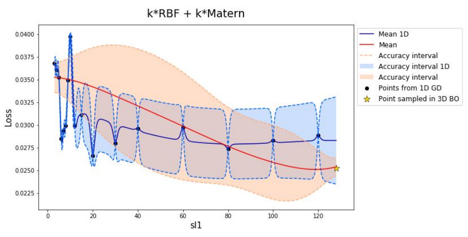
Figure 3.20: Representation of the prediction of the validation loss for different number of filters in the first convolutional layer. In red and orange are represented the predictions made by the 3D BO when considering one slice along sl2 and sl3 corresponding to the values for which the best point of the 3D BO is obtained. The best point of the 3D BO is represented by the yellow star. In black are represented the points evaluated by the 1D GS and in blue and light blue are represented the prediction made by the GP performed on the black points.



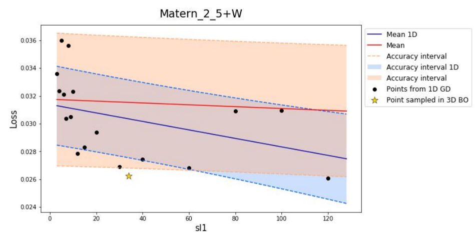
(a) *Predictions when using the weighted RBF kernel times the weighted RBF kernel plus the white kernel.*



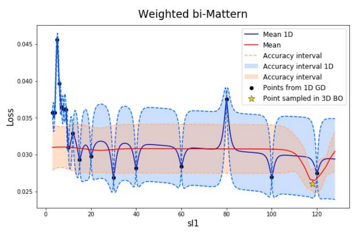
(b) *Predictions when using the weighted RBF kernel plus the weighted RBF kernel.*



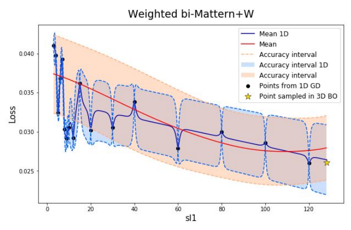
(c) *Predictions when using the weighted RBF kernel plus the weighted Matern kernel with* $\nu = 2.5$.



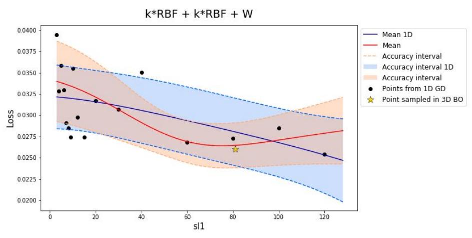
(d) *Predictions when using the Matern kernel with* $\nu = 2.5$ *plus the white kernel.*



(e) *Predictions when using the Matern kernel with* $\nu = 2.5$ *plus the weighted Matern kernel with* $\nu = 1$.



(f) *Predictions when using the Matern kernel with* $\nu = 2.5$ *plus the weighted Matern kernel with* $\nu = 1$ *plus the white kernel.*



(g) *Predictions when using the weighted RBF kernel + the weighted RBF kernel + the white kernel.*

Figure 3.21: Representation of the prediction of the validation loss for different number of filters in the first convolutional layer. In red and orange are represented the predictions made by the 3D BO when considering one slice along sl2 and sl3 corresponding to the values for which the best point of the 3D BO is obtained. The best point of the 3D BO is represented by the yellow star. In black are represented the points evaluated by the 1D GS and in blue and light blue are represented the prediction made by the GP performed on the black points.

In this case we run a 3D GS for 6 different values of sl1, sl2 and sl3 (3, 6, 12, 42, 75, 100), to generate in total 216 points in the 3D space, then this process is repeated 10 times in order to have for each of the 216 points a mean and a standard deviation. In this way it is possible to generate a target through which comparing the results of the already performed 3D BO. More specifically, a new matrix that contains all the possible combinations of sl1, sl2 and sl3 (with a final size equal to 216×3, where 216 corresponds to the total number of combinations of sl1, sl2 and sl3 considered and 3 corresponds to the total number of coordinates in the space: sl1, sl2 and sl3) is generated and it will be used as input of the already fitted 3D Gaussian process during the 3D BO. In so doing, for every 3D BO with one of the selected kernel, the associated 3D GP that is fitted during the optimization is used in order to make the predictions for each combination of sl1, sl2 and sl3 of the matrix. These predictions can be compared with the generated target.
This comparison can be done in different ways. The simplest idea can be to evaluate, for each 3D GP, and so for each BO model, and for each combination of sl1,sl2 and sl3, a difference in terms of a mean and also a difference in terms of a standard deviation. The difference in terms of a mean is the difference between the mean given by the target (μ_t) and the mean obtained after the predictions made with the kernel fitted during the specific 3D BO ($\mu_{\beta o}$): $|\mu_t - \mu_{\beta o}|$. Instead, considering the standard deviation, it is the difference between the standard deviation given by the target (σ_t) and the standard deviation evaluated by making the predictions with the fitted kernel ($\sigma_{\beta o}$): $|\sigma_t - \sigma_{\beta o}|$. These operations allow to generate, for each kernel, two vectors of size 216 containing, for each combination of sl1, sl2 and sl3, the difference between the mean of the target and of the 3D GP and the difference between the standard deviation of the target and of the 3D GP. Once created these two vectors the simplest way to compare all the 3D BO models is to evaluate the norm 2:

$$\sqrt{\sum_{i=1}^{N} |a_i|^2}$$

where a corresponds to one of the vectors containing the difference in terms of mean or in terms of standard deviation for each combination of sl1, sl2 and sl3.
At the end of this comparison we will have two numbers associated to each BO model with different kernels that tell us the goodness of the model. It is expected to have a small value of the norm 2 both when a is the vector associated to the differences of the mean and when a is the vector associated to the differences of the standard deviation. The results are shown in *Table 3.1* and it is possible to observe that one of the best kernel when comparing the mean is **k·RBF + W** while comparing the standard deviation is **k·RBF + k· Matern**, however it is worth noting that in terms of standard deviation the difference between the k·RBF + k·Matern and the k·RBF + W is not very significant and it is of the order of 10^{-3} while in terms of mean it is of the order of 10^{-1} . This suggests that both these kernels could be used, and k·RBF + W seems to have low differences for what concerns the mean.

Table 3.1: Comparison between the 3D BO using different kernels in terms of mean and standard deviation (std). Matern kernel is indicated with M

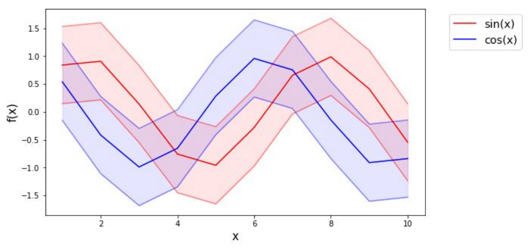
Kernel	mean	std
M($\nu = 2.5$)	0.985	14.608
RBF	1.038	0.494
k·RBF+W	0.969	0.256
RBF+k·W	1.880	1.562
M($\nu = 2.5$)+M($\nu = 1$)	1.004	14.899
M($\nu = 2.5$)+k·M($\nu = 1$)	0.995	0.257
k·RBF·k·RBF+W	0.976	0.259
k·RBF+k·RBF	0.999	0.255
k·RBF+k·M($\nu = 2.5$)	1.031	0.254
M($\nu = 2.5$)+W	0.991	0.255
M($\nu = 2.5$)+M($\nu = 1$)+W	1.081	0.339
M($\nu = 2.5$)+k·M($\nu = 1$)+W	1.057	0.254
k·RBF+k·RBF+W	1.012	0.255

Let us now consider a more general method to compare the 3D GP predictions and what is expected from the target. In particular, what we expect is that the predictions evaluated by the 3D GP overlap with the predictions made after the GS (target). In order to compare the results in a more significant way using the information related to how much the two plots for each model (represented in *Fig. 3.20* and *Fig. 3.21*) are overlapped, two parameters can be evaluated:

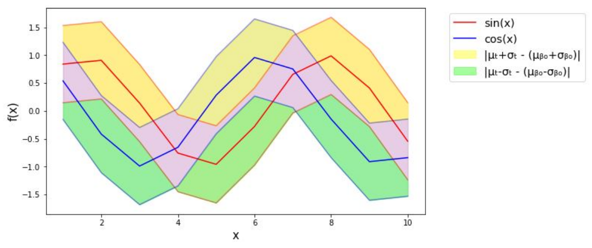
- the difference between the two top extremes of the target ($\mu_t + \sigma_t$) and of the 3D GP predictions ($\mu_{\beta o} + \sigma_{\beta o}$): $|\mu_t + \sigma_t - (\mu_{\beta o} + \sigma_{\beta o})|$;

- the difference between the two bottom extremes of the target ($\mu_t - \sigma_t$) and of the 3D GP predictions ($\mu_{\beta o} - \sigma_{\beta o}$): $|\mu_t - \sigma_t - (\mu_{\beta o} - \sigma_{\beta o})|$;

In this way it is possible to have an estimate of the regions that are not overlapped. For the purpose of clarification only, in *Fig. 3.22* is shown a generic plot with two functions that in some regions are overlapped and in other regions are not overlapped. The yellow regions represent the difference that we evaluate between the two top extremes ($|\mu_t + \sigma_t - (\mu_{\beta o} + \sigma_{\beta o})|$), while the green regions represent the difference evaluated between the two bottom extremes ($|\mu_t - \sigma_t - (\mu_{\beta o} - \sigma_{\beta o})|$). As it is possible to observe these regions represent the non overlapped regions and for this reason what we have is that the smaller the two regions are, the better the model is.



(a) *Plot of two simple functions, sine and cosine, that are overlapped in some regions.*



(b) *Plot of two simple functions, sine and cosine, in yellow is highlighted the difference between the two top extremes while in green is highlighted the difference between the two bottom extremes.*

Figure 3.22: Visual representation of the difference between top and bottom extremes.

In order to directly compare these regions, the norm2 of the difference between the two top extremes and then the norm2 of the difference between the two bottom extremes can be evaluated. Finally, in order to have only one number that allows to understand the accuracy of the BO models, it is possible to sum these two parameters and finally to compare this sum for all of the BO models. The quantity used for the comparison is termed error in terms of area (E_a) and the best model should have the smallest possible value associated to this quantity. The results are shown in *Table 3.2* and it is possible to observe that the smallest E_a is obtained for **k·RBF·k·RBF+W**. This result is not in disagreement with the result found with the previous criterion. In fact the two kernels differ in the value of E_a only at the third decimal value. This is due to the fact that the RBF part of the k·RBF·k·RBF+W is strictly related to the power 2 of the RBF part of k·RBF+W.

Table 3.2: Comparison between the 3D BO using different kernels in terms of area. Matern kernel is indicated with M.

Kernel	E_a
M($\nu = 2.5$)	29.404
RBF	1.164
k·RBF+W	0.239
RBF+k·W	4.911
M($\nu = 2.5$)+M($\nu = 1$)	30.026
M($\nu = 2.5$)+k·M($\nu = 1$)	0.251
k·RBF·k·RBF+W	0.237
k·RBF+k·RBF	0.291
k·RBF+k·M($\nu = 2.5$)	0.347
M($\nu = 2.5$)+W	0.270
M($\nu = 2.5$)+M($\nu = 1$)+W	0.828
M($\nu = 2.5$)+k·M($\nu = 1$)+W	0.360
k·RBF+k·RBF+W	0.265

At the end of this study where different kernels were taken into consideration it is possible to conclude that in this specific classification problem the kernel that better describes the correlation between the points of the target function is k·RBF+W, or something that is strictly related to it. As a consequence for the next step the kernel will be fixed to **k·RBF+W**.

3.4 Results of Bayesian optimization

After having determined the most suitable target function and kernel for this classification problem and having fixed a general stopping criterion useful to avoid the over-training, it is possible to perform the 3D bayesian optimization in order to finally find the ideal combination of the number of filters in the first and second convolutional layer and of the number of neurons in the dense layer that gives the best performances of the CNN.
In this section I will show the results obtained with the BO using a stopping criterion based on the past history of the validation loss during the training, minus the validation loss as the target function and a kernel, determined in *Section 3.3.5*, that is a sum of two terms, the RBF kernel and the white kernel:

$$k(x_a, x_b) = \alpha e^{-\frac{|x_a - x_b|^2}{2\ell^2}} + \beta\delta(x_a - x_b)$$

The 3D Bayesian Optimization is performed by using as parameters the number of filters in the 1^{st} (sl1) and 2^{nd} (sl2) convolutional layers and the number of neurons in the dense layer (sl3). The kernel, after having fitted the GP during the 3D BO, becomes:

$$0.00603^2 \cdot RBF(length\_scale = 17.7) + WhiteKernel(noise\_level = 1e^{-10})$$

and the result is shown in *Fig. 3.23.*

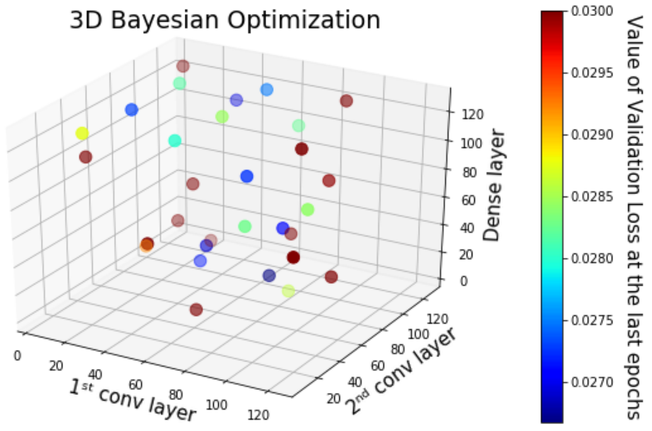


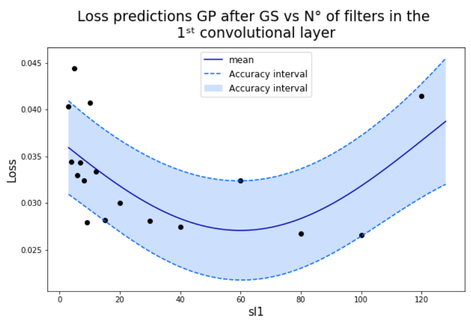
Figure 3.23: Evaluation of the validation loss for different combination of sl1, sl2 and sl3. The colorbar represents the value of the validation loss, the highest value in dark red and the smallest one in dark blue.

The best value of the validation loss, equal to **0.0267**, is obtained with the following combination: $sl1 = 85$, $sl2 = 63$, $sl3 = 28$. At this point it is also possible to perform the GP for only one of

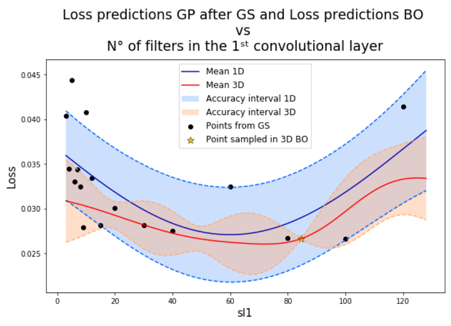
the three parameters. In particular it is possible to fix the other two parameters to the best value found with the 3D BO in order to compare the 2D representation of the loss as a function of sl1, sl2 and finally sl3.
It means that first it is possible to perform the GP for different sl1 by fixing $sl2 = 63$ and $sl3 = 28$. The associated kernel will become:

$$0.0413^2 \cdot RBF(length\_scale = 100) + WhiteKernel(noise\_level = 2.2e^{-5})$$

The result of GP is shown in *Fig. 3.24a*, it is also compared with the result obtained considering one slice of the 3D BO (shown in *Fig. 3.24b*).



(a) *Loss as a function of the number of filters in the* 1^{st} *convolutional layer.*



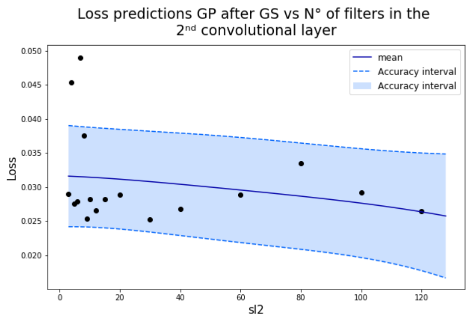
(b) *Loss as a function of the number of filters in the* 1^{st} *convolutional layer in the 3D case, orange, and in 1D, blue.*

Figure 3.24: Comparison of the results obtained with the GP after a small GS along sl1, on the left, and with the 3D BO with a cut along sl2=63 and sl3=28, in order to have a Loss as a function of sl1, on the right.

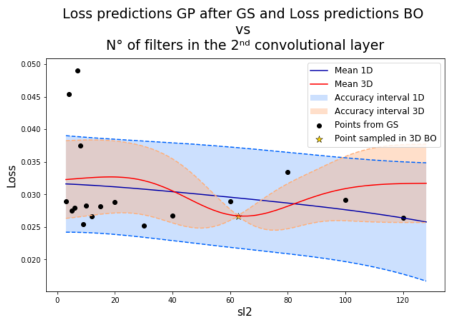
The same can be repeated for sl2 by fixing $sl1 = 85$ and $sl3 = 28$. The associated kernel in this case will be:

$$kernel = 0.0236^2 \cdot RBF(length\_scale = 100) + WhiteKernel(noise\_level = 4.96e^{-5})$$

The result of BO is shown in *Fig. 3.25a* and it is also compared with the result obtained after the 3D BO as done for sl1 (see *Fig. 3.25b*).



(a) *Loss as a function of the number of filters in the 2nd convolutional layer.*



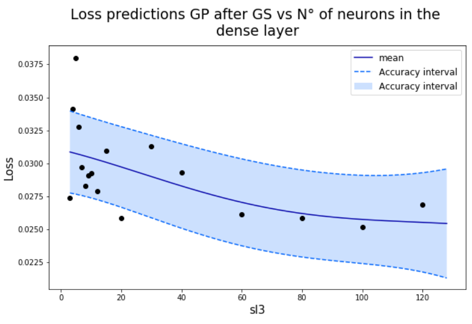
(b) *Loss as a function of the number of filters in the 2nd convolutional layer in the 3D case.*

Figure 3.25: Comparison of the results obtained with the GP after a small GS along sl2, on the left, and with the 3D BO with a cut along sl1=85 and sl3=28, in order to have a Loss as a function of sl2, on the right.

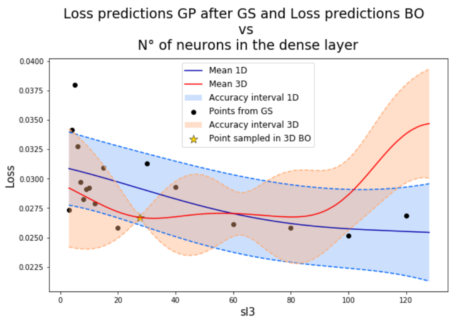
Finally, with sl3 it is possible to fix sl1 and sl2 to be 85 and 63 respectively. The associated kernel will be:

$$kernel = 0.0224^2 \cdot RBF(length\_scale = 100) + WhiteKernel(noise\_level = 8.56e^{-6})$$

The result of BO is shown in *Fig. 3.26a*, it is also compared with the result obtained with the 3D BO as done for both sl1 and sl2 (see *Fig. 3.26b*).



(a) *Loss as a function of the number of neurons in the dense layer.*



(b) *Loss as a function of the number of neurons in the dense layer in the 3D case.*

Figure 3.26: Comparison of the results obtained with the GP after a small GS along sl3, on the left, and the comparison with the 3D BO with a cut along sl1=85 and sl2=63, in order to have a Loss as a function of sl3, on the right.

As it is possible to observe from these results, the 3D GP does not match exactly the results obtained with the 1D GP. This could be related to the choice of the function that models the correlation between points that, in this case, could be a too simple model even if we observed that the kernel we are using is better than others. However, the results obtained with the selected kernel and target function reproduce in a very good way the behaviour of the target function that it is expected by the known results. For this reason, these results are enough for our purpose. They also allow to understand that the loss function that we are trying to minimize, in a certain region of the topology space, does not strongly depend on the number of filters in the convolutional layers or on the number of neurons in the dense layer. More precisely it means that the value of the loss function is always around 0.032 ± 0.006 when *sl1*, *sl2* and *sl3* are not too small (10) or too high (110). These considerations can be very useful for the implementation of the network on FPGA, since it will be possible to modify the number of filters in the convolutional layers and of neurons in the dense layer in order to guarantee the implementation of the network on FPGA without loosing too much in terms of performances. However, due to the fact that the best performances, loss equals to 0.0267, of the CNN are obtained for sl1 = 85, sl2=63 and sl3 = 28 and that performing this kind of optimization gives the optimal bias-variance tradeoff, because allows to find a model that is neither too complex nor too easy, we will keep this architecture and we will try to implement it on FPGA.

3.5 Summary

This chapter provided an explanation of different tuning techniques for the optimization of the hyper-parameters related to the size of the network with a more detailed attention to the Bayesian optimization. Then, different strategies used in order to determine the target function and the kernel function that were used in the practical use of the Bayesian optimization were explained. These selected target function and kernel are the **validation loss** and the **weighted RBF plus the white kernel** respectively. In doing this selection of the functions useful for the Bayesian optimization process it was also introduced the criterion used in the training process to avoid the over-fitting problem. Finally, the Bayesian optimization was performed and the number of filters in the 1^{st} and 2^{nd} convolutional layers and the number of neurons in the dense layer were found to be equal to 85, 63 and 28 respectively.
At this point, all the hyper-parameters of the neural network are optimized. The final and ideal architecture of the CNN for the classification of EM and HAD particles is:

- Input layer of size $21 \times 21 \times 3$
- A 2D convolutional layer, with 85 filters, a 3×3 kernel, a ReLu as activation function, a pooling layer with size 2×2, a dropout layer with a fraction of unit's layer to dropout equal to 0.2;
- A 2D convolutional layer, with 63 filters, a 3×3 kernel, and ReLu as activation function, a pooling layer with size 2×2, a dropout layer with a fraction of unit's layer to dropout equal to 0.2;
- A Flatten layer;
- Dense layer, with 28 neurons and a ReLu as activation function, a dropout layer with a fraction of unit's layer to dropout equal to 0.2;
- Dense layer, with 2 neurons as output, and a Softmax as activation function.

with a batch size = 32, the presence of the dropout layers and without any L1, L2 regularizer. The method described up to this point to select the hyper-parameters of the neural network is

universal and re-applicable and it will be re-used later in this study for an additional optimization (in *Section 4.2.2*).
Moreover, this architecture needs to be implemented on FPGA in order to observe how many resources are needed for the implementation and how much time the associated hardware requires in order to receive the input and give the output (latency). In order to observe these number of resources and this latency we need to exploit the tools described in *Section 1.4.2* and in *Section 1.4.3*, called hls4ml and Vivado HLS, that will allow the conversion from the Keras model into the C++ code and the conversion from the C++ code into the HDL code respectively.

Chapter 4

Vivado implementation

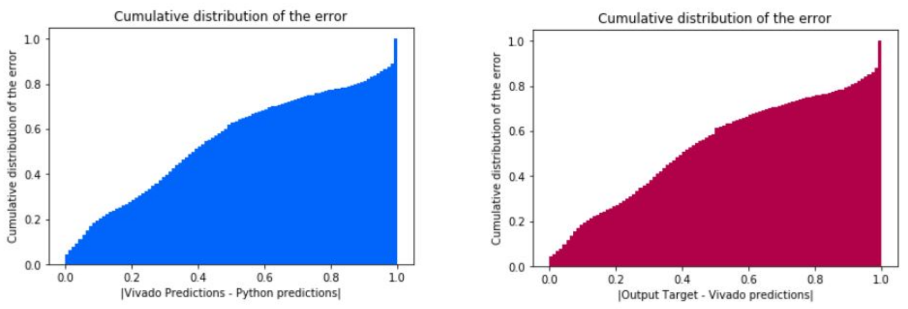
This chapter aims at describing the use of the hls4ml tool in order to convert the Python model of the CNN into a C++ code and the use of the tool Vivado HLS in order to convert the C++ code into an HDL code. These tools are used one after the other. In particular, once the C++ code is generated, it can be opened with Vivado HLS in order to perform the so called C-simulation (csim). This simulation allows to simulate the behaviour of the C++ code, which is the translation of the Python model, and to generate outputs that, if the conversion (including the compression of the weights and the biases of the network from a floating point precision to the fixed point precision) has been done correctly, must correspond to the predictions made by the original Python model. In fact, the output of the C-simulation is compared to the predictions made by the original model in order to observe whether the output of the network is still correct after the first conversion. Once the output is assessed as correct, the next step is the synthesis of the model, step where the C++ code is converted into an HDL code. Finally, when the synthesis is completed, it is possible to have information about the approximated number of resources and latency. This is an approximation and does not correspond to the real number of resources and the real latency. In order to have their real value it is possible to extract the IP core using Vivado HLS and load it in a different tool called Vivado.
In more details, even in this case we started to implement on FPGA a simple network, such as the FNN. Once succeeded with this task, we moved to the implementation of a more complex network, the CNN, applying the learnt information.
Section 4.1, describes the implementation of the FNN on FPGA, showing what are the modifications needed in the C++ code in order to find an output that reproduces exactly the predictions made by the original Python model. *Section 4.2* introduces the implementation of a very small CNN at the beginning and then it reports the problems that have been encountered in the implementation of the CNN on FPGA using Vivado HLS for the conversion into the HDL code. They were mainly due to the presence of size limits for what concerns the implementable network through Vivado HLS. Thereafter, in *Section 4.2* are also introduced the compression techniques that we needed to apply in order to implement our CNN on FPGA, such as the reduction of the size of the particles images, and, as a consequence, the new necessary optimizations of the network and their results are presented. Finally, this new network is implemented on FPGA and the estimated resources and latency are extracted.

4.1 Comparison between Vivado and Python results for a FNN

The implementation of the neural network on FPGA, and in particular the conversion of the Python code into a C++ code and then into an HDL code, can require some modifications of the original network in order to reach better performances when converting the code. In order to understand this aspect we decided to start with the implementation of a very simple ANN. We selected a FNN with 3 layers with respectively 20, 10 and 1 neurons, a ReLu as activation function, except for the last layer where a sigmoid is used, Adam as optimizer and an input of size 17 that corresponds to the most informative features of the clusters of particle energy described in *Section 1.2.1.* After having trained the model, we tried to convert the Python code into the C++ code with a Reuse Factor (RF) equal to 15 and with the default settings of hls4ml, in particular by using:

$$ap\_fixed < 16{,}6 >$$

where 16 stays for the total number of bits and 6 indicates the bits used for the integer part. One of these 6 bits is used for the sign, in order to convert the floating point format of the Python model into the fixed point precision used by the HDL for the weights, the biases and the predictions for each layer. Since Python, and in general the IEEE standard for the double precision floating point format, works with up to 52 decimal values, such a representation can result in the loss of some details supported by the Python language and, as a consequence, in a compression.
Once the C++ code is generated, it is possible to use Vivado HLS in order to perform the C-simulation and to observe the output after the conversion. The cumulative distributions of the absolute value of the error either between the predictions of the Vivado HLS model and the predictions of the Python model and between the predictions of the Vivado HLS model and what is expected from the target are shown in *Fig. 4.1a* and in *Fig. 4.1b* respectively. Ideally, in the absence of errors these plots would reach one almost immediately, meaning that the only errors appearing are extremely small.

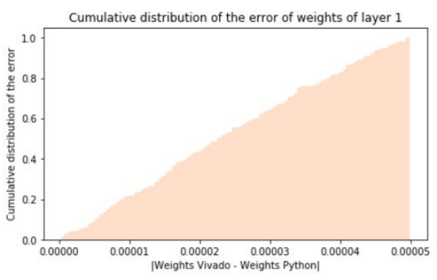


(a) *Cumulative distribution of the absolute value of the error between the predictions of the Vivado HLS model and the predictions of the Python model.*

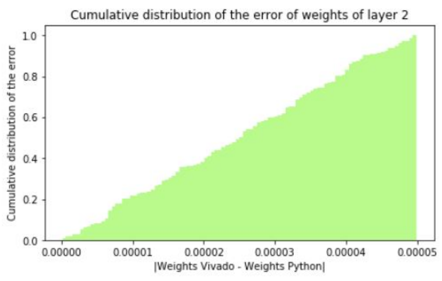
(b) *Cumulative distribution of the absolute value of the error between the predictions of the Vivado HLS model and the target.*

Figure 4.1: Cumulative distribution of the absolute value of the error between the predictions made by Vivado HLS and by Python, on the left, and between the predictions made by Vivado HLS and the target, on the right.

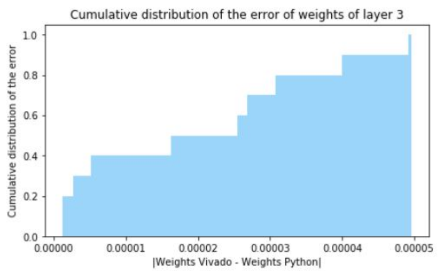
What it is possible to observe from these plots is that the output of the Vivado HLS model is not correct. In fact, there is a significant error when comparing it either with the predictions made by the original Python model and with the expected target. In order to figure out where this error is coming from it is possible to first compare the weights of the original Python model and the compressed ones generated during the conversion into the C++ code, to see if there is a mismatch after the compression for each layer. As it is possible to see from *Fig. 4.2* the output errors do not come from an error in the compression of the weights, since they differ by a factor of 10^{-5} only.



(a) *Cumulative distribution of the error of the weights for the first layer.*



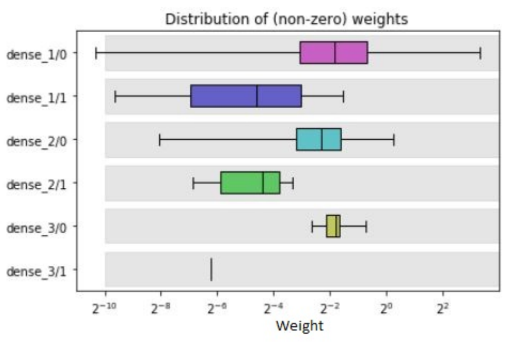
(b) *Cumulative distribution of the error of the weights for the second layer.*



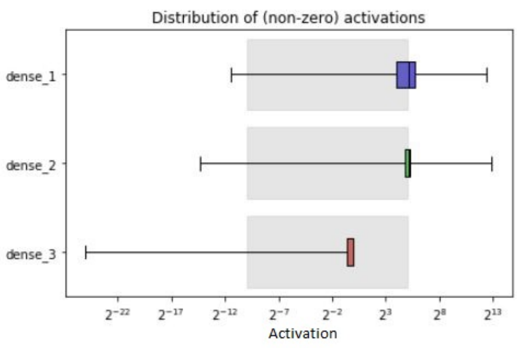
(c) *Cumulative distribution of the error of the weights for the last layer.*

Figure 4.2: Cumulative distribution of the absolute value of the error between the weights of the first layer, second layer and last layer generated by Vivado HLS and the ones of the original Python model.

Once it is ruled out that the problem is related to the compression of the weights, in order to understand why there are these errors on the output it is possible to use the Python files provided in the hls4ml package. In particular, they allow to visualize how the parameters in the Keras model are interpreted during the translation into the hls4ml model in terms of predictions and weights. This is because the Keras models are characterised by floating point numbers that need to be translated into the fixed point precision that is the way employed by Vivado HLS to interpret them. More specifically, two plots can be considered, one that describes the weights for each layer in the Keras model and the associated precision in the hls4ml model and the second one that shows the prediction made after each layer in the Keras model and the associated precision in the hls4ml model. This comparison is a verification method that is used in machine learning problems to check the effectiveness of the model. One example of these plots is shown in *Fig. 4.3*, where the coloured rectangles, representing the ranges of values for the weights and for the predictions of the Keras model, are obtained using the boxplot format. Using this format, each rectangle extents from the lower to the upper quantile values of the vector containing the weights or the predictions. The line in the middle of this rectangle corresponds to the median. In addition, the black lines outgoing from each rectangle represent the range of values. It is worth mentioning that in the case of a simple FNN it was sufficient to use the 25-th percentile and the 75-th percentile as the lower and upper quantile for the representation of the range of values, while these numbers will be modified later for the analysis of the CNN.



(a) *Weights distribution.*



(b) *Activations distribution.*

Figure 4.3: Representation of the ranges of values for the weights and for the activations (corresponding to the predictions) of the Keras model, coloured rectangles, and the hls4ml model precision in gray.

As it is possible to observe from *Fig. 4.3*, it seems that the distribution of the Python weights fits the hls4ml model precision because the coloured rectangles are inside the gray intervals. On the other hand, observing the activations distribution, which corresponds to the predictions distribution, the rectangles are not always inside the gray intervals, i.e. the precision used in that case is not good enough. This problem can be solved, in two different ways: by increasing the fixed point precision and as a consequence the number of bits used, or by modifying the Keras model. The first method requires an higher number of bits that could imply an higher number of resources. For this reason, due to the fact that we have a limited number of resources on FPGA, we decided to try to use the second approach first. For what concerns the Keras model, it is possible to think about a reduction of the range of values assumed by the predictions. This can be done by introducing a *normalization* of the input values in order to have a mean equal to 0 and a standard deviation of 1. In addition, in order to improve the performances as well, it is also possible to add a normalization, not only on the input data, but for each layer in the neural network in order to have a negligible change of the distribution of values at each iteration. In this way, better performances and faster convergence (an example in terms of loss is shown in *Fig. 4.4*) are achieved. The normalization performed after each layer is called batch normalization and it is used to normalize (i.e. mean 0 and standard deviation 1) the output of the activation function for each layer [56].

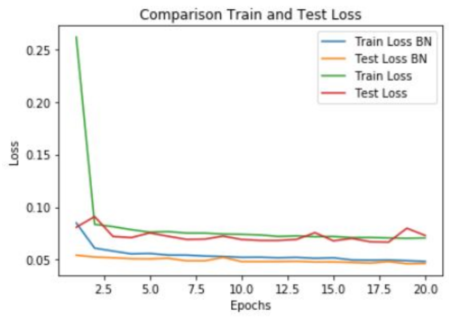
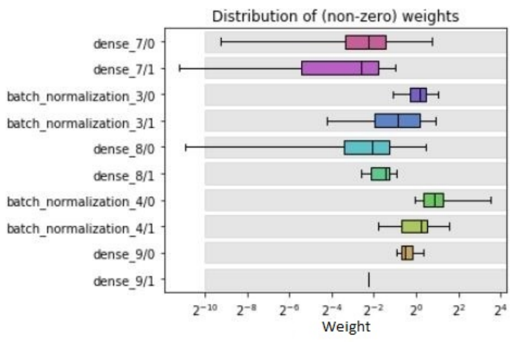
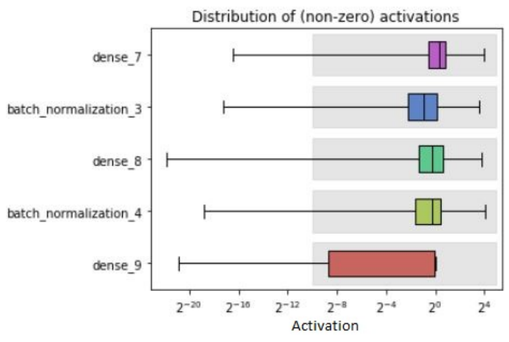


Figure 4.4: Comparison of the loss as a function of the number of epochs for a simple FNN (Train and Test Loss) and for the same FNN but with an input normalization and a Batch normalization after each layer (Train and Test Loss BN).

Once the normalization is performed, comparing the values assumed by the weights and by the predictions of the Keras model with the hls4ml, the precision gives a better result, as it is possible to see in *Fig. 4.5*. In this case both the weights and the predictions of the Keras model have values contained in the precision interval of the hls4ml model. This means that, in order to have an optimal result when the Keras model is translated into the hls4ml model, it is possible to normalize both the inputs of the network and the output of each layer in the model.



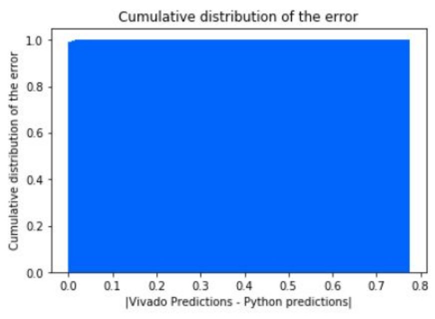
(a) *Weights distribution.*



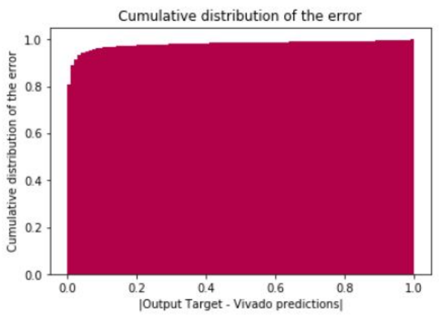
(b) *Activations distribution.*

Figure 4.5: Representation of the ranges of values for the weights and for the predictions of the Keras model, coloured rectangles, and the hls4ml model precision in gray.

Thanks to this result we expect that now, comparing the output of the Keras model and the output obtained after the compression into a C++ code, the error between the two results will become approximately zero. *Fig. 4.6* shows that the presence of the normalization on both the input of the network and after each layer reduces the range of values assumed by the weights and the predictions of the Python model allowing a correct compression with the same floating point precision ($ap\_fixed < 16{,}6 >$).



(a) *Cumulative distribution of the error between the predictions made by the Vivado HLS model and by the original Python model.*



(b) *Cumulative distribution of the error between the target and the predictions made by the Vivado HLS model.*

Figure 4.6: Cumulative distributions of the error between the target, the predictions made by Vivado HLS and the ones made by the original Python model. This result is obtained when using both the input and batch normalization in the Python model.

Another possibility is to normalize only the input of the neural network instead of considering a batch normalization layer after each activation function, in order to see if the batch normalization is necessary or not. The first things that can be observed are the performances of the network in terms of the loss as a function of the number of epochs (see *Fig. 4.7*).

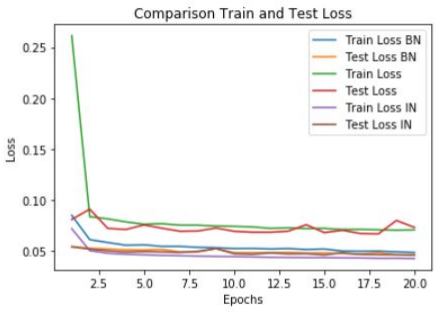
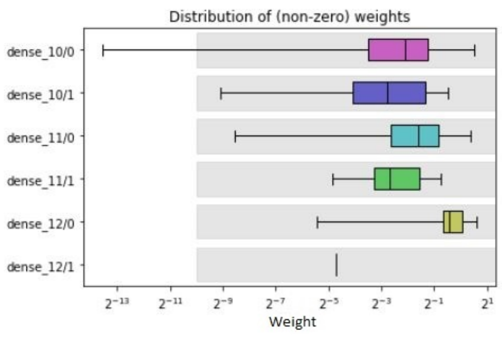


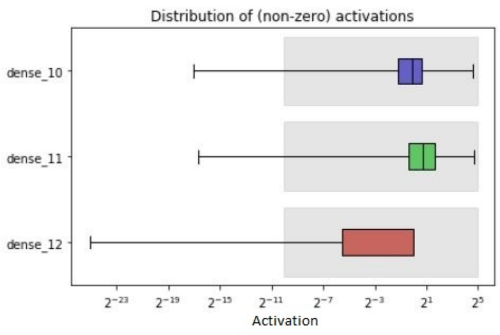
Figure 4.7: Comparison of the loss as a function of the number of epochs for a simple FNN (Train and Test Loss), for the same FNN but with an input normalization and a batch normalization after each layer (Train and Test Loss BN) and finally for the same FNN with only the input normalization (Train and Test loss IN).

In this case the training loss, when only the inputs are normalized, is smaller, and thus better, than the case where the batch normalization layer is added after each activation function. However, considering the validation loss, the two cases are exactly equal. Due to the fact that the parameter of interest is the value of the loss once the network is trained and so the validation loss, let us see how the performances, like the predictions, the AUC and the WP_99, of the neural network with only the normalized input change once compressing it with hls4ml.
As it is possible to observe from *Fig. 4.8* also in this case the weights and the predictions of the Keras model have values that are inside the precision interval of the hls4ml model. In fact, the distribution of the error is equal to the previous case and the output after the conversion into a

C++ code is correct also in this case (see *Fig. 4.9*).

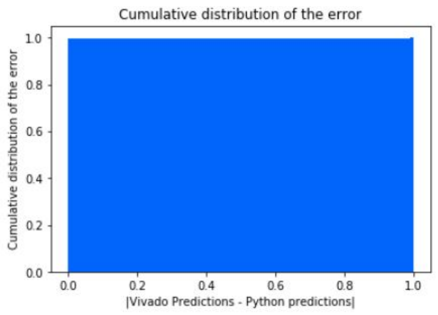


(a) *Weights distribution.*

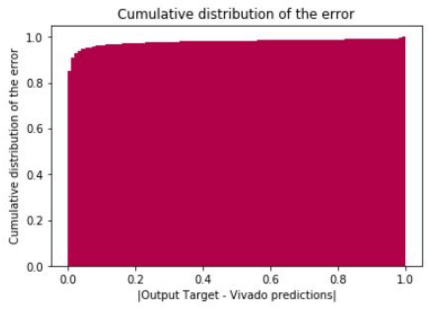


(b) *Activations distribution.*

Figure 4.8: Representation of the ranges of values for the weights and for the activations (corresponding to the predictions) of the Keras model, coloured rectangles, and the hls4ml model precision in gray. In this case the Python model is characterized only by the input normalization.



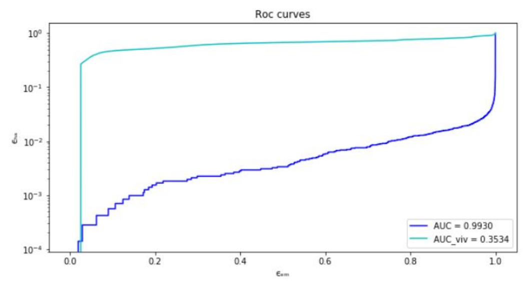
(a) *Cumulative distribution of the error between the predictions made by the Vivado HLS model and by the Python model.*



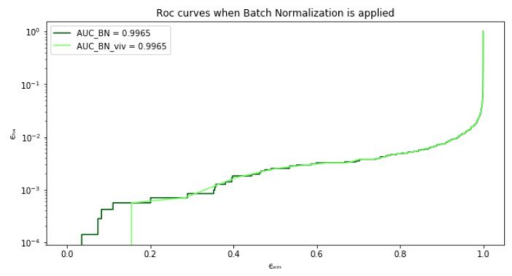
(b) *Cumulative distribution of the error between the target and the predictions made by the Vivado HLS model. In this case the Python model is characterized only by the input normalization.*

Figure 4.9: Cumulative distributions of the error between the target, the predictions made by Vivado and the ones made by Python. This result is obtained when using only the input normalization in the Python model.

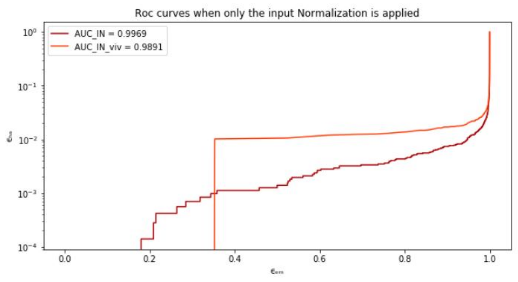
At this point, another parameter can be checked to compare the performances of these three networks and select which of them is the best to be implemented on FPGA while keeping the performances. This parameter is the already mentioned ROC curve and more specifically its related area (AUC). In particular, the ROC curve for the Keras model and for the hls4ml model are compared for each of the network, without any normalization, with both input and batch normalization and with only input normalization, in *Fig. 4.10a*, *Fig. 4.10b*, *Fig. 4.10c* respectively. Besides, the values of the AUC are made explicit in *Table 4.1*, where it is possible to observe that the performances are kept unchanged only in the case where both input and batch normalization are applied.



(a) *ROC curves of the Keras model (blue) and of the hls4ml model (light blue) for the simple FNN.*



(b) *ROC curves of the Keras model (dark green) of for the hls4ml model (light green) for the FNN with both input and batch normalization.*



(c) *ROC curves of the Keras model (red) and of the hls4ml model (orange) for the FNN with only the input normalization.*

Figure 4.10: Comparison of the ROC curves between the Keras model and the hls4ml model.

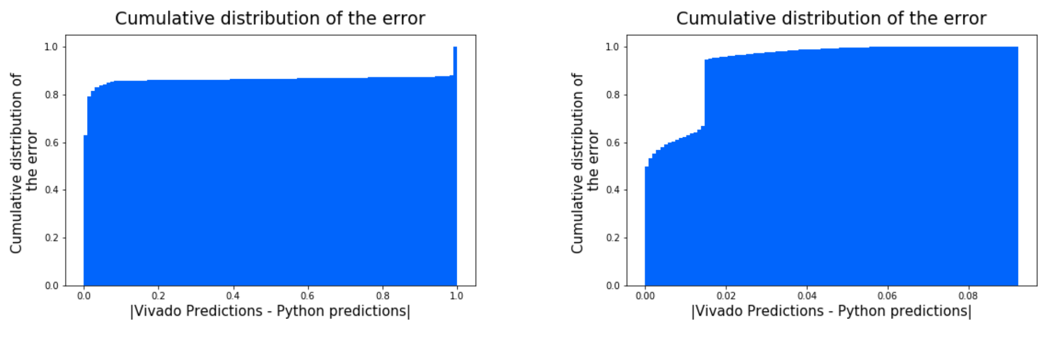
Table 4.1: Comparison of the performances of the networks, in terms of AUC, without any normalization, with both input and batch normalization and with only input normalization.

	AUC (No normalization)	AUC (Input and batch normalization)	AUC (Input normalization)
Keras	0.9930	0.9965	0.9969
Vivado HLS	0.3534	0.9965	0.9891

The conclusion that can be extracted after this last comparison is that, due to the equal results obtained in terms of output for the network where both the input and batch normalization are applied and for the network where only the input normalization is applied, both these networks can be taken into account. However, when we observe the comparison in terms of performances, it is possible to note that they are kept exactly equal, after the compression, only in the case where both the input and batch normalization are applied.
For these reasons, for the next steps where a more complicated network will be analysed, both the **input** and **batch normalization** will be taken into consideration from the beginning.

4.2 Comparison between Vivado and Python results for a CNN

The implementation of a Convolutional Neural Network on Vivado HLS is a little more complicated than implementing a Fully connected Neural Network. The convolution operation is more computationally expensive than the simple multiplication. It means that while implementing a FNN does not require a big attention to the number of neurons in the network, in the case of CNN if the number of neurons or filters is too big there may be problems of dimension and Vivado HLS will not be able to convert the model into an HDL. First of all, considering the information extracted from the implementation of a simple FNN, I tried to implement the model summarized in *Section 3.5*, with in addition the input normalization and the batch normalization after each layer as an additional feature. However, with the current version of Vivado HLS (Vivado 2019.2) it is not possible to convert this model into an HDL code, because the model is too big and the actual size of the model requires a huge usage of memory. What it can be done at this point is to try to compress the actual network to reduce this memory usage. Before doing that, we decided to use one of the structure employed in the example of the conversion provided by the hls4ml library, in order to observe if the output of the C++ code matches with the predictions of the Python model for this more complicated network. The provided model is composed by: 1 convolutional layer (with only 2 filters and a ReLu as activation function) and the output layer (with only two neurons and a softmax as activation function), Adam as optimizer and categorical crossentropy as loss and we used the images of the particles with size $21 \times 21 \times 3$ as input. The output for this simple model that is obtained after the C-simulation was compared with the Python predictions. The results show that there were some errors in the conversion of the Keras model into the C++ code because, observing the differences between the two outputs, it was possible to observe that there is an error of 1 (see *Fig. 4.11a*) at least for 20% of the times.



(a) *Comparison output Vivado HLS and Python.* **(b)** *Comparison output Vivado HLS and Python after adding AP_RND and AP_SAT.*

Figure 4.11: Comparison of the predictions between the Keras model and the Vivado HLS model.

The main problem of this error is the default interpretation of the weights in Vivado HLS. In particular, if the fixed point precision is fixed to be $< 16,6 >$, where 6 bits stay for the integer part, one of them is for the sign and 16 are the total bits, what Vivado HLS does is:

- if the value of the weight is higher than 2^5 it assigns to this weight a negative value;
- if the value of the weight is smaller than -2^5 it assigns to this weight a positive value.

This interpretation, in this specific case, is not the optimal one because some weights in the convolutional layer can be also of the order of 300 and the assignment of a value that is opposite in sign explains the reason why there is an error of exactly 1. There are two possibilities to solve this issue:

- increase the number of bits for the integer part;
- modify the way in which Vivado HLS interprets these high values or small values.

To increase the number of bits is not optimal when implementing on FPGA because it means that the hardware components will need higher number of bits, i.e. higher area. Instead, the second possibility can be more interesting in our case, it is indeed possible to specify how to convert the floating point into a fixed point precision by adding the following two parameters:

- *AP_RND* that allows to round numbers with significant decimal values, in order to avoid the use of a very high number of bits for the decimal part;
- *AP_SAT*, that allows to saturate very high values to the maximum acceptable value, in this case 2^5, and very small values to the minimum acceptable value -2^5, without changing the sign.

The parameter *AP_SAT* could appear a limiting parameter because numbers such as 100 and 300 are interpreted as 32 even if they are very different. However, in this specific case there are not a lot of weights that assume very high or very small values and for this reason it is reasonable to use this method without increasing the fixed point precision[1]. Another reason to use the second method instead of the first is that the order of magnitude of the weights can be very different and it is difficult to exactly know it for each layer. As a matter of facts, the second criterion, where these information are not required to be known, is better and more general. By adding these two parameters the problem of an error equal to 1 for 20% of the times is addressed (see *Fig. 4.11b*). However, in the same picture, *Fig. 4.11b*, it is possible to observe that there is still a difference of 0.02 for 20% of the outputs. This is related to the complexity of the operation in the network and, indeed, by simplifying a little the model, substituting the categorical crossentropy with the binary crossentropy, that in our case can be done because we have a binary classification problem and it is possible to use 1 neuron only in the output layer and substitute the activation function of this layer with the sigmoid. As a resul, the error disappears completely (see *Fig. 4.12*).

[1]If the use of *AP_SAT* alone does not solve the problem it is possible to also increase the number of bits associated to the integer part in the fixed point precision.

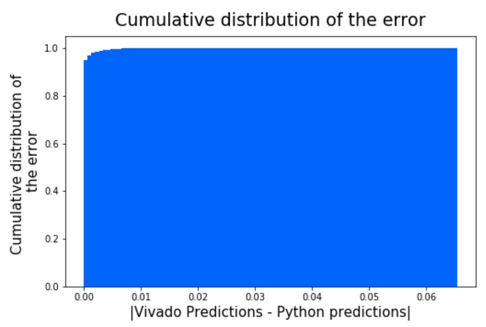


Figure 4.12: Comparison of the output between Keras and Vivado HLS models when using binary crossentropy.

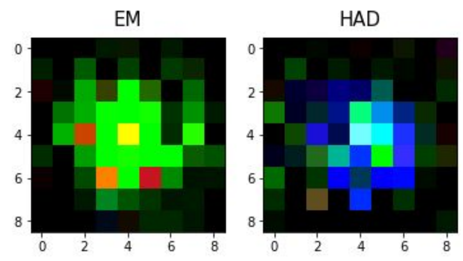
At this point that we understood what it can be done to solve the problem on the output when implementing the CNN with Vivado HLS, the main problem is that the conversion from C++ code into VHDL or Verilog is not supported for large networks on Vivado HLS 2019. In particular, the main issue of the structure found in *Chapter 3* is the size of the images that seems too large to be implemented.
It is important to specify that, at this point of the project, the network with the ideal images size ($21 \times 21 \times 3$) is optimized to reach optimal performances when dealing with the classification of EM and HAD particles and, once the implementation of large CNNs will be supported by Vivado HLS, it will be possible to directly implement this architecture on FPGA and to observe the related resources usage and latency.
However, due to the limitations imposed by the current available version of Vivado HLS, in order to solve the problem and go ahead with the FPGA implementation (even if the size $21 \times 21 \times 3$ was the optimal size for the images on the output of the detector) new datasets with images of size $9 \times 9 \times 3$ were created. This size was selected because it is the smallest that allows to keep enough information for the particles classification and to not loose too much in terms of characteristics of the particles in the images. Let us now analyse this new image size that will also require a new optimization of the CNN to find the hyper-parameters that give the best performances in this case.

4.2.1 New images size

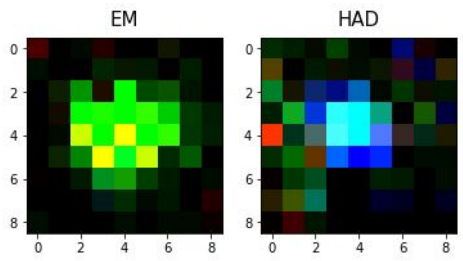
The size $9 \times 9 \times 3$ is smaller than the ideal case and this implies that some of the particle information can be lost. For this reason, it is better to consider all the possibilities available when generating the dataset in order to select which of them gives better results:

- Small pixel size: images where 1 pixel corresponds to 1cm in the virtual plane and use smaller granularity. The images will be characterized by 9 cm window (see *Fig. 4.13a*);

- Large pixel size: images where 1 pixel corresponds to 1.5cm in the virtual plane and use larger granularity. The images will be characterized by 13.5 cm window (see *Fig. 4.13b*).



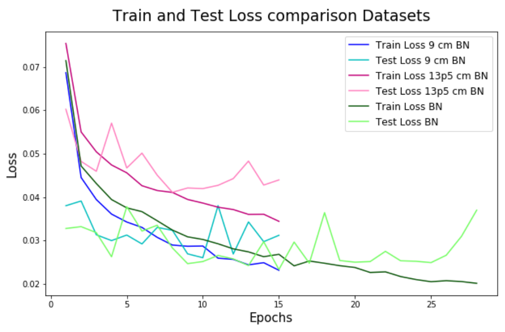
(a) *Images of EM and HAD particles with a size of* $9 \times 9 \times 3$ *and a window of 9cm.*



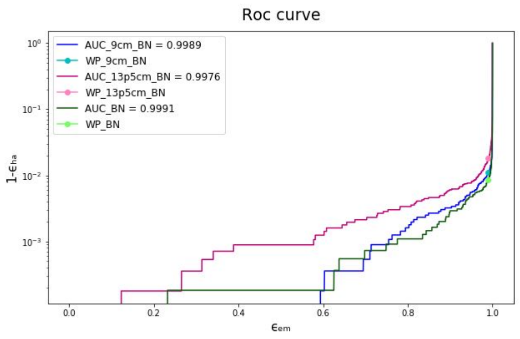
(b) *Images of EM and HAD particles with a size of* $9 \times 9 \times 3$ *and a window of 13.5cm.*

Figure 4.13: Images of EM and HAD particles of size $9 \times 9 \times 3$ with different granularity .

In order to select which of these datasets is better to not loose the performances with respect to the $21 \times 21 \times 3$ images I compared the 3 datasets in terms of validation loss and ROC curve (see *Fig. 4.14*) using the same structure of the network summarized in *Section 3.5*.



(a) *Comparison Train and Validation loss for the 3 datasets.*



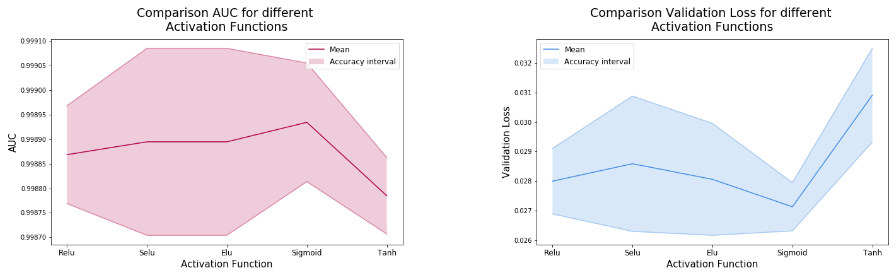
(b) *Comparison ROC curve for the 3 datasets.*

Figure 4.14: Performance comparison for the 3 datasets. In green the dataset with images size $21 \times 21 \times 3$, in blue the datasets with images size $9 \times 9 \times 3$ and a window of 9cm, in magenta the datasets with images size $9 \times 9 \times 3$ and a window of 13.5cm.

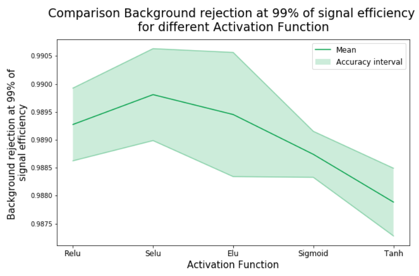
As it is possible to observe from *Fig. 4.14*, the dataset with images of size $9 \times 9 \times 3$ and a window of 9cm (blue curves) allows to reach performances that are almost equal to those obtained with the dataset with $21 \times 21 \times 3$ images size (green curves). For this reason, the dataset with 9cm window is selected and in the next section a small hyper-parameters optimization is performed to select the best architecture.

4.2.2 Small hyper-parameters optimization for the new images size

Considering the new dataset characterized by images with size $9 \times 9 \times 3$ and 9cm window, it could be useful to perform a study, similar to the one done in *Section 2.3* and in *Chapter 3*, in order to select the best hyper-parameters of the Convolutional Neural Network for the particles classification problem, where the images have different shapes compared to the previous case. As before, the first thing that can be selected is the activation function (see *Fig. 4.15*). In this case, in order to take into account the variability of the network, instead of running the CNN one time only for each possible activation function, I run each CNN 5 times for each activation function and I compared the loss, the AUC and the WP_99 in terms of mean and standard deviation.



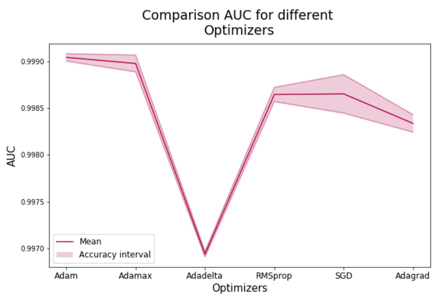
(a) *AUC comparison between different activation functions.* **(b)** *Loss comparison between different activation functions.*



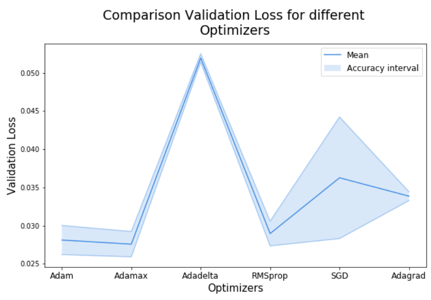
(c) *WP_99 comparison between different activation functions.*

Figure 4.15: Comparison between different activation functions.

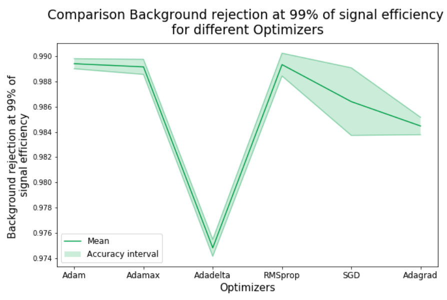
From these plots it is possible to understand that the *SELU* could give better results compared to the other activations considering the uncertainties, highest AUC and WP_99 and smallest validation loss. Fixing the activation function to the *SELU* it is possible to perform the same comparison for the optimizers (see *Fig. 4.16*). In this case, the optimizer that gives the best global result, is the *Adam*. Fixing the optimizers, the other hyper-parameter that can be fixed is the learning rate of the optimizers (see *Fig. 4.17*).



(a) *AUC comparison between different optimizers.*

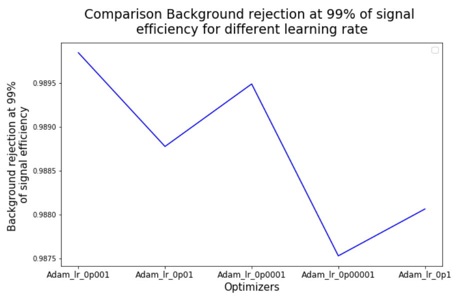


(b) *Loss comparison between different optimizers.*

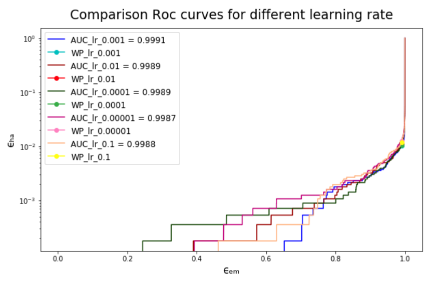


(c) *WP_99 comparison between different optimizers.*

Figure 4.16: Comparison between different optimizers.



(a) *WP_99 comparison between different Learning rate using Adam as optimizers.*



(b) *ROC curve comparison between different Learning rate using Adam as optimizers.*

Figure 4.17: Comparison between different learning rate

From *Fig. 4.17* it is possible to observe that the best learning rate is 0.001, that is the default value for Adam. The final step for this small optimization is the selection of the number of filters in the first and second convolutional layers and the number of neurons in the dense layer. In order to do that the Bayesian optimization with minus the validation loss as a target function and the weighted RBF plus the white kernel as the kernel function, described in *Chapter 3*, is used and the results are:

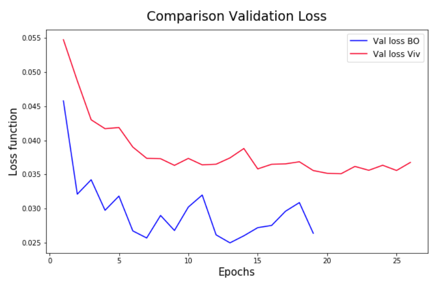
- Number of filters in the 1^{st} convolutional layer = **128**;

- Number of filters in the 2^{nd} convolutional layer = **128**;
- Number of neurons in the dense layer = **128**.

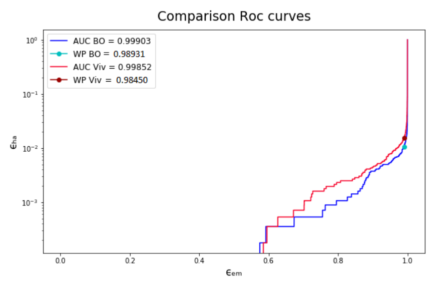
with a loss = **0.0259**. This result means that the selected range of parameters is too small and it is possible to find a better value of the loss by increasing the range of values for sl1, sl2 and sl3. As a consequence, it is not possible to be sure that larger numbers would not give better performances. However, this network is still too big, also with this smaller size of the images, and the conversion from the C++ code into the HDL code is still not supported by Vivado HLS and several problems related to excessive memory usage appear. For this reason, a more detailed study of how many filters and neurons are supported in Vivado HLS when the images size is $9 \times 9 \times 3$ is performed. The maximum number of filters or neurons in each layer that allows the conversion is very small, compared to what obtained after the BO, and corresponds to:

- Number of filters in the 1^{st} convolutional layer = 5;
- Number of filters in the 2^{nd} convolutional layer = 15;
- Number of neurons in the dense layer = 8.

Now that the maximum number of filters and neurons supported by the current version of Vivado HLS is discovered, it is necessary to understand if the compression of our network from $sl1 = 128$, $sl2 = 128$ and $sl3 = 128$ to $sl1 = 5$, $sl2 = 15$ and $sl3 = 8$ respectively implies a degradation of the performances of the network or the performances do not change significantly. In order to do that the two Python models can be compared in terms of validation loss and of the ROC curve. This comparison is shown in *Fig. 4.18*, where it is possible to observe that in terms of validation loss there is a variation that could be significant because it increases from 0.0259 to 0.0368 while in terms of the AUC the variation is not very significant and it decreases from 0.9990 to 0.9985.



(a) *Comparison in terms of the validation loss.*



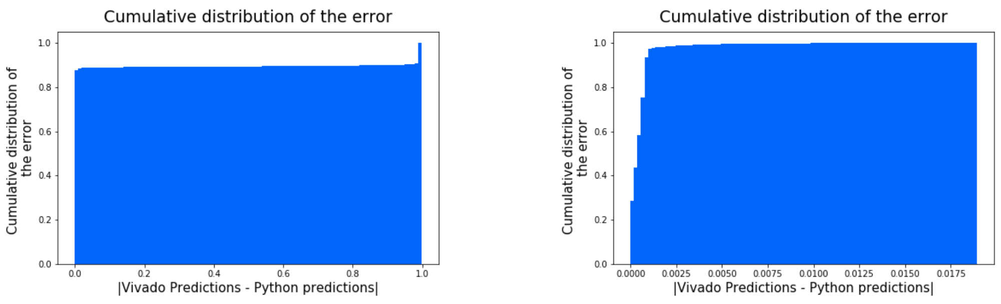
(b) *Comparison in terms of the AUC.*

Figure 4.18: Comparison between the best architecture found after the BO and the architecture with the highest number of filters and neurons that can be implemented with Vivado HLS, in terms of validation loss and AUC.

This result suggests that, even if the validation loss of the model is increasing, the AUC is always very high, that means that the network is still able to classify very well both EM and HAD particles. For this reason, this network could be a good compromise in terms of performances and size, in the framework of FPGA implementation. As a consequence, now it is possible to move to the implementation of this network using hls4ml and Vivado HLS.

4.2.3 Vivado implementation of the new network: sl1=5, sl2=15 and sl3=8

In order to implement the CNN on FPGA, we have seen that, in addition to the input and batch normalization, it is necessary to modify the way how Vivado HLS performs the conversion by adding the parameters AP_RND and AP_SAT to the traditional $ap\_fixed < 16{,}6 >$. Even if we observed that this modification completely solved the output problem in a very small network, it is better to check if the effect is the same also with this higher number of filters and neurons. In *Fig. 4.19* are shown the cumulative distributions of the absolute error between the predictions after the csim and the predictions made by the original Python model and what it is possible to observe is that also in this case this method seems suitable to implement the network.



(a) *Only $ap\_fixed < 16{,}6 >$ for the conversion.*

(b) *$ap\_fixed < 16{,}6 >$ plus AP_RND and AP_SAT for the conversion.*

Figure 4.19: Cumulative distributions of the absolute error between the predictions after the csim and the predictions made by the original Python model. On the left only $ap\_fixed < 16{,}6 >$ is used for the conversion, while on the right in addition to $ap\_fixed < 16{,}6 >$ are also used AP_RND and AP_SAT.

However, it could be interesting to check how these additional parameters modify the number of resources once converted the C++ code into an HDL code performing the synthesis. In order to do that, we compressed the same Python model into C++ codes using three different types of conversions from floating point to fixed point precision: $ap\_fixed < 16{,}6 >$, $ap\_fixed < 16{,}6{,}AP\_RND >$ and $ap\_fixed < 16{,}6{,}AP\_RND{,}AP\_SAT >$. Once the models are compressed, it is possible to synthesize them by selecting the FPGA of interest: xcku15p-ffve1760-2-e and by selecting a clock period, that in our case we fixed to 5 ns.
After the synthesis, it is possible to have an estimation of the number of resources that will be used in the hardware implementation, and the estimated latency. In *Table 4.2* are summarized the results in terms of resources and latency, where the latency is expressed in terms of number of clock cycles and the values highlighted in red represent values that exceed the total number of resources available on the selected FPGA.

Table 4.2: Comparison between the resources and the latency when using different types of parameters for the conversion from floating point to the fixed point precision.

Parameters	BRAM	DSP	FF	LUT	Total Latency	Theoretical Latency
Default	750	1454	84610	763139 (145%)	3964	451
AP_RND	750	1456	89559	804017 (153%)	3966	453
AP_RND and AP_SAT	750	1456	163381	1079057 (206%)	4002	489
Available	1968	1968	1045440	522720		

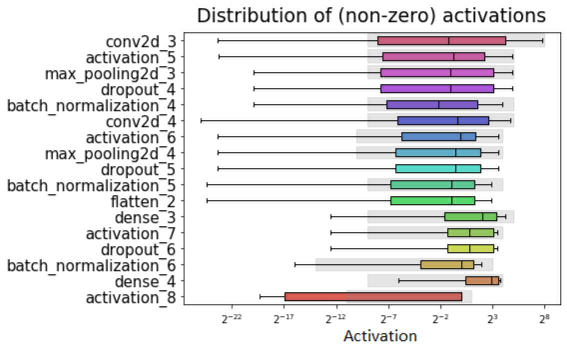
For what concerns the latency, it is worth mentioning that there is a problem in the way the pooling layer is synthesized with the hls4ml implementation. During the implementation of the pooling layers, a warning suggesting that they are not implemented with the same optimization methods, described in *Section 1.4.2*, used for all the other layers appears. This can be observed in *Table 4.3* where it is possible to see that hls4ml, for the implementation of the pooling, is processing the operations in a sequential way. In fact, it uses a rather low amount of resources compared to other layers of similar sizes, while having a huge latency at the same time.

Table 4.3: Comparison between the resources and the latency required by each layer in the selected CNN.

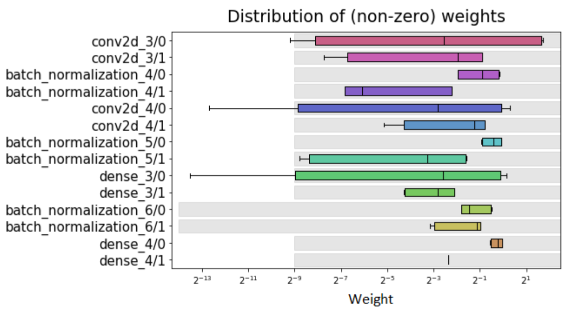
	BRAM	DSP	FF	LUT	Latency
Conv 2D	0	136	27218	591209	330
SELU	178	355	11718	26285	1
MaxPool	0	0	1505	8024	2131
Normalize	0	6	2512	1551	13
Conv 2D	0	670	9487	36945	21
SELU	120	240	7923	17775	1
MaxPool	0	0	1148	4962	1411
Norm	0	4	1807	1645	13
Dense	15	32	3664	4096	17
SELU	4	8	267	607	1
Norm	0	1	243	477	7
Dense	0	2	325	359	5
Sigmoid	1	0	14	163	2

Consequently, what is missing in the implementation of the pooling layer is the parallelization of its operations that, in *Section 1.4.2*, we have seen as the way to reduce the latency at the cost of increasing the resources. For this reason, in *Table 4.2* are quoted the total estimated

latency (including the pooling) and the theoretical latency considering the latency of the pooling layers similar to those of other layers of similar sizes (about 20 clock cycles). The total latency obtained with the pooling layers is obviously not affordable, while considering a possible parallelization of the pooling operations it would be about 10 times smaller. As it is possible to observe from *Table 4.2*, by adding only the AP_RND parameter to the type of conversion there is an increase of 8% in terms of LUT compared with the traditional conversion, while by adding also AP_SAT the increase is higher and of 61%.
These results suggest that modifying the conversion from the floating point into a fixed point precision by adding AP_RND and AP_SAT is not suitable for this kind of study because the number of resources, in terms of LUTs, increases a lot and it is already exceeded when we consider the traditional conversion with 16 bits. As a consequence the best choice, in this case, is to go back to the traditional conversion of the floating point into a fixed point precision and to try to modify for each layer the number of bits by observing the values assumed by the predictions after each layer and by the weights for each layer. This can be useful to reduce the number of bits when the values of the predictions or weights are small enough. In order to see the selected proper number of bits for each layer, *Fig. 4.20* and *Fig. 4.21* show the values of the weights and the predictions of the Python model together with the values that are correctly interpreted by Vivado HLS.



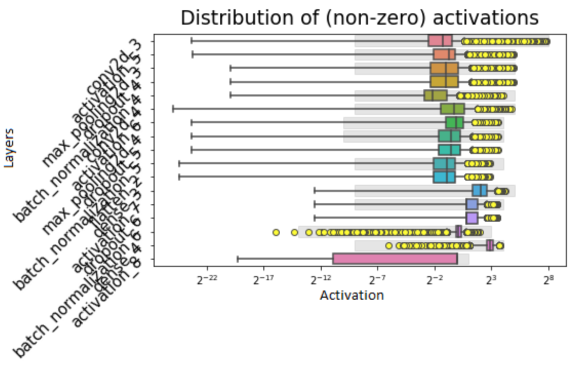
(a) *Distribution of the predictions.*



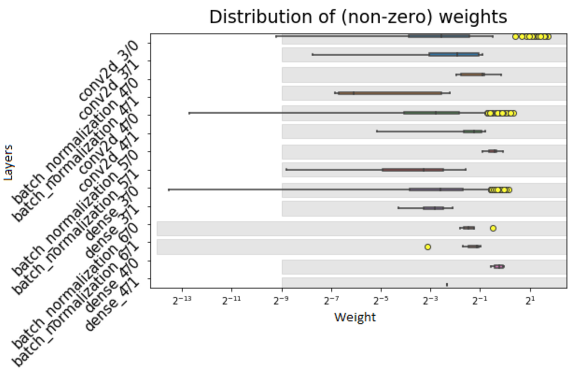
(b) *Distribution of the weights.*

Figure 4.20: Distributions of the predictions and of the weights, where the coloured rectangles represent the values assumed by the predictions and the weights of the original Python model and the gray rectangles represents the intervals inside which Vivado HLS is able to make a good conversion. The coloured rectangles are obtained using 1-th percentile and 99-th percentile as lower and upper quantiles.

In *Fig. 4.21* the coloured rectangles are obtained using 25-th percentile and 75-th percentile as lower and upper quantiles. While the yellow circles correspond to the values that are outside the black line intervals and are shown because we observed that it could be useful to cover them in order to have a good conversion from the Python model into the C++ code.
This is because when the network is characterized by more than two layers, even if only few values are not converted in a correct way in the first layers, the error that is generated can be propagated through the next layers and, as a consequence, it could be amplified on the output of the network. As a consequence we expect that, for each layer, when the gray interval is able to cover both the coloured rectangle shown in *Fig. 4.20* and the yellow circles shown in *Fig. 4.21* the conversion will be good.



(a) *Distribution of the predictions.*



(b) *Distribution of the weights.*

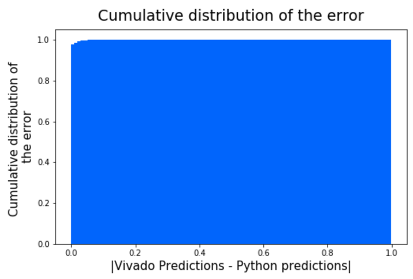
Figure 4.21: Distributions of the predictions and of the weights, where the yellow circles (and also the coloured rectangles) represent the values assumed by the predictions and the weights of the original Python model and the gray rectangles represents the intervals inside which Vivado HLS is able to make a good conversion.

In addition, observing *Fig. 4.20* and *Fig. 4.21* it is possible to see that the gray rectangles, representing ranges of values that can be correctly interpreted by Vivado HLS, cover different ranges for each layer. This is due to the selection of different fixed point precisions for each layer depending on the values assumed by both the coloured rectangles in *Fig. 4.20* and the yellow circles in *Fig. 4.21*.
Moreover, the ranges of values assumed by both the weights and the predictions after each layer for the Python model are always inside the gray rectangles except for the last layer. Considering the last layer, the region that is not covered is the one associated to values smaller than 2^{-10}. Due to the fact that on the output we expect a number smaller than 0.5 for HAD class and higher than 0.5 for EM, it is not meaningful to have so many decimal values for the predictions. This means that the range of values assumed by the weights and the predictions of the Python model are correctly interpreted by Vivado HLS.
The corresponding *ap_fixed* values used for the conversion from floating point to the fixed point precision for each layer are summarized in *Table 4.4*.

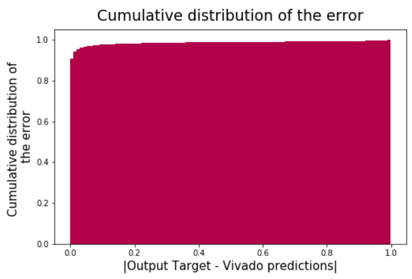
Table 4.4: Different ap_fixed used for different layers depending on the values assumed by the weights and by the activations of the Python model.

Layer	Total bits	Integer part
1st Conv 2D	18	9
1st SELU	15	6
1st max pooling	15	6
1st batch norm	15	6
2nd Conv 2D	15	6
2nd SELU	15	5
2nd max pooling	15	5
2nd batch norm	14	5
1st Dense	15	6
3rd SELU	14	5
3rd batch norm	18	4
2nd Dense	14	5
1st sigmoid	13	2

At this point, it is necessary to compare the predictions made by the Python model and those obtained after the C-simulation performed using Vivado HLS. The results are shown in *Fig. 4.22* and show that the conversion from the Python model into a C++ code is done in a correct way and there is no error between the two outputs.



(a) *Cumulative distributions of the absolute error between the predictions after the csim and the predictions made by the original Python mode.*



(b) *Cumulative distributions of the absolute error between the predictions after the csim and the target.*

Figure 4.22: Cumulative distributions of the absolute value of the output error between the Python model, the Vivado HLS output and the target.

The last thing that can be done to check the goodness of the model is to observe the ROC curve of the original Python model and the one obtained after the compression of this model into a C++ code. This comparison can be observed in *Fig. 4.23*. This picture shows that there is not a degradation of the performances of the network once it is compressed into a C++ code and it is synthesized on hardware, because the difference in terms of the AUC are of the order of 10^{-4}

and their ratio is equal to 99.94%.

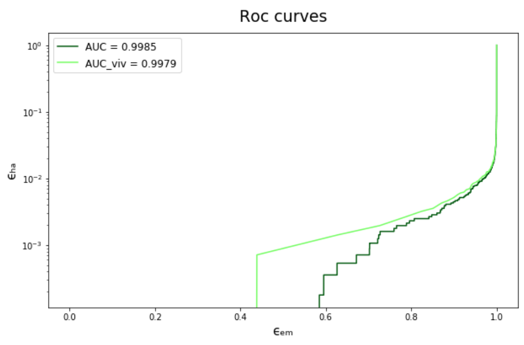


Figure 4.23: Comparison of the ROC curves of the original Python model and of the model obtained after the Vivado HLS csim.

At this point that the network can be implemented on FPGA, it is possible to observe the approximated number of resources and of the latency required by the hardware in order to implement this specific network. *Table 4.5* shows how many of each type of resources contained in the FPGA are required for the implementation compared to the available resources on the FPGA under consideration. The information that can be extracted from this table is that the number of LUT, even if it is only an approximation, is higher than the available resources on FPGA.

Table 4.5: Resource and latency usage when using as activation function the SELU. The latency is expressed in terms of number of clock cycle.

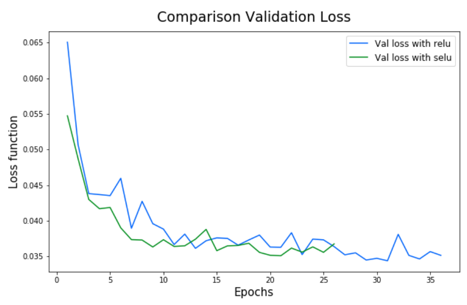
	BRAM	DSP	FF	LUT	Total Latency	Theoretical Latency
SELU	803	1449	87895	526458	3964	450
Available	1968	1968	1045440	522720		

In order to try to reduce the number of LUT, what can be done is changing the structure of the neural network architecture by simplifying it. In particular, it is possible to start by considering the activation function, that we selected to be the SELU. This activation function is not very simple to be implemented on hardware because, as already said in *Section 2.3.1*, it is defined as [35] :

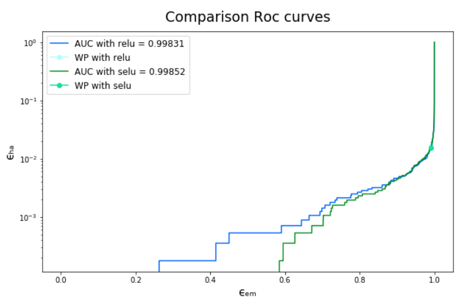
$$SELU(x) = \lambda \begin{cases} x, & \text{if } x > 0 \\ \alpha e^x - \alpha, & \text{if } x \leq 0 \end{cases}$$

This means that, when the values of x are negative, this function assigns to them very small numbers near to zero involving complicated operations to be implemented on hardware.
Provided that the behaviour of the SELU function is very similar to that of the ReLu and the only difference is that the ReLu assigns 0 to negative numbers, a reasonable consideration can be to substitute the current activation function for the first and second convolutional layers and

for the dense layer with the ReLu.
Once this is done, it is appropriate to check the degradation of the Keras model by evaluating the validation loss and the ROC curve and by comparing them with those obtained with the SELU as activation function (see *Fig. 4.24*). As it is possible to observe from the two pictures, the difference in terms of validation loss is not very significant, in particular the validation loss is better when using ReLu (0.0352) than when using SELU (0.0368), and the ratio between the two AUC is equal to 99.98%. This means that it is reasonable to substitute the SELU with the ReLu and check if in this case the number of resources will be reduced.



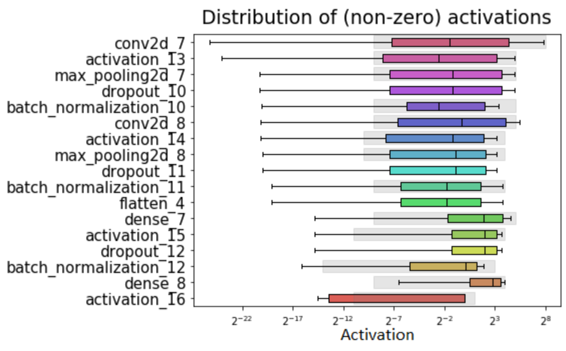
(a) *Comparison of the validation loss of the two Keras model when using the SELU as activation function, green curve, and the ReLu as activation function, blue curve.*



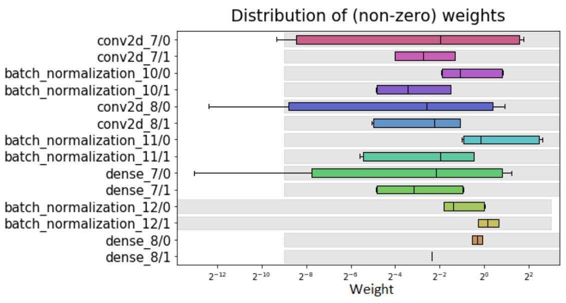
(b) *Comparison of the ROC curves of the two Keras model when using the SELU as activation function, green curve, and the ReLu as activation function, blue curve.*

Figure 4.24: Comparison of the Keras models when using SELU or ReLu as activation function, in terms of validation loss and ROC curve.

Also in this case, in order to perform the compression from the Keras model into the C++ code, we used different number of bits for each layer depending on the associated values of the weights and of the predictions. The plots to observe the range of the values assumed by the weights and the predictions for the Keras model and the ranges that are correctly interpreted by Vivado HLS are shown in *Fig. 4.25* and in *Fig. 4.26*. *Table 4.6*, instead, summarizes the values used for the conversion from the floating point into fixed point precision.

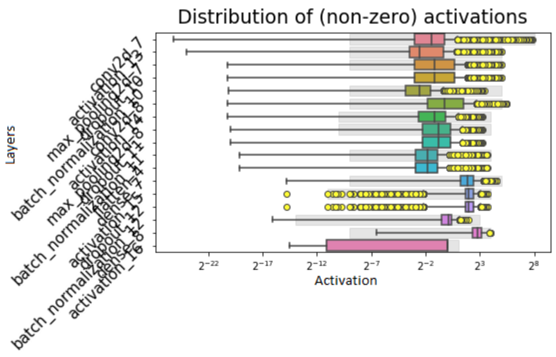


(a) *Distribution of the predictions.*

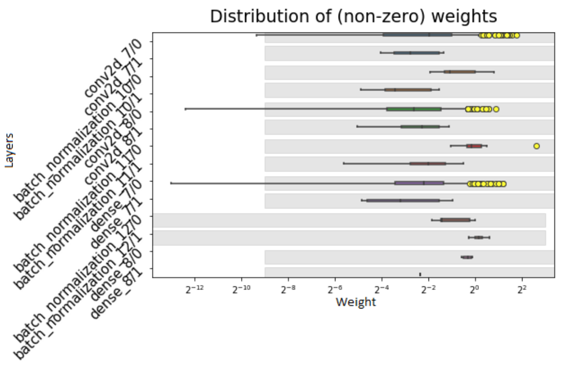


(b) *Distribution of the weights.*

Figure 4.25: Distributions of the predictions and of the weights, where the coloured rectangles represent the values assumed by the predictions and the weights of the original Python model and the gray rectangles represents the intervals inside which Vivado HLS is able to made a good conversion. The coloured rectangles are obtained using 1-th percentile and 99-th percentile as lower and upper quantiles.



(a) *Distribution of the predictions.*



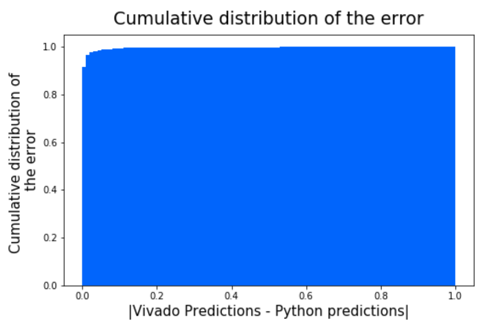
(b) *Distribution of the weights.*

Figure 4.26: Distributions of the predictions and of the weights, where the yellow circles (and also the coloured rectangles) represent the values assumed by the predictions and the weights of the original Python model and the gray rectangles represents the intervals inside which Vivado HLS is able to made a good conversion.

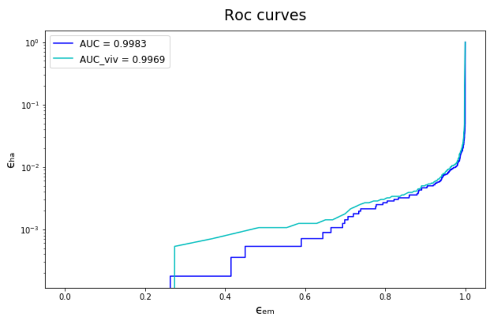
Now that the different numbers of bits are determined for each layer, what it can be done is (1) to compare, as it was done for the network with the SELU as activation function, the output of the original Keras model and the predictions made after the compression into a C++ code, and (2) to verify if the performances are kept (see *Fig. 4.27a* and *Fig. 4.27b* respectively). Observing these figures, it is possible to see either that there is not any error between the predictions of the original model and the predictions after the compression into a C++ code and also that there is not a significant loss in terms of performances, because the ratio between the AUC of the models is 99.86%.

Table 4.6: Different ap_fixed used for different layers depending on the values assumed by the weights and by the activations of the Python model.

Layer	Total bits	Integer part
1st Conv 2D	18	9
1st ReLu	15	6
1st max pooling	15	6
1st batch norm	15	6
2nd Conv 2D	15	6
2nd ReLu	15	5
2nd max pooling	15	5
2nd batch norm	14	5
1st Dense	15	6
3rd ReLu	16	5
3rd batch norm	18	4
2nd Dense	14	5
1st sigmoid	13	2



(a) *Cumulative distributions of the absolute error between the predictions after the csim and the predictions made by the original Python mode.*



(b) *Comparison of the ROC curves of the original Python model, blue curve, and of the model after the compression into a C++ code, light blue curve.*

Figure 4.27: Comparison of the predictions and of the ROC curve for the original Python model and for the model compressed into a C++ code and simulated with Vivado HLS.

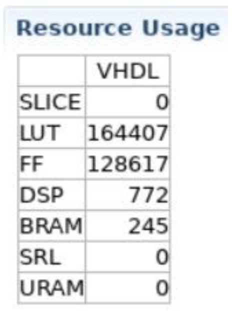
Having consolidated that substituting the SELU with the ReLu allows to have good performances also after the compression of the Python model, we can move to the synthesis process and observe the approximated number of resources, as well as the latency, required by the hardware to implement this network. *Table 4.7* summarizes the resources and the latency needed when implementing the network with the SELU and the ReLu, as well as the total number of them available on the FPGA of interest.

Table 4.7: Comparison between the resource and latency usage when using as activation function the SELU and the ReLu. The latency is expressed in terms of clock cycle.

	BRAM	DSP	FF	LUT	Total Latency	Theoretical Latency
SELU	803	1449	87895	526458	3964	450
ReLu	501	849	76709	506492	3959	445
Available	1968	1968	1045440	522720		

By comparing the number of resources when using ReLu and SELU it is possible to observe that using the ReLu allows to reduce the resources usage. The most interesting parameter to observe is the number of LUT, that in the case of the network with the SELU as activation function is higher than the total number of LUTs available on the FPGA. When using the network with the ReLu as activation function, instead, the required number of LUTs is smaller than the total available LUTs on the FPGA. This result suggests to keep the ReLu and to avoid the use of the SELU to guarantee a more reasonable resources usage.

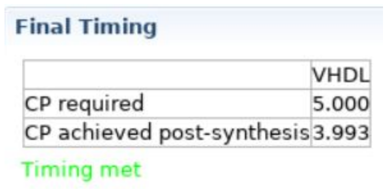
Once we have found a suitable neural network that can be implemented on FPGA and that gives good performances also after the implementation, in order to know exactly the number of resources and the latency necessary for its implementation, it is possible to extract, by using Vivado HLS, the IP core of the project and use it in a tool called Vivado. With this tool we are able to obtain the real number of resources and the real clock period required by the circuit. The final number of resources and the clock period, obtained with Vivado, are shown in *Fig. 4.28.*



Resource Usage

	VHDL
SLICE	0
LUT	164407
FF	128617
DSP	772
BRAM	245
SRL	0
URAM	0

(a) *Resources usage after performing the synthesis with Vivado.*



Final Timing

	VHDL
CP required	5.000
CP achieved post-synthesis	3.993

Timing met

(b) *Timing reached after the synthesis with Vivado.*

Figure 4.28: Resources usage and Latency.

As it is possible to observe from *Fig. 4.28* all the resources required for the implementation are very different from the approximated values extracted with Vivado HLS and are smaller than the total available resources on the FPGA of interest and the timing after the synthesis match the constraint of 5 ns reaching 3.99 ns. This means that 5 ns could be used as a clock period and the final latency of the circuit will be around 2.2 μs [2]. This number is obtained considering the latency for the pooling layers of the same order of magnitude of the ones of the other layers with similar size assuming a parallel implementation also for the pooling operations. However, with the current hls4ml implementation of the pooling operations, and as a consequence with a sequential implementation, the latency is always too big and of the order of 19.8 μs.

4.3 Summary

The purpose of this chapter was to introduce the methods used during this project to implement the ANN on FPGA using hls4ml and Vivado HLS. After having analysed the implementation of a simple ANN, the FNN, the batch normalization after each layer and the input normalization were added to the model described in *Chapter 3*. Then, before implementing this model, a simple architecture for the CNN was used in order to guarantee a correct output after the compression into a C++ code also when a more complicated model (CNN) is considered. This was done by adding two parameters, in the conversion, called AP_RND and AP_SAT. Once these simple studies were performed we understood that the selected network was too big to be converted into an HDL code using Vivado HLS.
For this reason different images size and different architectures with different number of filters and neurons were taken into consideration. When discovered the optimal image size ($9 \times 9 \times 3$) and the optimal architecture (sl1=5, sl2=15 and sl3=8, with SELU as activation function), that allow to keep the performances unchanged, we moved to the implementation of this network on FPGA, performing the conversion into a C++ code, the C-simulation and the synthesis in order to generate the HDL code. By observing the resources and latency after the synthesis we discovered that the use of a different activation function, such as the ReLu, allows to reduce the number of employed resources and, as a consequence, it has been fixed as the activation function for both the convolutional layers and the first dense layer. Finally, we used the Vivado tool to observe the real number of resources and the real clock period required for the implementation of this network on FPGA. As a conclusion we found that the network with sl1=5, sl2=15, sl3=8, with the input and batch normalization, with the ReLu as activation function for the convolutional layers and for the first dense layer and the binary cross-entropy as loss is the most suitable network for this particles classification problem. In fact, this network reaches very good performances before the implementation (AUC = 0.9983 and a validation loss = 0.0352) and allows to keep them once implemented on FPGA. It can also be implemented on FPGA with a number of resources that is sufficiently small for the implementation on the FPGA of interest (xcku15p-ffve1760-2-e) reaching a clock period of 3.99 ns and as a consequence a possible latency of 2.2 μs that is reasonable for the analysis of the particles on the output of the detector because, as we mentioned in *Section 1.1*, the constraints for this type of analysis are of the order of few microseconds.
However, when using this approach we are able to use at minimum 13 bits for few layers and, for all the other layers, a higher number of bits is required (as shown in *Table 4.6*). It could be very interesting to try to reduce the number of bits necessary for the conversion from floating

[2]This number is obtained by multiplying the latency in terms of the number of clock cycles, 445, with the fixed clock period equal to 5 ns.

point into a fixed point precision in order to see if the overall resources can be reduced more. This reduction of the resources could be of interest when the parallel implementation of the pooling layer operations will be supported by hls4ml. This is because, as already said, the parallelization of the operations implies either a reducion of the latency, making it suitable for this specific classification problem, but also an increase in terms of resources. This last effect could impact the feasibility of the hardware implementation if the number of resources required to implement the entire circuit becomes higher than those available.
In order to reduce the number of bits for the conversion and, as a consequence, the resources usage, it is possible to consider a different approach and, more specifically, a new Python library called QKeras that allows to quantize the weights and the biases of the network during the training process by using the so called quantized layers and quantized activations. This approach will be described in the next chapter.

Chapter 5

Quantized Neural Networks

In the previous chapter we have seen a methodology for the implementation of an ANN on FPGA based on the traditional training of the network and a subsequent quantization of the values of the weights and of the biases for each layer during the conversion from the floating point format of the Python model to the fixed point precision suitable for the hardware implementation. Considering this methodology we observed that it is impossible to significantly reduce the number of bits associated to the weights and to the biases. In fact, the minimum number of bits used for the conversion was 13.
In this chapter we are interested in exploring a new methodology where the weights and the biases will be quantized before the training process by using the so called quantized layers. These layers are defined in a library called QKeras [57]. This library allows to perform different types of quantizations such as exponent quantization, binary or ternary quantizers. The idea of using QKeras is to quantize some specific layers of the network. These layers are those that handle the data and perform the so called input data type changes, such as Dense or Convolutional layers or the Activation layers. All the other layers that instead manipulate the data without modifying their type, but ordering them in a different way, are kept unchanged and as a consequence cannot be quantized using this library. This last type of manipulation where the data are reordered is in general performed by layers such as the Max Pooling and Flatten. As a matter of fact, in order to quantize a model it is necessary to substitute the dense layers, the convolutional layers and the activation layers with the quantized dense (QDense) layers, the quantized convolutional (QConv2D) layers and the quantized activation (QActivation) layers respectively. It is worth mentioning that it is not necessary to quantize the entire model, it is also possible to quantize only the layers of interest. In particular, for the QDense layer and for the QConvolutional layer it is possible to specify two parameters called *kernel_quantizer* and *bias_quantizer* that allow to specify the type of quantization to be applied to the weights and the biases respectively.

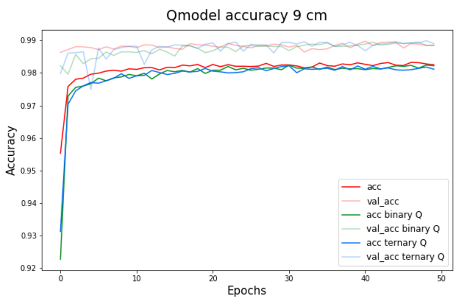
In this specific study we observed that the first convolutional layer requires a high number of resources. The first point of interest is the quantization of the first dense layer to reduce its associated resources and be able to retain more resources for the convolution operations. In this way, it would be possible to see if the total number of resources necessary for the implementation will be reduced. In addition, another reason to apply binary or ternary quantization only to one of the last layers is to avoid having errors propagating through the entire network. For these reasons, *Section 5.1* will describe two different networks with only the first dense layer quantized using in one case the binary quantization (-1 and 1) and in the second case the ternary quantization (-1, 0, 1) for the weights and the biases. These two networks will be compared with each other and with a non-quantized network in terms of Keras performances and also in terms of resources usage after the FPGA implementation. Then, another interesting thing is to observe how the performances of the network change when the entire model is quantized. In order to do that, in *Section 5.2* three quantized models with binary quantization, ternary quantization and a generic quantization (where it is possible to specify how many bits to use) will be considered and compared in terms of performances.

5.1 Quantization of the first dense layer

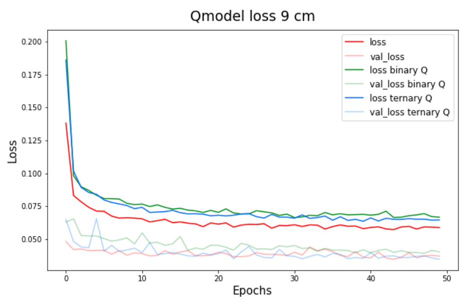
In this section the network with the structure defined in *Section 3.5* and with the modification, described in *Chapter 4*, necessary to guarantee the correct implementation on FPGA (such as sl1=5, sl2=15, sl3=8, input and batch normalization, a binary cross-entropy as a loss and a ReLu as activation function) will be considered.
The first thing that can be done is the quantization of only the first dense layer. For this purpose the dense layer characterized by 8 neurons will be substituted with the QDense layer with 8 neurons, the corresponding activation function will be instead substituted with the corresponding quantized ReLu[1] and two quantized model will be taken into consideration. The first model, that will be called for simplicity *binary model*, is characterized by the QDense layer with a binary quantization (-1,1) for its weights and biases while the second model, called for simplicity *ternary model*, is characterized by the QDense layer with the ternary quantization (-1, 0, 1) for its weights and biases. These two models were trained[2] and validated and their performances comparison is shown in *Fig. 5.1* and is summarized in *Table 5.1*. These results suggest that when only one layer of the model will be quantized there will not always be a degradation in terms of performances. In fact, considering the binary model there is a little degradation while considering the ternary model the performances are as good as the ones of the model where there is not any quantized layer, the loss is smaller and the accuracy is higher of a factor of 10^{-3}, the AUC is higher of a factor of 10^{-4} and the WP_99 is higher of a factor of 10^{-3}. It is important to notice that the value of the validation loss for the not quantized model is not in contradiction with the result obtained in *Section 4.2.3* because, as already said, when training a neural network we have a randomness component that propagates in the performances of the network.
Once understood that the performances of the network with one quantized layer do not present significant degradation and that in the case of the ternary model the performances are kept unchanged, it is possible to check what is the effect of this quantized layer in terms of resources usage and latency when implementing this network on hardware. Also in this case the same

[1]The quantized ReLu implemented on the QKeras library is a function that performs the ReLu and the quantization to a certain number of bits. More specifically it is possible to use a sigmoid to the inputs to normalize them or alternatively it is possibile to not normalize them and keep numbers up to $2^{integer} \cdot (1-2^{-bits})$ where integer stays for the number of bits associated to the interger part of the number and bits stays for the number of bits to perform quantization (`https://github.com/google/qkeras/blob/287e3ee8bd61fe1dd890fd6f3920e859842cf176/qkeras/quantizers.py#L1124`).

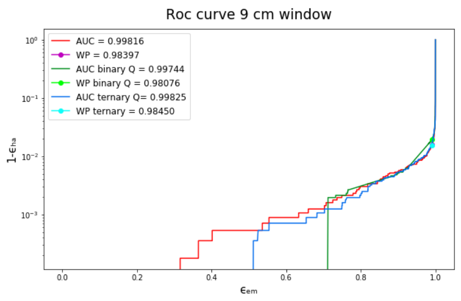
[2]In this case the tool INNATE was not used for the training because it does not support the QKeras library yet.



(a) *Comparison in terms of the Accuracy.*



(b) *Comparison in terms of the Loss.*[3]



(c) *Comparison in terms of the ROC curve.*

Figure 5.1: Comparison of the performances of the not quantized model, the binary model and the ternary model in terms of loss, accuracy and ROC curve.

Table 5.1: Comparison between the Keras performances when using different type of dense layer: not quantized, binary or ternary.

Type of Dense Layer	Val Loss	Val Accuracy	AUC	WP_99
Not Quantized	0.03731	0.98851	0.99816	0.98397
Binary	0.04058	0.98839	0.99744	0.98076
Ternary	0.03495	0.98900	0.99825	0.98450

FPGA (xcku15p-ffve1760-2-e), the same Reuse Factor (RF) equals to 15 and the same strategy of using different number of bits for each layer depending on their weights and predictions will be used. After the conversion of the python code into a C++ code, the C-simulation and a subsequent synthesis to generate the HDL code were performed. In this way the approximated number of resources and the approximated latency for each model are estimated (using Vivado HLS) and reported in *Table 5.2*. What it can be observed from this comparison is the decreasing number of BRAM, DSP and FF resources necessary to implement the quantized model compared to those necessary to implement the traditional Keras model. However, the number of LUT seems to increase and in order to verify the exact number of resources usage, as done in *Section 4.2.3*, the IP core can be exported using Vivado HLS and it can be used to synthesize the model with Vivado and to extract the real number of resources and clock period for each model (see *Table 5.3*).

Table 5.2: Comparison between the *approximated* number of resources and the approximated latency (expressed in number of clock cycle) when using different types of dense layer: not quantized, binary or ternary.

Type of Model	BRAM	DSP	FF	LUT	Total Latency	Theoretical Latency
Not Quantized	501	849	76709	506492	3959	445
Binary	379	820	75477	538928	3957	444
Ternary	379	813	72702	524153	3958	445
Available	1968	1968	1045440	522720		

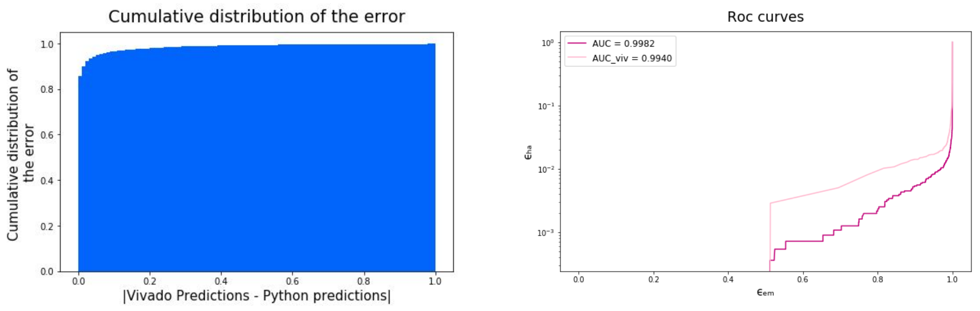
Table 5.3: Comparison between the *real* number of resources and the clock period when using different types of Dense layer: not quantized, binary or ternary.

Type of Model	BRAM	DSP	FF	LUT	Clock Period
Not Quantized	245	772	128617	164407	3.99 ns
Binary	191	767	128514	131858	3.72 ns
Ternary	191	687	125254	129997	3.70 ns
Available	1968	1968	1045440	522720	5 ns (constraint)

Comparing the real number of resources it is possible to see that the global effect of quantizing one layer of the model is to reduce the number of resources usage and to reduce the clock period required by the hardware. The fact that by quantizing only one layer of the network it is possible to keep unchanged the performances of the network and that it is possible to reduce the number of resources used, is of great interest for this study. It indeed suggests that this method is promising to further reduce the resources usage and the clock period required by the hardware used for the implementation. Finally, once understood that between the binary and ternary models the one that is more promising to be used instead of the non-quantized model is the ternary model (due to its better performances and smaller clock period and resources

[3]In this case the validation loss is not exactly equal to that obtained in the previous chapter (*Fig. 4.24a*) and it is around 0.037 because the network has been run another time and there is a variability in the model. This run was necessary because we need to train the network without INNATE due to its incompatibility with the QKeras library.

usage), it is possible to check if the predictions of the model before the compression into the C++ code match those made after the C-simulation (see *Fig. 5.2*). *Fig. 5.2* shows that the error in terms of predictions is almost zero and the ratio between the AUC before and after the compression of the model into a C++ code is 99.58%. Consequently this type of network, with only the dense layer quantized with a ternary quantization of the weights and of the biases, can be implemented on FPGA without loosing the performances and reducing both the resources usage and the latency.



(a) *Distribution of the absolute value of the error between the predictions made by the original python model and the compressed model after the C-simulation.*

(b) *Comparison in terms of the ROC curves, in light pink the results after the C-simulation while in magenta the results of the original python model.*

Figure 5.2: Comparison of the performances of the ternary model before and after the compression into a C++ code in terms of predictions and ROC curves.

This result is of relevant interest for the implementation on FPGA and the associated real time classification task. In fact, after this analysis, it is reasonable to think that by quantizing more than one layer it will be possible to implement on FPGA the quantized network reducing more the resources usage and the related latency. For this reason, in the next sections we will focus on the study of the quantization of the entire model.

5.2 Quantization of the entire model

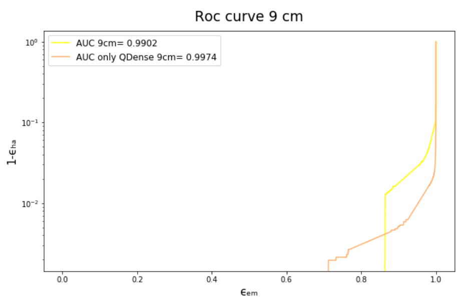
Now that we have demonstrated the astonishing potential of applying the quantization before the conversion into a C++ code and so directly during the training, it is possible to increase the number of quantized layers in the model and check, also in this case, the performances degradation and the changing in terms of resources usage and latency in the hardware implementation. In order to do that, we decided to consider first, in *Section 5.2.1*, two models: one with all the layers quantized, i.e. both the first dense layer and the two convolutional layers (with the associated activation functions) and a binary quantization for the weights and the biases, called *complete binary model* for simplicity, and the other one with all the layers quantized (with also the associated activation functions) and a ternary quantization for both the weights and the biases, called *complete ternary model*. Then, in *Section 5.2.2* we will consider the models used in the previous section (*Binary model* and *Ternary model*), but in this case we quantize also the convolutional layers using 4 bits for the quantization of their weights and biases.

5.2.1 Models with all the layers quantized with the same quantization: Binary or Ternary

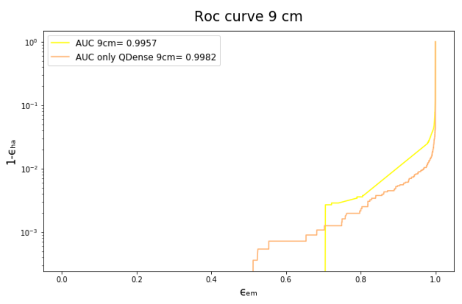
Starting from the models described in *Section 5.1* we decided to quantize not only the first dense layer but also the two convolutional layers and their associated activation functions. In this case both the convolutional layers, with 5 and 15 filters respectively, will be substituted with QConv2D layers with 5 and 15 filters respectively and with the same quantization as the one used for the Qdense layer for the parameters *kernel_quantizer* and *bias_quantizer*. Consequently, also in this case we consider two different models:

- The first one with the binary quantization for the kernel_quantizer and bias_quantizer for each layers (*complete binary model*);
- The second one with the ternary quantization for the kernel_quantizer and bias_quantizer for each layers (*complete ternary model*).

It is worth mentioning that the output layer is not quantized. This is because we are interested in comparing the performances of the model and, as a consequence, it is more significant to keep on the output values between 0 and 1, such as 0.99, and not only 0 and 1. Once these models are trained, we compared them with the corresponding model with only the dense layer quantized. The comparisons in terms of performances are shown in *Fig. 5.3a* for the binary quantization and in *Fig. 5.3b* for the ternary quantization.



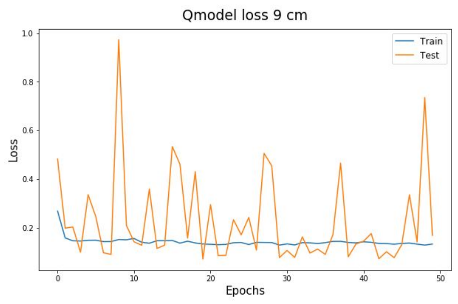
(a) *Comparison when using binary quantization for the weights and the biases for each layer of the QKeras model except for the last one.*



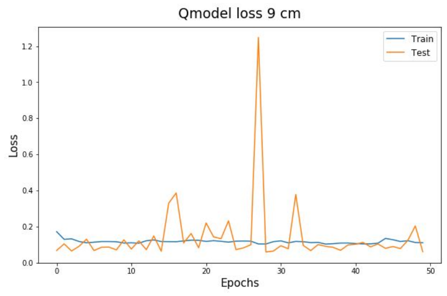
(b) *Comparison when using ternary quantization for the weights and the biases for each layer of the QKeras model except for the last one.*

Figure 5.3: Comparison of the performances in terms of the ROC curve when using only one layer quantized and when using all the layers of the model quantized except the output layer.

Figure 5.3 shows that the performances are kept when using only one or all the layers as quantized in terms of the ROC curve. In the case of binary quantization for the weights and the biases the ratio between the two curves is 99.28% while for the ternary quantization is 99.75%. However, observing the behaviour of the validation loss vs. the number of epochs for the *complete binary model* and for the *complete ternary model* it is possible to observe a lot of fluctuations and it seems that the models are not learning in the correct way, because the value of the loss does not become smaller as the number of epochs increases, see *Fig. 5.4*.
In fact, the values of the validation losses are higher than the cases where only one layer was quantized and are 0.16857 for the *complete binary model* and 0.06033 for the *complete ternary*



(a) *Binary quantization.*



(b) *Ternary quantization.*

Figure 5.4: Representation of the validation loss vs. the number of epochs when using all the layers of the model quantized, except the output layer, with the binary and the ternary quantization for the weights and the biases.

model. This means that we have a degradation of the validation loss that is of the order of 10^{-2} compared to the results obtained with only the dense layer quantized. This result suggests that this type of quantization for the first and second quantized convolutional layers is not enough to keep the performances of the model and, as a consequence, it is better to increase the number of bits for the quantization of the weights and of the biases in the QConv2D layers. This can be done by using, for example, 4 bits for the quantization. However, this type of quantization is not yet supported by hls4ml for the conversion of the Python model into a C++ code and thus, in order to see if we are going in the correct direction, we can observe how many resources and latency are required for the hardware implementation of the model where all the layers (except the last one) are quantized, even if we know that the performances are not kept.
For this purpose we compressed the two models (*complete binary model* and *complete ternary model*) using hls4ml, then the generated C++ codes were simulated and synthesized by using Vivado HLS and finally, always using Vivado HLS, the IP core was exctracted and used in Vivado in order to obtain the real number of resources and clock periods (see *Table 5.4*).

Table 5.4: Comparison between the *real* number of resources and the approximated latency when using different types of dense and convolutional layers: not quantized, binary or ternary.

Type of Model	BRAM	DSP	FF	LUT	Clock period
Not Quantized	245	772	128617	164407	3.99 ns
Complete Binary	191	88	117355	125347	3.58 ns
Complete Ternary	191	113	117849	126073	3.76 ns
Available	1968	1968	1045440	522720	5 ns (constraint)

What can be observed in this case is that the number of DSP, FF, LUT resources is reduced compared to the case where only one layer was quantized. The clock period instead always meets the constraint of 5 ns, it is almost kept unchanged and does not vary significantly. These results suggest that when all the layers are quantized the resources can be reduced more than the case where only the dense layer is quantized. Consequently, it is reasonable to try to use one bits more for the weights and biases of the quantized convolutional layers in order to see if the performances will be kept in this case.

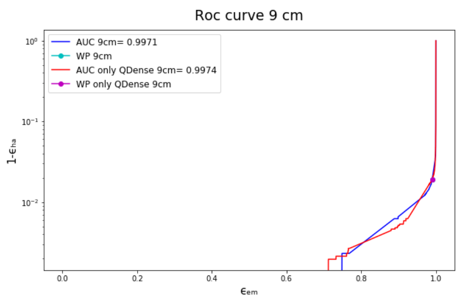
5.2.2 Models with all the layers quantized with different quantizations for dense and convolutional layers

The last thing that can be done to keep small resources and latency is to change the type of quantization for the weights and the biases of the QConv2D layers. This means that we will always consider the binary quantization or the ternary quantization for the weights and the biases of the quantized dense layer and, in addition, we will consider 4 bits for the quantization of the weights and of the biases of the two QConv2D layers and for the quantization of their associated activation functions. Moreover, the last layer is still not quantized in this case, in order to allow the comparison in terms of performances. The two models in this case will be:

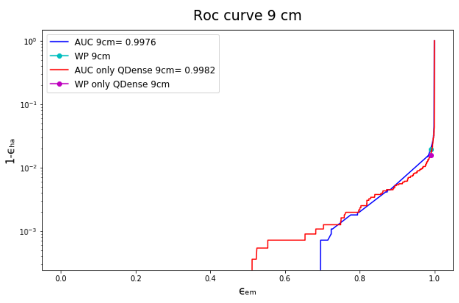
- The first one with the binary quantization of the weights and biases for the quantized dense layer, 4 bits for the quantization of the weights and the biases of the quantized convolutional layers and 4 bits for the quantization of the activation functions associated to the QConv2D layers (*Binary 4 bits quantization*);

- The second one with the ternary quantization of the weights and biases for the quantized dense layer, 4 bits for the quantization of the weights and the biases of the quantized convolutional layers and 4 bits for the quantization of the activation functions associated to the QConv2D layers (*Ternary 4 bits quantization*);

Once the two models are defined, the only thing that can be done is the comparison of the performances with the case where only the dense layer was quantized, because the quantization with a specific number of bits, as already said, is not supported by hls4ml. The comparison of the performances in terms of ROC curve between the *Binary 4 bits quantization* model and the *Binary model* and between the *Ternary 4 bits quantization* model and the *Ternary model* are shown in *Fig. 5.5a* and in *Fig. 5.5b* respectively.
More precisely, the ratio between the ROC curves for the *Binary 4 bits quantization* and the *Binary model* is 99.97% while for the *Ternary 4 bits quantization* and the *Ternary model* is 99.94%. This suggests that both the models present very good performances. Moreover, the validation loss for these models presents a more correct behaviour and decreases when the number of epoch increases reaching 0.0408 for the *Binary 4 bits quantization model* and 0.0539 for the *Ternary 4 bits quantization* model (see *Fig. 5.6*).
As a conclusion, when adding one more bit to the quantized convolutional layers for the quantization of the weights and of the biases, the performances of the models are good and comparable with those obtained with the non-quantized models and with the models with only the dense layer quantized. This suggest that quantizing the network directly from the training, by using the QKeras library, is a very promising approach to reduce the number of resources usage and the latency for the classification of EM and HAD particles problem.

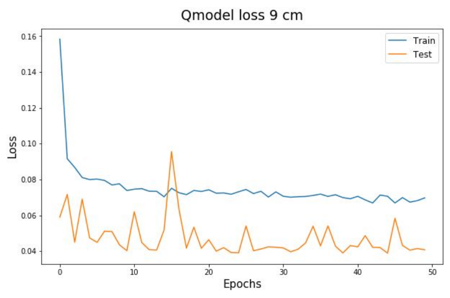


(a) *Comparison when using binary quantization for the weights and the biases for the quantized dense and 4 bits for the quantized convolutional layers of the QKeras model except for the last one.*

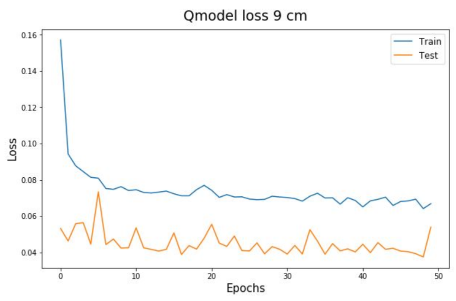


(b) *Comparison when using ternary quantization for the weights and the biases for the quantized dense and 4 bits for the quantized convolutional layers of the QKeras model except for the last one.*

Figure 5.5: Comparison of the performances in terms of the ROC curve when using only one layer quantized and when using all the layers of the model quantized except the output layer.



(a) *Binary quantization.*



(b) *Ternary quantization.*

Figure 5.6: Representation of the validation loss vs. the number of epochs when using all the layers of the model quantized (except the output layer) with the binary and the ternary quantization for the weights and the biases of the QDense layer and using 4 bits for the quantization of the weights and of the biases for the QConv2D layers.

The implemented network is not an approximate quantization of the optimal network but rather the optimal quantization network. This procedure allows to reduce the resources usage in the most natural way.

5.3 Summary

In this chapter we have seen a different approach for the quantization of the ANN. Instead of quantizing the model only during the compression from the Python model into the C++ code, e.g. once it is already trained and validated, it is possible to use the quantized layers and to specify the type of quantization for their weights and for their biases.
This type of quantization allows to reduce the number of bits for the conversion of these layers and consequently to reduce the number of resources usage and the latency when implementing the model on FPGA. Approximately, the higher the number of layer that will be quantized, the

lower the number of resources usage and the latency. From this analysis, we have observed that for this specific particles classification problem, when only the dense layer was quantized, the performances of the network were kept, in particular when the ternary quantization was used for the kernel_quantizer and for the bias_quantizer. In addition we observed that, when all the layers were considered as quantized layers, the ternary and binary quantization for the weights and the biases of the quantized convolutional layers are not enough to guarantee a model with high performances and the reasonable number of bits that can be used is rather 4.
However, with the current version of hls4ml it is not possible to convert the QKeras model with a specific number of bits for the weights and the biases into a C++ code and, subsequently, it was not possible to test if the resources and the latency would be effectively reduced. This last study, related to this new type of quantization using QKeras, should be a starting point to explore more the quantization of different layers with a specific number of bits for the weights and the biases and to verify, once hls4ml will be updated to support this conversion, the effective number of resources usage and the latency.

Chapter 6

Conclusions

The purpose of this study was to exploit an alternative method based on deep neural networks for the classification of EM and HAD particle on the output of the first stage of the HGCAL Trigger Primitive Generator (TPG) in the Level-1 trigger system. In this chapter I will provide the obtained results and some possible future works that might be done to improve this study.
The final result of this project is the confirmation of the possibility to use a new innovative approach, based on Deep Neural Networks, for the real time analysis of the particles generated during the proton-proton collisions in the LHC.
I have first shown how the use of a simple Fully Connected Neural Network, able to learn the relationship between input features and a target output, can be promising to substitute the traditional methods adopted for the classification of EM and HAD particles, based on hand-made algorithm such as Boosted Decision Trees on the output of the stage 2 of the TPG. In fact, we have found that using this architecture without any detailed optimization allows to reach optimal performances, i.e. an accuracy of the order of 99.56%.
Then, I have shown that the Convolutional Neural Network is a promising neural network architecture that could be implemented on existing hardware (FPGA) and could be used after the stage 1 of the HGCAL TPG in the Level-1 trigger by substituting the stage 2, that consists in a hand-made clustering algorithm able to reconstruct and to classify clusters of energies with a constraint of few microseconds dictated by the rate of the collisions. In fact, the main information that can be extracted from this analysis is that, after a careful optimization of the hyper-parameters of a Convolutional Neural Network using different techniques such as the hand-tuning followed by a Bayesian Optimization, it is possible to find the optimal architecture for this specific particle classification task. The optimality of the network that is reached is not only in terms of performances, but also in terms of resources usage and latency required for the implementation on the FPGA (see the first row of *Table 6.1*). The found network allows to reach an **accuracy** of **99.69%** after the implementation and, for what concerns its latency, we have found that it is of the order of $19.8\mu s$.

Table 6.1: Resource and latency usage. The latency is expressed in terms of number of clock cycle.

	BRAM	DSP	FF	LUT	Total Latency	Theoretical Latency
Standard NN	245	772	128617	164407	3959	445
Ternary	191	687	125254	129997	3958	445
Available	1968	1968	1045440	522720		

However, we observed that most of the latency (3500 clock cycles) came from the pooling layers. It is an unexpected result due to the fact that the pooling operation is a simple evaluation of the maximum, an easier operation than the matrix multiplication.
Observing the hls4ml implementation, we discovered that the pooling layer is not implemented in the same way as all the other layers. In particular, its operations are kept serialized and it is not possible to parallelize them with the current available version of hls4ml. For this reason, we have also taken into account the pooling layers by considering that their operations were parallelized as done for all the other layers and we associated to them a latency of the same order of magnitude (20 clock cycles) of the layers with similar size. By doing this considerations, it was possible to find a **latency** that is more reasonable, of the order of $\mathbf{2.2\mu s}$. This number seems very promising for the real time classification of the particle on the output of the L1-trigger and, for this reason, it could be of interest to study in more details the parallel implementation of the pooling layer operations using hls4ml in terms of future works.
On the other hand, I also exploited a new type of artificial neural network where some of the layers or all the layers have been already quantized before the training process. This approach seems interesting when the network has to be implemented on FPGA. In fact, we observed that by quantizing just one dense layer it is possible to have a reduction of the resources needed on FPGA while keeping the performances and the latency of the network unchanged if compared to the traditional neural network model (see the second row of *Table 6.1*).
Starting from this consideration, we decided to explore the quantization of all the layers of the network to further reduce the resources. Along this dissertation, we observed that, in order to not loose the performances of the network when quantizing all the layers, it was necessary to use more than 3 bits for the quantization of the convolutional layers (more precisely, 4 bits).
However, the implementation of this type of network quantization using hls4ml is still in progress and at the moment allows to consider only the binary and the ternary quantization. For this reason, we were not able to perform the implementation of the network with also the convolutional layers quantized. This last observation suggests to explore in more details this new type of artificial neural network in the future.

In addition to these results, this thesis also provides a definition and a practical use of a well defined methodology for the optimization of deep neural networks that takes into consideration the trade-off constraint between the software performances and the resources and latency usage imposed by the hardware implementation.

Furthermore, this document involves the testing of the innovative tool provided by the LLR laboratory to speed up the training process, called INNATE. The latter is used during all the training processes in the present project, showing its effective functioning with either different Neural Network architectures and optimization techniques.

6.1 Future works

Following the results obtained in this study, there are still some further investigations that could be considered:

- **Parallel implementation of the pooling operations using hls4ml**
  During the implementation of the neural network on FPGA we observed that the pooling layer is not synthesized in the same way as the other layers and, in particular, it is synthesized in a sequential way. We expect that it could be possible to reduce a lot the time required by the hardware associated to this layer when considering a parallelization of its operation during the synthesis made by hls4ml;

- **Synthesis of QKeras layers characterized by a generic number of bits using hls4ml**
  We observed that the quantization of the convolutional layers with a number of bits higher than three allows to keep the performances of the quantized network comparable with respect to the traditional ones. We expect that, due to the less number of bits (4) required by the quantized convolutional layers compared to those needed when converting traditional NN models, the resources usage once implementing this type of quantized network on FPGA will be smaller.

Appendix A

Optimizers algorithms

A.1 SGD

The Stochastic Gradient Descent (SGD), [58], is a method used to evaluate the minimum of a specific type of function. It is similar to the Gradient Descent that is based on the evaluation of the full gradient during each iteration that will be used to define the direction in the space where we have to go in order to find the minimum. However, the SGD does not evaluate the full gradient for each iteration but it evaluates only a certain number of samples, reducing the computational cost of performing this operation. Let us consider a convex minimization problem where the function $f(x)$ to be minimized is proper, closed and strongly-convex:

$$f^* = \min_{x \in \mathbb{R}^p} f(x)$$

The algorithm of the SGD can be written as follow:

1- Select a starting point x_0 and the value of a parameter, γ_t that define the step size called learning rate;

2- For k = 0,1,.., N-1, where N is the number of the iteration, perform:

$$x_{t+1} = x_t - \gamma_t \hat{g}_t$$

where γ_k is a positive number and $\hat{g}_t$ is an unbiased estimate of the full gradient: $\mathbb{E}[\hat{g}_t] = \nabla f(x_t)$. The main disadvantage of this algorithm is the fact that the learning rate is fixed.

A.2 Adagrad

The Adaptive Gradient Algorith [59], called Adagrad, is a method where the learning rate is adapted at each iteration depending on the value assumed by the past gradients. Hence, the main advantage, compared to the SGD, is that the update of the learning rate is done in an adaptive way. What it is possible to say in general is that with high global learning rate this method requires more iterations to converge than SGD due to the fact that the step size decreases. Instead, if the global learning rate decreases, this method starts to have better accuracy and lower losses with respect to the stochastic methods due to the fact that it does not keep the step

size, i.e. the learning rate, constant but it updates the learning rate depending on parameters. Its main disadvantage is that the global learning rate is divided by a factor that depends on the sum of the squares of the past gradients (element-wise product between past gradients) and so the learning rate, at each step, decreases of a quantity that increases. For this reason, when the global learning rate is too slow it does not perform well as the other adaptive methods because it requires a large number of iterations to converge.
The algorithm of the Adagrad can be written as follow [59]:

1- Initialize x_0, r=0;

2- For t = 0,1,.., N-1, where N is the number of the iteration, perform:

$$r = r + \hat{g}_t \odot \hat{g}_t$$

$$x_{t+1} = x_t - \frac{\gamma_t}{\delta + \sqrt{r}} \hat{g}_t$$

where $\hat{g}_t$ is an unbiased estimate of the full gradient, γ_t is the global learning rate, δ is the damping coefficient and τ is the decaying parameter.

A.3 RMSprop

The Root Mean Square Propagation (RMSProp) algorithm is very similar to AdaGrad and adds an exponentially decaying average of the squared gradient that allows to remove the problem that appears in AdaGrad, that implies a significant reduction of the learning rate at each iteration. The RMSProp allows to have a redistribution of the weights between the past gradients and the squared gradient. So it has the same advantages of AdaGrad with respect to the stochastic methods but it adds the advantage, previously mentioned, compared to AdaGrad. The algorithm of the RMSprop can be written as follow [60]:

1- Initialize x_0, r=0;

2- For t = 1,.., N-1, where N is the number of the iteration, perform:

$$r = \tau r + (1-\tau)\hat{g}_t \odot \hat{g}_t$$

$$x_{t+1} = x_t - \frac{\gamma_t}{\delta + \sqrt{r}} \hat{g}_t$$

where $\hat{g}_t$ is an unbiased estimate of the full gradient, γ_t is the global learning rate, δ is the damping coefficient and τ is the decaying parameter.

A.4 Adam

The Adaptive Moment Estimation (Adam) algorithm is a kind of combination of two already mentioned methods: the Adagrad and the RMSprop. It is an improvement compared to the others mentioned methods. It is very similar to RMSProp because it still adds the exponentially decaying average of the squared gradient but it also adds the exponentially decaying average of

the gradient. So it combines the RMSProp and the Momentum. In this way when the global learning rate is low it should be better than the other methods because it reaches higher accuracy after a small number of iteration and it is able to escape from local minimum (as the case of momentum). Instead, when the global learning rate is too high it does not perform well and the stochastic methods can present better accuracy.
The algorithm of the Adam can be written as follow [40]:

1- Initialize x_0, m_1=0, m_2=0;

2- For t = 0,1,.., N-1, where N is the number of the iteration, perform:

$$m_1 = \beta_1 m_1 + (1-\beta_1)\hat{g}_t$$

$$m_2 = \beta_2 m_2 + (1-\beta_2)\hat{g}_t \odot \hat{g}_t$$

$$m_1 \leftarrow \frac{m_1}{1-\beta_1^{t+1}} \qquad m_2 \leftarrow \frac{m_2}{1-\beta_2^{t+1}}$$

$$x_{t+1} = x_t - \gamma_t \frac{m_1}{\delta + \sqrt{m_2}}$$

where $\hat{g}_t$ is an unbiased estimate of the full gradient, γ_t is the global learning rate, δ is the damping coefficient, β_1 is the first order decaying parameter and β_2 is the second order decaying parameter

A.5 Adamax

Adamax algorithm is a variant of the Adam algorithm in which the norm l2 of the past gradients (also written as $\hat{g}_t \odot \hat{g}_t$) is substituted with the infinity norm [40].
The Adamax algorithm can be written as follow [40]:

1- Initialize x_0, m_0=0, u_0=0;

2- For t = 1,.., N-1, where N is the number of the iteration, perform:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\hat{g}_t$$

$$u_t = max(\beta_2 u_{t-1}, |\hat{g}_t|)$$

$$x_t = x_{t-1} - \frac{\gamma}{1-\beta_1^t}\frac{m_t}{u_t}$$

where $\hat{g}_t$ is an unbiased estimate of the full gradient, γ is the global learning rate, β_1 is the first order decaying parameter and u_t is the exponentially weighted infinity norm.

A.6 Adadelta

The Adadelta algorithm is an improvement of Adagrad. Its main difference compared with Adagrad is that it does not update the learning rate using the accumulated past gradients that imply a continuous reduction of the learning rate as the number of iteration increases, it instead uses a fixed window of past gradients (that means a small number of past gradients compare to the number of iterations) [61].
The Adadelta algorithm can be written as follow [61]:

1- Initialize x_0, $E[g^2]_0 = 0$; $E[\Delta x^2]_0 = 0$

2- For t = 1,.., N-1, where N is the number of the iteration, compute:

$$\begin{gathered} the\ \ gradient\ \ g_t \\ the\ \ accumulate\ \ gradient\ \ E[g^2]_t \\ \Delta x_t = -\frac{RMS[\Delta x]_{t-1}}{RMS[g]_t} g_t \\ x_{t+1} = x_t + \Delta x_t \end{gathered}$$

where g_t is the gradient, $E[g^2]_t$ is the average of the squared gradients at time t and RMS is the root mean square.

Bibliography

[1] S. Chatrchyan et al. “Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC”. In: *Physics Letters B* 716.1 (2012), pp. 30–61. ISSN: 0370-2693. DOI: https://doi.org/10.1016/j.physletb.2012.08.021. URL: http://www.sciencedirect.com/science/article/pii/S0370269312008581 (cit. on p. iv).

[2] G. Aad et al. “Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC”. In: *Physics Letters B* 716.1 (2012), pp. 1–29. ISSN: 0370-2693. DOI: https://doi.org/10.1016/j.physletb.2012.08.020. URL: http://www.sciencedirect.com/science/article/pii/S037026931200857X (cit. on p. iv).

[3] *The Phase-2 Upgrade of the CMS Endcap Calorimeter*. Tech. rep. CERN-LHCC-2017-023. CMS-TDR-019. Geneva: CERN, Nov. 2017. URL: https://cds.cern.ch/record/2293646 (cit. on pp. iv, 4).

[4] *The Phase-2 Upgrade of the CMS Level-1 Trigger*. Tech. rep. CERN-LHCC-2020-004. CMS-TDR-021. Final version. Geneva: CERN, Apr. 2020. URL: http://cds.cern.ch/record/2714892 (cit. on pp. iv, 3, 5, 6).

[5] CERN. *The Large Hadron Collider*. URL: https://home.cern/science/accelerators/large-hadron-collider (cit. on p. 2).

[6] Dan Guest, Kyle Cranmer, and Daniel Whiteson. “Deep Learning and Its Application to LHC Physics”. In: *Annual Review of Nuclear and Particle Science* 68.1 (Oct. 2018), pp. 161–181. ISSN: 1545-4134. DOI: 10.1146/annurev-nucl-101917-021019. URL: http://dx.doi.org/10.1146/annurev-nucl-101917-021019 (cit. on pp. 2, 12).

[7] The CMS Collaboration et al. “The CMS experiment at the CERN LHC”. In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08004–S08004. DOI: 10.1088/1748-0221/3/08/s08004. URL: https://doi.org/10.1088%2F1748-0221%2F3%2F08%2Fs08004 (cit. on p. 2).

[8] The ATLAS Collaboration et al. “The ATLAS Experiment at the CERN Large Hadron Collider”. In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08003–S08003. DOI: 10.1088/1748-0221/3/08/s08003. URL: https://doi.org/10.1088%2F1748-0221%2F3%2F08%2Fs08003 (cit. on p. 2).

[9] CERN. *Detector*. URL: https://cms.cern/detector (cit. on p. 2).

[10] Werner Herr and B Muratori. “Concept of luminosity”. In: (2006). DOI: 10.5170/CERN-2006-002.361. URL: https://cds.cern.ch/record/941318 (cit. on p. 3).

[11] Wikipedia contributors. *Barn (unit) — Wikipedia, The Free Encyclopedia*. 2020. URL: https://en.wikipedia.org/w/index.php?title=Barn_(unit)&oldid=968734092 (cit. on p. 3).

[12] Cheuk-Yin Wong. *Introduction to high-energy heavy-ion collisions*. Erratum. Singapore: World Scientific, 1994. DOI: 10.1142/1128. URL: https://cds.cern.ch/record/241251 (cit. on p. 7).

[13] Peter Zhang. "Neural Networks for Classification: A Survey". In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 30 (Dec. 2000), pp. 451–462. DOI: 10.1109/5326.897072 (cit. on p. 12).

[14] Yann LeCun, Y. Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521 (May 2015), pp. 436–44. DOI: 10.1038/nature14539 (cit. on p. 12).

[15] Isabelle Guyon. "Applications of neural networks to character recognition". In: *International Journal of Pattern Recognition and Artificial Intelligence* 5.01n02 (1991), pp. 353–382 (cit. on p. 12).

[16] Jinsa Kuruvilla and K. Gunavathi. "Lung cancer classification using neural networks for CT images". In: *Computer Methods and Programs in Biomedicine* 113.1 (2014), pp. 202–209. ISSN: 0169-2607. DOI: https://doi.org/10.1016/j.cmpb.2013.10.011. URL: http://www.sciencedirect.com/science/article/pii/S0169260713003532 (cit. on p. 12).

[17] U. Rajendra Acharya, Shu Lih Oh, Yuki Hagiwara, Jen Hong Tan, Muhammad Adam, Arkadiusz Gertych, and Ru San Tan. "A deep convolutional neural network model to classify heartbeats". In: *Computers in Biology and Medicine* 89 (2017), pp. 389–396. ISSN: 0010-4825. DOI: https://doi.org/10.1016/j.compbiomed.2017.08.022. URL: http://www.sciencedirect.com/science/article/pii/S0010482517302810 (cit. on p. 12).

[18] Celia Fernández Madrazo, Ignacio Heredia, Lara Lloret, and Jesús Marco de Lucas. "Application of a Convolutional Neural Network for image classification for the analysis of collisions in High Energy Physics". In: *EPJ Web of Conferences*. Vol. 214. EDP Sciences. 2019, p. 06017 (cit. on p. 12).

[19] Zhao Yanling, Deng Bimin, and Wang Zhanrong. "Analysis and study of perceptron to solve XOR problem". In: *The 2nd International Workshop on Autonomous Decentralized System, 2002*. 2002, pp. 168–173 (cit. on p. 15).

[20] K. Hornik, M. Stinchcombe, and H. White. "Multilayer Feedforward Networks Are Universal Approximators". In: *Neural Netw.* 2.5 (July 1989), pp. 359–366. ISSN: 0893-6080 (cit. on p. 16).

[21] Grace Lindsay. "Convolutional Neural Networks as a Model of the Visual System: Past, Present, and Future". In: *Journal of Cognitive Neuroscience* (Feb. 2020), pp. 1–15. ISSN: 1530-8898. DOI: 10.1162/jocn_a_01544. URL: http://dx.doi.org/10.1162/jocn_a_01544 (cit. on pp. 18, 20).

[22] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36 (1980), pp. 193–202 (cit. on p. 18).

[23] Andreas Müller Dominik Scherer and Sven Behnk. "Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition". In: (Sept. 2010) (cit. on p. 20).

[24] Frédéric Magniette. *Innate tutorial. (It should be released soon.)* (Cit. on p. 21).

[25] Douglas Thain, Todd Tannenbaum, and Miron Livny. "Distributed computing in practice: the Condor experience". In: *Concurrency and computation: practice and experience* 17.2-4 (2005), pp. 323–356. URL: https://research.cs.wisc.edu/htcondor/doc/condor-practice.pdf (cit. on p. 21).

[26] Gregory M. Kurtzer, Vanessa V. Sochat, and Michael Bauer. "Singularity: Scientific containers for mobility of compute". In: *PLoS ONE* 12 (2017) (cit. on p. 21).

[27] Frédéric Magniette. *innate*. URL: https://llrgit.in2p3.fr/online/innate (cit. on p. 22).

[28] J. Duarte et al. "Fast inference of deep neural networks in FPGAs for particle physics". In: *Journal of Instrumentation* 13.07 (July 2018), P07027–P07027. ISSN: 1748-0221. DOI: 10.1088/1748-0221/13/07/p07027. URL: http://dx.doi.org/10.1088/1748-0221/13/07/P07027 (cit. on pp. 22–24).

[29] Jennifer Ngadiuba (CERN. *Deep learning on FPGAs for L1 trigger and Data Acquisition*. URL: https://indico.cern.ch/event/587955/contributions/2937529/attachments/1683932/2706842/HLS4ML_CHEP2018_Ngadiuba.pdf (cit. on p. 24).

[30] Xilinx. *Vivado Design Suite User Guide: High-Level Synthesis*. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf (cit. on pp. 25, 26).

[31] Tom Fawcett. "Introduction to ROC analysis". In: *Pattern Recognition Letters* 27 (June 2006), pp. 861–874. DOI: 10.1016/j.patrec.2005.10.010 (cit. on pp. 28, 29).

[32] Yaoshiang Ho and Samuel Wookey. "The Real-World-Weight Cross-Entropy Loss Function: Modeling the Costs of Mislabeling". In: *IEEE Access* PP (Dec. 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2962617 (cit. on p. 30).

[33] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. Pag. 175. MIT press, 2016 (cit. on pp. 30, 39).

[34] Rajeswari Ponnuru, Ajit Kumar Pookalangara, Ravi Keron Nidamarty, and Rishabh Kumar Jain. *CIFAR-10 Classification using Intel® Optimization for TensorFlow**. Intel® AI Developer Program. 2017. URL: https://software.intel.com/content/www/us/en/develop/articles/cifar-10-classification-using-intel-optimization-for-tensorflow.html (cit. on p. 37).

[35] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. *Self-Normalizing Neural Networks*. 2017. arXiv: 1706.02515 [cs.LG] (cit. on pp. 39, 101).

[36] Keras. *Layer activation functions*. Available online. URL: https://keras.io/api/layers/activations/ (cit. on p. 39).

[37] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. 2015. arXiv: 1511.07289 [cs.LG] (cit. on p. 39).

[38] Jeremy Jordan. *Setting the learning rate of your neural network*. URL: https://www.jeremyjordan.me/nn-learning-rate/ (cit. on p. 40).

[39] Jean-Baptiste Sauvan Frédéric Magniette Alexandre Hakimi. *Neural-network Topology Bayesian Optimization for HEP*. URL: https://indico.in2p3.fr/event/19768/attachments/56045/74353/BOLabo.pdf (cit. on pp. 41, 48).

[40] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG] (cit. on pp. 41, 123).

[41] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html (cit. on pp. 41, 42).

[42] Christopher M Bishop. *Pattern recognition and machine learning*. Section 5.5. springer, 2006 (cit. on p. 41).

[43] Jacob Reinhold. *Dropout on convolutional layers is weird*. URL: https://towardsdatascience.com/dropout-on-convolutional-layers-is-weird-5c6ab14f19b2 (cit. on p. 42).

[44] Lin Xiao. "Dual Averaging Methods for Regularized Stochastic Learning and Online Optimization". In: *Journal of Machine Learning Research* 11.88 (2010), pp. 2543–2596. URL: http://jmlr.org/papers/v11/xiao10a.html (cit. on p. 43).

[45] Stuart Geman, Elie Bienenstock, and René Doursat. "Neural Networks and the Bias/Variance Dilemma". In: *Neural Computation* 4 (Jan. 1992), pp. 1–58. DOI: 10.1162/neco.1992.4.1.1 (cit. on p. 48).

[46] Roberto Calandra, André Seyfarth, Jan Peters, and Marc Peter Deisenroth. "An experimental comparison of Bayesian optimization for bipedal locomotion". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2014, pp. 1951–1958 (cit. on p. 49).

[47] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization". In: *The Journal of Machine Learning Research* 13.1 (2012), pp. 281–305 (cit. on p. 49).

[48] Wikipedia contributors. *CMA-ES — Wikipedia, The Free Encyclopedia*. [Online; accessed 10-August-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=CMA-ES&oldid=967047119 (cit. on p. 49).

[49] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*. Vol. 2. 3. MIT press Cambridge, MA, 2006 (cit. on pp. 50, 52, 57, 69).

[50] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. "Taking the Human Out of the Loop: A Review of Bayesian Optimization". In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175 (cit. on p. 50).

[51] Richard Bellman. "Dynamic programming". In: *Science* 153.3731 (1966), pp. 34–37 (cit. on p. 51).

[52] Peter Roelants. *Understanding Gaussian processes*. URL: https://github.com/peterroelants/peterroelants.github.io/blob/master/notebooks/gaussian_process/gaussian-process-tutorial.ipynb (cit. on p. 52).

[53] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on pp. 55, 57, 59).

[54] Christian Rakotonirina. *On the Cholesky method*. 2009. arXiv: `0906.0165 [math.NA]` (cit. on p. 55).

[55] Albert Alonso. *Bayesian Optimization*. URL: `https://github.com/fmfn/BayesianOptimization` (cit. on p. 57).

[56] Niranjan Kumar. *Batch Normalization and Dropout in Neural Networks with Pytorch*. URL: `https://towardsdatascience.com/batch-normalization-and-dropout-in-neural-networks-explained-with-pytorch-47d7a8459bcd` (cit. on p. 84).

[57] Claudionor N. Coelho Jr., Aki Kuusela, Hao Zhuang, Thea Aarrestad, Vladimir Loncar, Jennifer Ngadiuba, Maurizio Pierini, and Sioni Summers. *Ultra Low-latency, Low-area Inference Accelerators using Heterogeneous Deep Quantization with QKeras and hls4ml*. 2020. arXiv: `2006.10159 [physics.ins-det]` (cit. on p. 108).

[58] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. arXiv: `1609.04747 [cs.LG]` (cit. on p. 121).

[59] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *Journal of Machine Learning Research* 12.61 (2011), pp. 2121–2159. URL: `http://jmlr.org/papers/v12/duchi11a.html` (cit. on pp. 121, 122).

[60] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31 (cit. on p. 122).

[61] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. 2012. arXiv: `1212.5701 [cs.LG]` (cit. on p. 124).