MSE | MASTER OF SCIENCE IN ENGINEERING

UNIVERSITÉ DE GENÈVE
FACULTÉ DES SCIENCES
Département de physique nucléaire et corpusculaire

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master's Thesis
Academic Year 2025-2026

Orientation : Data science (DS)

Analyze and Optimize pipeline for Cherenkov Telescope SST-1M

Autor

Hugo Varenne

Supervisor

Upegui Posada Andres
MSE Teacher

Representative

Matthieu Heller
DPNC - Department of Nuclear and Particle Physics

Version : 1.3.0

Geneva, Switzerland // TM, 2025-2026

Acknowledgements

I would like to thank my supervisor, Andres Upegui Posada, and the representative, Matthieu Heller, for allowing me to work on this project. I always wanted to work on an astrophysics project. Additionally, I have really appreciated your guidance, feedback, and support. You have helped advance the thesis through your constructive criticism and comments.

In the meantime, I want to thank Laurent Gantel and Jakub Kvapil, who have always been available to review my work and answer my questions promptly. They also provided some ideas and their point of view on the thesis, which was very helpful.

I also want to thank Laetitia Guidetti, my colleague working on a parallel thesis in the same field. I had the Opportunity to chat and exchange important information with her that helped me with my thesis.

Finally, I want to thank Tjark Miener, with whom I interacted regarding the CTLearn Library and the data used in the thesis. He helped me to understand and use these things.

Version history

The report got majors updates during the project to adapt its content to standards and best practices. Table below gives an overview of each version major changes

Version	Details
1.0.0	Initial project structure
1.1.0	First draft and chapters
1.3.0	Updates followings comments
1.3.0	Completed version

Contents

1 Executive Summary **1**

2 Context **3**
2.1 Structure of the report . . . 5

3 Materials and methods **8**
3.1 Gamma ray . . . 8
3.1.1 Air shower . . . 9
3.1.2 Cherenkov light . . . 11
3.2 Triggering system . . . 12
3.3 Reconstructed data . . . 12
3.3.1 Type of primary particle . . . 13
3.3.2 Energy of primary particle . . . 13
3.3.3 Arrival direction of the primary particle . . . 13
3.4 CTLearn library . . . 13
3.5 IACT . . . 13
3.6 Working Environment . . . 15
3.6.1 Baobab . . . 16
3.6.1.1 Production environment . . . 17
3.6.1.2 SLURM . . . 18
3.6.1.3 Dedicated space . . . 20
3.6.2 Calculus . . . 21

4 Data **22**
4.1 Prediction tasks . . . 22
4.1.1 Particle Classification . . . 22
4.1.2 Energy Regression . . . 23
4.1.3 Direction Regression . . . 23
4.2 Generation of data . . . 24
4.3 Format of files . . . 26
4.3.1 Content of HDF5 file . . . 27
4.4 Compliance with the server . . . 33
4.4.1 Transferring to the cluster . . . 33
4.4.2 Conversion of data . . . 33
4.4.3 Moving Data across the cluster . . . 34
4.4.4 Merging files . . . 34
4.4.5 Reducing data clutter . . . 34

5 MLOps approach **35**
5.1 Task Breakdown . . . 36

5.2 Configuration files . . . 37
5.3 Report Generation . . . 38
5.3.1 Generic Section . . . 39
5.3.1.1 General information . . . 39
5.3.1.2 Model and Runtime Metrics . . . 40
5.3.1.3 Graphics . . . 41
5.3.2 Particle Classification . . . 43
5.3.2.1 Evaluation Metrics . . . 43
5.3.2.2 Graphics . . . 45
5.3.3 Energy Regression . . . 49
5.3.3.1 Evaluation Metrics . . . 49
5.3.3.2 Graphics . . . 51
5.3.4 Direction regression . . . 54
5.3.4.1 Evaluation Metrics . . . 54
5.3.4.2 Graphics . . . 56
5.4 Comparison of models . . . 59
5.4.1 Metrics modifications . . . 60
5.4.2 Graphics modifications . . . 61

6 Models **65**
6.1 Reference . . . 65
6.2 CTLearn Models . . . 66
6.2.1 ResNet . . . 66
6.2.2 CNN . . . 69
6.2.3 Custom model . . . 71
6.3 Tasks . . . 74
6.3.1 Prepare model . . . 74
6.3.2 Training . . . 74
6.3.3 Testing . . . 78
6.4 Model Optimization . . . 79
6.4.1 Pruning . . . 80
6.4.2 Quantization . . . 82
6.5 Towards New Models . . . 84

7 Challenges **85**
7.1 Link GPU to Tensorflow . . . 85
7.2 Accelerate training speed . . . 86
7.3 CTLearn library . . . 86
7.4 Quantization of complex models . . . 87
7.5 SLURM Time Limitation . . . 87

8 Conclusion **90**
8.1 Conclusion . . . 90
8.2 Limitations . . . 91
8.3 Future improvements . . . 91
8.4 Personal reflection . . . 92

9 Declaration of honor **93**

10 References **94**

11 Appendix **100**

1 Executive Summary

Gamma-ray astronomy plays a key role in understanding the highest-energy events in the Universe, such as supernova remnants, pulsars, and active galactic nuclei. A gamma ray doesn't make it to Earth's ground because of a collision with the atmosphere. Instead, they produce **extensive air showers** that emit **Cherenkov light**. This is what is detected by Imaging Atmospheric Cherenkov Telescopes (IACTs). Cherenkov light helps reconstruct the properties of the primary particle (a gamma ray at the origin): **type**, **energy**, and **arrival direction**. This task stays challenging due to the complexity of the data and the overwhelming amount of noise and background (hadronic) events.

This thesis focuses on the application of Machine Learning, Deep Learning and **MLOps techniques** to data from the **SST-1M** (Small-Sized Telescope) of the Cherenkov Telescope Array Observatory, using the **CTLearn library** as the core framework. The main objectives are multiple: first, to evaluate and prepare a production environment for the deep learning model generation process; and second, to boost the overall model development workflow by introducing software engineering and MLOps best practices into an environment traditionally driven by astrophysics research.

The work covers three main reconstruction tasks: particle classification, energy regression, and arrival direction regression. Simulated SST-1M data are processed, transformed, and configured to ensure compatibility with CTLearn and the production environment, with particular attention to data formats, storage constraints, and computational efficiency on high-performance computing infrastructure. Several Deep Learning architectures, including **CNNs** and **ResNet** models, are evaluated and compared.

Without focusing on model performance, this thesis makes a significant contribution by integrating an MLOps-oriented approach. The project introduces task decomposition, configuration-driven experimentation, automated report generation, and systematic model comparison tools. These additions improve reproducibility, traceability, and scalability of experiments, and facilitate more in-depth analysis of model behavior across tasks and configurations. Model optimization techniques, such as pruning and quantization, are also explored to assess their capacity to reduce computational costs.

The tools implemented are well integrated into the working environments and add value by improving the understanding of models' strengths and weaknesses. Model generation is more reliable and reproducible than before, while still maintaining the defined parameters. With these, MLOps principles show a real improvement to the current solution and an interest in further enhancing the model generation pipeline.

In conclusion, this thesis demonstrates that MLOps is well-suited for gamma-ray event reconstruction using SST-1M data and the CTLearn library, while highlighting the importance of structured workflows and automation in large-scale scientific machine learning projects. The tools and methodologies developed provide a starting point for future improvements to CTLearn-based analysis and serve as a bridge between astrophysics research and modern machine learning engineering practices.

2 Context

Gamma rays (or gamma radiation) are a form of electromagnetic radiation coming from space. Gamma rays at such high energies are produced by interactions between cosmic rays and matter, magnetic fields, or light. Cosmic rays are created by different kinds of astronomical events or objects, such as black holes, solar flares, supernova explosions, or nebulae. They travel through space. Gamma rays are unique because they have the smallest wavelengths and are the most energetic type of ray in the electromagnetic spectrum (0.1 - 100 TeV).

Detecting such high energies from Earth is complicated. The atmosphere blocks cosmic and gamma rays, which enables life as we know it today. As a result, there are two possibilities (Figure 2.1) to detect them: Space telescopes (e.g., Fermi-LAT) or Ground-based Cherenkov telescopes (e.g., CTA, HESS). Space telescopes are currently too small to stop VHE (Very High Energy) gamma rays and do not collect enough to make detection efficient.

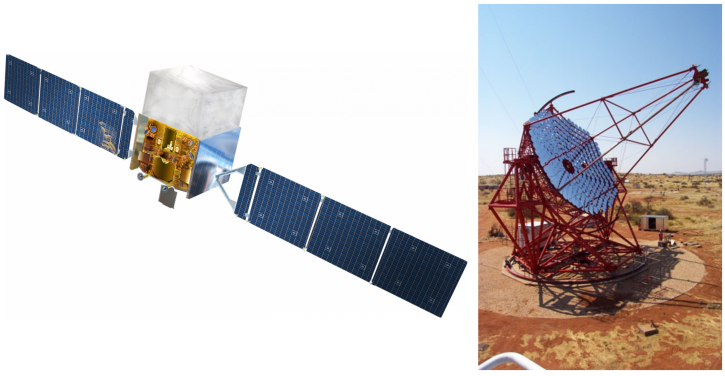

Figure 2.1: Fermi-Lat satellite of the left and H.E.S.S. telescope on the right. [21]

CTLearn is the library providing Machine Learning and Deep Learning solutions to work with imaging atmospheric Cherenkov telescopes, and is the central core of this thesis. The work will be conducted with Ground-based Cherenkov telescopes and the data they collect about events occurring in the sky.

How can they detect events even with the atmosphere blocking them (see figure 2.2)? This is where Cherenkov radiation comes in. The idea is to use the atmosphere as part of the detector. When the gamma ray collides with the atmosphere, it absorbs the primary particle and creates a reaction called extensive air showers. Air showers produce multiple secondary particles that, when their speed exceeds the speed of light in the air, generate Cherenkov light, a faint blue light captured by the IACTs (Imaging Atmospheric Cherenkov telescopes). These air showers come with two distinct variations : the electromagnetic ones (gammas) and the hadronic ones

(protons, electrons, etc.).

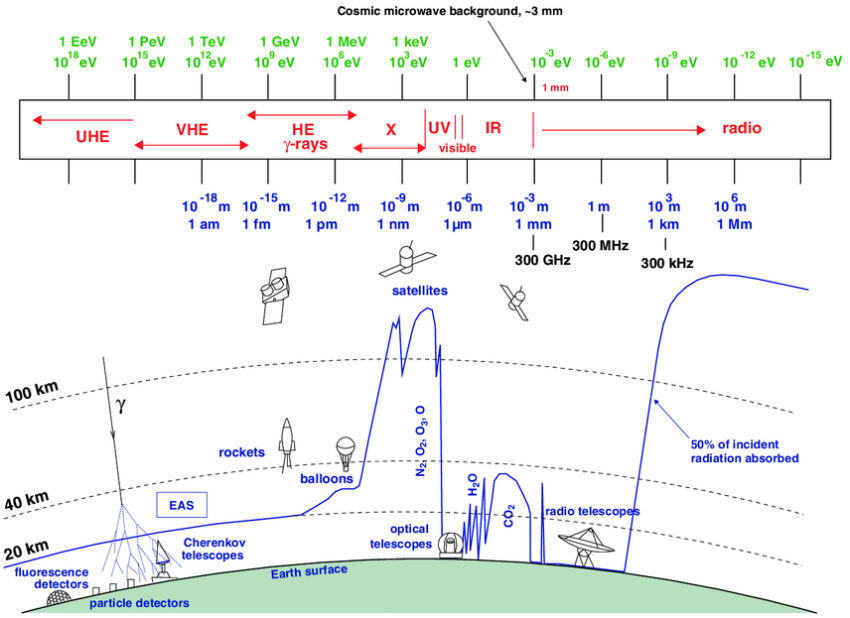

Figure 2.2: The continuous line indicates the height at which a detector can receive half of the incoming radiation. Only the photons of optical and radio frequencies can be directly measured on Earth. Other wavelengths are (partially) absorbed in the atmosphere. In the case of gamma-rays, an indirect measurement with the Imaging Atmospheric Cherenkov Technique is possible. [47]

The difference comes from the way the shower develops. The shower with a gamma as origin will have a smooth lateral distribution, whereas having hadrons as origin results in a clumpy lateral distribution.

A camera will capture what the telescope detects and store information to be treated as data for future usage (e.g., with CTLearn). A captured gamma-ray is displayed in the figure 2.3.

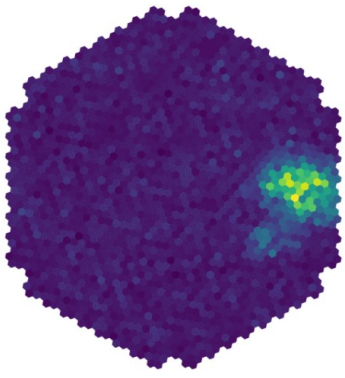

Figure 2.3: Captured gamma-ray by IACT and displayed. The part with colors amplified is where the gamma-ray has landed. [21]

Three important pieces of information need to be extracted from the data to understand the event correctly: the type of particle, the energy, and the arrival direction of the primary particle. Initially, some machine learning models, such as Random Forest, were used to predict these

values. The limitations of ML and the need for high precision led specialists to explore other solutions. Deep Learning addresses these needs. Its ability to analyze 2D images with tools like convolutional layers and its complexity make it a good candidate to improve AI-driven event analysis. Currently, the best model is a ResNet, which works for all three tasks described earlier. However, there is still potential for better models and results.

The thesis intervenes with the idea of improving the current solution in place. Not only could a better model be found, but the current environment to generate and evaluate these models could be improved. The library and the system in place were introduced by astrophysicists and not computer scientists. A computer science point of view could improve the global usage of the CTLearn library and model utilization by providing standardization, tools, and automation in the model handling process. It could also bring new interesting ideas to work on to further improve the analysis of air showers.

At the end of the thesis, it's expected to provide tools (in the form of scripts or others) and a potential new model to predict the required information.

2.1 Structure of the report

In order to have a better overview of the content of the report, the structure and content of the main chapters will be described here.

Context

The context aims to give an overview of why this thesis exists. It introduces many concepts and main elements treated during the thesis. It gives information about the current situation and the stakes of the thesis. This context is really close to an executive summary in form.

Materials and Methods

Material and Methods (3) is a section regrouping elements that need to be understood in order to get deeper into the thesis. It's a prolongation of the context, developing the mentioned concepts and tools that are sufficiently important for the thesis to have dedicated space. It's here to explain the physics part of the thesis.

It includes physics concepts like the air showers (3.1.1) or the Cherenkov light (3.1.2). These elements are essential to understand the stakes of projects like this thesis. Additionally, other concepts such as the telescope and the trigger system are worth mentioning details about them, thus not being directly part of the thesis.

Another element included here is the space dedicated to the environment of the thesis (3.6). Multiple workspaces (local, cluster, etc.) and tools have been used to work on this thesis. This means some schemes and information need to be provided to understand the roles and interactions between these environments.

Data

Data (4) from IACT has a lot of complexity and makes it difficult to work with. Therefore, a major part of the thesis needed to be dedicated to analyzing the data and related elements. It includes a description of metrics to extract and how the data is artificially generated to provide

enough information to train models. It also includes processing steps for the data to be used by CTLearn models in the most optimal way. Data format and content, like waveforms and frames, are an essential part to understand and need to be detailed.

Reconstructing gamma rays (4.1) is the purpose of the CTLearn project. Models should be able to extract three different types of data : particle type, energy, and direction. Their purposes need to be thoroughly explained and understood.

Data comes in multiple formats (4.3), zipped or unzipped. Some specifications need to be provided to understand the format, purposes, and their content. The content of the files needs to be carefully explained to understand their purpose. Related to that, data are simulated, meaning there is a need to understand how the generation works (4.2).

The data weren't directly in an adapted format to work with. Some processing steps (4.4) have been executed through the thesis in order to improve the efficiency and compliance of their usage with the CTLearn library. Each step is described with the objective in mind when introducing it.

MLOps approach

MLOps (5) approach and integration is one of the main axes of the thesis. It regroups details about this methodology and what contribution related to this that have been integrated as part of the thesis. It includes tools like the report generation or even concepts like the task breakdown. This concept is fully detailed as an introduction. MLOps is an important aspect to take into account in order to optimize and to ease the process of model generation.

The concept of task breakdown (5.1) is the first aspect of the methodology integrated. Adapting the structure (going from Jupyter Notebook to scripts) to a production environment is a related subject to this.

The concept of configuration files (5.2) is another aspect of MLOps. It has many benefits and enables better tracking of experience. Details about how it operates can be found. This concept is provided with examples of configuration files for each task in the appendix.

The most substantial part of the MLOps integration in the thesis is the tool dedicated to generating a report (5.3). This tool is a first step to get an overview of model performances with specific metrics and graphics depending on the task. It also relies on generic information useful, like the inference time per event. The report helps to understand where the model is better and can be used to compare with other models.

Another interesting tool is the comparison of models (5.4). This tool generates a report with metrics and graphics, on the basis of the report generation tool, that goes further in the comparison of models by comparing multiple models together with results next to each other, making it easier to analyze.

Models

Models (6) are one of the main axes explored during the thesis. It regroups everything related to model creation and usage, as well as related subjects such as optimization. This is an essential part of the thesis as model performance and improvements are the core of it.

To get an idea of the performances of models created during the thesis, some reference metrics and graphics from the literature related to CTLearn has be summarized and centralized as a goal for future models generated to beat (6.1).

Some pre-built models are provided by the CTLearn library and were mostly used during tests and implementation of tools (6.2). These models and their specificity have been listed and detailed, as well as configurable options that can be set to modify them. Additionally, a custom model option is available for models with different architectures to be used. This part includes a description of a template dedicated to fit these custom models with CTLearn library requirements (6.2.3).

Tasks (6.3) related to the model generation need to be detailed, as the features and configurable parameters may be technical. Three tasks compose the model generation : preparation, training, and testing of the model. It explains which interactions with the CTLearn are required to make it work.

Model optimization (6.4), in the case of the thesis, was only explored in surface. Techniques like pruning or quantization are essential, depending on resource restrictions the environment of the model may have. Some details and examples of implementation can be found for more details on that.

Challenges

This section (7) is dedicated to explaining the main challenges encountered during the thesis. Challenges appear at different places and can be various. Challenges are decomposed into multiple sections : description of the issue, possible solutions, and final choice.

Conclusion

The conclusion of the report (8) provides an overview of all the work accomplished in the project. This is divided into two types of feedback: a more formal feedback on the results obtained, what the results were, what was achieved, and how it fitted in with the schedule, and a more personal feedback on the process of working on the project. This is also the place where limitations (8.2) encountered are mentioned and explained, as well as any future improvements (8.3) that might be envisaged.

Bibliography, Appendix

Sections are also provided to reconcile all the sources (9), figures, and appendices (11) that were used in this project. These different chapters are intended for this purpose. Mentioning all these elements supports the work that has been carried out and can provide insights into potential future projects. Appendix (11) contains images, scripts, and documents that explore further aspects of the thesis or elements that couldn't be included in it.

3 Materials and methods

Materials and Methods section is dedicated to details element introduced in the context of the thesis. It also reflects astrophysical elements that will be mentioned throughout the report. It regroups the physics concepts used and explains them for readers from another domain.

It includes unorganized and unrelated notions such as tools, concept or even objects related in any way to the subject of the thesis.

3.1 Gamma ray

Cosmic-rays [21] can be seen as atomic particles accelerated to extraordinarily high energies, at almost the speed of light. They constantly bombard the Earth from space, coming from space objects like galaxies, nebulae, pulsars, or even black holes. During their travel time, they are accelerated and deflected by magnetic fields, making it difficult to understand their origin and track them. The composition of cosmic rays is mostly protons and helium nuclei (99%) with a small percentage of heavier nuclei (1%) [16].

Gamma rays can be seen as the child of cosmic rays, generated when cosmic-ray interaction with matter, radiation fields, or magnetic fields. This is the most energetic form of electromagnetic radiation. The interest in them lies in the fact that they preserve directional information from the interaction point and can be detected by telescopes. The travel direction of this type of ray is in a straight line because it's insensitive to astrophysical objects and magnetic fields. It helps to identify objects (figure 3.1) that accelerated cosmic rays and generated gamma rays. There are multiple processes in existence that cover the interaction of the cosmic ray with objects to generate a gamma ray. These processes [9] are called the Leptonic processes (for electrons) and the hadronic processes (for protons and nuclei).

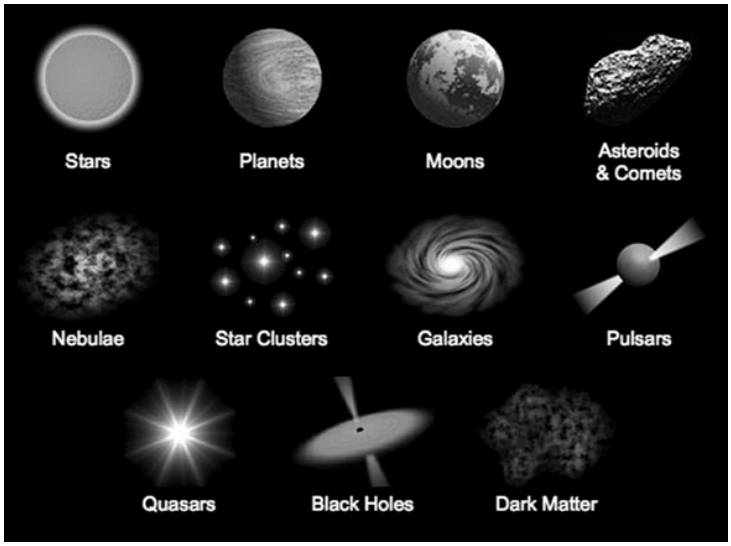

Figure 3.1: Non-exhaustive list of astrophysical objects than can collides with cosmic-ray and generate gamma-ray. [8]

Information carried by a gamma ray is crucial to understanding the spatial objects. Among the information, some of them are more interesting, like the arrival direction or the energy. These types of rays enables to study the high-energy astrophysics and more precisely magnetic fields, particle interaction, or dark matter. By observing the sky for multiple hours with a telescope, a tremendous amount of gamma rays are captured, making it possible to build maps like the figure 3.2.

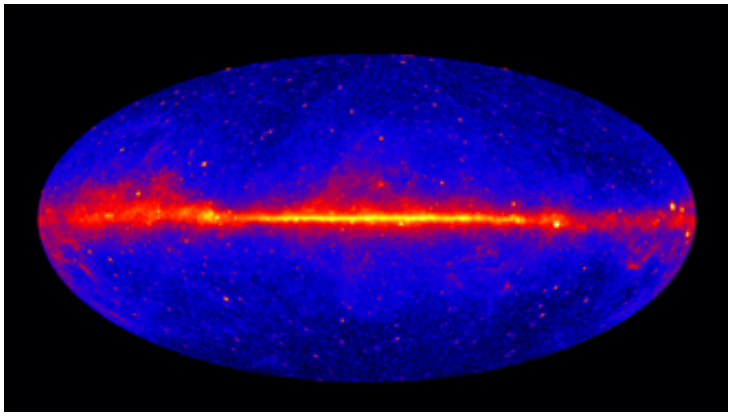

Figure 3.2: Gamma ray observation of the sky captured by the Fermi telescope for 5 years of observation. The line at the middle represents the Milky Way. [11]

They cannot be directly detected from the Earth because of their interaction with the atmosphere.

3.1.1 Air shower

Air showers [42] [21] result from the interaction of high-energy particles with the atmosphere, like gamma rays. This interaction creates a cascade of subatomic particles called extensive air showers. Air showers are the way to observe high-energy particles from Earth, as the primary particle cannot reach it without being absorbed. This cascade occurs in a chain reaction with

secondary particles (subatomic). These particles travel at a high speed, faster than the speed of light in the air, creating what is called the Cherenkov light. It can be compared to the blow-up produced after passing the speed of sound.

An air shower longitudinal development is almost five kilometers and is produced around 10 km above the sea level. The figure 3.3 displays an example of shower development from the gamma ray until it reaches Earth.

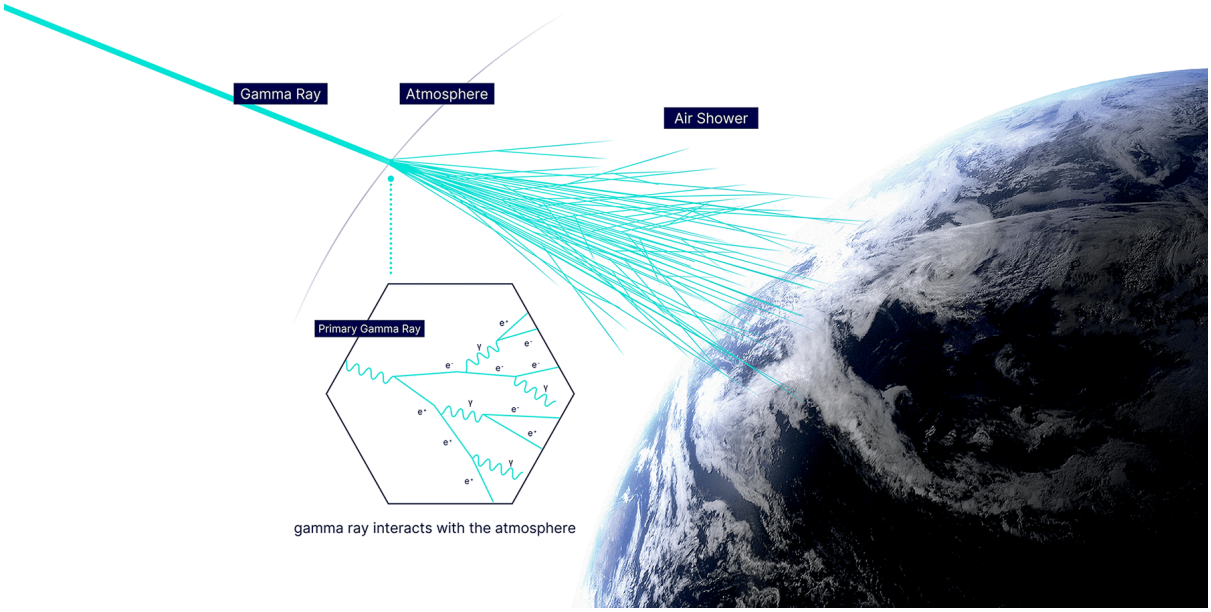

Figure 3.3: Gamma ray shower development from collision with the atmosphere to Earth's ground. [21]

Two types of air showers are generally produced : electromagnetic showers and hadronic showers. The type is determined by the primary particle that interacts with the atmosphere. The figure 3.4 displays a visual distinction between the two types of showers.

Electromagnetic showers [27] occur when gamma rays collide with the atmosphere. It requires specific gamma-ray particles to be considered. The particles are electrons, positrons, and photons. It's a simpler, more predictable shower than the hadronic one. The global air showers are compact and have a smooth development. It forms through pair production, in which a gamma ray is converted into an electron-positron pair. These particles then emit gamma rays via Bremsstrahlung, which in turn produces additional electron-positron pairs. This repeated process leads to the development of the electromagnetic air shower.

Hadronic shower [6] occurs when a cosmic ray collides with the atmosphere. It's the dominant type of showers occurring. It's unstable and irregular, and difficult to work with because the energy and direction are poorly reconstructed. The complexity comes from the stronger nuclear interactions. When it collides with the atmosphere, it produces various secondary hadrons, such as pions, kaons, or other types of hadrons. Unstable secondary hadrons like this decay into muons, neutrinos, and even gamma rays. This type of shower mixes various particle forms, indicating its instability.

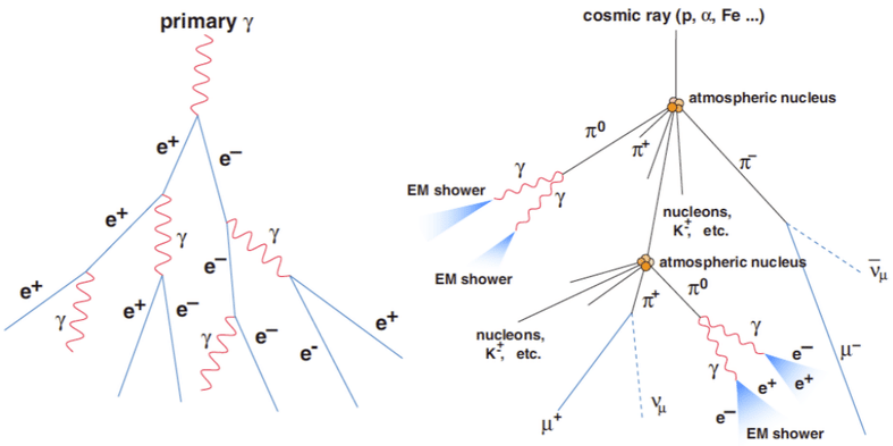

Figure 3.4: Representation of an electromagnetic shower (left) and an hadronic shower (right). [41]

The type of shower that is interesting is the electromagnetic one. They are initiated by gamma rays and are less complex to analyze than hadronic ones. Electromagnetic showers follow a more regular pattern, which makes it easier to extract properties of the primary particle. However, this type of flux is far more suppressed than the hadronic showers.

3.1.2 Cherenkov light

Cherenkov light [21] appears during air shower development in the atmosphere when secondary particles exceed the speed of light in a medium. This effect is similar to the sonic boom produced when the speed of sound is exceeded. Specifically, exceeding the speed of light is possible when light travels through materials like water or air. As a result, electromagnetic radiation is emitted, observed as a faint blue glow lasting only a few nanoseconds. The blue color appears because the radiation is emitted at shorter wavelengths in the visible spectrum (around 400nm). Figure 3.5 shows Cherenkov emission in air shower development. Beyond atmospheric showers, this phenomenon is also visible in other settings; for example, it occurs in nuclear reactors, where particles are highly energetic.

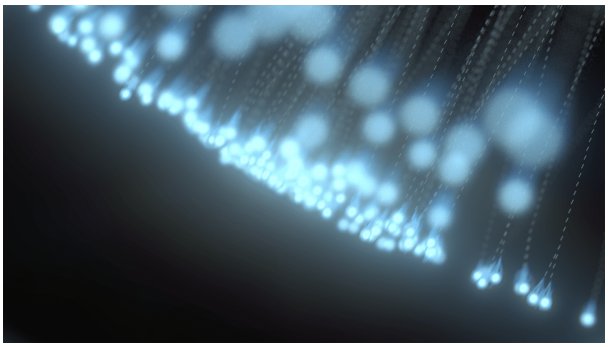

Figure 3.5: Visual representation of Cherenkov light emitted by secondary particles from an air shower. [21]

This Cherenkov flash is what IACTs detect. Although the light lasts only a very short time, it

has time to spread out on the ground over a large area, allowing telescopes to detect it. The light is collected by a large reflective dish and focused onto single-photon-sensitive, ultra-fast cameras. The imprint of the Cherenkov flash in the cameras carries the information about the type, energy, and arrival direction of the primary particle, allowing it to be reconstructed.

3.2 Triggering system

The triggering system [43] is a solution implemented to manage the overwhelming amount of data captured by the telescope and its camera, which would otherwise be impossible to store. Camera samples signal at high speed (hundreds of MHz) with a large number of pixels, but only a tiny fraction of the captured data contains gamma-ray events; the rest is pollution of the Night Sky Background and unexpected events. Therefore, some filtering is needed via a real-time trigger system that saves events only if they are relevant.

The trigger system [17] needs to be extremely fast and reliable to filter data effectively. The trigger system relies on multiple levels, increasingly selective the deeper it gets. The event is saved only after it passes through the entire trigger system.

The triggering system for the SST-1M telescopes, as described in this thesis, is integrated into DigiCam, a fully digital camera system. DigiCam digitizes signals for the camera readout and trigger system. It uses an FPGA and is highly configurable, allowing for customizable thresholds and scalability.

Digicam triggering system is composed of three main levels [43]. The first is a pixel-level trigger that applies a threshold to suppress low-level noise and fluctuations. It also identifies potential pixels that contain Cherenkov light. The second level is more at the camera level, where clusters of simultaneously activated pixels are identified. The signal is summed up to determine if it exceeds a global threshold. It helps to suppress NSB and select events with patterns consistent with air showers. The last level is an array/stereo trigger where multiple telescopes are combined to analyze events that pass to this point. It checks whether multiple telescopes triggered the same event within the same time window. It prevents local noise or single-telescope fluctuations from being detected as an event.

3.3 Reconstructed data

Telescopes and cameras do not observe particles directly, but only consequences of its interaction of the primary particle with the atmosphere. As stated previously, it's the Cherenkov light produced by the shower that is observed by instruments.

It means the recorded data cannot be interpreted directly. It's an issue because an outrageous amount of cosmic rays and gamma rays pass through the atmosphere, along with noise and background light, complicating the extraction of valuable information from the gamma rays. Refer to the gamma-ray chapter to see its importance (3.1).

Based on data collected by telescopes and cameras, it's possible to reconstruct the particle's parameters at the origin. The parameters that need to be identified to understand the particle at the origin are its type, energy, and arrival direction. These parameters can be used for scientific analysis of the gamma-ray origin.

3.3.1 Type of primary particle

The reconstructed parameter is the type of primary particle. The type is known from the morphology of the air showers. As with the two shower types previously seen, the parameter determines which category the air shower belongs to. The two categories are electromagnetic and hadronic. The first one can be identified as a signal that is smooth, elongated, and compact (gamma rays). This is also from this type of air shower that other parameters, like energy and direction, are calculated. The other one is identified as the background. It's a more complex and irregular air shower due to decays and nuclear interactions (cosmic rays).

The distribution of the two types is highly disproportionate, with the hadronic type being massively more prevalent.

3.3.2 Energy of primary particle

The second reconstructed parameter is the energy. Its value is estimated from the total amount of Cherenkov light produced by the air shower, or from the shower's shape and impact distance from the telescope. This parameter is generally reconstructed from gamma-ray (electromagnetic) events to obtain accurate results. It's generally expressed in TeV.

3.3.3 Arrival direction of the primary particle

The last parameter is the arrival direction of the primary particle. The idea is to get from the air shower the original trajectory of the primary particle. Because gamma rays travel in a straight line, their direction helps determine the origin of the gamma ray, most likely an astronomical object. The direction is reconstructed from multiple elements, such as the Cherenkov light pattern, image orientation, and event-detection timing. It can be even more precise when combining multiple telescopes' observations of the same event, with multiple angles of visualization (stereo).

3.4 CTLearn library

The CTLearn library [34] is a Python package that provides Deep Learning solutions for IACT. It handles multiple data levels from CTAO, such as waveforms and images. These models help to reconstruct events using information from DL1 data-level in HDF5 format. It produces neural networks for the three main reconstruction parameters : particle type, energy, and arrival time.

This package is still under development and maintained through regular releases with new features. It uses TensorFlow to generate models and works with generic domain packages like ctapipe or DL1DataHandler.

3.5 IACT

To detect gamma-ray events, CTAO, or the Cherenkov Telescope Array Observatory, is using multiple types of instruments. It includes telescopes, satellites, and other types of cameras located at different sites around the world. Among these instruments, IACTs (Imaging Atmospheric Cherenkov Telescopes) are ground-based telescopes used to detect very-high-energy

gamma-ray events by detecting the Cherenkov light emitted when these events interact with the atmosphere (air showers). IACT combines a telescope with a segmented mirror and a camera oriented to capture the image formed by the mirror (each mirror corresponds to one pixel). It records the behavior of the Cherenkov light distribution by looking at its shape, orientation, and intensity.

The interest of the ground observation [3] [5] with these telescopes is because of its high angular resolution and sensitivity. It helps to have precise details and identify a simple source of the gamma-ray event. This approach has constraints, such as the fact that it only works at night for a short time due to light pollution from stars or Earth. This also includes the fact that the detection isn't directly on the particle but on the Cherenkov light produced by the particle's collision with the atmosphere. The use of telescopes makes the field of view very small, limiting sky coverage.

The following figure 3.6 provides an illustration of the IACTs.

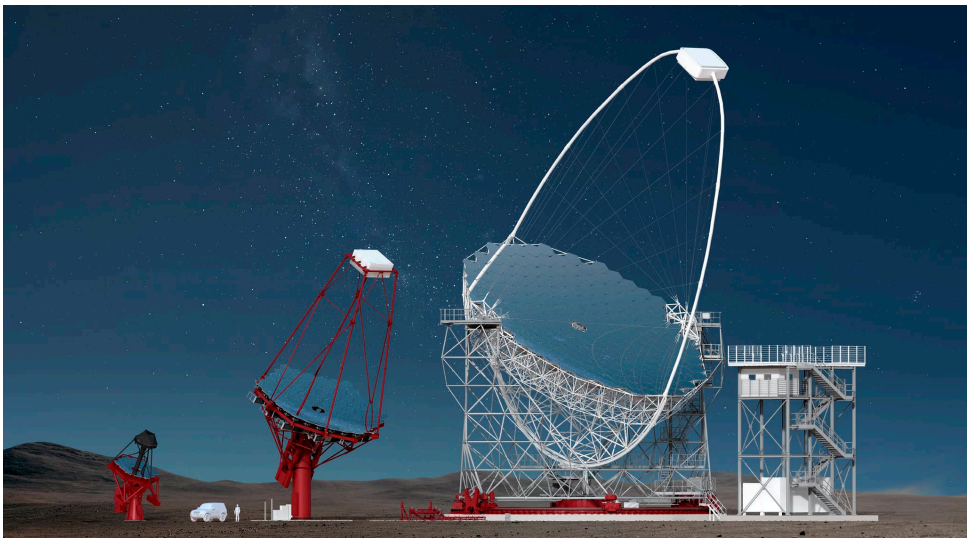

Figure 3.6: IACT telescopes. Three categories of telescopes are represented here. The difference lies in their size and number of mirrors. From left to right : SST, MST, LST. [21]

The telescopes available through IACT can be listed [21].

- **LST** : Four Large-Sized Telescopes (45 meters) in the northern hemisphere with a low range of energy (20 to 150 GeV)
- **MST** : 23 Medium-Sized Telescopes (27 meters) across the world for an energy range of 150 GeV to 5 TeV
- **SST** : 37 Small-Sized Telescopes (9 meters) in the southern hemisphere for energy above 5 TeV

SST-1M telescopes

In the thesis, data from the SST-1M telescopes are used; it is worth mentioning additional details about them. SST-1M telescopes [22] fall in the category of small-sized telescopes with

a single mirror. As stated earlier, this type of telescope covers an energy range of a few TeV to 300 TeV.

The telescope uses a Davies-Cotton optical design to provide good imaging performance for a large FoV (Field-of-View) and limited time dispersion. The specifications of the telescope can be found in the figure 3.7. A key element of the SST-1M is its camera technology, using Silicon Photomultipliers (SiPMs). It allows operation under bright moonlight, extending the operating window. The camera has 1296 pixels and is equipped with a triggering system (DigiCam). Some other elements, such as wavelength filters or light concentrators, help reduce the Night Sky Background and maximize Cherenkov light.

Optical properties	Focal Length	5600 ± 5 mm
	f/D	1.4
	Dish diameter	4 m
	Mirror Area (*)	9.42 m^2
	Mirror Effective Area(*)	6.47 m^2
	Hexagonal Mirror facets	780 ± 3 mm
	Preliminary on-axis PSF real optical parameters	0.07°
	PSF (80% of FoV@ 4° off-axis)(**)	0.21°
Camera Characteristics	Camera (depth x width)	60 cm x 90 cm
	Total pixel number	1296
	Pixel linear size	23.2 mm
	Pixel angular size	0.24°
	FoV	9.1°
	Photosensors PDE	> 30%
	Sampling frequency	250 MHz
	Readout rate	0.6-1 kHz
	Time Spread RMS	< 0.25 ns

Figure 3.7: SST-1M telescope. This telescope is part of a project that includes the University of Geneva. Specifications of the telescope are also provided to get a better idea of how the telescope and the camera capture information. [51] [22]

Having one telescope only enables working in mono mode. To enter stereoscopic mode, an important element for observing air showers from different angles, two telescopes are required. Two of these telescopes are currently working at the Ondřejov Observatory in Czech Republic.

3.6 Working Environment

Several environments and servers will be used during the thesis. Tracking which environment is used at each step is important for clarity. Key components will be described in subsections.

The working environment changes throughout the project. The first environment is dedicated to data discovery and model testing for a variety of tasks. It uses a small data subset, operates locally, and is intentionally unorganized. Its primary function is to conduct initial tests and explore techniques and tools relevant to the thesis. Figure 3.8 displays this environment.

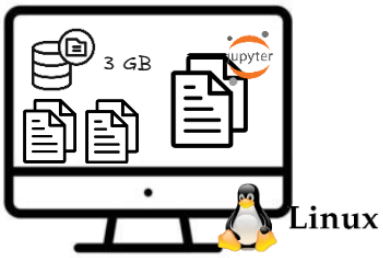

Figure 3.8: Usage of a Linux virtual machine (WSL) to test elements. The tests and explorations are run in Jupyter Notebook to execute steps independently. The amount of data used is very small due to space constraints and low complexity requirements. Configuration files, models, and libraries are completely unstructured and mixed together.

The second environment extends the first by using larger infrastructure and more data. It's available on a cluster and built for direct production use. The structure is clean, with no unnecessary files or elements. Some interactions between files are shown in Figure 3.9.

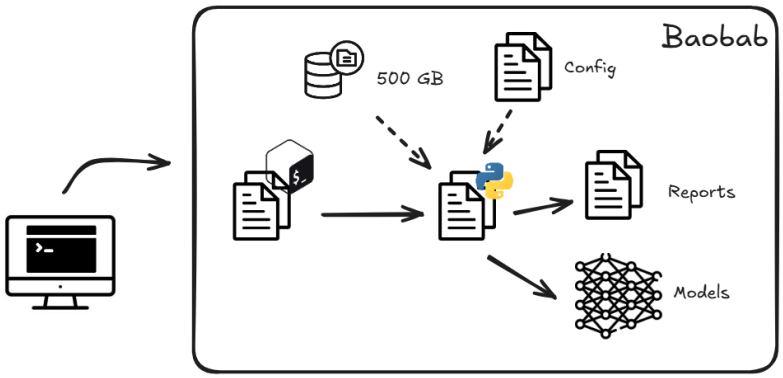

Figure 3.9: Environment on the cluster. It's close to a production environment. Access to it is via SSH on a terminal. Files on the cluster are highly structured to make it clean and easy to understand. To execute jobs, a bash script must be associated with the corresponding Python script. Each Python script represents a single task. They allow the generation of reports and models and store them on the cluster. To generate these elements, Python scripts rely on a large amount of data and configuration files. The data is structured to be easy to use.

3.6.1 Baobab

Baobab is one of the three high-performance computing clusters provided by the University of Geneva [19]. This cluster is optimized for parallel computation and provides GPU and CPU resources. This server is available for researchers who require significant infrastructure for their tasks. The resources available on the cluster are documented in the support [20].

The interest in using such infrastructure is multiple. A large amount of data is used to train models, and these models can be complex enough to strain a personal computer's computational

resources. A personal computer has limited resources (space, memory, CPU capacity, etc.), and processing large amounts of data and complex models can take significant time. The cluster will reduce these constraints and provide tools to run jobs on the different resources allocated.

To work with the same environment used locally, it's necessary to forward it to the cluster. This environment is detailed in the section below (3.6.1.1).

It's also recommended to work with scripts to execute the desired tasks. Indeed, many researchers access the resources every day, thereby requiring parallel execution and resource allocation. To solve these issues, the server uses a resource manager called SLURM (3.6.1.2).

In addition to computational resources, the cluster also provides storage for a large amount of data. The spaces used during the thesis are detailed below (3.6.1.3).

The cluster can be accessed remotely via SSH, and an SSH key must be provided to link the account. The account was requested at the start of the thesis to support the cluster. Files can also be easily transferred from local to the cluster and vice versa by using the SCP command [10] with the SSH key.

3.6.1.1 Production environment

To ensure reproducibility and a better understanding of script behavior, having a similar environment on the local machine and on the production server is a must. Therefore, it's a requirement to provide the local configuration and packages on the cluster. A model system, Lmod [30], is in place on the cluster, allowing the loading of prebuilt configurations for different types of tasks. However, the current modules aren't sufficient to comply with the local environment and the specific packages used. Therefore, another solution is possible: prepare its own container to run tasks, called a Singularity [24].

A Singularity container is used to run complex applications (with many packages and specific versions) on a cluster. It allows for simple, reproducible application handling. It takes the form of a container. The container is documented in a single file, with each library and its corresponding versions documented.

To generate a Singularity container from a local environment, the first step is to export the current environment configuration as a YAML file.

```
conda env export > ctlearn.yml
```

Then, you should have a file containing every dependency and corresponding versions, as displayed on the figure 3.10. For future steps, it's important to remove the "prefix" section from the YAML file.

```yaml
name: ctlearn
channels:
  - pytorch
  - anaconda
  - conda-forge
dependencies:
  - _libgcc_mutex=0.1=main
  - _openmp_mutex=4.5=4_kmp_llvm
  - aom=3.6.0=h6a678d5_0
  - astropy=6.1.3=py310h5eee18b_0
  - astropy-base=6.1.3=h0df7b8e_0
  - astropy-iers-data=0.2025.8.25.0.36.58=py310h06a4308_0
  - pip:
      - absl-py==1.4.0
      - annotated-types==0.7.0
      - anyio==4.10.0
      - argon2-cffi==25.1.0
      - argon2-cffi-bindings==25.1.0
      - arrow==1.3.0
      - asttokens==3.0.0
      - astunparse==1.6.3
      - async-lru==2.0.5
      - attrs==25.3.0
      - babel==2.17.0
```

Figure 3.10: Example of environment configuration saved in a YAML file. Generally, there are three main sections: the channels (package repositories), the dependencies (installed via Conda), and the subsection for pip dependencies (installed via pip).

The file needs to be copied to the server to continue building the singularity. Once it's done, the last step can be executed. It implies a series of commands to load a module prepared to help build the singularity, and the corresponding command to store the container in a single file: ctlearnenv.sif here.

```
ml purge
module load GCCcore/13.3.0 cotainr
cotainr build ctlearnenv.sif --base-image=docker://ubuntu:22.04
    --accept-licenses --conda-env=ctlearnenv.yml -v
```

The Singularity is now ready to execute a task by calling it as shown in the example below.

```
apptainer exec ctlearnenv.sif python3 -c "print('Hello World')"
```

3.6.1.2 SLURM

SLURM (Simple Linux Utility for Resource Management) [48] is a workload manager used on HPC instances from UNIGE. It's an open-source, highly scalable tool for managing resources and scheduling jobs across clusters. It's there to address the issue of allocating resources to many researchers working on the cluster simultaneously. This tool is very popular because of its key features: allocation of access to resources for a duration of time, a framework to start, execute, and monitor tasks on allocated nodes, and, finally, arbitrary resource contention using a queue system.

To put this in place, resources on the clusters are split across nodes and assigned to partitions based on similar computational nodes. These nodes can be CPUs, GPUs, or other computational resources. Having this system helps when a task (job in Slurm terminology) needs to be executed

by selecting an available space that meets all requirements for correct operation. The use of SLURM in this thesis is at a high level, using only a few of the available features; more details can be found on their website [48]. At some point, it could be expected to run the different tasks of the thesis together by chaining them.

The details of the run configuration are provided in a bash script explained in the chapter below. This script is then handled by SLURM using the following command. This command will add the job to the queue and start it when resources become available.

```
sbatch script.sh
```

Some other commands not mentioned here can be used to monitor and analyze job execution in the tool.

Bash Script Structure

The most efficient way to execute tasks on SLURM is to use bash scripts. These scripts will provide some details about the execution of the task. Each task has a dedicated script. The structure is still the same across all of them.

The Figure 3.11 displays an example of a script to interact with SLURM.

```
#!/bin/bash
#SBATCH --job-name=reduce-data-amount
#SBATCH --time=12:00:00
#SBATCH --partition=shared-gpu
#SBATCH --gpus=nvidia_geforce_rtx_3090:1
#SBATCH --mem=24GB

### Remove limit of files for training
ulimit -n 65535

### Execute the job
apptainer exec --nv \
  --bind /usr/local/cuda/targets/x86_64-linux:/usr/local/cuda/targets/x86_64-linux \
  --bind /usr/local/cuda/lib64:/usr/local/cuda/lib64 \
  --bind /srv/beegfs/scratch/shares/upeguipa/SST1M \
  ctlearnenv.sif bash -c '
export LD_LIBRARY_PATH="/usr/local/cuda/targets/x86_64-linux/lib:/usr/local/cuda/lib64:${LD_LIBRARY_PATH:-}"
python3 scripts/extract_only_images.py --input_dir /srv/beegfs/scratch/shares/upeguipa/SST1M/data/protons_diffuse/reduce_train/
'
```

Figure 3.11: Example of a bash script used to run a job on SLURM. It's separated into three sections: a section related to the job configuration for SLURM, additional commands for the task, and the main command to run the corresponding Python script.

The first section of the script describes the job to run and the resources required to proceed. The information is directly read by SLURM when a job starts. The list of attributes provided in the figure 3.11 shows the most important ones from a job configuration. Others exist but aren't used for this scenario.

- **job-name** : Name of the job (shown when monitoring tasks)
- **time** : Limit of time for the task to be completed
- **partition** : A group of computational nodes where the task is executed
- **gpus** : Type and amount of GPUs required for this task
- **mem** : Memory required from the computational resource

The second part is rather small in this case. This contains all additional commands to run before the Python script for it to work. In this case, many files are used; therefore, the limit

on the number of files a script can work with needs to be increased. That's the purpose of the "ulimit" command.

The third and last part contains the command to execute the Python script, the core element of the task. It can be broken down as follows.

The command starts by executing in a container runtime, as previously seen.

```
apptainer exec
```

The first parameter of this function is intended to allow the NVIDIA GPU to handle the task. Related to that, there is also the binding of the CUDA libraries to make it work along with the GPU. GPU identification by the task has been a major challenge; more details can be found in Chapter 7.1.

```
--nv \
--bind /usr/local/cuda/targets/x86_64-linux:/usr/local/cuda/targets/x86_64-linux \
--bind /usr/local/cuda/lib64:/usr/local/cuda/lib64 \
```

There are additional bindings to do, depending on where the dependent resources can be found. Here, for example, data is stored on another file system, so the Python script must bind to it to access it.

```
--bind /path/to/wanted/data
```

The end of the container execution command is then provided with the singularity file and the desired task type. In this case, it needed to execute another command to link the CUDA library to the Python script by defining a variable. Once all of the above is done, the script can be provided with the required input parameters.

```
ctlearnenv.sif bash -c '
export LD_LIBRARY_PATH="..."
python3 scripts/extract_only_images.py --input_dir ...
'
```

3.6.1.3 Dedicated space

As mentioned earlier, data is stored on the cluster to enable more efficient model generation. The only issue with this is that, depending on where the data is stored, some constraints can appear in terms of space or reading speed. That's why two different spaces are used.

The first one is the personal space. This is the fastest way to read the data, as the scripts and data are kept close (locally). The issue with the personal space is its limited capacity of 1 TB, which is inconvenient when handling larger amounts of data.

The second is a shared space on a BeeGFS[31] file system. All UNIGE researchers working on SST-1M satellite images use this system. It offers a larger storage capacity (10 TB), but the read speed is significantly lower because BeeGFS is optimized for parallel writes rather than fast individual reads.

Depending on thesis progress and specific needs, each is used to generate models.

3.6.2 Calculus

Calculus is a server that hosts data for the SST1M Cherenkov telescopes. A lot of data is stored there for various applications (on the order of TB). For the thesis, simulation data are the main concern, including gamma and proton showers (gamma diffuse/proton diffuse) and gamma points. These different types of data will be useful depending on the task we want to generate a model for.

4 Data

Data is an essential part of this thesis. It's one of the most important subjects to discuss. The complexity of the data used to train and test the model is important. It's mostly composed of images obtained from a telescope, capturing events from the sky. A whole chapter details how the data is retrieved from the telescope (2). Here, a detailed analysis of the data used is provided, describing the format, content, and processing.

4.1 Prediction tasks

Gamma-ray detection and identification imply extracting three distinct types of information from the image produced by the detected event. It includes detecting the particle type, the event energy, and the event direction. These three elements help to understand the event and its related behavior for further analysis. Each of these values is associated with a separate task that a model needs to be trained on. It helps the model to be more precise as it has only one focus. The different values depend on each other to be predicted; more details are below.

These tasks have two modes: **mono**, which uses data from only one telescope, and **stereo**, which uses data from at least two telescopes. In stereo mode, data from multiple telescopes observing the same event are combined to predict reconstructed values. Data handling depends on the specific configuration used.

The data used for these tasks falls into two categories: **Diffuse-like** events are randomly distributed over a defined region in the sky, generally used to train a model for generalization. It's also the most realistic form of events; **Point-like** events that originate from the same point in the sky and are generally used for evaluating gamma-ray reconstruction.

4.1.1 Particle Classification

Particle classification is the most essential task to perform on events. The task itself is to classify between gammas and protons (hadrons) the events that are provided to it. For now, based on the data, there are only these two classes. Right now, this is a 2-class binary classification problem, but it could become a multiclass problem in the future, depending on the model's precision.

This task is the central one of the thesis. It can be used as a filter to extract gamma events, which the other two use. Also, when working with triggers, this task is used to achieve a higher level of filtering with a less accurate model, and then, through the trigger levels, to have a precise model working only on filtered data. Background noise, such as moonlight or light from

the ground, will also falsify the results. Removing this noise or at least ignoring it should be considered.

The task analyzes gamma diffuse and proton diffuse events. Usually, the event distribution is highly skewed, with protons accounting for about 95

The model will not generate the class it predicted for the event, as most models do, but rather compute a probability that the event is a gamma event. Having something like this allows for a better understanding of the model's behavior and more precise results. It also enables the use of additional methods to evaluate the model using specific metrics and graphics (e.g., Brier score). The downside is that a threshold must be defined to assign a class to each event. Sometimes people think the threshold is always 0.5, and that below it the class should be protons and above it gammas. This isn't that simple. It can vary depending on factors such as the desired precision, the model's purpose (focus on recall or accuracy), ... This threshold can be trained, and metrics and graphics can be used to determine the best one.

The task is to accurately detect all gamma events without missing any. Misclassifying a proton as a gamma only results in a loss of time, but missing a gamma event is critical, as it could lead to the loss of important information from space.

4.1.2 Energy Regression

This second task is quite different from the first one. A regression model is built to predict the energy produced by a gamma event. This indicates that only gamma events (or at least expected) are provided to train the model. The model will try to approximate the energy of the provided event to the ground truth.

The regression means that different metrics and graphics will be used to evaluate the model. It also means the model's performance will never be exact, and it needs to be analyzed to identify potential bottlenecks or misbehavior in its predictions.

The energy of an event can be in a very wide range of energies: 0.1 - 100 TeV. This means the model needs to work with different energy scales and perform well on each. The evaluation process is therefore updated to assess the model's performance across different energy scales, using bins and other level separations.

As previously mentioned, this task is performed after particle classification. It's expected to work only with gamma events; that's why the training is conducted on gamma diffuse events only and the testing on gamma point events.

4.1.3 Direction Regression

This is the final task and is similar to the energy task. In this direction regression task, the model must predict the origin of an event by estimating its coordinates. The model's input is expected to contain only gamma events. The objective is to have the model estimate coordinate values as accurately as possible to the true direction from which the event originated.

The particularity of this model is the fact that two values need to be predicted. The coordinate system used for this model is the Altitude-Azimuth system, specialized for locating objects in the sky. The model's performance will never be exact, and it needs to be analyzed to identify potential bottlenecks or misbehavior in its predictions.

A coordinate system as an output of the model means the evaluation is quite special. The coordinates need to be evaluated individually and combined into the coordinate system to identify which coordinate is less accurate and the model's global accuracy. The predictions are provided in degrees and are limited to 0-90° for altitude and 0-360° for azimuth. The evaluator should be aware of the dimensionality to choose appropriate metrics and graphics to visualize the model's details.

As previously mentioned, this task is performed after particle classification. It's expected to work only with gamma events; that's why the training is conducted on gamma diffuse events only and the testing on gamma point events.

For the direction regression task, there are two implementation modes. In camera direction mode, available only in mono, the arrival direction of the air shower is predicted in camera coordinates (degrees or radians). In sky direction mode, available only in stereo, the true sky arrival direction is predicted using horizontal coordinates, combining measurements from multiple telescopes to locate the event's origin.

4.2 Generation of data

Training models on complex task like the reconstruction of gamma-ray events requires large amounts of data. Data cannot be labeled by hands because of the complexity and time required to proceed. Therefore, another way must be introduced : Simulations. A simulation tries to artificially produce air showers for a gamma-ray events respecting certain restrictions and configuration. The advantage is that the ground truth (particle type, energy, direction) is already known and can be used to evaluate predictions tools and train models. The only issue with simulation is that it can't completely reproduce complex situations that occurs in reality such as the noise detail.

The simulation framework used for IACT data consists of many steps to provide a complete simulation [4]. The simulation works in two distinct steps using two different tools. It allows to generate realistic simulated data to analyze particle behavior and to train models efficiently. The figure 4.1 details which components does what on the simulation.

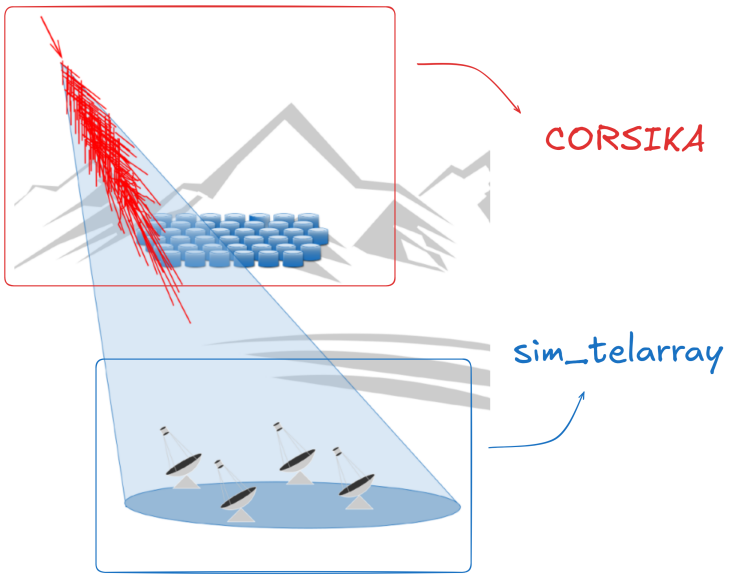

Figure 4.1: Visualization of parts of the simulation. The simulation can be separated in two elements : CORSIKA and sim_telarray. The first one focus on initializing a particle and build interaction with the atmosphere. It includes the generation of the air shower and related Cherenkov light. On the other hand, the second component focus on simulating the detector part : camera and telescope. It includes the triggering level system from it. [42]

The first component is the CORSIKA (COsmic Ray SImulations for KAscade) tool [12]. This tools simulates the interaction of primary particles with the atmosphere. It tracks the development of the air shower and related Cherenkov light emissions resulting from the interaction. The simulation is executed to have generated Cherenkov lights hit the detector layout. The simulation can be configured with many parameters like the particle type, energy range, altitude of the interaction or others. The treatment of hadronic and electromagnetic interactions needs to be separated. It provides also optimization processes to reuse generated shower but with different impact points.

The second component is the sim_telarray. This tool is oriented to simulate the telescope optics and camera response, in short, the hardware side. It simulates the detection of the events by the telescopes. It takes Cherenkov light generated by CORSIKA as an input to simulate the telescope and camera behavior. There are also lots of parameters that can be provided to vary the event detection such as the noise from the transmission and from Night Sky Background. It also provides a triggering system just like in reality to faithfully reproduce the detection. It finalizes by digitizing detected elements into waveforms samples adapted to future use cases.

The whole simulation process follows a Monte Carlo simulation approach [56] [33]. It tries to simulate randomness using probability distributions. It generates multiple samples according to theses probabilities on multiple parameters. With this, you can get an estimated behavior of a system without computing every possibilities, allowing to gain lots of resources and time. Complementary images are available with figure 4.2 to explain the Monte Carlo simulation.

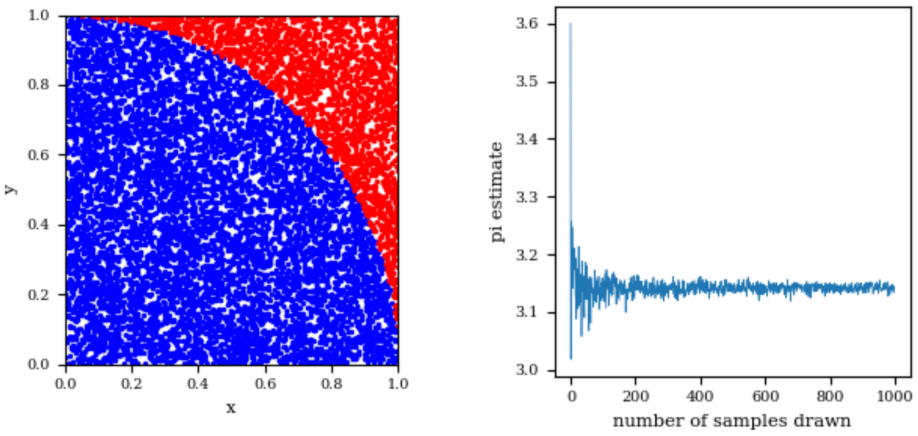

Figure 4.2: Monte Carlo simulation. Using the randomness of parameters, simulations have been produced in order to estimate the value of π in this situation. It doesn't need to compute every possibilities to approximate π as the right graphic displays. [39]

It starts by taking random samples for input to the system. As an example, for a IACT simulation, it refers to particle type, energy and direction but also context configuration such as the noise and others elements. Then the generation starts by doing many simulations with random parameters to cover statistically most of the scenarios. With all the simulation, it's, then, possible to check distributions, variance or uncertainties of the system simulated.

With IACT, this approach is essential because interactions of particles with the atmosphere are stochastic and the air shower development varies because of the total randomness of reality.

4.3 Format of files

The detected gamma-ray events are stored in various places across the working environment (Calculus, Baobab, etc.).

There are two types of files encountered in this thesis: .simtel.gz and .h5.

The first one is a compressed CTA simulation file produced by "sim_telarray", one of the simulation tools. As it's said, the thesis uses simulated data rather than real data. The reasons behind this choice are documented in the following chapter (4.2). It contains the raw simulated data, including every detail related to event generation. The generated files are too large to keep in raw format, so Gzip is used to compress them. This allows us to keep a massive amount of data on Calculus.

The second type of file is the .h5 extension. This second type of file is generally obtained after decompressing .gz files. The file's content depends on how the conversion from .gz to .h5 files was performed. Refer to the chapter dedicated to the conversion to see how it was done (4.4.2) and which elements are kept. To decompress the files, the CTAPIPE library is used as a standard approach to maintain a uniform data structure.

4.3.1 Content of HDF5 file

HDF5, or Hierarchical Data Format [15], is, as its name suggests, a hierarchical container for managing and storing data. It has been formatted to easily manage various data, is easily maintainable and understandable by others, offers high performance for reading and writing data, handles large datasets, and supports various data structures using a metadata system. This format is widely used by scientists and industries. Having everything stored in a single place isn't an issue, as HDF5 is optimized for selective access via chunks.

This is exactly what's needed for IACT Data, given the variety and complexity of the data. Its design is built to enhance performance, scalability, and reproducibility, which are essential for the data used. It could be compared to an independent filesystem at this point. Some libraries help handle this format in detail, including all the features related to it. This will not be explored in detail as it is out of the scope of this thesis.

The structure of an HDF5 file is quite simple to understand. It's close to a directed graph in visual, with nodes linked together to form a hierarchy. It includes three distinct elements: groups, datasets, and attributes. Groups can be viewed as directories or folders and contain a category of data (e.g., simulation configuration). It stores other groups and datasets. Datasets can be seen as files that store information in multiple dimensions and data types. It contains detailed values about a specific aspect of data. Attributes are information in a key-value format. It can be used to indicate data provenance, versions, or descriptions. Additionally, attributes can be used at different levels and directly connected to groups for metadata. The figure 4.3 displays an example of HDF5 file architecture.

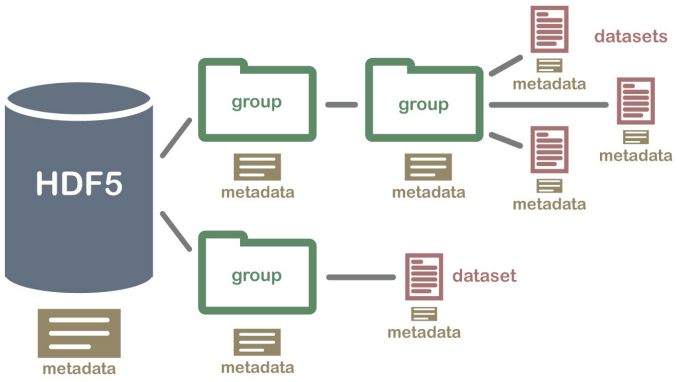

Figure 4.3: Example of HDF5 file structure with three elements (group, dataset, attributes) employed. By connecting groups and datasets, a structured dataset can be created, with optimal separation of data while keeping everything in a single file. Understanding and working with this format is easy because of the metadata distributed at each level of the hierarchy. [18]

Depending on the simulation and information kept, the content of the HDF5 file might change. Each chunk of data is dedicated to a specific state of gamma-ray events or simulation/configuration parameters. Here is a list of groups that can be found in these files [49].

- **Configuration** : Processing parameters and software settings to produce data
- **R0** : Raw waveforms in each pixel (uncalibrated)

- **R1** : Calibrated waveforms (in photoelectrons, pedestal subtracted)
- **DL0** : Event-level pixel data before image parameterization.
- **DL1** : Integrated charge and peak position of the waveform in each pixel (Hillas parameters, frames).
- **DL2** : Reconstructed event parameters (energy, direction, primary type)
- **Simulation** : Simulation and Monte Carlo parameters

Data range from waveforms (R1) to reconstructed event parameters (DL2) via multiple processing and prediction steps. This data can be manipulated at different levels to yield different results, depending on the use case. There are some additional steps, such as DL3 (IRFs) and DL4, but they aren't addressed in this thesis. The figure 4.4 gives an overview of the data level workflow. It takes a data-oriented, event-wise account, starting from DL0.

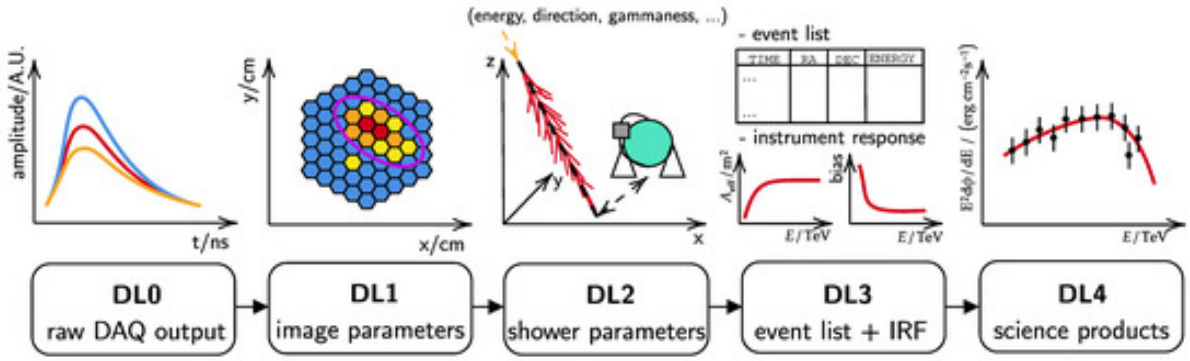

Figure 4.4: Workflow of data level (excluding R0 and R1). It shows that information extraction of various types is performed at every stage of the data using previous results. DL3 and DL4 are just for the show as they aren't part of the thesis. [37]

Configuration

The configuration group described how the simulation and its associated data were generated. It contains simulation metadata, enhancing its reproducibility. The first thing it includes concerns the instrument used for the simulation. It includes optics for the telescope or camera geometry. There are also simulation details for the Monte Carlo, such as the noise level, the range of features from the air shower, the simulation schedules, and even the observation position. It also includes where the telescope is pointing and other aspects that help to understand how the simulation is conducted. Metadata is also important in this part, as it is used by multiple processes, such as the CTLearn library, to verify the quality and success of the simulation file.

R0 - Raw waveforms

R0 data [12] is the starting point of gamma-ray events. It corresponds to the lowest level of IACT data. It contains raw detector output directly from the telescopes, as well as values from the camera electronics, without any modification or processing. This data is in the form of waveforms, one for each pixel, per gain channel, and per a defined time slice. It represents the on-line streamed raw data. A gain channel is used to read pixels with different amplitudes and determine the optimal way to read them. It includes parameters and information on events directly from the hardware and on digitization-related topics. Most of the time, it's not used

for simulated data, as simulations are more stable this way and require less parametrization. These waveforms are also called ADC (Analog-to-Digital Conversion) waveforms, representing the digitized time evolution of the signal measured in a camera pixel [2]. Figure 4.5 represents R0 data in the most minimalist way.

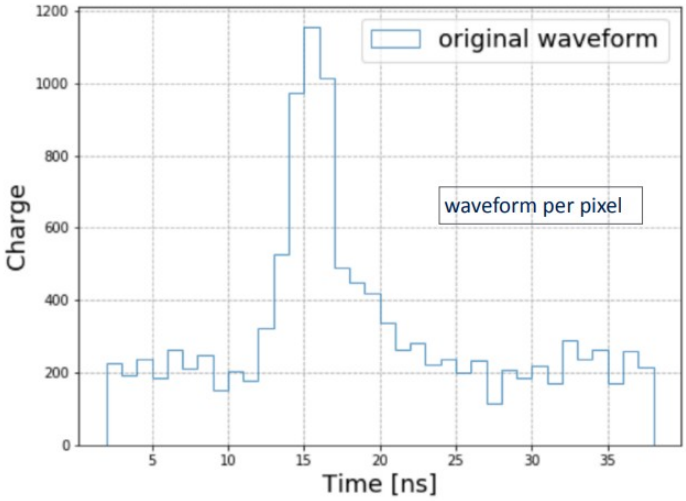

Figure 4.5: Example of R0 waveforms. It's a representation of the ADC counter and the signal for a pixel at a specific time slice, with an integer value for each nanosecond. It's a representation of the data detected by the telescope's camera in the purest form. [42]

The data used in the thesis does not include any datasets related to this part; therefore, its content isn't detailed.

R1 - Calibrated waveforms

R1 data is also waveforms, and it's the second level of data in the CTAO environment. It's either directly obtained from the simulation (depending on settings) or processed from R0 data in two steps. An offset per pixel is subtracted, and the signal is converted from ADC counts to photoelectrons. Doing these two steps is primordial if parameters and patterns need to be extracted from the waveform. The noise is slightly reduced by proceeding like this.

The HDF5 group here includes the event ID, the event time, the selected gain channel, the waveform, and other interesting event details. The figure 4.6 shows the processing performed on R0 data to obtain calibrated waveforms.

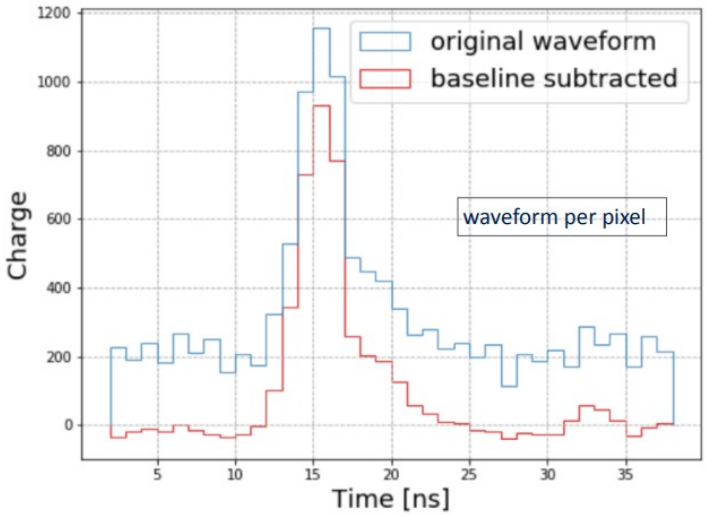

Figure 4.6: Example of R1 waveforms. These are calibrated waveforms with a clear distinction between the noise and the air shower visible. The photoelectrons are represented by the y-axis. [2]

DL0 - Archived data

This level of data isn't addressed in the thesis, but it's worth mentioning. This level is meant to retain only important data and reduce the amount of information by keeping only a small percentage. DL0 enables this reduction by using tools like the optimal gain selection. This small percentage is then used to reorganize the data into a structured format for feature and pattern detection. It converts the information into events standardized across telescopes' cameras. Globally, this can be seen as a preparatory step for analysis.

This step is optional, which is one reason only a few documents mention it. It serves as a bridge between the hardware data (camera and trigger system) and the physics interpretation and data processing.

The data used in the thesis does not include any datasets related to this part; therefore, its content isn't detailed.

DL1 - Image parameters

DL1 data changes the current image representation from pixel waveforms to scalar quantities and per-event pixel arrays. The frame is a representation of the per-event pixel arrays, with scalars corresponding to an image from it. It extracts the charge integration, the pulse time, the gain channel, the Hillas parameters, and some event details, such as the telescope used or the triggering type. It helps provide context for the event. The figure 4.7 illustrates the charge integration and the pulse time extraction, where the information is contained.

The per-event pixel array is also cleaned at this data level to remove noise and keep only the Cherenkov light. It uses the pixel charges and arrival times to proceed. The cleaning of the array isn't part of the thesis, but it's still worth mentioning. There are multiple configurations to proceed to clean arrays/images. Refer to this thesis for more details [13]. The cleaning allows for the separation and highlighting of detected events.

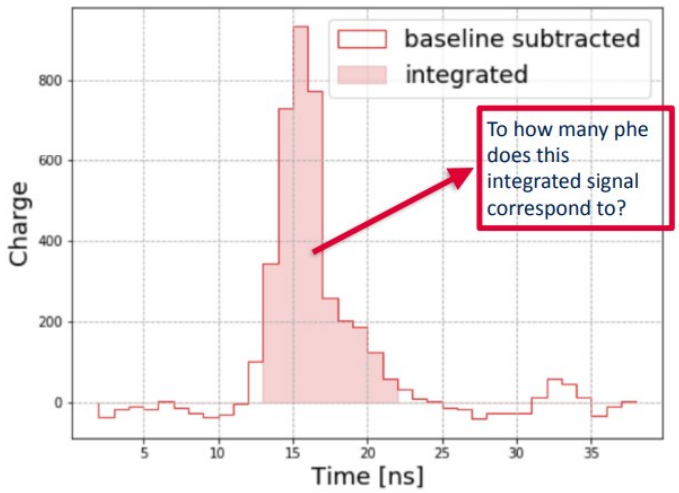

Figure 4.7: Extraction of the valuable information from the waveform. It accounts for the signal's peak and keeps a small window around it. Then the charge is calculated by adding up the charge in photoelectrons. This returns the pixel value. [42]

Figure 4.8 displays an example of Hillas parameters and how the parameters are determined.

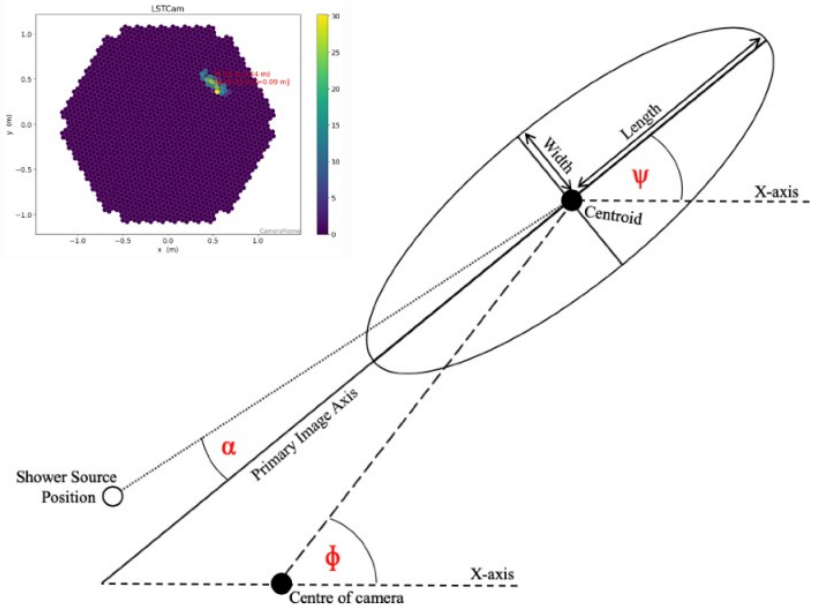

Figure 4.8: Hillas parameters extraction. This displays various parameters computed during a Hillas parametrization for each event in a per-event pixel array. The parametrization is done on an event from a clean array, as shown in the picture. Some examples of parameters : Size, Width, Length, Center of Gravity, Leakage, etc. [42]

If a comparison is needed, DL0 waveforms (time series) grouped together form a video, and DL1 extracts the most important information from the video to build a single image. It does that by performing charge integration, taking only the peak of the waveform (event) signal, and extracting it as a single value. It makes DL1 compact and uniform by significantly reducing the information.

These elements are used to reconstruct metrics such as particle type, energy, or direction using algorithms and machine learning models, by providing them with these parameters. On the other hand, Deep learning models tend to take the array/image to learn the features.

This group contains multiple types of information, such as event triggers, the Hillas parametrization, images, and image charge. It also has some statistics about images. It stores information per-telescope, since images from one telescope differ from those of another.

DL2 - Reconstructed events

DL2 is dedicated to reconstructing events. It summarizes the event by calculating information like the particle type, energy, and direction of the air shower. This is obtained by analyzing telescope observations (one or multiple) and applying them to algorithms or models.

In the case of Deep Learning models, they work directly with pixel arrays to generate this information. The figure 4.9 displays an example of pixel array representation. This representation needs some cleaning and conversion to a usable format for neural networks. The transformation step is required because the current CNN infrastructure operates on rectangular logic when the pixel array/frame is in hexagonal pixels.

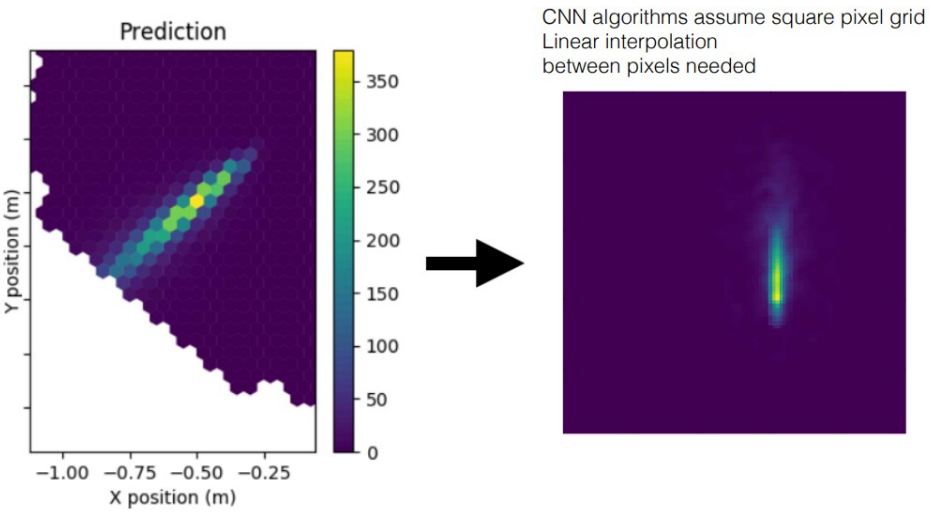

Figure 4.9: Example of an event in pixel array or frame. It appears on the left side of the pixel grid in the telescope image. These pixels are hexagonal and need to be converted to rectangular ones to work with deep neural networks and classic libraries. [42]

It's used to evaluate a model's performance and determine whether it succeeded in identifying important patterns and features. The data is compact and gives straightforward results and statistics for detailed analysis and for the conclusions to be drawn.

The group is composed of the model's results. During the thesis, models were trained on only one task at a time, so the group contains only the reconstruction results for a single task.

Simulation

The simulation group isn't at the data level as previously seen. This group is dedicated to the truth of Monte Carlo simulations and stores ground-truth elements, such as parameters

from simulated air showers. It also includes additional information, such as the distribution histogram of an air shower, simulated images from the camera, or Hillas parameters derived from them.

It's useful to evaluate neural network results against reality and gauge their precision. This data is only available in simulation, as in real data, there isn't enough labeled data to train a model. Events would need to be reconstructed by hand, which would take too much time.

4.4 Compliance with the server

The data available in Calculus (3.6.2) cannot be used directly to train models and requires processing before being used on the server. This processing step is necessary for multiple purposes, as explained below.

The data used to generate the model and, therefore, transferred from Calculus are displayed in this chapter (3.6.2). The steps to obtain correctly processed data were executed separately throughout the project, resolving issues as they arose to work with the data. The order in the subsections reflects the order of the data processing steps.

Normally, to use information, only data transfer and conversion are required. When going into details, issues with the location and size of the data were raised, requiring modifications.

4.4.1 Transferring to the cluster

The first step in data processing is a migration. It is the process of getting the data stored in Calculus to the Baobab cluster for daily use. This is a one-time process, as the data is used to explore models and isn't a production phase where data is transferred after it is generated. Data migration is handled with a single command that transfers the desired data from a folder.

```
rsync -av -e "ssh -i {SSH KEY}"
--files-from=<(ls | sed -n '{start_file},{end_file}')
. {cluster_address}:{destination_baobab}
```

This command, when executed in the folder, will take a determined number of files by position (from the starting point to the ending point) and send them to the cluster. The cluster address needs to be provided before the folder where you want to store the data. To know which data is transferred, refer to this chapter (3.6.2).

4.4.2 Conversion of data

The data, after being transferred from Calculus, isn't ready for use. Indeed, the data is in a ZIP format, which is not usable by CTLearn. Therefore, the files need to be unzipped.

A script using the **ctapipe** library will convert the Simtel file (zip) to an HDF5 format. The script doing the transformation is available in the appendix. The script allows for precise control over what to keep from the zip file when unzipping. Depending on what is wanted, images, parameters, or waveforms can be ignored during the decompression. The script applies procedurally to each specified file separately, not the most effective way to proceed, but it could eventually be parallelized across multiple CPUs. The script is built so that only ".simtel.gz" files are unzipped.

The files aren't combined during the transformation, ensuring the same number of compressed and decompressed files at the end of the process. The Simtel files aren't suppressed at the end of the script because, depending on the use case, other information that isn't kept for a given scenario may be necessary for another.

In addition, another file is created when decompressing: the provenance log. This file is there to track the origin of an event. It includes the processing steps applied to it, the source of the data, and other similar details. The file size is large because of that. It's not required to have it to use the data. It can be suppressed to save space (as done for the thesis), but additional information about the data provenance is lost in the process.

4.4.3 Moving Data across the cluster

Depending on the size and computational requirements of the models, the data location on the cluster changes. At one point, the available space for data migration wasn't enough. Therefore, a new location needed to be found. At another time, the speed at which files could be read wasn't sufficient when generating models due to the location. Therefore, a new location had to be found as well. This operation has been done a few times, depending on resource and time availability. Some details about the spaces used on the cluster can be found in the corresponding chapter (3.6.1.3).

Moving data across the cluster is not complicated. It relies on a single command to move all data from a specified folder to another location (commands **cp** or **mv**).

4.4.4 Merging files

Data on Calculus is stored in thousands of small files containing events. Each of these files has different simulation configurations. This is an issue when files need to be read for multiple tasks, as it significantly slows down the processes and tasks that depend on them. An issue related to that is well documented here (7.2). To reduce this I/O bottleneck, a tool in the ctapipe library is available to safely merge files together : **ctapipe-merger**. The script is designed to directly run a command with SLURM, so the merge works correctly. There is a possibility to indicate how many files the merge should result in. By providing a directory, it will merge all HDF5 files into a defined number of HDF5 files (in their original order).

4.4.5 Reducing data clutter

Simulation data contains a lot of information about events, such as their format, simulation parameters, and even the waveforms associated with them. Depending on the case, some information contained in the files might be more useful than others. It could be useful to remove this unwanted information to gain space on the cluster. With this in mind, a script has been created to extract only the event frames and parameters. Waveforms aren't used during model training, making them unwanted data. The script cut the file to keep only the wanted tables and information. This process allows to cut almost 90% of the file as waveforms take the majority of them. This process is executed for each file separately. It keeps only three sections: configuration (simulation environment, like the telescope), dl1 (event frames), and simulation (simulation configuration).

5 MLOps approach

MLOps or Machine Learning Operations [54] [7] is a methodology created to make machine learning models (deep learning and others too) more reliable, reproducible, deployable, and maintainable in production. It's an extension of the DevOps [53] set of practices, but oriented towards data and models. It focuses on the practical implementation and management of models.

Nowadays, large datasets are used to train various models to find the most efficient way to predict a value, as in this thesis. MLOps is the solution to handle this. Its aim is to optimize the use of AI models and the data they rely on to save time and resources. Without MLOps, developing an AI solution requires significant resources, a lot of time, and a really deep understanding of the process for managing models and data. MlOps provides automation, reproducibility, and efficiency in the development and production environments through the deployment, monitoring, and maintenance of models. Figure 5.1 shows the traditional MLOps life-cycle of a model.

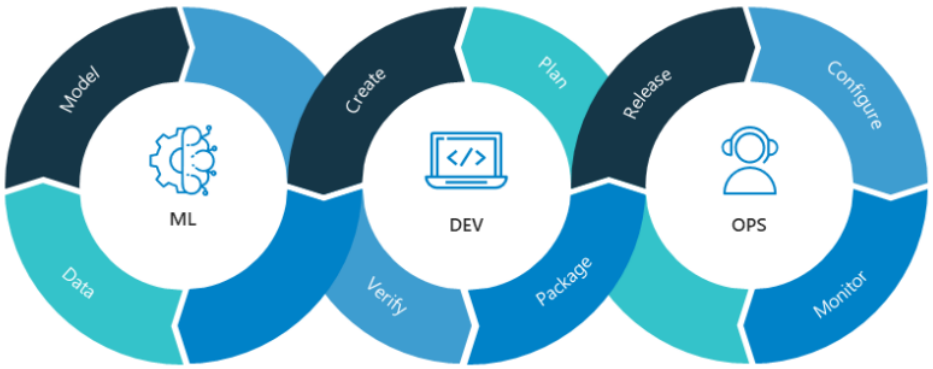

Figure 5.1: MLOps cycle representing the life-cycle for a model and the related data in order to deploy a model in a production environment. Deploying a model in a production environment isn't the end of the process; monitoring and maintenance are ongoing, leading to model refinement. This means the cycle can restart at some point. [32]

MLOps relies on a list of principles to build an adequate MLOps environment [55].

- **Collaboration** : Make communication between data scientists, engineers, and relevant parties easier to understand the process in place
- **Continuous improvement** : An iterative approach where models are monitored and evaluated in order to stay accurate with the use case

- **Automation** : Automate repetitive tasks to avoid losing time with configuration
- **Reproducibility** : Ability to reproduce an experiment to analyze/debugging and compare results
- **Versioning** : Tracking of changes from one model to another
- **Monitoring and observability** : Analysis of model performance
- **Governance and security** : Secure access to model and data across the pipeline.
- **Scalability and security** : Scalable infrastructure for a growing amount of data or increasing model complexity

As part of the thesis, some of these aspects are prioritized, including automation, reproducibility, and monitoring of model results by analyzing and comparing them. Another major part of the MLOps approach concerns data handling. In this scenario, data has been prepared for model usage in a separate state as the CTLearn library processes them differently. For more information, refer to the dedicated chapter (4). Among the elements of MLOps addressed in the thesis are experiment tracking with tools for generating reports or comparing models. For reproducibility and automation, tasks have been separated, and the code has been implemented to quickly understand the experiment's parameters and how to reproduce them. The automation stems from the fact that tasks can be chained and rerun in case of unexpected behavior, without affecting the entire process. Handling models requires a detailed, defined structure to facilitate their takeover and use.

MLOps also helps close the gap between astrophysicists (in the thesis scenario) and data scientists, so that each side can easily understand and use the other's contributions.

This is a wild area, with multiple ways to incorporate this aspect. The tools and techniques below provide an overview of thesis elements related to an MLOps integration in the CTLearn environment. The environment doesn't provide a fully adequate MLOps environment, as it could take forever, but it introduces some interesting elements that can be easily integrated into the CTLearn library.

5.1 Task Breakdown

One of the first aspects of the MLOps approach is to divide the pipeline into subtasks to handle them separately. The general idea was to handle the entire pipeline to generate a model within the same task, thereby chaining everything after another. This is an interesting first approach to generating a model, as everything is in the same place and the code runs to completion in a single pass. This approach has some limitations when manipulations or reruns are required. It's also not convenient because it's harder to identify where a problem occurs.

As we move towards an MLOps methodology, this approach needs to change. The idea here is to separate this global job executed in a Jupyter Notebook into smaller components, executing a sub-task, and dedicate them only to it. These tasks are split into their respective Python scripts and can be executed separately, in compliance with the task's requirements. For example, the testing task requires a model that has already been trained to work (so the training task should have been run at least once). The decomposition in sub-tasks is represented in the figure 5.2.

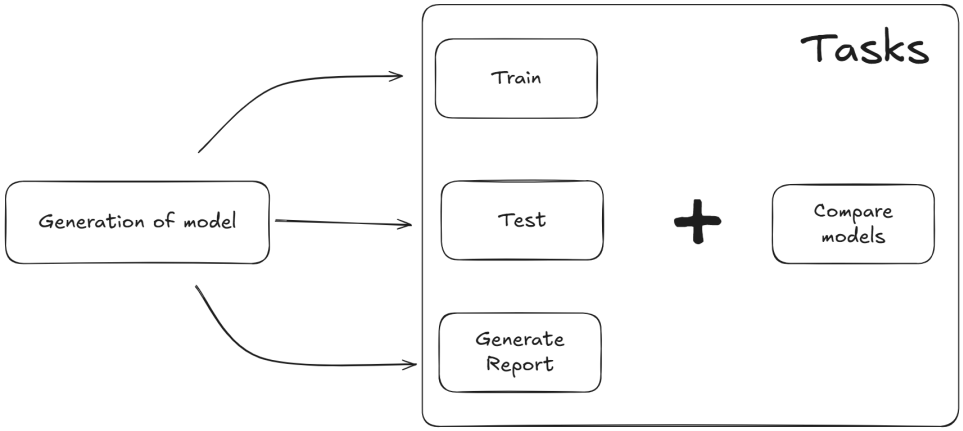

Figure 5.2: Task decomposition of the pipeline to generate a model and monitor it. The pipeline executed in a Jupyter Notebook is separated into three subtasks: training, testing, and model report generation. This ensures greater flexibility when executing tasks and is better suited for monitoring and maintenance. Another task is included in this picture : comparison of the models. This task is outside the pipeline because it requires multiple models. It's considered in the task decomposition as its usage is the same as the other sub-tasks.

Breaking down tasks is useful for a pipeline because it makes tasks independent, fault-tolerant, and scalable. The tasks are then easier to monitor, maintain, and update in an environment where new features are often implemented, and multiple models are generated. It also allows reproducibility and versioning of the tasks and models generated. The fault-tolerance is also important. Depending on the environment in which the script is run, restrictions may affect the task's success, such as limited time or resources. Having tasks separated prevents the entire pipeline from failing and limits environmental issues, as tasks are not run simultaneously. In case of failure, the task can be easily rerun without affecting the rest of the pipeline. Information about the failure is easier to identify (place, what).

The task decomposition could be extended in the future by including the data processing in the pipeline. In the thesis, data are processed at an earlier stage and are never updated thereafter. For more details, refer to the dedicated chapter (4.4).

5.2 Configuration files

Another element that can be included in the MLOps methodology is configuration files. Configuration files are essential to prevent code modifications after each pipeline run. MLOps methodology states that the code and the model logic should not change from one experience to another. The only changes should be provided by the configuration file. An overview of a configuration file is displayed with the figure 5.3.

The configuration files already available are examples of configurations for various types of models, with comments on possible modifications. The file structure is also a good way to understand the pipeline's needs and behavior with the CTLearn library. The separation has been made, so the tasks can look only at their allocated parameters to execute.

```yaml
prepare_model:
  model_type: 'ResNet' # ['SingleCNN', 'ResNet', 'LoadedModel']
  tasks: ['type'] # ['type', 'energy', 'cameradirection', 'skydirection']
  input_shape: [96, 96, 2] # Shape to determine
  num_classes: 2 # 2 for type classification, 1 for energy and direction regression
  temp_dir: '/home/users/v/varenneh/models/type/temp/' # Place to store temporary model (intermediate steps)
  CTLearnModel:
    attention_mechanism: null
    attention_reduction_ratio: null

#  CustomModel: # Only if Loaded Model
#    model_filename: "ComplexModel"
#    model_name: "ComplexName"

training_model:
  TrainCTLearnModel:
    output_dir: "/home/users/v/varenneh/models/type/resnet_batch64/" # Where model is saved
    input_dir_signal: "/home/users/v/varenneh/data/gammas_diffuse/train/" # Gamma values to use
    file_pattern_signal: ["gamma_*.h5"]
    input_dir_background: "/home/users/v/varenneh/data/protons_diffuse/train/" # Hadrons values to use (not for regression)
    file_pattern_background: ["proton_*.h5"]
    model_type: 'LoadedModel' # ['SingleCNN', 'ResNet', 'LoadedModel'] (same as in prepare_model)
    reco_tasks: 'type' # ['type', 'energy', 'direction'] (same as in prepare_model)
    n_epochs: 100
    batch_size: 64
    overwrite: true
    quiet: false
    percentage_per_epoch: 1.0
    log_level: "DEBUG"
```

Figure 5.3: Example of configuration file for a model generation. The model's task is particle classification of telescope images. The structure of the configuration file is important and must be maintained to work correctly. New parameters can be added based on the desired configuration and the available attributes in the CTLearn library.

The configuration is also a good way to track the experiment, especially when combined with other tools like Git. If the MLOps is extended to this point, it may be possible to see which changes improved the model with versioning. Also, when it comes to comparing models, it could be easily done by comparing configurations. The configuration files are saved in the same place as their model, allowing an additional description of the generated model. Reproducibility is ensured because all parameters are directly available. It's also useful in scenarios where an error is detected in the pipeline, as rerunning problematic tasks with the same configuration file keeps the experiment consistent and still accurate. These files are also a safer, more transparent way to work with an external library like CTLearn than to work directly in a Python script.

5.3 Report Generation

Report generation is the last step of the model generation pipeline. It can be seen as a way to control the model generated and its associated performance. It provides insights that experts can analyze in a centralized place to determine whether the model generated is accurate as expected.

Sections below describe the different elements that can be found in the report used to evaluate a specific model. There are multiple sections available in the report depending on the task. Some graphics and metrics are dedicated to a specific reconstructed value and therefore do not work or display for other reconstructed values.

- **Generic Section** : Metrics and graphics in every report for details about the model and runtime performances

- **Particle classification** : Metrics and graphics dedicated to this task
- **Energy regression** : Metrics and graphics dedicated to this task
- **Direction regression** : Metrics and graphics dedicated to this task. It's the same for both Camera direction and Sky direction regression.

Like other tasks, the report is generated from a Python file, with all metrics and graphics ready for use. It requires a fully trained and tested model in order to work.

Reports help advance the MLOps approach by providing detailed validation and centralized information. This can be used for user decisions or for automated decisions made by specialized tools. Graphics provides a debugging view of the model to identify where problems occur. It's also a collaborative tool, as different stakeholders can see what concerns them (astrophysicists with performance, engineers with learning curves, etc.). It provides an overview of the model's output, enhancing its reproducibility.

The report generated is used to compare and get a quick overview of the model performance. There are also enough metrics and graphics to identify potential weaknesses of the model. It reveals if the model is unstable or overfitting. The visual aspect used comes directly from a template generated by ChatGPT (for quick usage). The metrics, on the contrary, were chosen after analyzing possible ways to evaluate the different tasks on the internet and in the CTLearn Manager library [28].

5.3.1 Generic Section

In a report on different types of models, some generic information is displayed regardless of the task type. This information provides additional details unrelated to the model's performance. They are here to enhance reproducibility and to provide insight into the environment used to generate and use the model.

5.3.1.1 General information

When building a report, some general information is useful to be displayed. This information is the same across different model types, helps understand the environment used to produce the model, and could aid reproducibility.

The timestamp of the report's generation and a clear definition of the task the model is built for are examples of information displayed.

The number of epochs is an important additional piece of information to understand the model. It describes how many times the dataset is passed through the model. It helps to learn patterns and features progressively. In the report, the information about the epochs includes the scheduled number of epochs for training and the number of epochs actually used. This number can be lower than the first one if early stopping has been configured in the model.

In the same area, the batch size is also an important piece of information to understand the behavior of the training. It describes the number of samples combined to update model weights when training. It reduces the number of model updates during training by averaging batch losses.

The last information provided is the number of training and testing events used. The data environment is quite large and complex, which can lead to issues when reading data. This information helps determine whether the data was handled consistently across models during training and testing.

5.3.1.2 Model and Runtime Metrics

To compare and evaluate different models, it's necessary to look at how well they perform and the resources required to make them work. With this in mind, some metrics have been defined to get an idea of the requirements for a model. These metrics can relate to the size of a model, the number of operations required to compute a result, or the time needed to complete a task.

These metrics are shared and used for each task as they display an overview of a model (without specifics).

Depending on the environment and the model's use case, the importance of some metrics and their values can shift slightly. For example, the importance of metrics isn't the same for a model on an FPGA or a model with a lot of computation resources.

Number of layers and Number of parameters

These two metrics have a common purpose: determine the complexity of a model. The number of layers, or model depth, is an essential factor in explaining the model's complexity. The number of parameters scales with the number of layers and provides an indication of each layer's complexity.

Having more layers and parameters allows the model to identify more features and interesting patterns in the training data, thereby yielding better results. On the contrary, having fewer layers and parameters reduces complexity, which can be useful in infrastructures with limited resources.

For similar performance, the model with fewer parameters and layers should always be chosen to reduce memory usage and model complexity.

Estimated GFLOPs

This metric is related to the previous one (5.3.1.2). It also evaluates the complexity of a model, but with a different angle. Instead of directly measuring complexity and memory usage, it calculates the GFLOPs required to predict a value. This approach is more production-oriented and helps to understand the computational costs of the model.

GFLOPs corresponds to Giga Floating Point Operations, Floating Point Operations being the unit of mathematical operations required for one forward pass (for a CNN model). The number of GFLOPs varies from model to model depending on the complexity, architecture, or data used to predict it. This unit is directly related to a model's inference speed and the energy consumption associated with it. These two elements are very useful when the environment and use case of a model are important.

A lower GFLOP value means the model infers faster and consumes less energy. Minimizing this value is interesting when working with limited infrastructures.

Training Time, Inference Time and Inference per event

These three metrics are related, but they differ little. The objective with them is to calculate the time required to train and test an operational model, ready for use.

The training time includes loading the data, building the model with the CTLearn library, and training it. Data loading is performed to adapt the provided data to a format that a TensorFlow model can be trained on. The data format is a bit special for telescope data in the scope of the thesis, and adjustments need to be made to make it work (Refer to chapter 4.3 for more details). Each of these steps takes a certain amount of time, depending on the size of the dataset and the configuration of the model (number of epochs or model complexity).

The testing time includes only one task: model testing. It means predicting each event contained in the provided files and generating a corresponding output file containing the predictions.

The inference per event is the mean time the model takes to infer a single event. It is simply calculated by taking the total number of events and the total testing time.

It helps to get an idea of what is taking the most time between training and testing a model. The inference per event is an essential metric, depending on the use case, to know which model can make predictions faster. Minimizing this metric is the best thing for production purposes. It's also often used as a trade-off against the model's precision. It's difficult for both of them to be excellent, so a focus on one of them is needed.

5.3.1.3 Graphics

Graphics are pretty useful for displaying a lot of information and for understanding it quickly. These kinds of elements help recognize patterns and compare different models. It helps to determine where problems could lie and also communicate more clearly. These graphics will be used for every task and give general information about a model.

Training Loss

This plot shows how training and validation loss evolve over epochs. It shows the model's behavior over time and indicates whether it is learning correctly.

The main idea is to minimize loss over epochs (both for training and validation) and achieve a smooth curve. Irregularities in the curve are due to unstable training and sometimes require modifications to the structure or the batch size. A significant disparity between the two curves could also indicate issues with the training, such as underfitting, overfitting, or vanishing/exploding gradients.

This is a useful graph when discussing the training purpose and provides hints about problems in evaluating the model using metrics and other graphics.

Figure 5.4 displays an example of a training loss graphic.

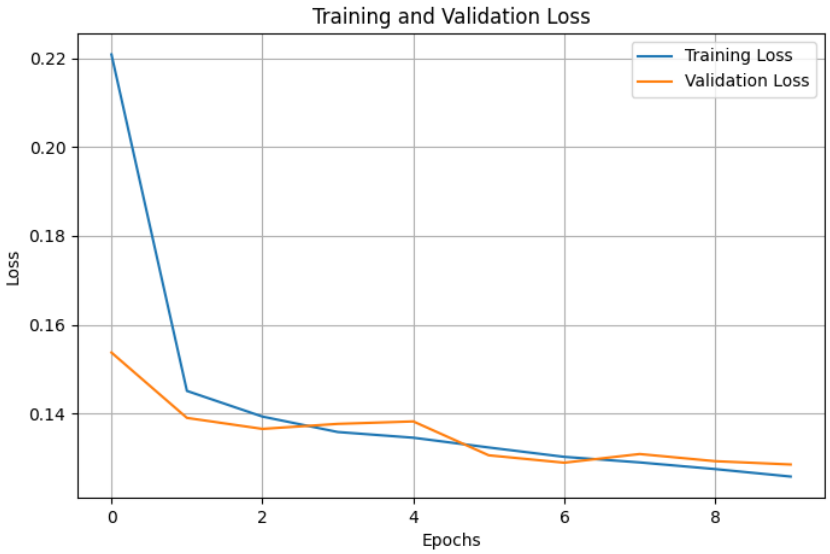

Figure 5.4: Graphic containing the training loss and the validation loss obtained through epochs. This helps identify potential problems during training, such as underfitting or overfitting. This could also indicate that the current number of epochs isn't enough to converge.

Model Summary

This one is a bit tricky. It's not really a graphic but rather an architectural display directly from a function available in TensorFlow: **model.summary()**. This command displays the model's structure as text, showing connected layers with their names, types, output shapes, and the number of parameters associated to them. Additionally, the total number of parameters and the number of trainable parameters are displayed alongside the model size. This helps understand the model's architecture in depth and its input/output format.

Figure 5.5 demonstrates an example of model summary display.

```
Model: "CTLearn_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input (InputLayer)          [(None, 96, 96, 2)]       0

 ThinResNet_block (Function  (None, 1024)              5357600
 al)

 fc_energy_1 (Dense)         (None, 512)               524800

 fc_energy_2 (Dense)         (None, 256)               131328

 energy (Dense)              (None, 1)                 257

=================================================================
Total params: 6013985 (22.94 MB)
Trainable params: 6013985 (22.94 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 5.5: Model summary containing information about layers of the model. Additional information, such as the total number of parameters and the model size, is provided.

5.3.2 Particle Classification

Particle classification is one of the tasks addressed in this thesis. For more details, refer to the chapter (4.1.1). This section includes metrics and graphics specifically designed to evaluate this task and provide insights from the results.

5.3.2.1 Evaluation Metrics

To compare and evaluate different models, it's necessary to assess how well they perform their tasks. With this in mind, some metrics are defined to provide a global overview of results based on multiple criteria. These metrics can reveal different problems encountered with the data, and, depending on the use case, have varying levels of importance and priority to address. Classification metrics can be relatively simple to obtain or complex, depending on the case.

In this particular classification, it's important to be aware of one thing. The classification is provided as a probabilistic prediction. It means the prediction isn't a class label but rather a probability that the event belongs to a class (in the task case, it's a gamma probability). However, common classification metrics work with class labels rather than probabilities. For this, a decision threshold is needed to classify the values. In this classification scenario, with a binary classification scenario (gammas or hadrons), it's really simple to classify the values. Values below the threshold are considered hadrons, and values above the threshold are considered gammas.

Accuracy

Accuracy is one of the most common metrics for evaluating classification performance. It calculates the percentage of values correctly classified across the entire dataset. This metric

is really simple, but it can sometimes lead to misinterpretation, especially given the results and the use case.

In this use case, the positives are the gammas, and the negatives are the hadrons.

$$accuracy = \frac{\text{true positive} + \text{true negative}}{\text{true positive} + \text{true negative} + \text{false positive} + \text{false negative}}$$

Recall

The recall (also called True Positive Rate) is another common metric used to evaluate classification. It has a slightly different behavior. It measures the model's ability to correctly classify all values in a specific class. This metric can be very valuable in scenarios where it's critical not to miss any positives, even if it increases false positives. It focuses on false negatives.

In this use case, the positives are the gammas, and the negatives are the hadrons. We therefore focus on classifying all gamma events.

$$recall = \frac{\text{true positive}}{\text{true positives} + \text{false negative}}$$

Precision

This metric is not used directly in the report, but rather as a subpart of another metric: F1-Score (5.3.2.1). This metric works similarly to recall. The metric, this time, measures the model's capacity to recognize values for a specific class. It focuses on false positives.

In this use case, the positives are the gammas, and the negatives are the hadrons. We therefore focus on the ability to correctly classify gamma events.

$$precision = \frac{\text{true positive}}{\text{true positives} + \text{false positive}}$$

F1-Score

This metric serves as a trade-off between two other metrics seen earlier: precision and recall. Normally, only one of these two values can be maximized while penalizing the other. Therefore, an optimal solution is needed to balance both. That is the F1-Score.

$$\text{F1-Score} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

Brier Score

The Brier Score differs from previous metrics. Instead of focusing on binary classification metrics like accuracy or recall, it directly uses probabilistic predictions to assess the model's quality. It's used to evaluate the model's calibration. This metric also penalizes confidence in a wrong prediction.

With other metrics, there is no distinction between 0.51 and 0.87 (both gammas in this case), even though the second value is better. Brier Score solves this. It measures how closely a model's predicted probabilities match the ground truth. It is the mean squared error between the probabilities and the ground truth, and it is often referred to as a cost function.

$$\text{Brier Score} = \frac{1}{N} * \sum_{i=1}^{N} (p_i - y_i)^2$$

5.3.2.2 Graphics

Graphics are pretty useful for displaying a lot of information and for understanding it quickly. These kinds of elements help recognize patterns and compare different models. It helps to determine where problems could lie and also communicate more clearly. These graphics will help clarify the classification.

ROC Curve

ROC Curve (Receiver Operating Characteristic curve) is a graphical plot used for particle classification. It represents a model's ability to identify differences in a binary-class problem (in the thesis case, gammas versus protons/hadrons). It uses the trade-off between the True Positive Rate (recall; 5.3.2.1) and the False Positive Rate (the proportion of negative instances incorrectly classified as positive) to construct the curve. It's a useful graphic because it doesn't account for class imbalance, unlike other metrics, which are useful for real-time applications where gamma events are rare.

It can be used, for example, to determine an appropriate threshold for a probabilistic classification model (see 5.3.2.1) or to compare models.

The ROC Curve also uses the AUC (Area Under the Curve) metric to summarize the curve's performance. It corresponds to the probability that the model classifies a random gamma event as higher than a random hadron.

Figure 5.6 is a clear example of a ROC curve, with a legend showing the AUC value.

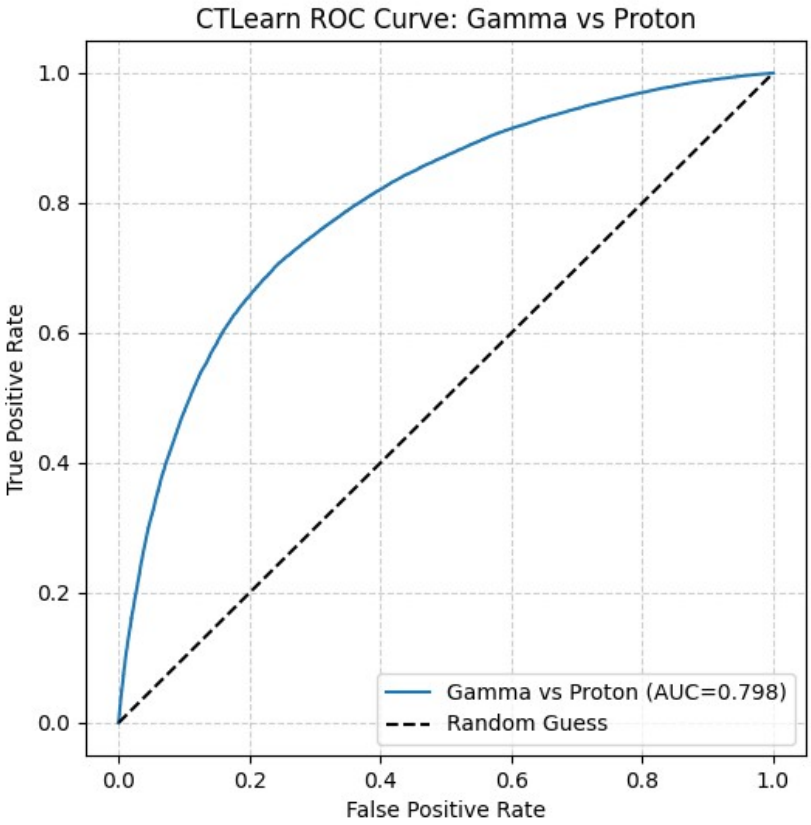

Figure 5.6: ROC Curve graphic displayed using TPR and FPR. There is also a diagonal displaying random guess values. The legend displays the model label and the AUC to get a better overview of the model performance on the curve.

If the curve is close to the random-guess diagonal, it means the model is guessing randomly. The closer to the top-left corner, the better the performance.

Gammaness Distribution

This graphic will display the distribution of data (gamma and hadrons separately) over the probabilistic predictions. The probability score obtained after classifying events is called the gammaness, and it's what is displayed here. It's helpful to see if the model can separate the gammas and protons clearly. It's also helpful to choose a threshold for the classification, as it shows more directly at what level of gammaness the gamma are most correctly classified.

The figure 5.7 displays with two distinct colors the distribution of gamma/hadron events at each level of gammaness.

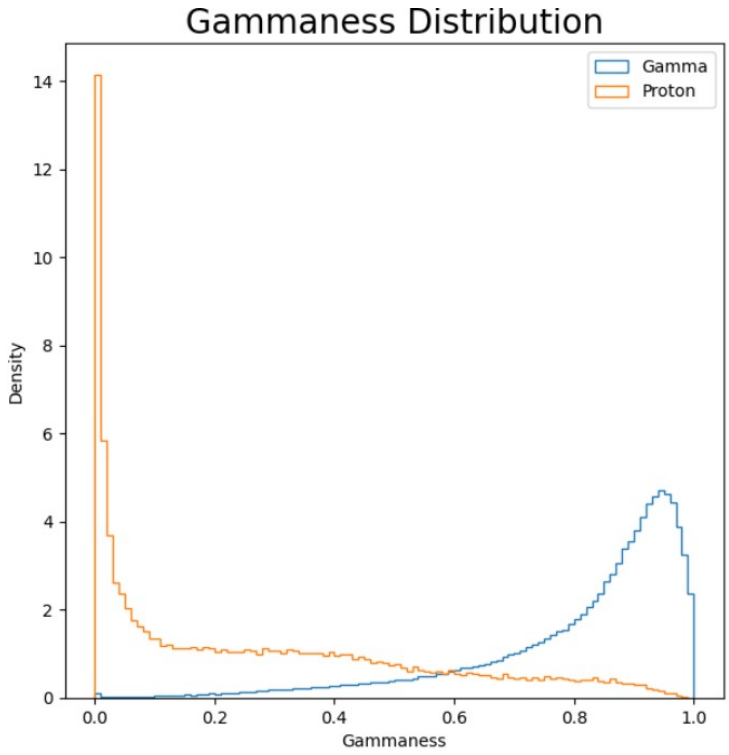

Figure 5.7: Gammaness distribution graphic where the distribution for each type of particle can be seen. The axes are gammaness and the event density at each gammaness level. In the example, the separation between gamma rays and hadrons is evident. It's not clear, meaning some improvements could be made.

Confusion Matrix

The confusion matrix is a table used to display classification results. Using this table, it's possible to calculate a model's accuracy, recall, or precision by comparing predicted classes with ground-truth classes. It also provides hints about potential imbalance in the dataset and the number of events classified correctly and incorrectly for each class. It requires the predictions to be labeled with classes and not probabilities.

The figure displays a confusion matrix for the classification problem of the thesis (gamma versus proton/hadron).

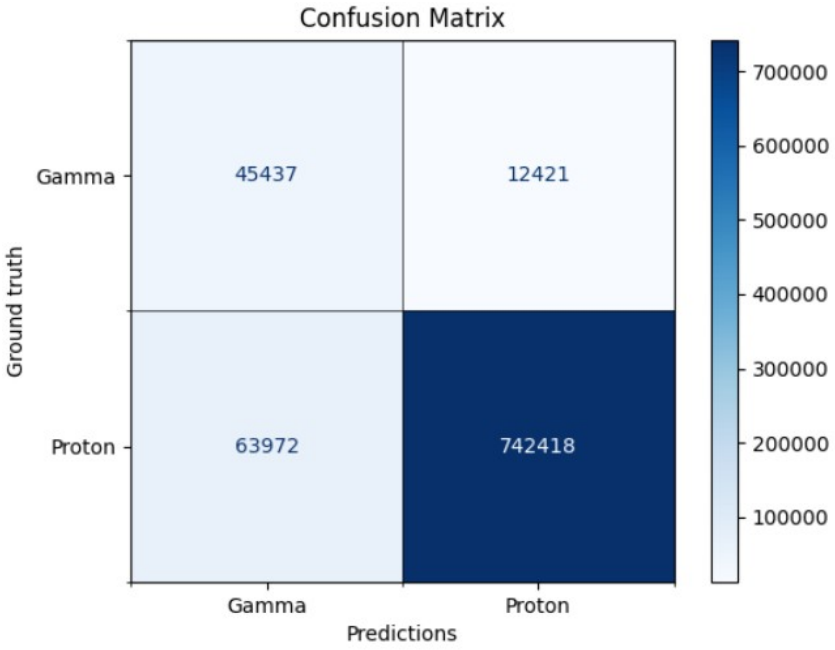

Figure 5.8: The confusion matrix displays a 2x2 matrix (in the task scenario, but could be more) describing gamma and hadron events classification. It shows if values were correctly evaluated and the corresponding number of values associated with the scenario. Colors are also displayed to get a quick idea of where most of the data falls. The x-axis displays columns corresponding to predicted gamma and hadron events, and the same goes for the y-axis for the ground truth.

Calibration probability

This graphic is related to the Brier score (5.3.2.1). The Brier score provides an overview of a model's calibration; this graphic provides more detail on probability mismatches. It measures a model's reliability by evaluating probabilistic predictions based on the likelihood that an event belongs to a class.

A model can be accurate but not well calibrated. For example, if the prediction probability is 0.8, it's expected that the model is correct 80

The graphic groups predictions by probability range. A mean probability is, then, computed alongside the true fraction of gamma events contained in this bin. With this, observations can be used to determine whether model confidence and calibration are well calibrated at each probability level.

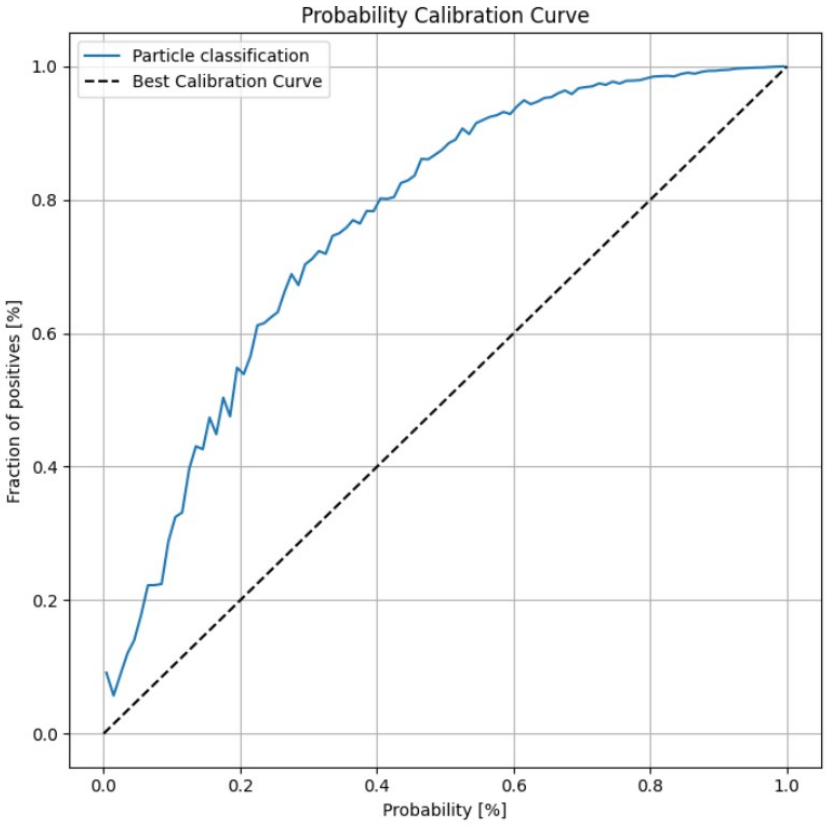

Figure 5.9: Calibration probability graphic displays a curve corresponding to the calibration for a certain probability. It is based on the probability (on the x-axis) and the fraction of positives (the percentage of gamma events in a given sample). A default diagonal is provided to display the best possible calibration. The curve can be above or below the diagonal, depending on the calibration.

The black diagonal indicates the objective and the best possible calibration value. It's expected to get as close to this as possible. Going under the diagonal means the model isn't confident, and going above it means the model is overconfident.

5.3.3 Energy Regression

Energy regression is one of the tasks addressed in this thesis. For more details, refer to the chapter (4.1.2). This section includes metrics and graphics specifically designed to evaluate this task and provide insights from the results. This task is normally conducted after doing the classification task. Therefore, the evaluation is mostly conducted only on gamma values.

5.3.3.1 Evaluation Metrics

To compare and evaluate different models, it's necessary to assess how well they perform their tasks. With this in mind, some metrics are defined to provide a global overview of results based on multiple criteria. These metrics can reveal different problems encountered with the data, and, depending on the use case, have varying levels of importance and priority to address. Regression metrics can vary and be complex depending on the task.

MSE

MSE (Mean Squared Error) is one of the most common metrics for evaluating the accuracy of regression models. It measures how far the model predictions are from the ground truth. To do so, it takes the average squared difference between predicted values and the ground truth. The difference is called the residual.

It penalizes large differences between the predicted and actual values. It means it's sensitive to outliers. The lower this metric's value, the more accurate the model.

$$\text{MSE} = \frac{1}{N} * \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

R^2

R^2 (Coefficient of determination) is a metric used to analyze the variation in the data. It shows how well the model explains the data's variability. It evaluates the quality of a regression model.

It measures the percentage of variation the model can capture. The higher the value, the better the model in general (but more sensitive to overfitting). This metric can't be used alone because of its limitations (it's sensitive to noise or doesn't work with non-linear relationships). These limitations can result in a coefficient of determination below 0, indicating that the model performs worse than a simple random prediction.

$$\text{R}^2 = 1 - \frac{\sum_{i=1}^{N} (y_i - \hat{y}i)^2}{\sum i{=}1^{N} (y_i - \bar{y})^2}$$

RMSLE

RMSLE (Root Mean Squared Logarithmic Error) is similar to MSE (5.3.3.1) but measures the difference between the logarithms of predicted and ground-truth values. It's useful because it focuses on relative error rather than absolute error, like MSE.

It isn't sensible to outliers like MSE (5.3.3.1) because of the scaling brought by the logarithms. This metric is well adapted for energy regression because of the multiple orders of magnitude (0.1-100TeV) that are predicted. Using relative error focuses more on the percentage of error than on the absolute value of error.

$$\text{RMSLE} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (\log(1 + \hat{y}_i) - \log(1 + y_i))^2}$$

MAE

MAE (Mean Absolute Error) is a metric dedicated to measuring the average difference between predictions and ground truth. It doesn't account for underestimation or overestimation, unlike other metrics. This metric is easy to interpret and robust to outliers. The only concern with this metric is its limited insight. This metric is similar to MSE.

Easy to interpret: if the value is 0.5 TeV, it means the predictions are, on average, off by 0.5 TeV from the ground truth.

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|$$

5.3.3.2 Graphics

Graphics are pretty useful for displaying a lot of information and for quickly understanding it. These elements help recognize patterns and compare different models. It helps to determine where problems could lie and also communicate more clearly. These graphics will help clarify the energy regression.

Energy Distribution

The energy distribution graph is a rather simple graphic. It's a histogram showing the distribution of ground-truth events across an energy range. This graphic can display the different types of particles to highlight their differences. No predictions are implicated in this one.

The figure 5.10 shows the content of this graphic.

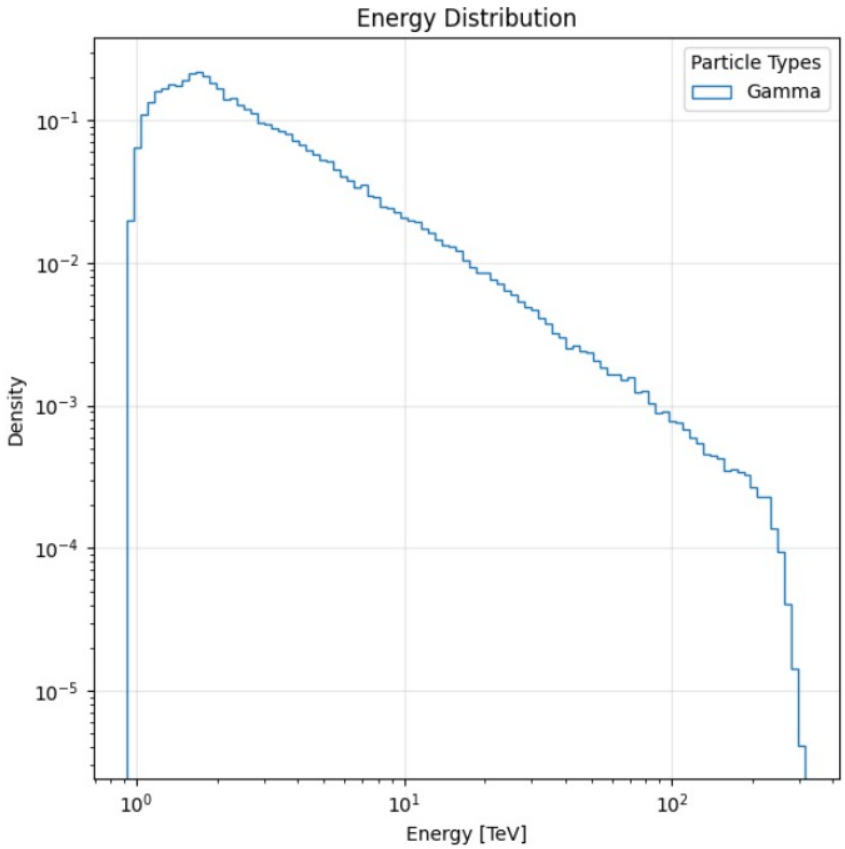

Figure 5.10: Distribution of events over energy ranges (bins). The histogram helps to get an idea of where events are concentrated. The x-axis shows the energy range for ground-truth values on a log scale, and the y-axis displays the number of events per bin.

Migration Matrix

The migration matrix is also a graphical tool for evaluating energy regression models. It shows how the true energy is converted into reconstructed energy (with bins). It shows how ground-truth values are reconstructed. It shows the distortion (divergence) the model imposes on the data.

This is used to show the reconstruction of the energy spectrum between real and predicted data. This is generally used for gamma-point data, but can be extended to hadrons/protons. The

events are distributed across bins (based on ground truth and prediction scales) to make it more readable.

The generated matrix should have its data concentrated along the diagonal, indicating perfect resolution of the events. This can be visualized with the figure 5.11. This graphic helps visualize where the model misinterprets events and assigns the wrong value. The warmer the matrix bin color, the more events it contains.

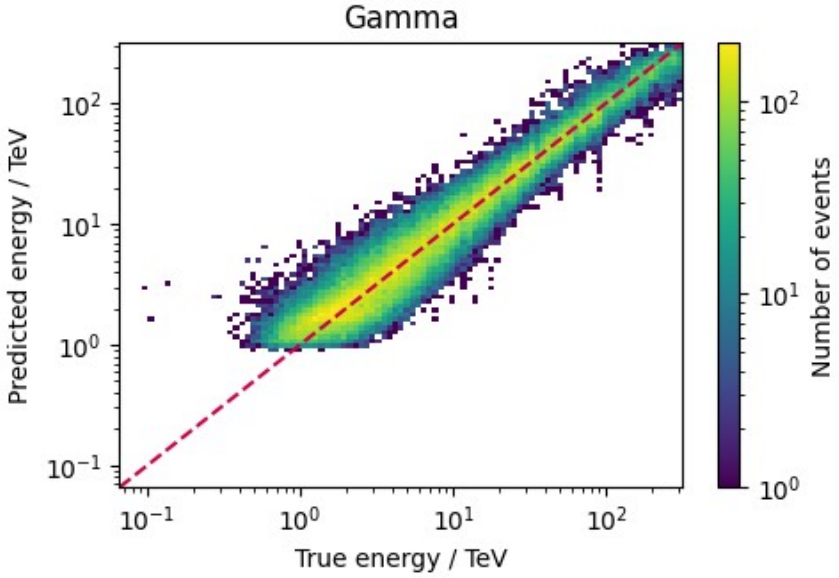

Figure 5.11: Migration matrices are useful to visualize an overview of events, energy predictions. By grouping them in bins, the graphic becomes more readable and interpretable. The scales are logarithmic to accommodate the range of possible prediction values. The axes for ground truth and predictions are displayed in TeV units (energy). A color scale has been added to provide more insight into where most of the events fall. The ideal result is to have events concentrated along the diagonal, describing the perfect predictions.

Multiple pieces of information can be gained from this graphic. A large diagonal means an uncertainty in the reconstruction of events, while a narrow one means a good reconstruction. The bias can also be assessed by comparing the reconstructed diagonal to the perfect one. Being above means the model is overestimating, and being below means it is underestimating. Other small details can be gleaned from analyzing the graphic (e.g., at what level of energy it works better, patterns, …).

Energy Resolution

The energy resolution is a graphic used to evaluate energy regression models. This graphic displays a curve showing how precisely a model can reconstruct the energy of gamma events. It shows the distribution of the relative error across the energy range as a percentage. Like this, it's possible to determine whether a model performs better in high- or low-energy ranges. It helps to understand the model's misbehavior within the energy range.

The calculation is based on the standard deviation of the Gaussian distribution of relative errors between the reconstructed/predicted energy and the true energy. The lower the energy resolution is, the better the spectral reconstruction (figure 5.12).

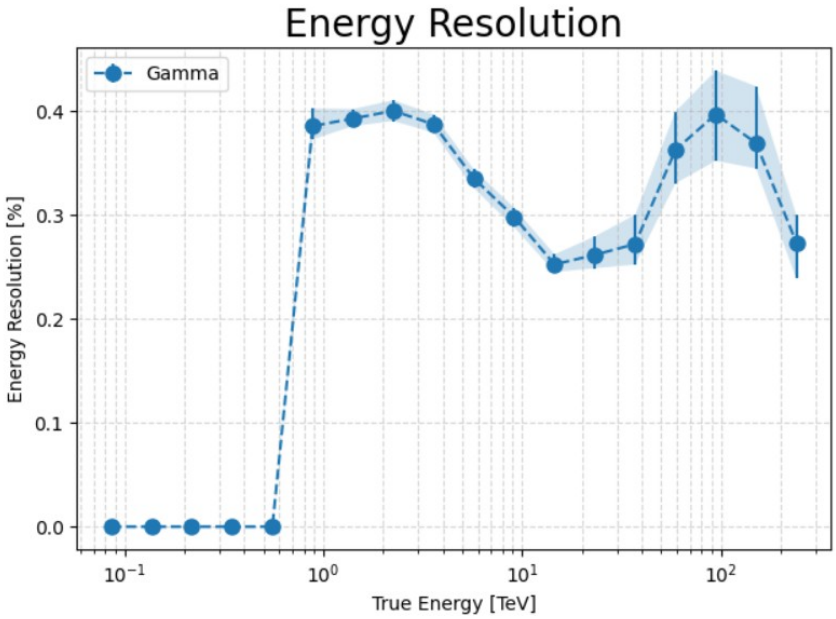

Figure 5.12: Energy resolution graphic computes how precise the model for reconstructing the energy value, depending on the energy scale (ground truth energy). The x-axis displays the real data on a logarithmic scale. For visualization purposes, the values are averaged in bins for defined ranges. The y-axis displays the spread (energy resolution) as a percentage of the predicted value. For each point on the graph, there is a shading to show the upper and lower bounds of the bin (68% containment interval) for the events contained in the bin.

It helps to see where the model performs better. At lower energies, it may be difficult to achieve good resolution, as it can be mistaken for noise. It's the same for high energies, where there are only a few events.

Bias/Standard Deviation

This graphic displays two curves: the bias and the standard deviation. It's normally computed as a single metric value for the whole dataset, but working with a wide range of energy makes it interesting to split the metrics into a curve for a defined range of energy (bins).

The bias is the metric that calculates how far, on average, a model's predictions are from the ground truth. It measures whether the model over- or underestimates the energy. In bins, the value shown on the graph is the mean relative error as a measure of bias.

The standard deviation (or spread) measures how much the residuals vary around the average error. It measures the model's precision. It gives the statistical uncertainty of the model. It can be seen as a precision metric per energy bin.

The trade-off between the two is important to know if a model is accurate and precise. Normally, with a better bias, the standard deviation is worse, and vice versa. Using bins, the observations can indicate which energy range is causing issues in the model. It can be seen in the figure 5.13.

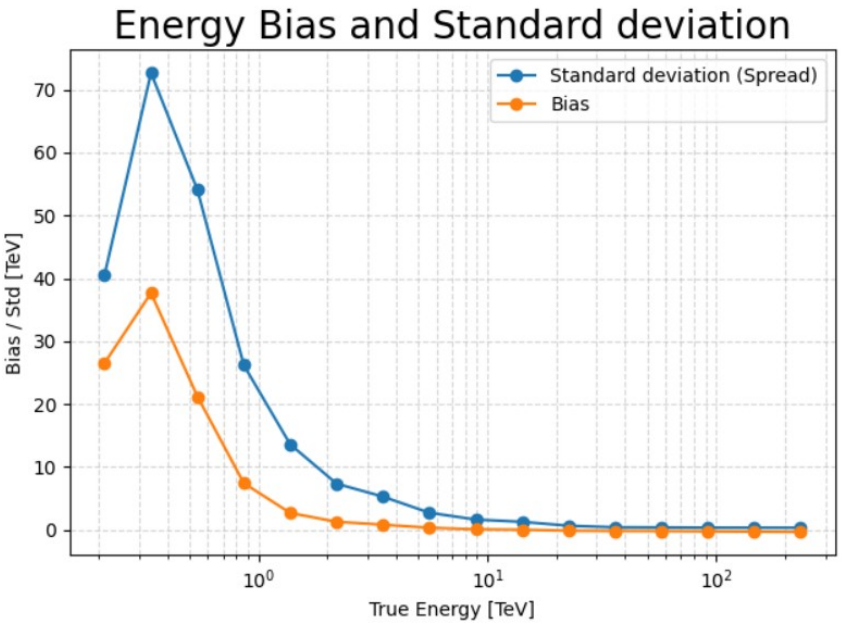

Figure 5.13: Bias and Standard deviation graphic. Two curves are displayed here : the bias and the standard deviation. The results are displayed using bins average their ground truth values and reconstructed energy errors. The x-axis shows the bins (range) on a log scale, and the y-axis shows the difference in TeV between the standard deviation and the bias.

5.3.4 Direction regression

Direction regression is one of the tasks addressed in this thesis. For more details, refer to the chapter (4.1.3). This section includes metrics and graphics specifically oriented towards evaluating this task and gaining insights from the results. This task is normally conducted after doing the classification task. Therefore, the evaluation is mostly conducted only on gamma values.

As a reminder, it's important to note that there aren't just one but two values predicted as outputs here: azimuth and altitude. These two predictions will be evaluated together or separately depending on the metric and graphic used.

The direction regression has two variants : Sky direction and Camera direction. It doesn't change the type of graphics used. Refer to chapter 4.1.3 for more details.

5.3.4.1 Evaluation Metrics

To compare and evaluate different models, it's necessary to assess how well they perform their tasks. With this in mind, some metrics are defined to provide a global overview of results based on multiple criteria. These metrics can show different problems encountered with the data and, depending on the use case, have different levels of importance and priority to comply with. Regression metrics can vary and be complex depending on the task. The measure used for every metrics is degrees. Radians were also considered but it's less precise.

MDAE

MDAE (Mean Directional Absolute Error) is a metric measuring the average angular error between predicted events and the ground truth. To compute the directional error, the

azimuth and altitude must be combined and converted to 3D vectors. This helps ensure a generic implementation of the metrics afterward, working across different coordinate types and preventing issues with angles, a common problem when working with coordinates.

$$y_i^t \text{ and } \hat{y}_i^t = \begin{bmatrix} x & y & z \end{bmatrix} = \begin{bmatrix} \cos(\text{alt})\cos(\text{az}) & \cos(\text{alt})\sin(\text{az}) & \sin(\text{alt}) \end{bmatrix}$$

With the converted values, it's easier to calculate the metric, as the interaction between the prediction and ground truth vectors is a simple dot product.

This metric is easy to interpret and robust to outliers. The only concern with this metric is the limited insight it provides. This metric is similar to MSE (5.3.3.1) and MAE 5.3.3.1). It's also rotation-invariant, meaning it doesn't depend on a coordinate system to work (using 3D vectors).

Easy to interpret because if the value is 0.5 degree, it means, on average, the predictions are off by 0.5 degree from the ground truth (in angular units).

$$\text{MDAE} = \frac{1}{N}\sum_{i=1}^{N} \arccos(y_i^t \cdot \hat{y}_i^t)$$

RMSDE

RMSDE (Root Mean Squared Directional Error) is a metric that also relies on the conversion to 3D coords (5.3.4.1) because it involves both azimuth and altitude. This metric measures the average angular error but penalizes larger errors. It's similar to MSE (5.3.3.1) but for directional evaluation.

This metric is sensitive to outliers and aware of distortions in dimensions introduced by the conversion of 3D coords. It's used in combination with other metrics to detect whether the model makes large errors.

$$\text{RMSDE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} \arccos(y_i^t \cdot \hat{y}_i^t)^2}$$

MAE - Circular

This metric focuses on evaluating one of the values predicted : the azimuth. The name includes 'circular' because the range of the azimuth angle is 0-360 degrees. This is the mean absolute error for this particular feature. The metric is similar to MAE (5.3.3.1) in all respects, with a twist to handle the circular nature of the data. The distance between one degree and 359 degrees should be two, not 358, as it should be computed using only MAE. That's why a minimum selection is introduced to handle this aspect.

The metric will report how far the predicted azimuth is from the ground truth in degrees.

$$\text{MAE Azimuth} = \frac{1}{N}\sum_{i=1}^{N} \min\left(|y_i - \hat{y}_i|, ; 360^\circ - |y_i - \hat{y}_i|\right)$$

MAE - Altitude

This metric is the same as the azimuth, but for the altitude this time. Here, the handle of the circular aspect isn't necessary, which means that the metric is just like the classic MAE (5.3.3.1).

The metric will provide how off the prediction of altitude is from the ground truth in degree.

$$\text{MAE - Altitude} = \frac{1}{N}\sum_{i=1}^{N}(|y_i - \hat{y}_i|)$$

5.3.4.2 Graphics

Graphics are pretty useful for displaying a lot of information and for understanding it quickly. These kinds of elements help recognize patterns and compare different models. It helps to determine where problems could lie and also communicate more clearly. These graphics will help clarify the energy regression.

Degree Performance Curve

This graphic is really simple but can offer interesting insights into a model's precision range. The idea behind the graphic was to see how imprecise the model can be while still achieving interesting accuracy. A curve is displayed to get a precision at different level of thresholds in degrees. The accuracy is then displayed for each threshold. It can help to note how precise the model is in range of degrees.

The degree of error is defined using the MDAE (5.3.4.1) metric, which measures the difference in 3D coordinates between predictions and ground truth.

Figure 5.14 shows an example of a curve that can be obtained.

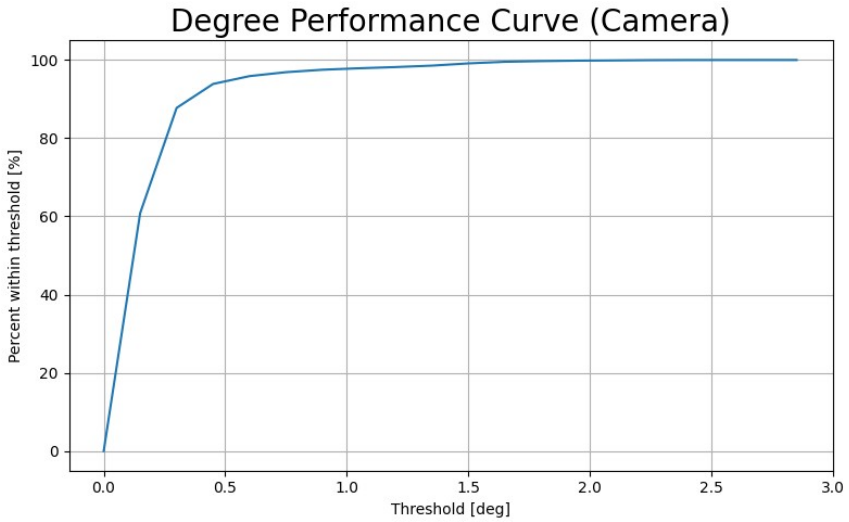

Figure 5.14: Curve showing the evolution of the accuracy with lower thresholds over time. The x-axis displays the arbitrary threshold, and the y-axis displays the accuracy at each threshold.

Altitude-Azimuth Distribution

This graphic uses visual code from the migration matrix to display the predictions for a direction regression model. It helps to visualize event density in a heatmap using bins. The axis corresponds to the two types of predicted coordinates : azimuth and altitude.

This graphic helps identify patterns in event predictions. As the telescope normally points in the same direction, a pattern should appear showing the density of events around the telescope's pointing area. The direction the telescope is pointing is indicated by a cross. This indicates the area around which events should be detected. If it diverges from it, it could indicate a bias in the model.

The choice of graphic (figure 5.15) is made to concentrate around the area where the telescope is pointing. If no pointing reference was given, using a sky plot was also considered.

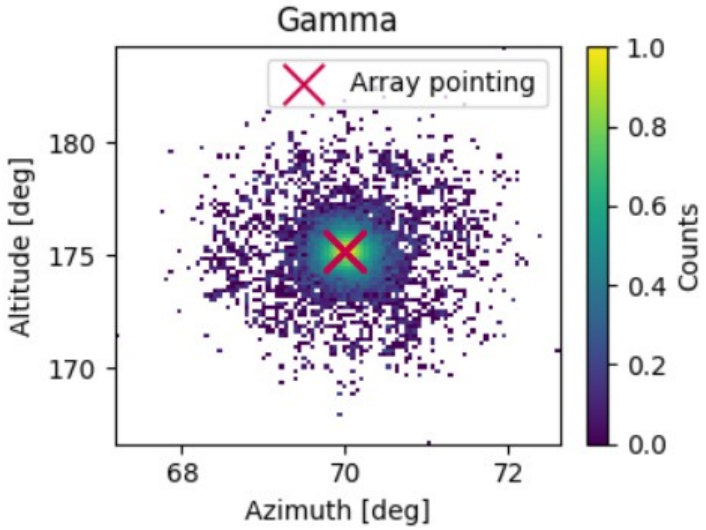

Figure 5.15: Distribution graphic showing repartition of the events in 2D coordinates system. The x-axis shows azimuth, and the y-axis shows altitude. The axes are in degrees, and the values are computed using bins. The density per bin is shown using a color bar. The cross represents the telescope pointing area, and events should normally be detected around it. Depending on the use case, the events can be displayed in different graphics depending on their type.

Angular Resolution

This graphic is used to evaluate direction regression models. It works in a similar way to the energy resolution (5.3.3.2). It measures how precisely the direction of an event is reconstructed. The direction, like other metrics, is constructed by combining the two predicted coordinates : azimuth and altitude. The events are regrouped into bins corresponding to the ground-truth energy. With this in mind, the angular resolution is calculated in degrees and displayed on the energy range scope. Like this, it's possible to know whether a model performs better in high- or low-energy ranges, thereby understanding model misbehavior.

This graphic is shown in Figure 5.16.

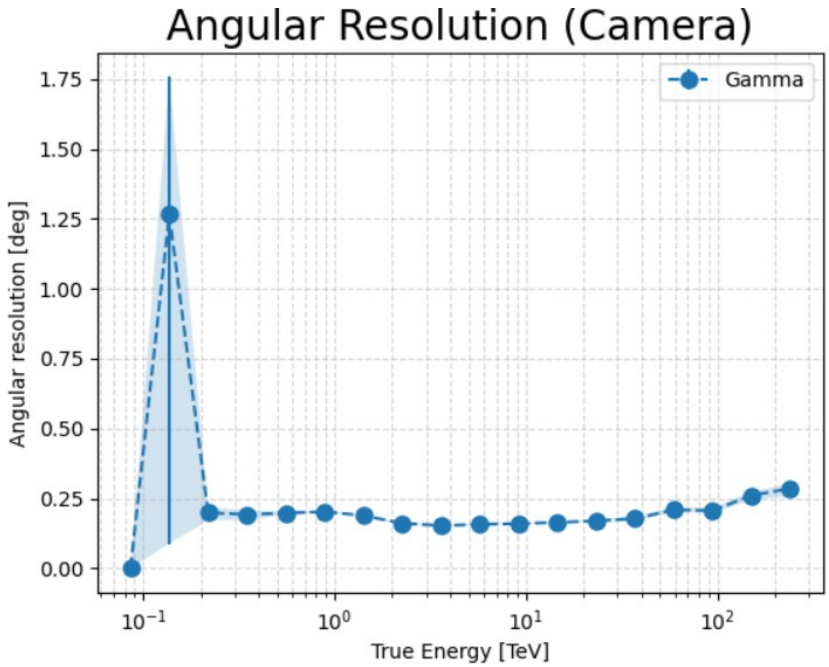

Figure 5.16: Angular resolution graphic computes how precise the model for reconstructing the angular (degree) value, depending on the energy scale (ground truth energy). The x-axis shows the ground-truth energy of events on a logarithmic scale. For visualization purposes, the values are averaged in bins for defined ranges. The y-axis displays the spread (angular resolution) in degrees of the predicted data. For each point on the graph, there is a shading to show the upper and lower bounds of the bin (68% containment interval) for the events contained in the bin.

Angular Bias/Standard Deviation

This graphic works exactly the same way as the one for the energy (5.3.3.2). Refer to it for more details. The graphic displays the standard deviation and bias per bin for both predicted coordinates: azimuth and altitude. It helps identify whether one of the coordinates is causing more problems than the other. The values are displayed in degrees.

The figure 5.17 shows the different curves for this graphic.

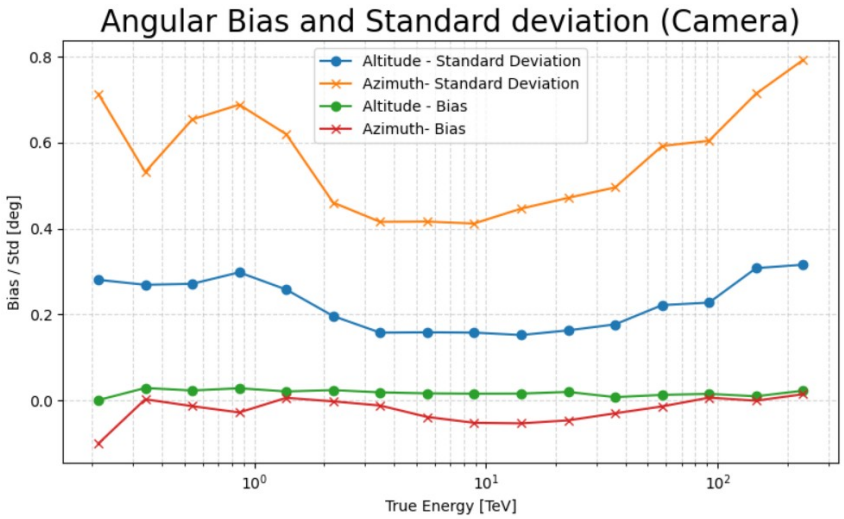

Figure 5.17: Bias and Standard deviation graphic for the two coordinates predictions. Each of them displays the bias and the standard deviation corresponding to it. The difference between the curves is in the point type and the colors used. They are displayed using bins for the ground truth energy of events. The x-axis shows the bins (range) on a log scale, and the y-axis shows the difference in degrees between the standard deviation and the bias.

5.4 Comparison of models

The other tool implemented as part of the thesis is the comparison of models report. It's similar to the report generation as it's also a report and it uses almost all the graphics and metrics detailed in the chapter dedicated to it (5.3). There are differences in the graphics and metric displays, as the comparison of the models provided multiple values for the same graphics and metrics. Some rearrangements are needed to ensure a proper comparison of models. These changes are explained below.

The comparison of the models is an extended control system for report generation. Instead of evaluating the performance of a single model, it's possible to compare multiple models simultaneously in this scenario. It enables decision-making and identifies the best candidates for a given task. It's also enabling objective evaluation, as a report on only one model is subjective and doesn't determine whether the model is better or not. It's also useful because the model can be used in different scenarios/environments, and depending on those, one model may be better than another (e.g., the requirements for a triggering system aren't the same as those for a server usage model). This tool is rather useful for a decision-making system.

The comparison is performed only among models for the same task. It doesn't display graphics for models of different tasks. Report generation is dedicated to a model, and model comparison is the same, but for a specific task and related models.

Some general information is available without comparing the model on the comparison report. It includes the task for which models are compared, the number of training and testing events, and the number of models involved in the comparison. A dedicated chapter isn't necessary as these values explain themselves quite simply.

5.4.1 Metrics modifications

Regarding how to handle metrics, there are many changes from the report implementation. In the model report, there were two sections dedicated to metrics: Model and Runtime metrics, and Performance metrics. The same two sections are present in the comparison of the model with the associated metrics. To compare metrics, two approaches have been implemented: a brute-force comparison of the metrics and a ranking system for the metrics. The combination of the two approaches allows for precision and quick understanding of the metrics and which models are better, depending on the desired use case.

The brute comparison delivers detailed metric values. It helps to compare the level of differences in metric results between models. Figure 5.18 shows a table with metrics comparison for each model. A row is associated with a model as a column to a metric. This display has a quick overview and easier comparison of models' metrics.

Performance Summary

model	MSE	R^2	RMSLE	MAE
cnn_batch64	685.245	0.311	26.177	8.571
resnet_bottleneck	991.712	-0.689	31.491	10.678
resnet	635.286	0.447	25.205	7.705
resnet_batch64	613.277	0.430	24.764	7.508
full_cnn	751.043	0.069	27.405	9.350

Figure 5.18: Example of table for Performance metrics. It contains a first column listing the model names as references. Then the metrics are provided with their values. This table is for a deeper comparison of a model to see whether it outperforms on a specific metric.

The ranking comparison, on the other hand, provides an idea of performance relative to other models across metrics. It uses the same table system as the brute comparison and is very helpful for quickly identifying the model with the best performance when many models are compared. To further accelerate understanding of the table, a green background is applied to the model with the best value for a determined metric. This can be seen through Figure 5.19.

Ranking Summary

model	MSE	R^2	RMSLE	MAE
cnn_batch64	3	3	3	3
resnet_bottleneck	5	5	5	5
resnet	2	1	2	2
resnet_batch64	1	2	1	1
full_cnn	4	4	4	4

Figure 5.19: Example of table for Performance metrics. This one is dedicated to a ranking system. It rather shows rankings compared to other models for each metric, rather than their values. It helps quickly identify which model performs better.

5.4.2 Graphics modifications

As mentioned earlier, the model comparison reuses graphics from the report generation. Some of the graphics, depending on the task, aren't really relevant for comparison and are therefore not used in this tool. Displaying graphics for this tool replaces the report-generation graphics to provide a clean comparison of the models. It combines the graphics from each model into a single graphic for comparison.

There are two different ways to combine these graphics. The first one, shown in Figure 5.20, combines the values from the models into a single graph. This is normally used for line plots or similar plots when values from different models don't overlap and remain legible on the same graph. The scale used for each model output is the same, unlike the second approach. The second one, displayed by Figure 5.21, provides sub-graphics for each model inside the same model. The graphics are directly comparable to those generated in their respective reports. A matrix is a good example of a graphic that requires this approach.

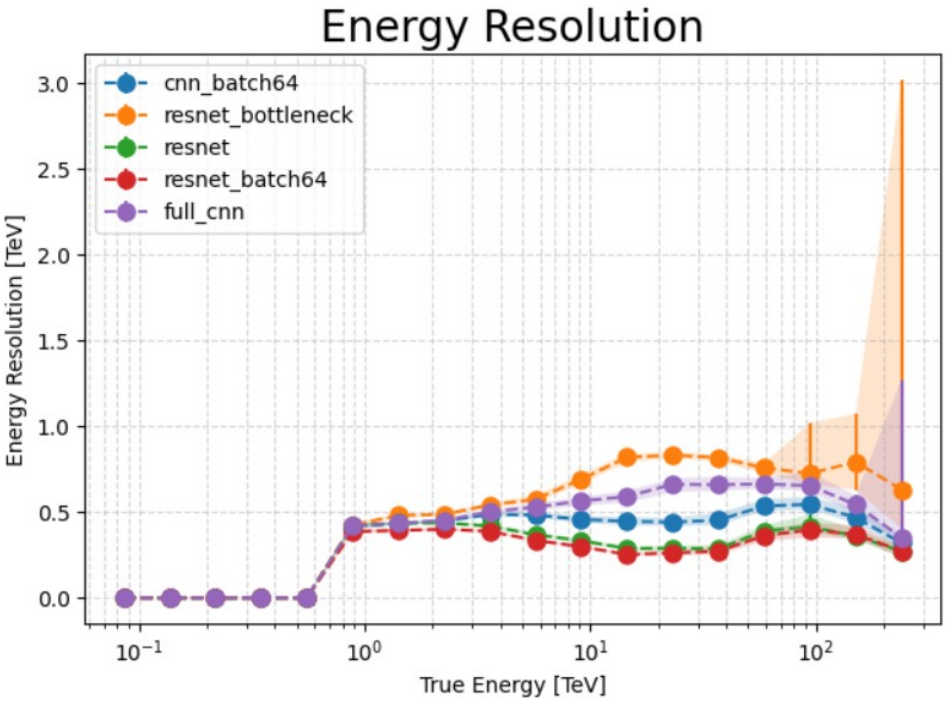

Figure 5.20: Example of a combined graphic in the comparison of models (Energy Resolution). A single graphic is displayed, with lines for each model showing the energy resolution over energy bins. The same scale is used for every line, and comparisons can be easily made as elements are next to each other. This approach is used for simple graphic displays.

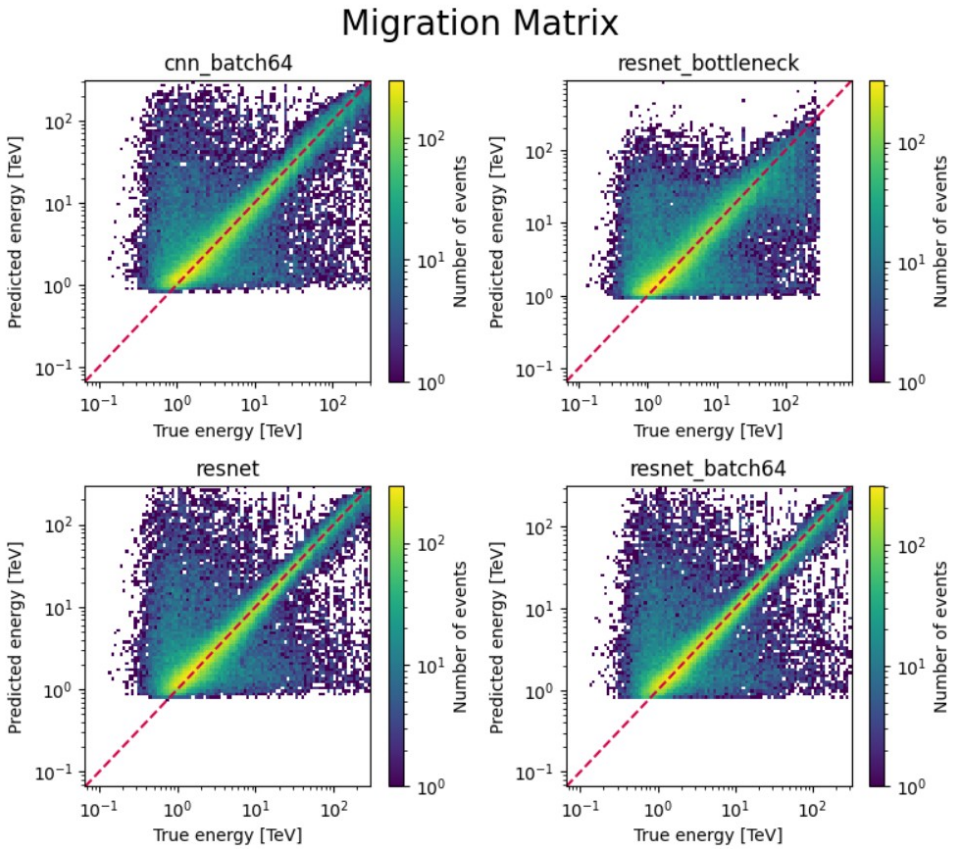

Figure 5.21: Example of a subgraph comparison (Migration Matrix). Graphics with complex features aren't combined; they're placed side by side to make comparisons as easy as possible. The plot is identical to the model report one. The scale might vary depending on the subgraph displayed.

To give an idea of the graphics used for each task, the list below provides details and links to all the graphics in the generated file. This list also specifies whether the graphics are merged, as in figure 5.20, or separated using subplots, as in figure 5.21. An example of each task comparison is also available in the appendix.

- **Particle classification**
  - **Confusion Matrix** (5.3.2.2) - Separated graphs.
  - **ROC Curve** (5.3.2.2) - Combined graph.
  - **Calibration Curve** (5.3.2.2) - Combined graph.
  - **Gammaness Distribution** (5.3.2.2) - Separated graphs
- **Energy regression**
  - **Energy Distribution** (5.3.3.2) - Separated graphs.
  - **Migration Matrix** (5.3.3.2) - Separated graph.
  - **Energy Resolution** (5.3.3.2) - Combined graph.
  - **Energy Bias and Standard Deviation** (5.3.3.2) - Separated graphs

- **Direction regression**
  - **Precision in degrees** (5.3.4.2) - Combined graph.
  - **Angular Distribution** (5.3.4.2) - Separated graph.
  - **Angular Resolution** (5.3.4.2) - Combined graph.
  - **Angular Bias and Standard Deviation** (5.3.4.2) - Separated graphs

6 Models

CTLearn needs to analyze telescope images and find key features to understand events and objects in space. The library thought Deep Learning models were an interesting solution to explore. The library offers multiple model implementations for training and predicting the key features required, depending on the task.

This implies that models are a major part of the CTLearn library and, therefore, of the thesis. Models built here aim to be the best for their respective tasks. Looking at research papers, some reference results can serve as a starting point for models developed during the thesis, with the goal of being beatable.

Models and related subjects, such as optimizing them with FPGA deployment or code for generating models, are examples of what this section covers. The thesis is tied to the CTLearn library, and some modifications are expected to the library. Some sections describe modifications and interactions with the library.

The models used in the thesis are also described with precision, including the parameters that can be modified, the default architecture, and the main concept of each model.

6.1 Reference

All models trained and tested in this thesis have one goal: to beat the reference model reported in research papers, or at least get as close as possible to its performance on every task. This isn't the main goal, obviously, but producing a similar result could be a good indicator that the thesis's approach is effective.

The reference model varies from one paper to another; choosing the model to beat and the performance metrics is important. Usually, the models used as references in papers use less data than the models in this thesis, implying that some interesting results could come out of this. Some parameters also need to be taken into account for the reference, such as the batch size or the number of epochs, to better understand it.

The first reference is the CNN-RNN model from [36], which combines CNN and RNN layers. This model takes multiple images from the same shower as input to determine the event's particle type. This model is conducted only for a **particle classification** task. Experiments were conducted on multiple telescopes, but the focus here is on SST-1M telescope data (same as the thesis data). It covers energy from 20 GeV to more than 300 TeV and has a balanced dataset across particle types. The model obtained a global accuracy of **0.809 %** and an AUC of **0.893 %** on the test set. More details about these metrics are displayed in the figure 6.1.

The model was trained using 40000 batches of 16 events, fewer than the models from the thesis.

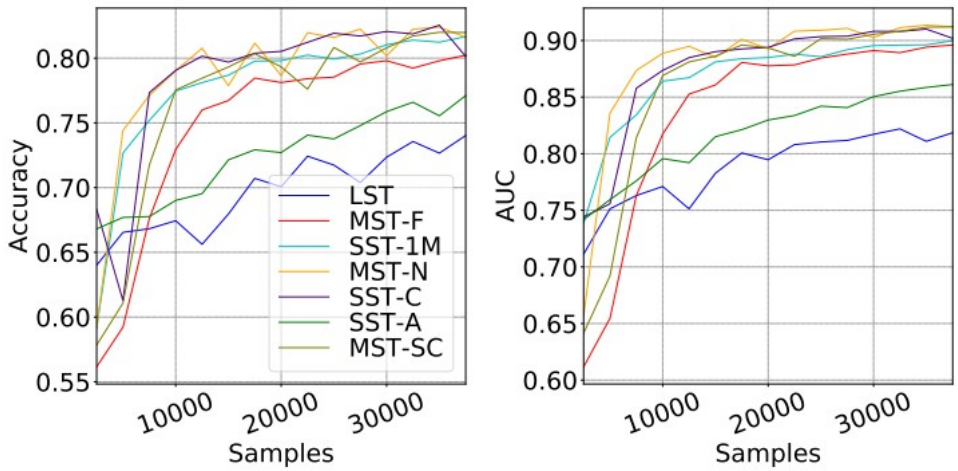

Figure 6.1: Accuracy and AUC metrics over the number of samples for each telescope. The main focus is on the SST-1M, the telescope from which the data used in this thesis are retrieved. The accuracy for this telescope is converging slightly above 0.8, and the AUC is converging to almost 0.9.

6.2 CTLearn Models

CTLearn library contains lots of features. Among them, there are pre-built models ready for the library to train and test. These models are useful because the library adapted them to its requirements and constraints. These pre-built models can be customized by providing a configuration file specifying the desired architecture features, such as the number of layers or the use of attention layers.

An additional model type, the LoadedModel, allows loading custom models while respecting library restrictions such as layer names, the separation between the head and the backbone, and other constraints. A custom model template has been created to avoid incompatible models and is documented in a dedicated chapter.

6.2.1 ResNet

The ResNet model available in the library can be considered the reference model with the best results for every task in the thesis. It's also a complex model with a consequent number of parameters and layers. The computational resources required by this model are higher than those of others.

The ResNet (Residual Network) [46] [45] architecture addresses challenges that very deep neural networks face. Normally, the deeper the model, the harder it is to train; vanishing and exploding gradients are more common, and performance can decrease. The idea is that the model learns residual mappings to train more effectively by using a skip connection, allowing information to flow directly across layers. This means that the residual mapping is combined with the convolutional layers' outputs to achieve better results while preserving the initial information. It allows better results and fewer issues when working with deeper models. The whole concept is visualized through figure 6.2.

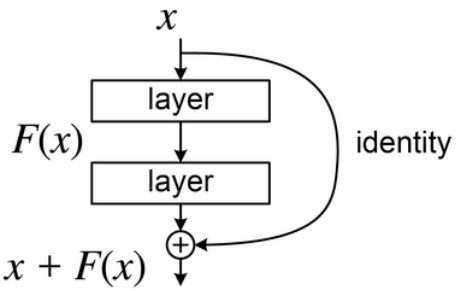

Figure 6.2: Main concept of the residual block. The idea is to propagate the input with an identity mapping to skip some layers. The input is then combined with the convolution layer outputs via an element-wise Add operation. It helps prevent vanishing/exploding gradients and limits performance degradation in very deep networks. This type of block is generally combined with other blocks to form a full network. [46]

The default configuration of the ResNet model is a complex architecture with multiple blocks.

The residual blocks defined in the CTLearn library use two formats: a bottleneck format and a basic format. A parameter is defined to switch between them. The difference between them lies in the number of filters and layers used, but the fundamental concept is the same for both. The residual block makes models non-sequential and complex. The figure 6.3 displays the bottleneck residual block and gives details about it.

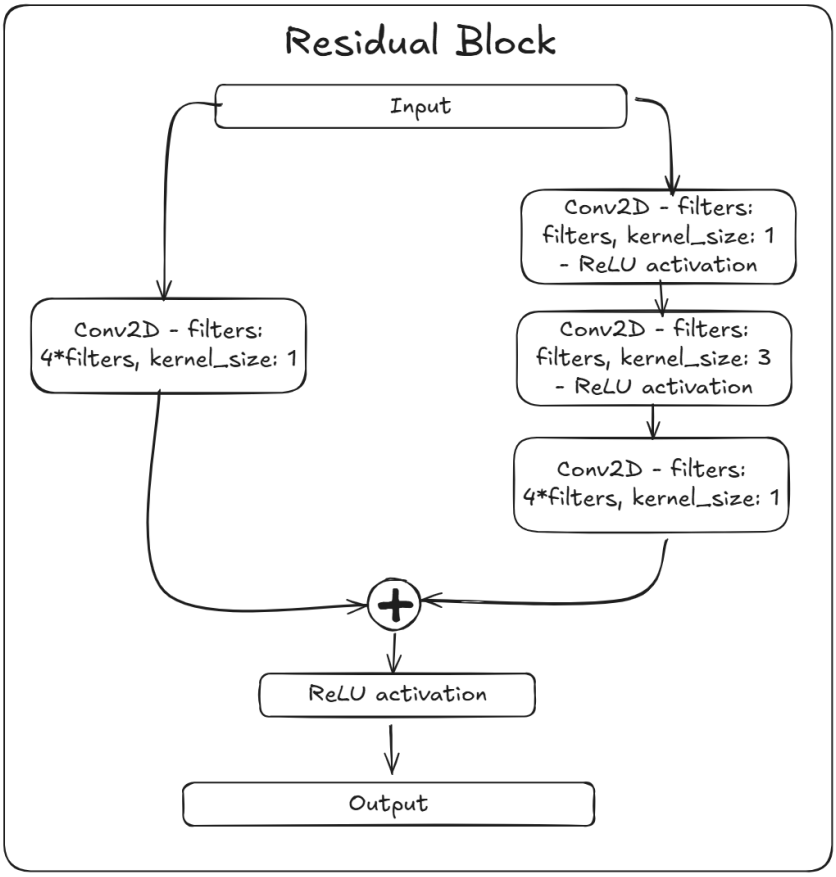

Figure 6.3: Default Residual block in CTLearn library. The residual block helps maintain computational efficiency and stability in deeper networks. This is the default residual block used to build a ResNet model with the CTLearn library. It's a configurable block that leverages residual features by passing the input directly through an identity mapping/projection shortcut. The shortcut is represented by this identity map, which is combined with the normal convolutional pathway via addition. A ReLU activation is then applied to stabilize the network. The activation function's output is then passed to the next block/layer, completing the residual block step. Some elements of the residual block are configurable. First, there is the number of filters that can be changed per block. An attention layer can also be added at the end of the convolutional pathway and before the add operation.

The central element of the ResNet network from CTLearn has been described. To have a full, deep network, the model stacks multiple residual blocks. The figure 6.4 represents the architecture of the default model of the CTLearn library. It includes layers that represent stacked blocks with a defined number of filters. The residual blocks are only in the backbone, as the head still has a Dense layer to converge to the output. The number of stacked blocks and filters can be changed in the configuration file. This is represented by the same parameter as the CNN model, **architecture**.

```
CTLearnModel:
    architecture: [
        {"filters": 48, "blocks": 2},
        {"filters": 96, "blocks": 3},
        {"filters": 128, "blocks": 3},
        {"filters": 256, "blocks": 3},
```

```
]
```

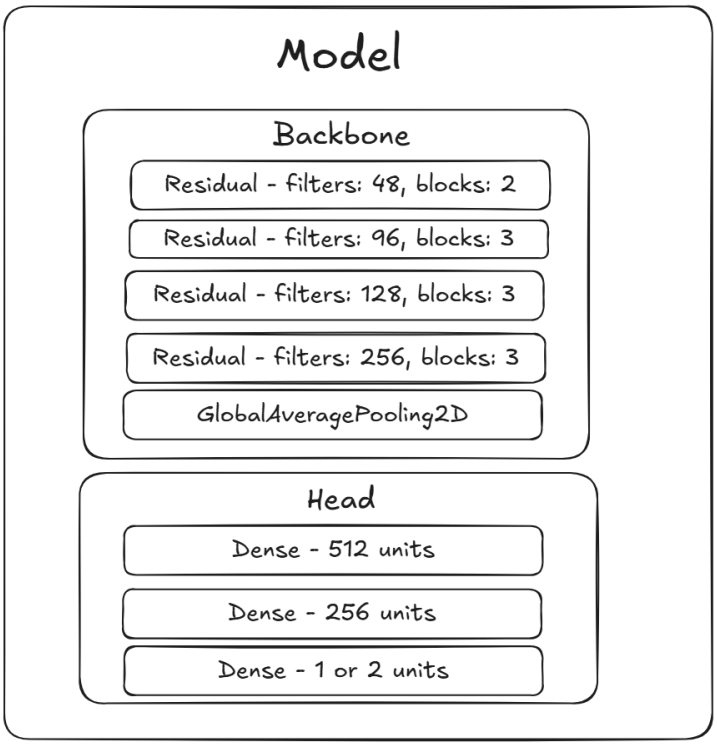

Figure 6.4: Default ResNet Architecture on CTLearn library. The architecture combined residual blocks from the figure 6.3. The idea is to build a very deep model by chaining a series of residual blocks. It allows training a very precise model with fewer parameters and less computation time than a model without residual connections would require. The head part is a classic combination of Dense layers that can be defined separately from the blocks.

With this, understanding the ResNet model and its behavior should be clear. Having a high-complexity model is useful for checking whether the entire pipeline and metric calculations work correctly. It avoids having to face these issues later when using complex customs models. This model has served as the reference when building tools or testing metrics within the CTLearn library. In some scenarios, the model's complexity revealed errors that weren't apparent with simpler models.

6.2.2 CNN

The CNN model [23] [26] available in the library can be considered the simplest possible model. It contains a few layers, really simple, just to build a model and test the whole pipeline of training and testing a model.

A CNN (Convolutional Neural Network) model is a type of Deep Learning architecture used for computer vision and image processing. The model's layers are 2-dimensional grids to match matrix-like data (e.g., spatial or temporal data). This allows for the extraction of complex features and patterns. This architecture generally relies on multiple layers to work. Convolutional layers are the main feature of CNNs, with their 2D filters able to extract patterns. MaxPooling and other layers, along with classic Dense layers, can be used for this type of architecture.

Figure 6.5 displays an example of architecture for a CNN.

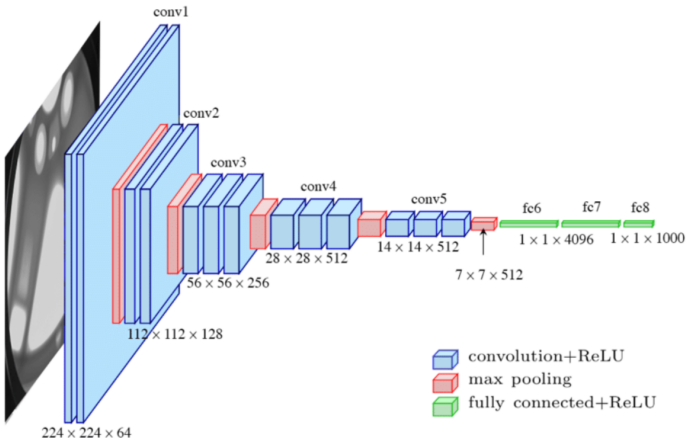

Figure 6.5: Example of VGG16-CNN Network with multiple convolutional layers and max pooling layers. Along the network, the matrix size is reduced to a single output at the end, yielding the prediction. This type of architecture extracts a lot of information from an image, from high-level features/patterns to detailed low-level features. [26]

Regarding the implementation of the CTLearn library, the CNN model's backbone (and head) can be modified using the configuration file to adapt to the use case. The only thing is that the layer type is still the same. It's always Conv2D layers for the backbone and Dense layers for the head.

To change the default backbone layers, the **architecture** parameter of the CTLearnModel instance must be set as shown in the example below. Some other parameters, such as the pooling type or normalisation, can be changed. Refer to the CTLearn library for more details.

```
CTLearnModel:
    architecture: [
        {"filters": 32, "kernel_size": 3, "number": 1},
        {"filters": 32, "kernel_size": 3, "number": 1},
        {"filters": 64, "kernel_size": 3, "number": 1},
        {"filters": 128, "kernel_size": 3, "number": 1},
    ]
```

As said earlier, the head can also be modified. This time, the parameter to use is **head_layers** (be sure to respect the output dimensions). These parameters define the number and size of head Dense layers. The task needs to be precise.

```
CTLearnModel:
    head_layers: {"type": [512, 256, 2]}
```

Figure 6.6 displays the default model used during the thesis with details about the implementation. The figure doesn't include all the details about activation functions or MaxPooling layers, but the figure's caption clearly explains them.

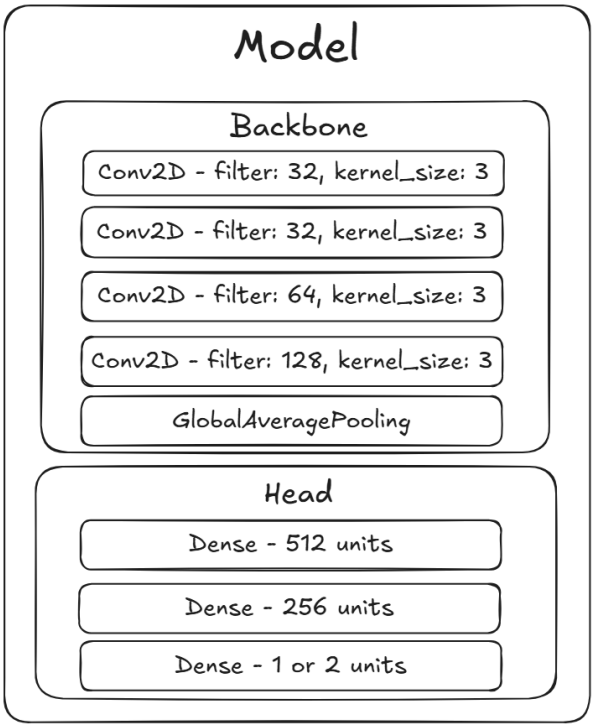

Figure 6.6: Default CNN implementation for CTLearn library. The model is split into two parts: the backbone and the head. The backbone consists of four convolutional layers, with the important parameters listed. Each of these layers is followed by a MaxPool layer to reduce the matrix's dimensions. The activation function for each layer is the ReLU activation function [44], and the padding is defined so that the Conv2D layer doesn't reduce the dimension of the matrix (only MaxPool). The backbone concludes with a GlobalAveragePooling layer, averaging the spatial dimensions into a single dimension (height, width, and channel into one). This is similar to what flatten would do. The head has three layers that take data in one dimension, with a decreasing number of units across the layers, converging to one or two outputs, depending on the task (e.g., one for energy regression and two for particle classification). For the classification, a SoftMax layer is added to finalize the model. For the head, the same activation function as the backbone is used.

6.2.3 Custom model

The custom model is the third option for using a model with the CTLearn library. In contrast to the other two types of models, this isn't a prebuilt model but rather a way to provide a custom model to train. This approach has many benefits, such as the flexibility to choose a model type and the option to fine-tune using pretrained models.

For uniformity, when performing the training task, a model is always provided via the LoadedModel implementation. The model, depending on whether it's a custom one, is built in another task to prepare it once and for all. Refer to this chapter for more details (6.3.1).

Template

The template to build a custom model is rather simple. The model is built so that only the layers need to be modified to make it work. The template is attached in the appendix (11).

There is a global class that contains the entire model and stores the initial parameters, such as

the input shape or the task type. With the attributes initialized in the class, the model is built by separating into two distinct sections: the backbone and the head. These elements are built separately in distinct classes. The network structure of the template is shown in Figure 6.7.

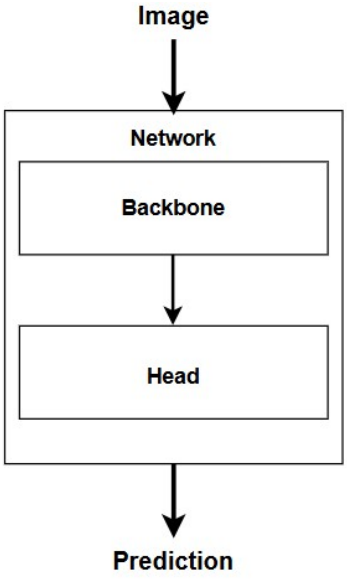

Figure 6.7: Network structure of the template for custom model. The network is divided into two parts: the head and the backbone. The Backbone handles high-level features and starts processing the input, while the Head handles low-level features and computes the model's predictions. Each of these elements has a distinct class to define the model layers.

The choice of this kind of structure for the network is due to parameters in the CTLearn library that allow removing the model's head and using the default head provided by CTLearn on the data. The idea is then to keep this separation of head and backbone between layers as flexible as possible to work with the package. It's also a useful way to separate fine-tuning/transfer-learning components from unmodified weights. The reusability of this is also very high, with the possibility of attaching different head parts to the backbone to observe potential differences in the results.

```
class ComplexModel(Model):
    def init(self, input_shape=(96, 96, 2), reco_task=""type, num_classes=10,
                                                dropout_rate=0.2, name=
                                                ""complex_test_block):
        super().init(name=name)

        # Initialize model parameters
        self.input_dim = input_shape
        self.task = reco_task
        self.num_classes = num_classes
        self.model_name = name
        self.dropout_rate=dropout_rate

        # Build backbone and head
        self.backbone = self.build_backbone()
        self.head = self.build_head(name=self.task)

        # Prepare and build global model for graphic usage
        inp = Input(shape=self.input_dim, name="input")
```

```
        x = self.backbone(inp)
        out = self.head(x)
        self._graph_model = Model(inp, out, name=f"{name}_functional")

    # Build by chaining different layers
    def call(self, inputs, training=False):
        x = self.backbone(inputs, training=training)
        out = self.head(x, training=training)
        return out
```

As a reminder, the backbone is the model's core network, containing the high-level features. It takes most of the resources and can be heavy. Most of the parameters are included in this part, which is one of the reasons why, when fine-tuning, the backbone is usually frozen. It's the starting point of the network where the data is provided.

```
# Backbone model (always used) and can be only element called if overwrite of head
                                                    is wanted
# Parameters can be changed or added (name MUST end with ""_block)
class ModelBackbone(Model):
    def init(self, dropout_rate=0.2, name=""backbone_block):
        super().init(name=name)

        # Define layers
        ...

        # Need to flatten before head
    def call(self, inputs, training=False):

        # Build the model by chaining different layers
        return ...
```

The head, on the contrary, can be seen as the end of the network, where predictions are computed using lighter, simpler layers. This is generally the part of the model that is fine-tuned.

Both of them are constructed by the network by wrapping their respective parts in a functional model (or nested model) so that they can be treated as a single block.

```
# Build backbone as functional layer (combine backbone layers as one for tensorflow
                                                    recognition)
def build_backbone(self, name=""backbone_block):
    inp = Input(shape=self.input_dim)
    backbone = ModelBackbone(dropout_rate=self.dropout_rate, name=name)
    out = backbone(inp)
    return Model(inputs=inp, outputs=out, name=name)
```

This approach provides a clear abstraction and enhances the reusability of global and nested models. By proceeding this way, it is necessary to override some functions from the Keras Model class for compatibility with the architecture in place.

To comply with the CTLearn library, some restrictions must be respected. For example, there can be only one input passed to the model for it to work. There are also naming constraints. CTLearn library searches for specific layers by looking at their names, such as the backbone, which must end with "_block" to work. Same thing for the head, where a correct task needs to be provided.

6.3 Tasks

As discussed in the chapter dedicated to the MLOps approach (5), the pipeline to produce a model is decomposed into multiple tasks. Some of them are completely related to the MLOps approach; however, the generic tasks related to generating the model are closely tied to the modeling part. These tasks directly interact with the CTLearn library and depend on it. They directly influence the generation of a model and its related results. Short explanations about these tasks are therefore required. To visualize all the tasks related to the model generation for this thesis, refer to figure 5.2.

6.3.1 Prepare model

Model preparation is an essential task to ensure uniform training. It's included in the model's training. The goal is to build a model that meets the requirements of the CTLearn library. This is the first step in creating the model. The model can be loaded with a default configuration from CTLearn (ResNet or CNN) or load a custom model created using the template provided earlier (6.2.3). For now, pretrained models cannot be passed in the preparation step, but with small changes, it could be possible.

The preparation is split into two cases: using a prebuilt CTLearn model or using Custom models. For custom models, the file and the class name of the model class need to be provided as additional elements. If prebuilt CTLearn models are used, a CTLearnModel instance is created that contains the model. Both scenarios save the prepared model to a temp folder for the training task to retrieve.

Some essential parameters need to be provided when preparing the models. Parameters displayed below are mandatory for this step. Additional elements, like the attention mechanism for ResNet models, aren't shown below. These additional elements are used to modify and adapt prebuilt models to certain situations.

- **Image shape** : Shape of images provided to the model and used as input layer
- **Type of model** : LoadedModel for custom models or ResNet/CNN for prebuilt models
- **Task** : Task for which the model is built. Depending on the task, the head layers and compiler can change.
- **Number of classes** : Related to the task. Corresponds to the number of classes as output (1 when not classification)
- **Temporary directory** : Place to store the prepared model.

With this, the model is ready for training with the CTLearn library. The model can also be used without the library, but some modifications to the data processing are required to make it usable by the model.

6.3.2 Training

The core task of model generation is model training. This task follows the model preparation. This previous step can be skipped if a model is already prepared and respects the constraints of the CTLearn library. This step was taken to minimize the number of changes when training

different models. The differences are managed by the preparation step and the configuration file. It enhances reproducibility and automation of model generation.

The training step retrieves the previously stored temporary model and provides it along with the configuration file to an internal CTLearn process, **TrainCTLearnModel**, which creates the training process with the configuration and corresponding model. To train the model, the only step is to run this process.

The training, just like other tasks, requires parameters to work correctly. These parameters are editable in the experiment's configuration file. Refer to the corresponding chapter (5.2) for more details. All the parameters are passed through a Config instance from the traitlets library [52]. The training tool from CTLearn offers many different configurations, so the focus has been on some specifications, while others are kept as default. Some parameters in the configuration file will not be described as they should not be touched.

- **Input data (signal)** : Data provided to the model containing gamma-ray events (Comes with a pattern parameter to filter the files).
- **Input data (background)** : Data provided to the model containing hadron events (Comes with a pattern parameter to filter the files).
- **Task** : Task for which the model is built. Depending on the task, the CTLearn library behaves differently.
- **Number of epochs** : Number of epochs to train the model
- **Batch Size** : Size of batches for training.
- **Output directory** : Directory where the model will be stored.
- **Percentage per epoch** : Percentage of the data used for each epoch (if not 100%, selected randomly)
- **Early Stopping** : Configuration to stop training earlier if changes from one epoch to another aren't enough.
- **Pruning of the model** : Parameters to prune the model
- **Mono/Stereo mode** : Whether to handle one telescope image or several telescope images at once.

For the training task (also testing task), modifications to the library are required as the process of pull requests and new package releases is not compatible with the limited time available for the thesis. Therefore, modifications have been made to specific files, and package redirection has been implemented. These modified files are stored in a dedicated tools space.

Regarding the training task, the train_model.py file has been modified to comply with the thesis's model usage. Among the modifications, two of the listed parameters were added. The percentage per epoch and the pruning option have been added. The pruning option is documented in a dedicated chapter for model optimization (6.4.1).

The percentage per epoch, on the other hand, is a feature added to address the issue of time required to generate a model (7.2). The idea is that, after separating the data into batches, a random portion of the batches, according to a given percentage, is selected for the current epoch training. The next epoch will take a different random portion of batches. This allows for

reducing the amount of data read during an epoch. This approach comes with another feature: model generalization. That can be either an advantage or a disadvantage, depending on the percentage defined. By training on different data in each epoch, the model won't overtrain, since it doesn't see the same data every time. However, if the percentage is too low, the model may not see some data during training. This approach has a downside. Batches selected in the earliest epochs will define the model's shape and carry greater importance than later epochs' data, as the learning rate decreases over time. This concept is visualized using figure 6.8.

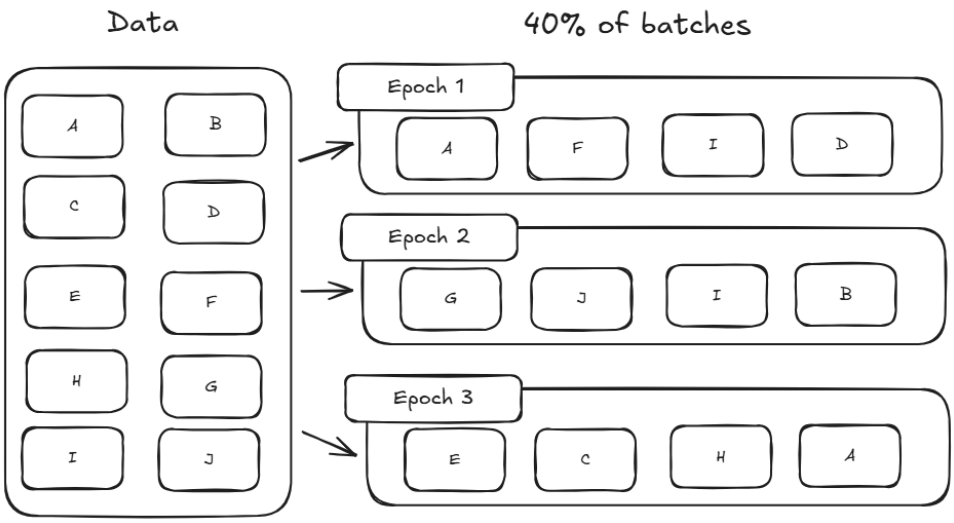

Figure 6.8: Principle of percentage of data used per epochs. The example displays the data separated into 10 batches. The model is then given the provided percentage and randomly selects 40% of the batches for a specific epoch. For each epoch, the batches selected are randomly chosen again. The randomness related to one epoch isn't influenced by the results of other epochs. It means it can have multiple instances of the same batch across epochs.

Another modification, not directly related to the training, has been made in the dl1_data_handler library to the DLDataReader class to parallelize file access. Explanation can be found in the corresponding chapter (7.2).

Here is an overview of the training steps with the CTLearn tool. The first step is to use a data reader to load and process the data. Then, the data is split into two sets: training and validation. After that, the callbacks are set up. Callbacks [50] are event listeners that run a task after each epoch, such as updating the learning rate or verifying whether early stopping requirements are met. This way, the tool's initialization is complete. The next step is to load the model and define the basic elements of the training, such as the learning rate and optimizers. After a few minor setup steps, such as pruning the model, the model is compiled, and training starts. Figure 6.9 shows the tasks the tool displays, along with the steps within the tool.

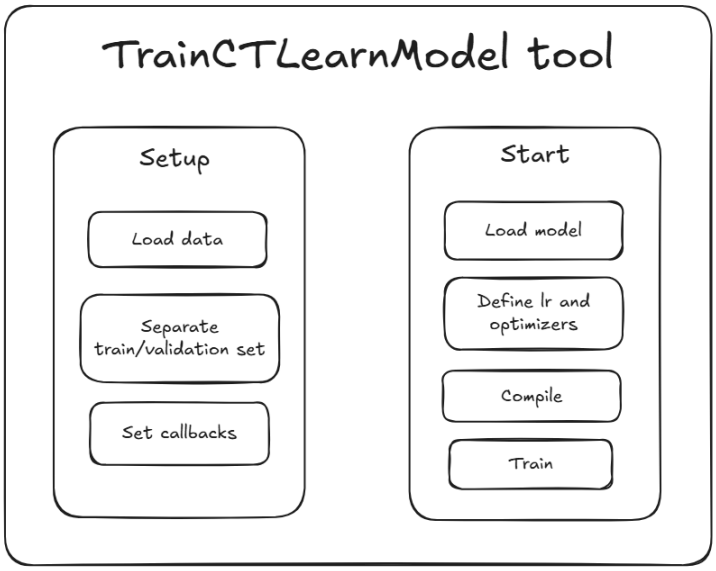

Figure 6.9: Repartition of steps within the TrainCTLearnModel tool. This tool has multiple features, including processing the data into an adapted format. It also prepares the environment to train the model with callbacks. The model is then loaded inside the tool, compiled, and trained.

This concludes the internal training process for the CTLearn library. There is still one element to discuss in this training task: the metrics. For both training and testing, some metrics are saved in a JSON file for additional information in the report and for model comparison.

The metrics stored are the following:

- **Training time** : Time in milliseconds to train a full model.
- **Training events** : Number of events used to train the model.
- **Number of parameters** : Number of parameters in the model.
- **Number of layers** : Number of layers in the model.
- **Estimated flops** : Number of FLOPs (Floating point operations per second) for one inference.

For each of these metrics, a complete description and its relevance are presented in a dedicated chapter (5.3.1.2). The idea here is to describe how to obtain this information.

The number of events and the time required to train the model are relatively easy to obtain. To calculate the time needed to train the model, a timer is started just before the run function of the TrainCTLearnModel instance and stopped just after the training ends. For the number of events, CTLearn provides an internal function that retrieves it.

```
self.training_events = model.dl1dh_reader._get_n_events()
```

Regarding the number of model parameters and layers, it is easy to retrieve. Keras provides an internal function for that. Retrieving the number of layers is more complicated and requires defining an entire function for it. Normally, to display the number of layers, there are some internal parameters that can be used. The issue with this is that it doesn't account for sub-layers from Functional layers or Nested models. To fix this, a recursive function is used to get

all layers by traversing the entire model architecture (including nested models) to obtain the actual number of layers. The figure 6.10 represents the function.

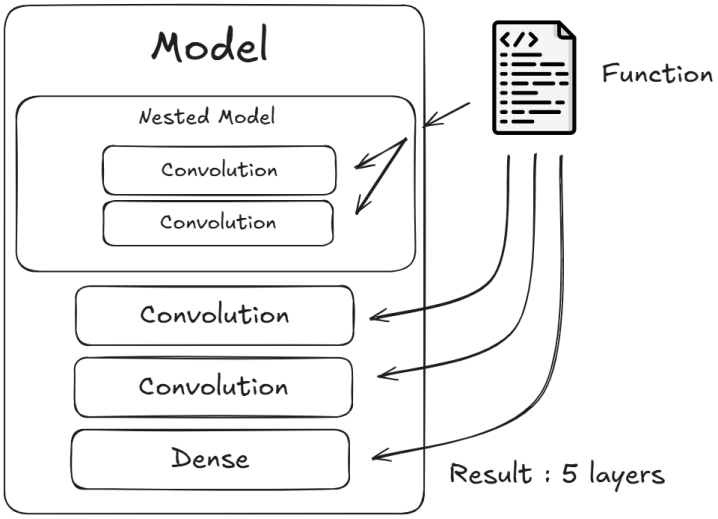

Figure 6.10: Representation of how the model retrieves the number of layers recursively. It goes inside the nested model/Functional layers to retrieve the layers. It's important to note that some layers aren't taken into account, such as the input layer or the nested model itself. It has no interest in counting an artificial layer.

For the number of FLOPs, it is even more complicated. Multiple solutions have been tried, but because of the complex architecture of models, most of the time it didn't work. The solution for this scenario was to use TensorFlow to convert the model to a graph function and use the TensorFlow Profiler [38] to extract the number of FLOPs. It's important to know that this method only estimates the number of FLOPs per inference. The steps can be visualized with the figure 6.11.

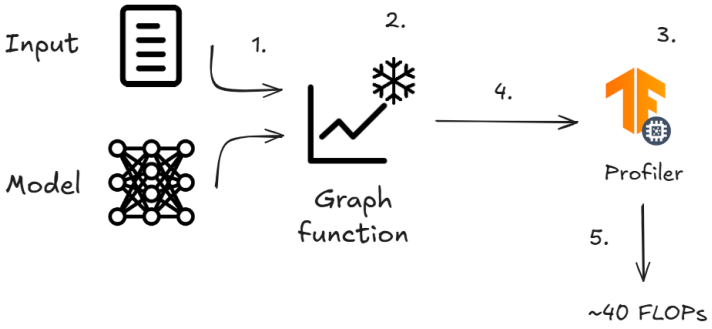

Figure 6.11: Process to obtain FLOPs from a model. 1. A dummy input is created and used together with the model to convert it into a graph function. 2. The graph is frozen, so it can't be modified with manipulation. 3. TensorFlow Profiler is set up. 4. The graph is passed to the Profiler to retrieve performance metrics. 5. The estimated number of FLOPs is extracted

6.3.3 Testing

The testing task is the logical next step after training a model. Once a model is available, it can be tested on the dedicated data. Normally, if the configuration file is correctly set up, there

should only be a few things to change. This is again to enhance reproducibility and automate the entire model generation process. It relies on two similar tools from the CTLearn library, just as with the training step: MonoPredictCTLearnModel and StereoPredictCTLearnModel. These two tools inherit from the same class, and the difference between them lies in the mode they handle.

Some parameters need to be set in order for this task to execute correctly. There are only a few of them to set.

- **Data path** : Path to the data folder (global data folder, not a specific one).
- **Mono/Stereo mode** : Whether to handle one telescope image or several telescope images at once.

Depending on the mode and the task defined in the configuration file, one of the tools is chosen and executed. This step will produce HDF5 files, just like the data provided, with an additional section dedicated to the model's predictions (DL2). They are stored in a dedicated folder within the model's directory.

Regarding the testing task, the predict_model.py file has been modified to comply with the thesis's model usage. It includes the two prediction tools. The applied modification is a fix for the particle classification model. The fix is described there (7.3).

For the training task, some metrics are calculated and stored in a JSON file for future use when generating a report or comparing models. The number of testing events and the total testing time (all inference time) can be easily retrieved by defining a timer and using the internal function from CTLearn to count events.

6.4 Model Optimization

Model optimization involves configuring models for specific tasks. It includes optimizing training by tuning hyperparameters or choosing the right optimizer/loss function, deploying models using compression, and related topics.

In this thesis, model optimization focused on model compression and computational cost efficiency. While CTLearn-generated models operate without resource issues on the cluster (3.6.1), this may differ in the production environment.

The production environment uses a filtering system using triggers to reduce the massive amount of data captured by the telescope (3.2). These triggers use an algorithm and small models to perform filtering on an FPGA. With this in mind, it has been considered using the models generated with the CTLearn library on the FPGA device as a new triggering system, aiming to replace the current filtering system. The only issue is that the models are currently too large and resource-intensive to be printed on the FPGA.

Model optimization becomes essential when adapting models for deployment on constrained hardware. Through compression and resource optimization, it may become feasible to run CTLearn-generated models on low-end devices such as FPGAs.

To optimize models for FPGA usage, it is important to align techniques with CTLearn's structure. Only methods compatible with this framework, and which do not require major modifications, such as distillation, have been tested, with pruning given particular attention.

6.4.1 Pruning

Pruning [40] [29] is an optimization technique used to optimize a model's size. The core idea is to remove unimportant parameters from a neural network to reduce the model's size and computational complexity. The parameters are represented as the node weights and biases. The removal of these parameters can take various forms, such as random weight removal or removing a percentage of less important parameters. The figure 6.12 displays an example of what pruning looks like.

This model optimization strategy works very well with Deep Learning networks, as they are generally over-parameterized with many layers and complex structures. The idea behind it is to shrink models by removing weights, neurons, and, more so, layers to reduce the model's size and the computational cost during inference. The removal is performed by setting the weights to 0, thereby removing their value and cost from the calculation.

The downside of pruning is a loss of accuracy, as removing parameters reduces it. Depending on the pruning configuration, accuracy can be reduced to the point that the model becomes unusable. Pruning must be tested and balanced between desired compression/inference and acceptable accuracy. Interestingly, pruning can generalize certain model types (e.g., well-trained models), making them better for production.

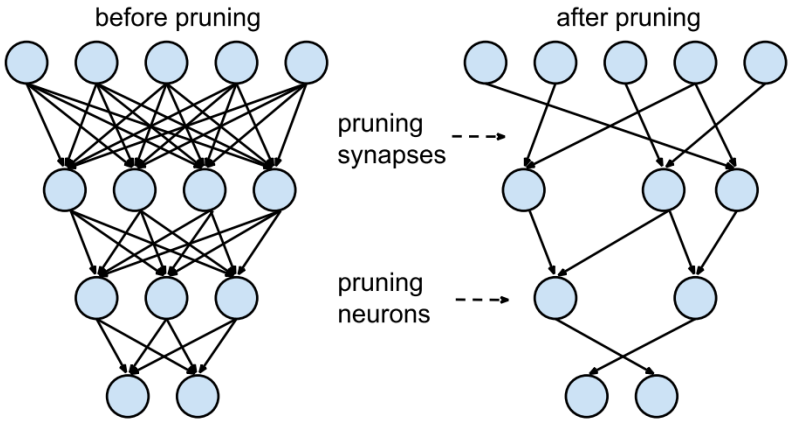

Figure 6.12: Pruning process where the model is displayed before and after the pruning optimization. Two levels of pruning are displayed here: the weights pruning represented by the synapses, where the connections are removed, and the neurons pruning when an entire matrix of weights is removed. [14]

There are two approaches [1] to pruning: train-time pruning and post-training pruning. As the names suggest, the first is conducted simultaneously with the training process, and the second is conducted after the model is fully trained. The train-time pruning makes pruning decisions simultaneously with weight updates, ensuring better network sparsity and identifying which weights are less important. It's done through the iterations. Post-training pruning is performed on the fully trained model. It's simpler to implement, but it could affect accuracy much more, since the pruning isn't optimal. On the contrary, the train-time pruning produces more efficient models, and the pruning process is optimized. However, the training step is a lot more complex to execute.

For the post-training model, there are multiple pruning subtypes. Unstructured pruning is a naive approach that uses a threshold to determine which parameters are removed. That means it helps denoise/generalize a model, but it doesn't improve inference time. The structured pruning, on the other hand, relies on removing the structure of weights, such as neurons or entire layers. This approach improves model inference significantly, but can drastically reduce accuracy depending on the configuration.

Pruning is a powerful optimization technique for models, especially beneficial in FPGA use cases, where it streamlines resource usage.

Implementation in CTLearn

To operate in accordance with CTLearn, modifications have been made to some library files. The idea was to allow users to configure pruning and add it to the model if they wanted to. The pruning configuration can be provided in the configuration file, where the parameters are specified.

The pruning method used is train-time pruning with the integrated TensorFlow tool **tfmot**. It's a TensorFlow module [35] that provides model optimization solutions for Keras models. This tool provides a configurable solution to implement pruning.

It uses the prune_low_magnitude function with a PolynomialDecay pruning schedule, which prunes the model more aggressively in the later stages to preserve early learning and avoid accuracy drops during training.

Three parameters can be modified in the configuration file.

- **Initial sparsity** : Indicates the sparsity (percentage of weights at 0) of the model when pruning begins. Generally, you want it to be at 0 for the first training.
- **Final sparsity** : Indicates the sparsity wanted at the end of the pruning
- **Begin step** : When to start pruning in the current epoch (after which number of batches). Generally, you don't want it to be zero to avoid unstable training

```
training_model:
    TrainCTLearnModel:
        pruning_model: # Pruning settings to use
            initial_sparsity: 0.00
            final_sparsity: 0.90
            begin_step: 1000
```

Some changes need to be applied to the TrainCTLearnModel tool from the CTLearn library to add this pruning feature. First, a callback is defined to count the current number of steps (batches) in the training. It's useful to know when to start and stop pruning.

```
class TrainCTLearnModel(Tool):
    def setup(self):
        ...
        # Pruning callback
        if self.pruning_model is not None:
            pruning_callback = tfmot.sparsity.keras.UpdatePruningStep()
            self.callbacks.append(pruning_callback)
```

Then, the pruning schedule is created and applied to the CTLearn model before training it with the prune_low_magnitude function.

The pruning isn't applied directly; instead, each layer of the model is wrapped, adding a mask to each weight. This way, the weights aren't directly affected, and the global pruning mask can be updated over the course of the epochs. It also means that, afterwards, you need to strip the pruning process, or the model will keep the mask and therefore end up being bigger than the original.

```
...
pruning_schedule = tfmot.sparsity.keras.PolynomialDecay(
initial_sparsity=self.pruning_model[""initial_sparsity],
final_sparsity=self.pruning_model[""final_sparsity],
begin_step=self.pruning_model[""begin_step],
end_step=end_step
)

# Apply pruning to the model.
self.model = prune_low_magnitude(self.model, pruning_schedule=pruning_schedule)
...
```

```
# Restore model state if pruning.
if ""pruning_model in config_training.TrainCTLearnModel:
    ...

    # Load pruned model

    with tfmot.sparsity.keras.prune_scope():
        pruned_model = tf.keras.models.load_model(pruned_model_path)
    # Strip model of pruning weights
    strip_model = tfmot.sparsity.keras.strip_pruning(pruned_model)
    # Save stripped model
    strip_model.save(pruned_model_path)
    ...
```

6.4.2 Quantization

The quantization [57] is another optimization technique. It's rather simple to understand. The idea is to reduce the model's memory requirements by representing the weights with lower precision (e.g., from 8-bit to 4-bit). It also improves the computational efficiency, resource consumption, and inference of the generated model. This alteration affects the model size because weights, neurons, or layers are encoded in smaller sizes, thereby reducing the global model size. Less precision comes with a downside: reduced accuracy. Another interesting aspect is that, by quantizing, the model could become able to run on more machines that couldn't handle high precision, for example.

The quantization process can be represented by the figure 6.13.

Figure 6.13: Quantization process where the left is encoded in FP32 (Floating Point 32), with 32 bits for each value, and converted to INT8, encoded in 8 bits. It means the value is going from 4 billion possibilities to only 256, considerably improving matrix multiplication. Two things can be observed here: the reduction in the number of bits and the conversion of the values to integers. The quantization parameters aren't provided here, but depending on the settings, the mapping function for integer values could change, yielding a totally different output. [25]

The quantization can be applied to multiple parts of the model. It could be to weights, activation functions, or biases. It means the quantization process needs to be correctly defined, and, to operate correctly, the same process needs to be applied to each weight of a layer, model, or neuron to preserve the model's logic and information.

Like the pruning, there are different ways [29] to incorporate quantization into a model. There are the Post-Training Quantization (PTQ) and the Quantization-Aware Training (QAT).

PTQ is conducted after the training and is relatively simple to implement. Quantization is applied quickly across the entire model. It also means the quantization might not be optimal, resulting in considerable accuracy drops. Generally, this approach uses calibration data (a subset of the dataset) to reduce this accuracy issue and the potential approximation errors that can impact the model. The calibration data will help determine the model's quantization parameters and mitigate approximation errors by evaluating the model's performance on it. Depending on the results, the parameters can be updated to preserve as much of the accuracy as possible.

QAT, on the other hand, is performed in parallel during training. It's done, so the model can adapt itself to work in lower precision levels. It helps the model compensate for the approximation errors it may introduce. It can even be used to fine-tune the quantization process later. Because it's conducted during the training, the training process is slightly longer, but it produces better results than a PTQ approach.

The implementation using the CTLearn library wasn't pursued due to issues with quantizing complex models. A dedicated chapter explains this challenge (7.4). The main idea was to first conduct the PTQ approach and then explore the QAT approach once the first quantization process stabilized.

6.5 Towards New Models

The models presented earlier are classic models/configurations available in the CTLearn library. The custom model implementation (6.2.3) with the provided template allows integrating new models with interesting features. New models could introduce interesting aspects of Deep Learning or introduce completely new architectures that might improve performance in classification and regression. It can also be seen as the first step toward exploring new possibilities through thesis work.

7 Challenges

During the thesis, various challenges needed to be resolved to ensure smooth progress. The aim was to find solutions or identify possible approaches. Some issues couldn't be resolved due to limitations, technicalities, or time constraints.

7.1 Link GPU to Tensorflow

A significant technical challenge during the thesis was incorporating GPU usage on the Baobab cluster (see Section refsubsec:baobab). Training neural networks on large datasets requires GPU acceleration, but proper TensorFlow configuration and compatibility were essential.

The first issue is that GPUs aren't automatically visible to the bash script. It's SLURM that handles this aspect, and you must provide detailed GPU configuration in the script to make it work. The second issue related to the first is that the versions of TensorFlow, CUDA, and cuDNN must match for it to work.

This often caused the task to run on the CPU because GPUs weren't detected. Multiple solutions were attempted, including assigning specific CUDA modules or directly referencing a PATH in the Python script.

Only after combining several solutions could the Python script detect the GPUs.

The first element introduced is the flag –**nv** when running the singularity. It exposes the host GPU devices and drivers to the container and enables NVIDIA GPU Support.

The next step was to bind the CUDA libraries. This mounts the host's CUDA libraries into the container to match the driver version.

```
--bind /usr/local/cuda/targets/x86_64-linux:/usr/local/cuda/targets/x86_64-linux
--bind /usr/local/cuda/lib64:/usr/local/cuda/lib64
```

The last step is to define a variable to force the dynamic linker to prioritize the host CUDA libraries. Because the host CUDA libraries are exposed in the container, TensorFlow might choose the wrong one. It prevents that scenario from happening.

```
export LD_LIBRARY_PATH="/usr/local/cuda/targets/x86_64linux/lib:/usr/local/cuda/
                                        lib64:${LD_LIBRARY_PATH:-}"
```

With these three elements combined, tasks can run on a GPU.

7.2 Accelerate training speed

A classic issue encountered multiple times in training large models is the time required to converge to a solution. Throughout the thesis, multiple solutions were implemented to reduce the time required to train a model.

- **Parallelization**
- **Merging files**
- **Data Location**
- **Percentage per epochs**

At some point in the thesis, lots of files were used to train models. It blocked data loading because thousands of files were read, causing an I/O issue. To solve this issue and take advantage of the cluster resources, the idea was to parallelize the reading steps in **dl1_data_handler** library. It introduced the usage of multiprocessing tools and Thread pools. Here is an example of parallelizing the loading process. The modified file is available in the appendix.

```
with ThreadPoolExecutor(max_workers=(min(8, cpu_count()))) as pool:
    for output in pool.map(partial(process_file, constant_args), file_list):
        results.append(output)
```

This solution was then canceled as the number of files containing data was drastically reduced. It's because a merging tool has been introduced to merge together files containing data (4.4.4). It's a solution to reduce the training time and issues in handling thousands of files.

Percentage-per-epoch is a concept in which only a few batches are randomly selected to train an epoch. This parameter has been added to the CTLearn training tool to reduce the training time. This concept is explained in the chapter dedicated to the training task (6.3.2).

Another factor reducing training time is the batch size. By increasing batch size, the number of batches is reduced, resulting in fewer operations per epoch. The issue with this is double. The first is that the model's precision is significantly reduced because less information is provided. An additional issue is that having a higher batch size requires more memory.

The last idea was to change the path where the data is stored. At some point. The data was stored on a remote and heavily shared file system (BeeGFS). It introduces I/O overhead due to increased latency and limited bandwidth. This issue was resolved by moving the data back to a local location.

7.3 CTLearn library

Throughout the thesis, manipulations were required in the CTLearn library to adapt and add new elements aligned with the MLOps approach, or to address challenges in the production environment. Waiting on merge requests and new releases wasn't ideal. Therefore, an idea was put in place.

It consists of keeping modified files locally and redirecting requests to them. The code points to these files, which in turn fall back to the CTLearn library files. These files were implemented to

help Python scripts use them. This avoids reliance on the system-installed library and enables quick maintenance and script reuse.

The implicated files were the tools **train_model.py** and **predict_model.py**, and, more specifically, the tools they included. It allows direct use of a library on the cluster without complicated procedures to use a modified library.

The modified files are available in the appendix.

7.4 Quantization of complex models

Among the model optimization techniques that can be applied to models, Quantization has been a particular challenge. Normally, when working with simple, small models, quantization is applied correctly to each layer without issue. The problem is that with complex nested models, the quantization step isn't applied correctly to sub-layers. The quantization process fails to reach deeper layers. It's desired to have a similar quantization process applied at each layer to maintain uniformity and knowledge of the applied quantization.

This issue couldn't be resolved in time, and the quantization was abandoned to progress in the thesis. A Jupyter Notebook is available in the appendix, describing all methods used to try to quantize a system.

7.5 SLURM Time Limitation

The last challenge encountered during this thesis concerns a limitation of the cluster environment. On the server, there is a time limit on a job to prevent a GPU unit from being monopolized. The time limit is fixed to 12 hours. After that, the job is killed without finishing its current execution.

When training Deep Learning models on a large amount of data, this becomes an issue, as models can take over 12 hours to converge. However, since the best epoch is saved, this was initially less of a concern, as the model could still be evaluated.

To solve this, two things must be put in place: modify the code to enable rerunning by restarting from the last epoch trained, and allow SLURM to automatically requeue the job if it didn't finish training the model. To finish training the model, the early stopping system must be used. The code has been updated to introduce rerunning verification, triggered when an interrupted temp model is detected in a model folder.

```
# Set Rerun state

self.rerun = False
self.interrupted_epoch = 0
# Check if the output directory exists
if self.output_dir.exists():
    self.log.info("Output directory exists - checking for resume state")
    # Check if a rerun is in process
    interrupt_path = os.path.join(self.output_dir, "interrupt")
    if os.path.exists(interrupt_path):
        self.log.info("Resuming state ...")
```

```
        self.rerun = True
```

This temporary model contains the model's last saved epoch, enabling resuming training from it. Because the model in this scenario is already compiled, some steps in its preparation are skipped, and it restarts training from the last epoch.

```
# Train and evaluate the model

self.log.info"(Training and …"evaluating)
self.model.fit(
    self.training_loader,
    validation_data=self.validation_loader,
    epochs=self.n_epochs,
    initial_epoch=self.interrupted_epoch,
    steps_per_epoch=self.steps_per_epoch,
    class_weight=self.dl1dh_reader.class_weight,
    callbacks=self.callbacks,
    verbose=2,
)
```

The issue arises from allowing SLURM to rerun a job just before it finishes. The idea was to send a signal to the Python script before the job time limit is reached to save the epoch and metrics. The problem is that SLURM sends the signal to the Python script, but it is never executed correctly due to the Singularity environment and the way the bash script is executed. Multiple approaches have been tried to solve this, but the end of the thesis ended this research.

Among the potential solutions was the signal-and-requeue system in SLURM, which allows executing a command before the job ends. In addition, the signal handler was executed directly in the Python script.

```
#SBATCH --signal=B:USR1@180
#SBATCH --requeue

# Time limit reached

trap ‘’handle_timeout USR1 TERM

handle_timeout() {
    echo "Timeout soon - saving state & "requeueing
    scontrol requeue $SLURM_JOB_ID
    exit 0
}
```

There are some side effects to this training. To clarify this section, first consider the management of reruns in the CTLearn library, which couldn't be optimized, which may lead to issues with this configuration. Second, testing was not possible due to the SLURM rerun issue. Additionally, the training step assumes the task will complete without interruption, so metrics are only calculated afterwards. If the training process is killed, training metrics (estimated flops, number of layers, etc.) won't be saved, leaving a blank space in model-based reports (Figure 7.1).

Figure 7.1: Side-effect of the time limit, where training metrics aren't saved, and consequently not displayed in reports.

This challenge couldn't be solved in time and was addressed too late. However, the section on manual training remains relevant: manual training can still be performed by rerunning the job. The corresponding code is available in the appendix and on GitHub in a separate folder.

8 Conclusion

This chapter contains elements necessary to evaluate the work done on the thesis. This is to evaluate the project's global progress. It will include sections on difficulties encountered and future improvements.

8.1 Conclusion

Gamma rays play a crucial role in understanding high-energy and astrophysical events. This thesis explores this subject to find new ways to analyze such phenomena. It uses a library, CTLearn, dedicated to building neural networks (a machine learning technique) to reconstruct gamma rays, especially for event classification, energy estimation, and direction reconstruction. This library has been designed from an astrophysicist's perspective and may lack some essential modeling behaviors. The thesis aims to understand physics from the library's data and architecture, and to combine that with the MLOps (Machine Learning Operations) methodology to improve the model generation process.

The objective of this thesis was to bridge domain-specific knowledge with modern machine learning engineering practices in order to improve the model generation process. For that, several contributions have been made. The first step was preparing the data for use on an HPC (High Performance Computing) cluster. It required processing and documentation to understand it. The second element was the introduction of MLOps tools and techniques to enhance the model generation process. Providing reporting tools to analyze the performance of models and compare them was an interesting aspect. Alongside this, manipulations have been made to the library and environment to generalize the process as much as possible, with task repartitioning or setting configuration files to avoid code modification. The last element is the possibility to easily provide new types of models for training, which was difficult until now.

The thesis has succeeded in highlighting the importance of MLOps in the model generation process and in providing a computer science perspective in a field mostly dominated by astrophysicists. The project can be regarded as a first step towards these ideas, serving as an overall proof of concept. It opens the door to future work and improvements to the current state of the CTLearn library and to model generation.

The thesis will be presented at the Swiss AI days conference as a poster to introduce gamma-ray astronomy and highlight the importance of MLOps when AI solutions are implemented to address a specific, detailed issue.

8.2 Limitations

The thesis has faced multiple limitations that have slowed or halted the advancement of the model generation and related tools.

- **CTLearn Library** : CTLearn library is a well-developed library about deep learning on IACT. However, the library isn't easy to understand and use. Manipulating the library is also difficult because of numerous interdependencies and a complex structure that breaks if changes aren't introduced meticulously.
- **Cluster limitations** : The cluster limited the training of models in multiple ways. First of all, the training isn't directly executed because Baobab (3.6.1) uses a queue system. The cluster also imposes time limits on specific tasks, blocking long training sessions multiple times. It can also be limited by available resources.

8.3 Future improvements

The thesis introduced the MLOps methodology and tools for managing model generation. This is only an introduction, and multiple improvements could be made to further enhance the CTLearn library and model generation, leading to better solutions for reconstructing gamma rays. With this limited time, it was only possible to scratch the surface of many of the aspects introduced in the thesis. Some ideas to explore in the future are listed below.

- **Get deeper in the MLOps approach** : Adding new MLOps features to further automate, optimize, etc., the generation of the model and its monitoring/maintenance. For example, automation with external tools like GitHub and CI/CD, or data and model versioning with Weights&Biases, could be an interesting starting point.
- **Find better models to reconstruct gamma ray** : The possibility has been enabled to use custom models with the template (6.2.3) to test new models. It could be explored and benchmarked to find the most suitable model and hopefully, a better one.
- **Implementing models on FPGA** : Going further in model optimization could be interesting. The triggering system could benefit from using high-performance, low-resource neural networks with low computational time.
- **Combine tasks** : Something mentioned at the start of the project, but never again is the combination of tasks. To centralize the prediction, it could be useful to have a single model predicting three outputs: type, energy, and direction.
- **Test on real data** : Up until now, the models were trained, tested, and compared on the basis of simulated data. It's important to verify if performance may vary when working with real data. Real data are a lot more unstable and random than simulated data.
- **Optimize model generation** : The model generation caused problems because of the time it took to train. Also related to that, the cluster has some time limitations that prevent standard model training. Looking at ways to optimize the model could be interesting.

8.4 Personal reflection

This type of project is a unique opportunity. I like astronomy, and working on something related to this field was exciting. At first, I didn't think about the difficulty of the subject. It started becoming clearer as the thesis progressed. I learned a lot about gamma rays and telescopes during my research. I'm frustrated because I'm starting to really understand the physics side of the thesis, only when it's about to finish.

This project helped me to improve my skills, especially the understanding of Deep Learning models and MLOps methodology. Related to that, I could improve my capacity to adapt to unforeseen events, such as limitations on the cluster, and to complex libraries like CTLearn. I learned to understand a library in detail and leverage it when building tools. I also learned new stuff, like working with large datasets and the challenges they entail, and how to work with a cluster (SLURM and a queue system).

The results of the thesis are also a bit frustrating, as I haven't had enough time to explore deeper into the implementation of the MLOps approach. I feel like I've only touched on various topics without going into detail.

I hope the solutions provided will help the CTLearn community and serve as a basis for future model development. I introduced the subject of MLOps in CTAO, and I hope it will continue to delve deeper into the methodology over time. I was happy to participate in global meetings with other collaborators to explore further the gamma-ray domain.

9 Declaration of honor

I, the undersigned Hugo Varenne, hereby declare on my honor that the submitted work is the result of my own personal effort. I certify that I have not engaged in plagiarism or any other form of fraud. All sources of information used and any author quotations have been clearly referenced.

I further declare that any use of artificial intelligence tools during the preparation of this report was limited strictly to assistance in checking and correcting the grammar of the English text. An additional use was to generate an initial template for the model report code, only intending to have an interface ready to use. Other than that, no AI tools have been used for generating code, ideas, or documentation.

Bibliography

[1] *A Comprehensive Guide to Neural Network Model Pruning | Datature Blog*. en. URL: https://datature.com/blog/a-comprehensive-guide-to-neural-network-model-pruning (visited on 01/19/2026).

[2] K. Abe et al. "A new method of reconstructing images of gamma-ray telescopes applied to the LST-1 of CTAO". en. In: *Astronomy & Astrophysics* 691 (Nov. 2024), A328. ISSN: 0004-6361, 1432-0746. DOI: 10.1051/0004-6361/202450889. URL: https://www.aanda.org/articles/aa/abs/2024/11/aa50889-24/aa50889-24.html (visited on 01/29/2026).

[3] J. A. Barrio. "Status of the large size telescopes and medium size telescopes for the Cherenkov Telescope Array observatory". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*. 10th International Workshop on Ring Imaging Cherenkov Detectors (RICH 2018) 952 (Feb. 2020), p. 161588. ISSN: 0168-9002. DOI: 10.1016/j.nima.2018.11.047. URL: https://www.sciencedirect.com/science/article/pii/S016890021831622X (visited on 01/30/2026).

[4] Konrad Bernlohr. "Simulation of Imaging Atmospheric Cherenkov Telescopes with CORSIKA and sim_telarray". In: *Astroparticle Physics* 30.3 (Oct. 2008). arXiv:0808.2253 [astro-ph], pp. 149–158. ISSN: 09276505. DOI: 10.1016/j.astropartphys.2008.07.009. URL: http://arxiv.org/abs/0808.2253 (visited on 01/30/2026).

[5] Alessio Berti. "Observation of the gamma-ray sky from the ground: the IACT technique". en. In: (). URL: https://agenda.infn.it/event/47777/contributions/269943/contribution.pdf (visited on 01/30/2026).

[6] Thorsten Buss et al. *CaloHadronic: a diffusion model for the generation of hadronic showers*. arXiv:2506.21720 [physics]. June 2025. DOI: 10.48550/arXiv.2506.21720. URL: http://arxiv.org/abs/2506.21720 (visited on 01/31/2026).

[7] Prince Canuma. *MLOps: What It Is, Why It Matters, and How to Implement It*. en-US. July 2022. URL: https://neptune.ai/blog/mlops (visited on 01/20/2026).

[8] *Celestial Bodies*. en-US. URL: https://unacademy.com/content/cbse-class-12/study-material/physics/celestial-bodies/ (visited on 01/31/2026).

[9] Matteo Cerruti. "Leptonic and Hadronic Radiative Processes in Supermassive-Black-Hole Jets". In: *Galaxies* 8.4 (Oct. 2020). arXiv:2012.13302 [astro-ph], p. 72. ISSN: 2075-4434. DOI: 10.3390/galaxies8040072. URL: http://arxiv.org/abs/2012.13302 (visited on 01/31/2026).

[10] *Commande SCP de Linux*. fr. Sept. 2021. URL: https://www.ionos.fr/digitalguide/serveur/configuration/commande-scp-de-linux/ (visited on 01/06/2026).

[11] *Gamma-ray astronomy*. en. Page Version ID: 1320843996. Nov. 2025. URL: https://en.wikipedia.org/w/index.php?title=Gamma-ray_astronomy&oldid=1320843996 (visited on 01/31/2026).

[12] PhD thesis Gasparetto. *CTA Hierarchical Data Model*. Tech. rep. Defines R0, R1, DL0, etc. Università degli Studi di Trieste, 2025. URL: https://arts.units.it/retrieve/e2913fdd-4852-f688-e053-3705fe0a67e0/thesis_PHD_Gasparetto_Reviewed_resized.pdf.

[13] Laetitia Guidetti. *VAE for detecting anomalies in Cherenkov telescope data*. en. 2025.

[14] Song Han et al. *Learning both Weights and Connections for Efficient Neural Networks*. arXiv:1506.02626 [cs]. Oct. 2015. DOI: 10.48550/arXiv.1506.02626. URL: http://arxiv.org/abs/1506.02626 (visited on 01/19/2026).

[15] *HDF5: HDF5 Data Model and File Structure*. URL: https://support.hdfgroup.org/documentation/hdf5/latest/_h5_d_m___u_g.html (visited on 01/29/2026).

[16] *Heavy Nuclei - an overview | ScienceDirect Topics*. URL: https://www.sciencedirect.com/topics/earth-and-planetary-sciences/heavy-nuclei (visited on 01/31/2026).

[17] M. Heller et al. "An innovative silicon photomultiplier digitizing camera for gamma-ray astronomy". en. In: *The European Physical Journal C* 77.1 (Jan. 2017), p. 47. ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-017-4609-z. URL: https://doi.org/10.1140/epjc/s10052-017-4609-z (visited on 01/31/2026).

[18] *Hierarchical Data Formats - What is HDF5? | NSF NEON | Open Data to Understand our Ecosystems*. URL: https://www.neonscience.org/resources/learning-hub/tutorials/about-hdf5 (visited on 01/29/2026).

[19] *High Performance Computing - e-Research - UNIGE*. en. Last Modified: 2025-02-20T12:44:23Z. July 2019. URL: https://www.unige.ch/eresearch/en/services/hpc (visited on 01/06/2026).

[20] *High Performance Computing - e-Research - UNIGE - Baobab*. en. July 2019. URL: https://doc.eresearch.unige.ch/hpc/hpc_clusters#for_advanced_users (visited on 01/06/2026).

[21] *Homepage*. en-US. URL: https://www.ctao.org/ (visited on 01/30/2026).

[22] *https://www.unige.ch/sciences/astroparticle/projects/sst*. en. Last Modified: 2024-05-27T10:12:23Z. Mar. 2020. URL: https://www.unige.ch/sciences/astroparticle/projects/sst (visited on 01/31/2026).

[23] *Introduction to Convolution Neural Network*. en. Section: Machine Learning. URL: https://www.geeksforgeeks.org/machine-learning/introduction-convolution-neural-network/ (visited on 01/13/2026).

[24] *Introduction to Singularity — Singularity container 3.5 documentation*. URL: https://docs.sylabs.io/guides/3.5/user-guide/introduction.html (visited on 01/06/2026).

[25] Florian June. *Model Quantization 1: Basic Concepts*. en. May 2024. URL: https://medium.com/@florian_algo/model-quantization-1-basic-concepts-860547ec6aa9 (visited on 01/20/2026).

[26] Ajitesh Kumar. *Different Types of CNN Architectures Explained: Examples*. en-US. Dec. 2023. URL: https://vitalflux.com/different-types-of-cnn-architectures-explained-examples/ (visited on 01/13/2026).

[27] E. A. Kuraev et al. "Bremsstrahlung and pair production processes at low energies, multi-differential cross section and polarization phenomena". In: *Physical Review C* 81.5 (May 2010). arXiv:0907.5271 [hep-ph], p. 055208. ISSN: 0556-2813, 1089-490X. DOI: 10.1103/PhysRevC.81.055208. URL: http://arxiv.org/abs/0907.5271 (visited on 01/31/2026).

[28] Bastien Lacave. *BastienLacave/CTLearn-Manager*. original-date: 2024-12-18T18:01:02Z. Dec. 2025. URL: https://github.com/BastienLacave/CTLearn-Manager (visited on 12/09/2025).

[29] Alessandro Lamberti. *Deep Learning Model Optimization Methods*. en-US. Mar. 2024. URL: https://neptune.ai/blog/deep-learning-model-optimization-methods (visited on 01/13/2026).

[30] *Lmod: A New Environment Module System — Lmod 9.0.5 documentation*. URL: https://lmod.readthedocs.io/en/latest/index.html (visited on 01/06/2026).

[31] marcel. *BeeGFS*. en-US. URL: https://www.beegfs.io/c/ (visited on 01/06/2026).

[32] Rick Merritt. *What is MLOps?* en-US. Sept. 2020. URL: https://blogs.nvidia.com/blog/what-is-mlops/ (visited on 01/20/2026).

[33] *Méthode de Monte-Carlo*. fr. Page Version ID: 228963734. Sept. 2025. URL: https://fr.wikipedia.org/w/index.php?title=M%C3%A9thode_de_Monte-Carlo&oldid=228963734 (visited on 01/30/2026).

[34] Tjark Miener et al. *CTLearn: Deep learning for imaging atmospheric Cherenkov telescopes event reconstruction*. Language: en. Mar. 2025. DOI: 10.5281/ZENODO.3342952. URL: https://zenodo.org/doi/10.5281/zenodo.3342952 (visited on 01/30/2026).

[35] *Module: tfmot | TensorFlow Model Optimization*. en. URL: https://www.tensorflow.org/model_optimization/api_docs/python/tfmot (visited on 01/19/2026).

[36] D. Nieto et al. *CTLearn: Deep Learning for Gamma-ray Astronomy*. arXiv:1912.09877 [astro-ph]. Dec. 2019. DOI: 10.48550/arXiv.1912.09877. URL: http://arxiv.org/abs/1912.09877 (visited on 01/07/2026).

[37] Cosimo Nigro et al. "Evolution of Data Formats in Very-High-Energy Gamma-Ray Astronomy". en. In: *Universe* 7.10 (Oct. 2021), p. 374. ISSN: 2218-1997. DOI: 10.3390/universe7100374. URL: https://www.mdpi.com/2218-1997/7/10/374 (visited on 01/29/2026).

[38] *Optimiser les performances de TensorFlow à l'aide du profileur | TensorFlow Core*. fr-x-mtfrom-en. URL: https://www.tensorflow.org/guide/profiler?hl=fr (visited on 01/12/2026).

[39] Jordan Pannell. *Monte Carlo estimation of pi*. en. Aug. 2020. URL: https://jordanpannell.co.uk/blog/Monte-Carlo-estimation-of-pi/ (visited on 01/30/2026).

[40] Souvik Paul. *Pruning in Deep Learning Models*. en. June 2020. URL: https://medium.com/@souvik.paul01/pruning-in-deep-learning-models-1067a19acd89 (visited on 01/19/2026).

[41] Mario Pecimotika. "Transmittance Simulations for the Atmosphere with Clouds". PhD thesis. Nov. 2018. DOI: 10.13140/RG.2.2.34140.95361/1.

[42] *Provided documentation by Unige on CTAO general understanding*. en. URL: None.

[43] P. Rajda et al. *DigiCam - Fully Digital Compact Read-out and Trigger Electronics for the SST-1M Telescope proposed for the Cherenkov Telescope Array*. arXiv:1508.06082 [astro-ph]. Aug. 2015. DOI: 10.48550/arXiv.1508.06082. URL: http://arxiv.org/abs/1508.06082 (visited on 01/31/2026).

[44] *ReLU Activation Function in Deep Learning*. en-US. Section: Deep Learning. URL: https://www.geeksforgeeks.org/deep-learning/relu-activation-function-in-deep-learning/ (visited on 01/13/2026).

[45] *Residual Networks (ResNet) - Deep Learning*. en-US. Section: Deep Learning. URL: https://www.geeksforgeeks.org/deep-learning/residual-networks-resnet-deep-learning/ (visited on 01/13/2026).

[46] *Residual neural network*. en. Page Version ID: 1314589246. Oct. 2025. URL: https://en.wikipedia.org/w/index.php?title=Residual_neural_network&oldid=1314589246 (visited on 01/13/2026).

[47] Konstancja Satalecka. "Multimessenger studies of point-sources using the IceCube neutrino telescope and the MAGIC gamma-ray telescope". PhD thesis. Oct. 2010.

[48] *Slurm Workload Manager - Quick Start User Guide*. URL: https://slurm.schedmd.com/quickstart.html (visited on 01/06/2026).

[49] *SST-1M-collaboration/sst1mpipe*. original-date: 2024-03-18T10:20:20Z. Jan. 2026. URL: https://github.com/SST-1M-collaboration/sst1mpipe (visited on 01/29/2026).

[50] Keras Team. *Keras documentation: Callbacks API*. en. URL: https://keras.io/api/callbacks/ (visited on 01/09/2026).

[51] *The first Cherenkov telescopes in the Czech Republic to observe the Crab Nebula - Akademie věd České republiky*. URL: https://www.avcr.cz/en/news-archive/The-first-Cherenkov-telescopes-in-the-Czech-Republic-to-observe-the-Crab-Nebula/ (visited on 01/31/2026).

[52] *Traitlets — traitlets 5.14.3 documentation*. URL: https://traitlets.readthedocs.io/en/stable/ (visited on 01/08/2026).
[53] *What Is DevOps? | IBM*. en. May 2025. URL: https://www.ibm.com/think/topics/devops (visited on 01/20/2026).
[54] *What is MLOps?* en-US. Dec. 2021. URL: https://www.databricks.com/glossary/mlops (visited on 01/20/2026).
[55] *What is MLOps? | IBM*. en. Apr. 2024. URL: https://www.ibm.com/think/topics/mlops (visited on 01/21/2026).
[56] *What is Monte Carlo Simulation?* en. Section: Artificial Intelligence. URL: https://www.geeksforgeeks.org/artificial-intelligence/what-is-monte-carlo-simulation/ (visited on 01/30/2026).
[57] *What is Quantization? | IBM*. en. July 2024. URL: https://www.ibm.com/think/topics/quantization (visited on 01/20/2026).

List of Figures

2.1 Fermi and HESS . . . 3
2.2 Picture of atmosphere transparency for different wavelengths . . . 4
2.3 Captured gamma-ray . . . 4

3.1 Astrophysical objects . . . 9
3.2 Gamma ray observation of the sky . . . 9
3.3 Gamma ray shower development . . . 10
3.4 electromagnetic/hadronic shower . . . 11
3.5 Cherenkov light . . . 11
3.6 IACT telescopes . . . 14
3.7 SST-1M telescope . . . 15
3.8 Usage of a Linux virtual machine (WSL) . . . 16
3.9 Environment on the cluster . . . 16
3.10 Example of environment configuration . . . 18
3.11 Example of a bash script used to run a job on SLURM . . . 19

4.1 Visualization of parts of the simulation . . . 25
4.2 Monte Carlo simulation . . . 26
4.3 Example of HDF5 file structure . . . 27
4.4 Workflow of data level (excluding R0 and R1) . . . 28
4.5 Example of R0 waveforms . . . 29
4.6 Example of R1 waveforms . . . 30
4.7 Extraction of the valuable information from the waveform . . . 31
4.8 Hillas parameters extraction . . . 31
4.9 Example of an event in pixel array or frame . . . 32

5.1 MLOps cycle . . . 35
5.2 Task decomposition of the pipeline . . . 37
5.3 Configuration file for a model generation . . . 38
5.4 Training loss and the Validation loss . . . 42
5.5 Model summary . . . 43
5.6 ROC Curve . . . 46
5.7 Gammaness distribution graphic . . . 47
5.8 Confusion matrix . . . 48
5.9 Calibration probability graphic . . . 49
5.10 Distribution of events over energy ranges (bins) . . . 51
5.11 Migration matrix . . . 52

5.12 Energy resolution . . . 53
5.13 Bias and Standard deviation graphic (energy) . . . 54
5.14 Performance degree Curve . . . 56
5.15 Alt/Az Distribution graphic . . . 57
5.16 Angular resolution graphic . . . 58
5.17 Bias and Standard deviation graphic (direction) . . . 59
5.18 Example of table for Performance metrics . . . 60
5.19 Example of table for Ranking Performance metrics . . . 61
5.20 Example of a combined graphic . . . 62
5.21 Example of a subgraph comparison . . . 63

6.1 Accuracy and AUC metrics over the number of samples for each telescope . . . 66
6.2 Main concept of the residual block . . . 67
6.3 Default Residual block in CTLearn library . . . 68
6.4 Default ResNet Architecture on CTLearn library . . . 69
6.5 Example of VGG16-CNN Network . . . 70
6.6 Default CNN implementation for CTLearn library . . . 71
6.7 Network structure of the template for custom model . . . 72
6.8 Principle of percentage of data used per epochs . . . 76
6.9 Repartition of steps within the TrainCTLearnModel tool . . . 77
6.10 Representation of how the model retrieves the number of layers recursively . . . 78
6.11 Process to obtain FLOPs from a model . . . 78
6.12 Pruning process . . . 80
6.13 Quantization process . . . 83

7.1 Side-effect limitation of time . . . 89

11 Appendix

- Git repository
- Jupyter notebooks
- Bash SLURM interaction template
- Environment configuration file
- Poster
- Template for custom models
- Configuration file for each task (template)
- Python scripts for each task (+ tools related)
- Python scripts for data processing
- Reports generated
- Modified scripts to enable rerunning