TRACK RECONSTRUCTION USING GRAPH NEURAL NETWORKS

Trabajo de Fin de Máster

para acceder al

MÁSTER INTERUNIVERSITARIO (UC-UIMP) EN FÍSICA DE PARTÍCULAS Y DEL COSMOS

Autor: Carlos Andres Cruz Carpio

Director: Pablo Martínez Ruiz del Árbol

Codirector: Rubén López Ruiz

Julio - 2024

Abstract

After more than a decade of data-taking of the experiments at the Large Hadron Collider at CERN, no evidence of New Physics has been found. A new accelerator, the High-Luminosity Large Hadron Collider is currently under construction, increasing the instantaneous luminosity by one order of magnitud. The new accelerator conditions generate a challenging environment for the detectors at the LHC. One of the most important difficulties that the CMS experiment will find is the extremely large hit occupancy in the tracker and the huge combinatorial problem associated to the track reconstruction. This is even more difficult for displaced topologies emerging from potential new, long-lived particles, where the traditional seeding methods cannot be used. This work explores the use of Graph Neural Networks as an alternative procedure to identify tracks in a detector. A simple track generator has been developed together with a simplified model of the CMS tracker, and the resulting tracks and hits are reconstructed using a dedicated GNN algorithm trained with a number of example tracks. The performance of the algorithm is tested to be higher than 80% even for 500 simultaneous tracks.

Key words: Track reconstruction, graph neural networks, Particle Detectors.

Resumen

Después de más de una década de toma de datos, los experimentos del Gran Colisionador de Hadrones en el CERN, no han encontrado evidencia de Nueva Física. Un nuevo acelerador, el Gran Colisionador de Hadrones de Alta Luminosidad está siendo construido actualmente, aumentando la luminosidad instantánea por un order de magnitud. Las nuevas condiciones del acelerador generan un entorno complejo para los detectores del LHC. Una de las dificultades más importantes que el experimento CMS va a encontrar, es la alta ocupancia de hits en el tracker y el consecuente problema combinatorial asociado a la reconstrucción de trazas. Esto es aún más acuciante para el caso de topologías desplazadas que puedan emerger de nuevas partículas con alto tiempo de vida. Este trabajo explora el uso de Redes Neuronales de tipo Grafo como un procedimiento alternativo para identificar las trazas del detector. Un generador de trazas sencillo junto con un modelo simplificado del tracker the CMS ha sido desarrollado, y los hits resultantes han sido reconstruidos utilizando un algoritmo GNN entrenado con un número de trazas de ejemplo. El desempeño del algoritmo ha resultado ser mejor que el 80% incluso para un caso con 500 trazas simultáneas.

Palabras clave: Reconstrucción de Trazas, Redes Neuronales, Grafos, Detectores de Partículas.

Acknowledgements

En primer lugar debo agradecer a la Fundación Carolina, por confiar en mí y permitirme acceder a este excelente programa. Quedo profundamente agradecido por esta oportunidad.

Este trabajo no hubiera sido posible sin los directores Pablo y Rubén. Les agradezco por la retroalimentación, los consejos y la paciencia a lo largo de estos meses de trabajo y en los momentos mas decisivos.

Por último y no menos importante a mi familia, de sangre y por elección, quienes estuvieron apoyándome y evitando que desfallezca. Se que el tan ansiado reencuentro será placentero.

Contents

1 Introduction **1**

2 Particle Tracking at the CMS detector **3**
2.1 The CMS Tracker 4
2.1.1 Pixel detector 4
2.1.2 Strip detector 4
2.2 Track Reconstruction 5
2.2.1 Local reconstruction 5
2.2.2 Kalman Filter 5
2.2.3 Global Reconstruction 6

3 Simulation Framework **9**
3.1 Tracker Geometry Builder 9
3.2 Track Propagation 10
3.2.1 POCA parametrization 12
3.3 Particle Generator 12

4 Graph Neural Networks **17**
4.1 Graph notions 17
4.1.1 Definition 17
4.2 Graph Neural Networks 18
4.2.1 The Message Passing Mechanism 18
4.2.2 GNN variants 19
4.3 PyTorch Geometric 20
4.4 Graph generation from a collision event 21
4.4.1 Point Generation 21
4.4.2 Edge generation 21
4.4.3 PyG Graph creation 22
4.5 Model Definition 24
4.6 Model Training 25
4.6.1 Loss function 25
4.6.2 Optimizer 25
4.6.3 Training and validation sets 26
4.7 Model Evaluation 26
4.7.1 Post prediction filter 26
4.7.2 Evaluation metrics 26

5 Results **29**
5.1 Results using an opening window of 20 degrees 29
5.1.1 Model training 29
5.1.2 Accuracy studies 30
5.2 Results obtained using an opening window of 45 degrees 32
5.2.1 Model training 32
5.2.2 Accuracy 33
5.3 Track accuracy comparison 34

6 Conclusions **37**

Bibliography **40**

Chapter 1

Introduction

Track reconstruction is an important part of many High Energy Physics experiments. Its goal is to identify the tracks and their parameters, position, direction and momentum, using particle signals from the detectors. It can take several steps to achieve this goal, and it will depend on various factors. Not only the experiment design is important, e.g. how detectors are arranged to collect signals, but also how these signals are treated, the algorithms implemented to select, discard, store or filter data, are also relevant to properly reconstruct tracks.

Since each experiment is different than the others, the tracking task will be different as well. Therefore, this work will be based solely on the Compact Muon Solenoid (CMS) detector. The CMS [1] is a general purpose detector at the Large Hadron Collider (LHC) [2], which has a broad physics programme from studying the Standard Model to searching extra dimensions and particles that could make up dark matter. A description of CMS tracker is given in Chapter 2.

The process of reconstruction in CMS is computationally challenging. Not all particles will be detected, only the tracks of stable or sufficiently long lived charged particles, such as electrons, protons or muons are visible in the tracking detectors. Short lived particles are reconstructed from their decay product [3]. In addition, the tracking software must run sufficiently fast to be used not only for offline event reconstruction (of $\approx 10^9$ events per year), but also for the CMS High-Level Trigger (HLT), which processes events at rates of up to 100 kHz.

Figure 1.1 shows the transverse plane of a generic detector, with long-lived particles (LLPs) decaying after traveling some distance from the interaction point. The inner layers, which are part of the tracker, would detect none or a small number of hits and this could lead to the decision to mistakenly associate to a different particle or discarding a potential track. In general, track reconstruction algorithms are optimized to detect particles coming from the vertex of the detector. This becomes evident taking into account that most of the tracking algorithms start reconstructing the track from hits in the first layers of the trackers (seeds) that may be missing for a displaced particle.

On the other hand, LLPs are an important physics case nowadays. After more than a decade of searches at the LHC, no Physics Beyond the Standard Model (BSM) has been

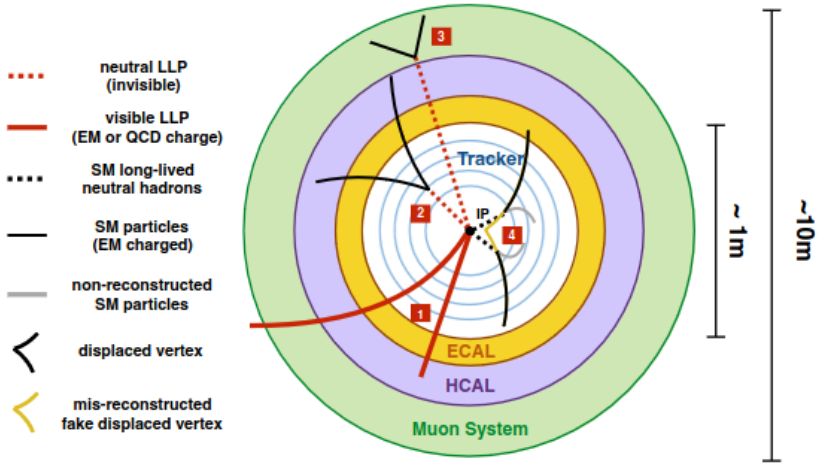

Figure 1.1: Cross section of a detector showing particle tracks and displaced vertex [4].

found. The answer could be that BSM is realized through long-lived particles that escape detection due to the difficulties explained above. The situation is even worse when it is extrapolated to the next High-Luminosity Large Hadron Collider (HL-LHC) [5], where the pattern recognition problem is even more complicated due to the large multiplicity of tracks. The combinatorial seeding and track building algorithms are inherently serial and scale quadratic or worse with detector occupancy. It is thus worthwhile to investigate new solutions such as methods based on Deep Learning [6]

This work intends to explore the Graph Neural Network approach with the PyTorch Geometric library in order to reconstruct tracks using particle hits but without including a seeding step, and therefore suitable for displaced tracks. To do this, a controlled environment is required, that is a simulation framework where tracks can be generated with desired parameters and detectors that can return the position of the particle.

The simulation framework is described in Chapter 3, including how it has been coded, its output and limitations. The basis of graph neural networks and the model employed is found in Chapter 4. And finally the results and conclusions will be presented in the last Chapters of this work.

Chapter 2

Particle Tracking at the CMS detector

This Chapter will describe the CMS tracker: geometric layout, detector arrangement, dimensions, etc, and the tracking process itself. The importance of this section lies on the next step of the work, this section establishes the basis for the simulation framework that will be presented in the next Chapter.

Roughly speaking CMS consists of four different systems. From the center of the detector, a central tracking system (tracker), where the position of charged particles is recorded, is located. Electrons and photons are then detected in an electromagnetic calorimeter. Increasing in radius, a hadronic calorimeter is placed to measure the properties of hadrons, and finally a muon system to detect and measure muons, can be found. [7]. A sketch of these systems is shown in figure 2.1.

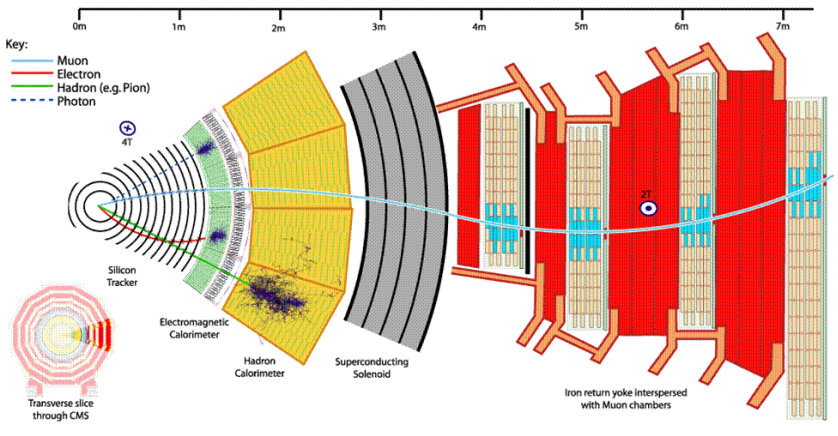

Figure 2.1: Slice of a cross section of CMS, showing the different systems and the particles they could detect.

2.1 The CMS Tracker

The following description of the CMS tracker is based on reference [8] dated on the year 2014.

The tracker can be visualized as a series of concentric barrels. The whole cylindrical structure has dimensions of 5.8 m in length and 2.5 m in diameter This volume is aligned with the LHC beam line and is surrounded by a 3.8 T magnetic field From this an axis can be set along the beam line, the Z axis, and draw an orthogonal plane, the transverse XY plane, with the X axis pointing to the center of the LHC ring and the Y axis pointing upwards. Thus, a three-dimensional coordinate system has been established and will be used in the simulation process afterwards.

The tracker is divided in two sub detectors, the pixel and trip tracker, both made of silicon. The pixel tracker comprises the first three layers up to 10.2 cm and the strip tracker has ten layers up to 110 cm. Both sub detectors have their endcaps at each end of the cylinder.

2.1.1 Pixel detector

This is the innermost section of the tracker, it consists of three layers with radii of 4.4, 7.3 and 10.2 cm. The pixel tracker provides three dimensional positions measurement with high resolution. It has about 66 million pixel with each having dimensions of the order of μm.

Because of its location it plays a major role in the reconstruction process. It takes part of the seeding process, establishing starting parameters of possible tracks with hits of this section, this process will be described later.

Figure 2.2 show a transverse plot of the tracker where the pixel section is clearly marked as **PIXEL**.

2.1.2 Strip detector

The strip tracker is the complement of the pixel tracker. It in turn can be divided in three parts. The tracker inner barrel (TIB), the tracker outer barrel (TOB) and the encaps. While it does not have the same resolution of the pixel tracker it does cover a larger area.

The inner barrel covers the region between $r < 55cm$ and $|z| < 118cm$ and the outer barrel covers the region $r > 55cm$ and $|z| < 118cm$. The endcaps are placed at both ends of the barrels.

In figure 2.2 the strip tracker constituents can be seen, labeled as **TIB**, **TOB** and **TEC**.

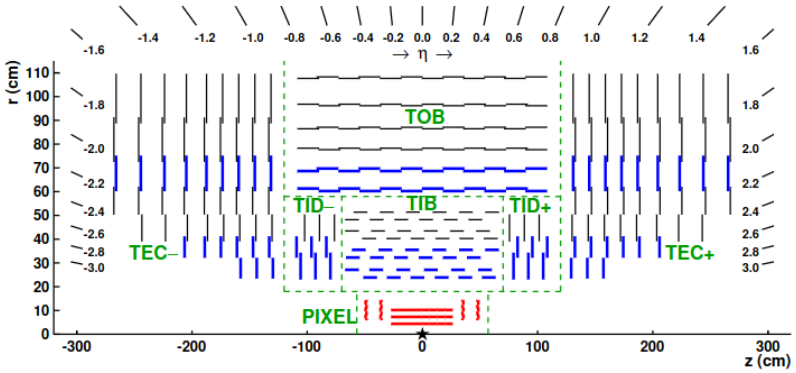

Figure 2.2: Schematic view of the CMS tracker extracted from [8] dated on 2014. The pixel tracker as well as the strip tracker can clearly noted

2.2 Track Reconstruction

2.2.1 Local reconstruction

The basic idea of the algorithms used in this step will be given, they can be skipped for the purpose of this work. Reference [8] covers them in detail.

The main goal of this step is to estimate positions and uncertainties of hits. Only signals that surpass a certain threshold in both the pixel or the strip tracker will be clustered and considered hits.

To do this two algorithms are applied. The first, a fast one, compares nearby pixels, and by studying the charge in the pixels the position of the cluster is estimated. The second algorithm appears due to the need to maintain the performance after high level radiation, and so the comparison is made with simulation templates, and again position can be estimated.

2.2.2 Kalman Filter

The Kalman filter is an algorithm to make estimations of an unknown variable or variables, based on previous observations over time and considering uncertainties or noise. It is widely used in navigation systems and signal processing.

Part of state space models, which describe the changing state of an object in space or time but only of interest at discrete instances, the Kalman filter can be understood as a sequence of alternating prediction and update steps. [3].

In the application of the Kalman filter to track reconstruction, the system equation is usually non linear, as the next chapter will show, so it will have a general form:

$$q_k = f_{k|k-1}(q_{k-1}) + \gamma_k, E[\gamma_k] = g_k, Var[\gamma_k] = Q_k \tag{2.1}$$

Where q_k is a state vector at an instance k, γ_k process noise, covariance matrix Q_k

The idea is that given some observations m_i, an initial state q_0 and a initial state covariance matrix C_0 all states can be estimated recursively by the Kalman filter. The estimated vector is obtained by different steps. A prediction, an update and a smoothing.

- Prediction: State vector and its covariance matrix are propagated to the next instance using the system equation.
- Update: Weighted mean of the predicted step and the observation m_i, it has an associated chi-square statistic χ^2.
- Smoothing: Propagates the full information in the last estimate back to all previous states.

2.2.3 Global Reconstruction

The tracking software used at CMS is known as Combinatorial Track Finder (CTF), it is an extension of the Kalman filter, previously described. Applying multiple iterations of the CTF sequence reconstructed tracks are obtained. The CTF process has four steps, seed generation, track finding, track fitting and track selection. [8]

Seed generation

Charged particles follow helical paths and therefore five parameters are needed to define a trajectory. This parameters will be given in detail in Chapter 3.

Seeds define the starting parameters. These seeds are built from three or two hits coming out of the inner part of the tracker with the assumption the particle originated near the interaction point.

Track finding

This process is based on the Kalman filter method. After defining seeds, track candidates are built by adding hits from successive layers.

Similar to the Kalman filter process here four steps are performed. Navigation, where the next layer is determined by using the parameters of the seeding stage. The second step looks for compatible hits with no more than three standard deviations. The third step groups hits and a χ^2 test is performed to check compatibility. The final step updates the trajectory, the original track candidates get a new compatible hit.

Track fitting

The previous stage yields a series of points and estimated track parameters. This stage refits the track by applying a Kalman filter and a smoother. Also, chi square χ^2 statistic is applied to search for outliers.

The process proceeds iterative from the innermost point to the outermost one. To obtain a better precision the filtering and smoothing sections use a Runge-Kutta propagator, a numerical method to estimate values after a given time step, with great accuracy.

Track selection

Because in a CMS event many different particles are present, the algorithm could provide fake tracks. Thus, track selection is made by analyzing the the number of layers that have hits, whether their fit yielded a good χ^2/dof, and how compatible they are with originating from a primary interaction vertex.

Figure 2.3 shows a scheme summarizing the described iterative process. After each successful step the used hits are discarded and new combination are considered.

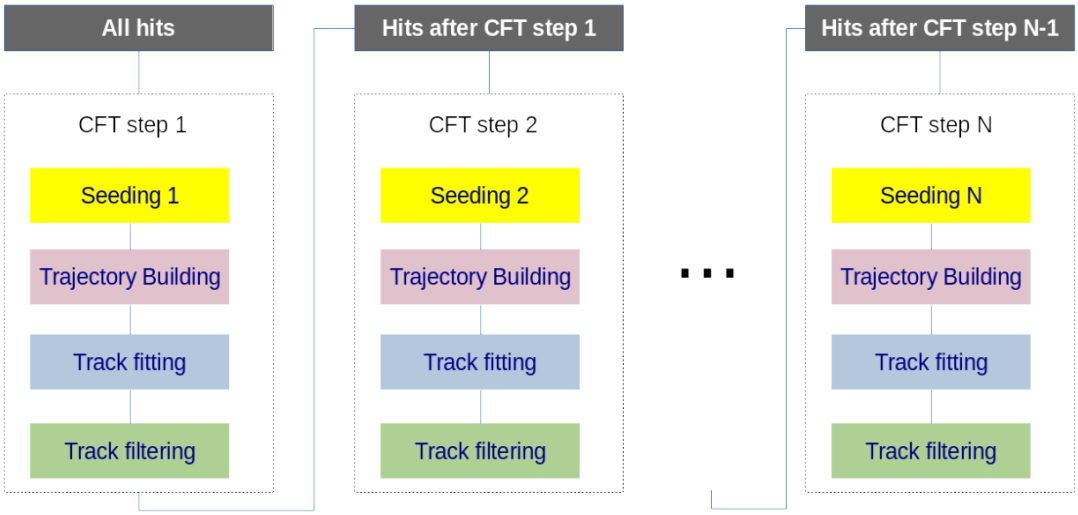

Figure 2.3: Scheme of the iterative process

Chapter 3

Simulation Framework

This work represents a proof-of-concept on the use of Graph Neural Networks for particle reconstruction. Due to the intrinsic complexity of the CMS Tracker and its real tracking procedure, a simulation framework has been devised, simplifying the CMS tracker geometry.

The framework is divided in three parts: the first is the *tracker builder geometry* where concentric polygonal layers resembling the CMS tracker with the capability to "detect" hits within them are built. The second is the *particle generator*, where the 4-momenta of the particles is obtained from random distributions. And the third and final is the *track propagation*, in which the trajectories of the particles are estimated.

It has been developed using python [9] and heavily relying on the ***numpy*** [10] and ***matplotlib*** [11] libraries. The code is designed following a object oriented scheme, where each part is a class object, and each class has several functions for the different required tasks.

3.1 Tracker Geometry Builder

As mentioned in Chapter 2 the tracker of CMS has a cylindrical shape consisting on concentric layers immersed in a magnetic field. The idea is to model the layers of detectors as concentric polygons. To do this, points over a circumference are generated. And then, two points are taken and a line is drawn between them.

The hits are calculated by expressing each side of one layer as a plane of the form $ax + by + cz + d = 0$. With the choice of building them around the z axis it turns out that the component c is always 0, making the equation simpler to work with. Here numpy plays an important role, as x,y are collections of values of a given length, numpy arrays; the calculation to find intersections is then straightforward, just plugging the equation, the arrays and filtering which points give a null output. Afterwards, another condition must be fulfilled in order to check that the point being tested belong to the desired portion of the polygon and not somewhere else.

Thus, the code for this part outputs the following images:

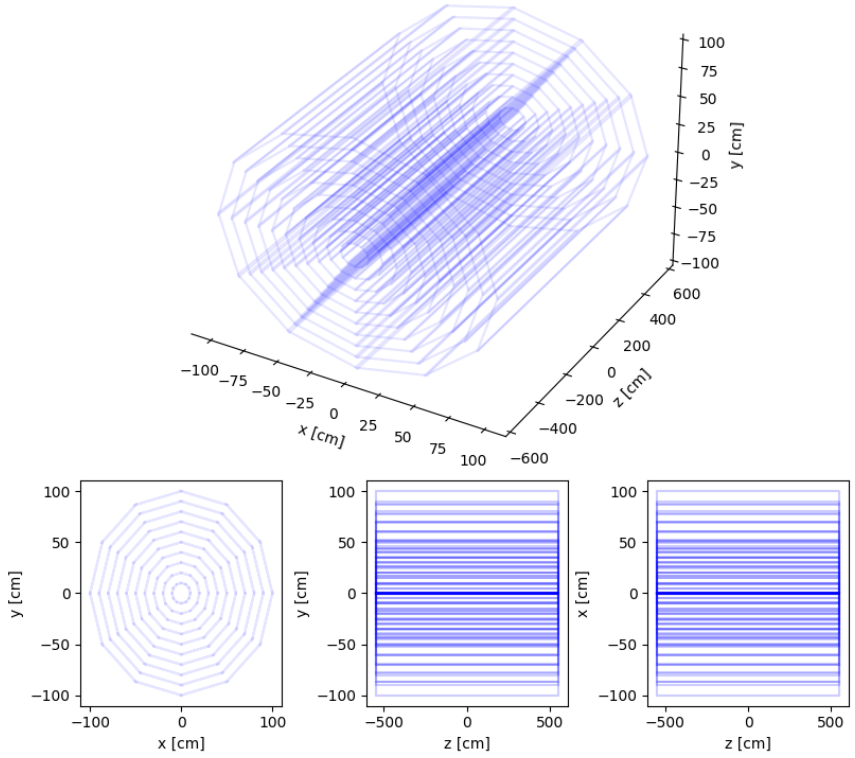

Figure 3.1: Tracker geometry in three dimensions as well two dimensions. 10 layers of 12 sided polygons with linearly increasing radius from 10 cm to 100 cm

3.2 Track Propagation

The equations of motion needed to model the tracks come from the relativistic Lorentz force because only charged particles in a homogeneous magnetic field are being considered. It is helpful to start with equation and consider a factor of proportionality that will depend on the desired units, for example reference [3] uses $k = 0.29979\ GeV/cT^{-1}m^{-1}$

$$\frac{d\vec{p}}{dt} = \frac{q}{m}\vec{p} \times \vec{B}, \tag{3.1}$$

where $\vec{p} = \gamma m\vec{v}$ is the momentum of the particle, q is an integer relating the elementary charge e.g. 1 for electron, -1 for muon. The solution of this differential equation is straightforward once $\vec{B} = Bz$ and $w = \frac{kqB}{\gamma m}$ are defined. So expanding 3.1 gives

$$\frac{dp_x}{dt} = \frac{q}{m}Bp_y \tag{3.2}$$

$$\frac{dp_y}{dt} = -\frac{q}{m}Bp_x \tag{3.3}$$

$$\frac{dp_z}{dt} = 0. \tag{3.4}$$

Differentiating once more with respect to time and combining the first two equations

results in a harmonic oscillator equation

$$\frac{d^2p_x}{dt^2} = -w^2p_x. \tag{3.5}$$

By proposing a solution of the type $p = A\cos(wt + \delta)$ two important parameters are defined. The transverse momentum p_T and the direction ϕ

$$\sqrt{p_x^2 + p_y^2} = A = p_T \tag{3.6}$$

$$\arctan(\frac{p_y}{p_x}) \equiv -\phi \tag{3.7}$$

So,

$$p_x = p_T\cos(wt - \phi) \tag{3.8}$$

$$p_y = -p_T\sin(wt - \phi) \tag{3.9}$$

$$p_z = p_{z0}. \tag{3.10}$$

Regarding p_z, it can be related to p_T quantity introducing the pseudorapidity η:

$$\eta = -\ln(\tan(\frac{\theta}{2})) \longrightarrow e^{-\eta} = \tan(\frac{\theta}{2}). \tag{3.11}$$

Now, using the trigonometric relations

$$\frac{e^{\eta} - e^{-\eta}}{e^{\eta} + e^{-\eta}} = \tanh(\eta), \tag{3.12}$$

$$\cos^2(x) - \sin^2(x) = \cos(2x) \tag{3.13}$$

$$\text{and } \cos(\theta) = \frac{p_z}{p}, \tag{3.14}$$

the next relation is obtained:

$$p_z = p_T\sinh(\eta). \tag{3.15}$$

So, the momentum spatial components are:

$$p_x = p_T\cos(wt - \phi) \tag{3.16}$$

$$p_y = -p_T\sin(wt - \phi) \tag{3.17}$$

$$p_z = p_T\sinh(\eta). \tag{3.18}$$

Recalling the definition of momentum $\vec{p} = \gamma m\vec{v}$ and carefully integrating once more

$$x(t) = x_0 + \frac{p_T}{\gamma kqB}[\sin(wt - \phi) + \sin(\psi_0)] \tag{3.19}$$

$$y(t) = y_0 + \frac{p_T}{\gamma kqB}[\cos(wt - \phi) - \cos(\psi_0)] \tag{3.20}$$

$$z(t) = z_0 + \frac{p_T}{\gamma m}\sinh(\eta)t. \tag{3.21}$$

Given the momentum in units of GeV/c, mass in GeV/c^2, B in T, positions in cm, and t in s, the set of equations will be:

$$x(t) = x_0 + 333.55 \cdot \frac{p_T}{\gamma q B}[\sin(wt - \phi) + \sin(\phi)] \text{ [cm]} \tag{3.22}$$

$$y(t) = y_0 + 333.55 \cdot \frac{p_T}{\gamma q B}[\cos(wt - \phi) - \cos(\phi)] \text{ [cm]} \tag{3.23}$$

$$z(t) = z_0 + 29.98 \cdot \frac{p_T}{\gamma m} \sinh(\eta) t \text{ [cm]}. \tag{3.24}$$

3.2.1 POCA parametrization

The found set of equations give the position of a charged particle inside an homogeneous magnetic field at a given time. However, the work it's not quite finished. The last step is to parametrize the track. The track are parametrized taking the Point of Closest Approach (POCA) as a reference. The POCA is the point of the track that is closest to the center of the detector $(0, 0, 0)$. This condition is achieved when the position and momentum are orthogonal in the transverse plane (XY) $\vec{x}(t_{POCA}) \cdot \vec{p}(t_{POCA}) = 0$. To generate the tracks, the starting point will be when $t_{POCA} = 0$.

The final parametrization involves six parameters including the POCA:

- d_{POCA}: distance in the transverse plane (XY) from the origin (0,0) to the track.
- d_z: distance from the origin to the track in the z direction.
- ϕ: direction in the transverse plane at the poca point.
- p_T: transverse momentum, (see equation 3.6).
- q Integer reflecting the charge of the particle, whether positive or negative will imply clockwise or anticlockwise movement.
- η: pseudorapity (see equation 3.11).

Applying the equations and conditions yields figure 3.2.

3.3 Particle Generator

For simplicity only muons and antimuons are being considered in this work. This sets the charge and mass to unchangeable values. -1 or 1 for muons or antimuons respectively, and its mass of 0.106 GeV [12].

However, the remaining track parameters cannot acquire all possible values and they must be constraint to a certain interval to be selected. To ensure a common behaviour for all the generated particles random uniform distributions were assigned:

Parameter	q	p_T	η	ϕ	d_{POCA}	d_z
Interval	[-1 1] (discrete choice)	[25, 100] GeV	[-2.4, 2.4]	[0,2π]	[0,0.5] cm	[-0.5, 0.5] cm

Table 3.1: Track's POCA parameters random uniform intervals

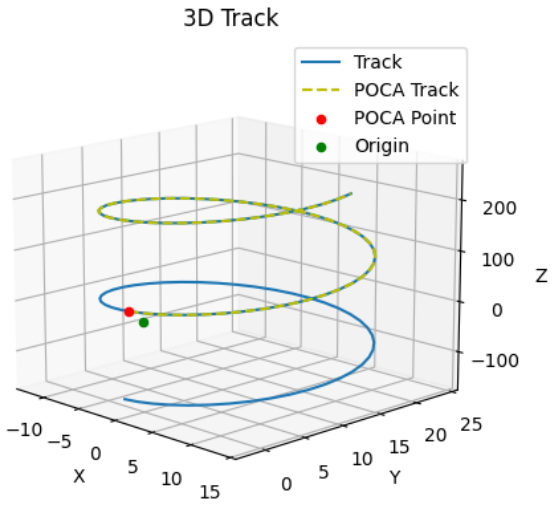

Figure 3.2: Example of parametrized track. The helix behaviour can be appreciated, as well as the POCA point. Whether using equation 3.22 as it is or with the POCA parameters returns the same curve

With this choice of values 3D momentum vector can be calculated with equations 3.16 and the particle's energy

$$E = \sqrt{p^2 + m^2} = \sqrt{p_T^2 + p_z^2 + m^2}. \tag{3.25}$$

Finally, the γ factor:

$$\gamma = \frac{E}{m}. \tag{3.26}$$

This way all the listed parameters as well as the remaining kinematic parameters are settled. In other words, tracks can be generated for muons or antimuons with specific values.

The final result of the simulation framework will be points of muon or antimuon tracks that belong to the intersection of the particles tracks and a series of polygonal layer, which resembles the CMS tracker. These tracks can be visually appreciated in Figures 3.3, 3.4 and 3.5.

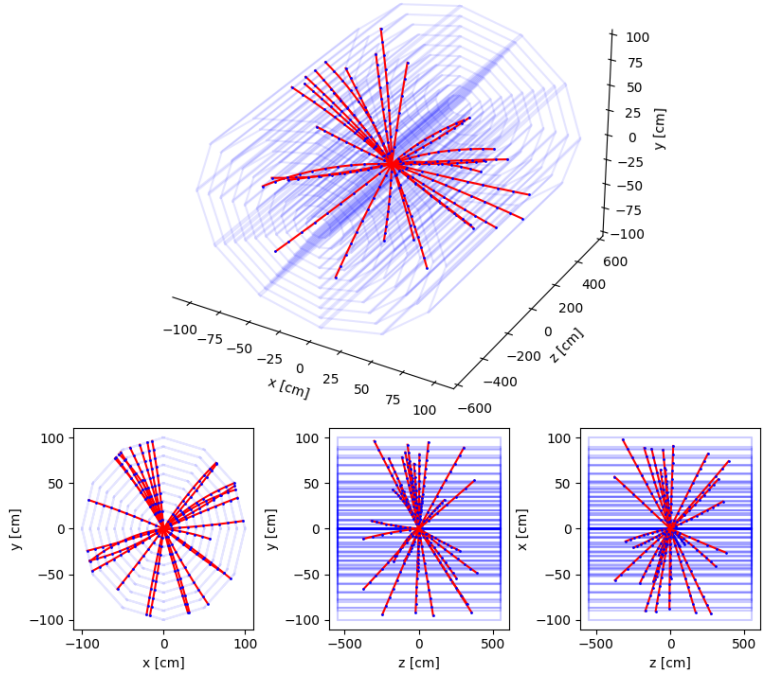

Figure 3.3: 30 tracks generated randomly with the stabilised constraints. A 3D view as well as all the 2D associated planes are appreciated

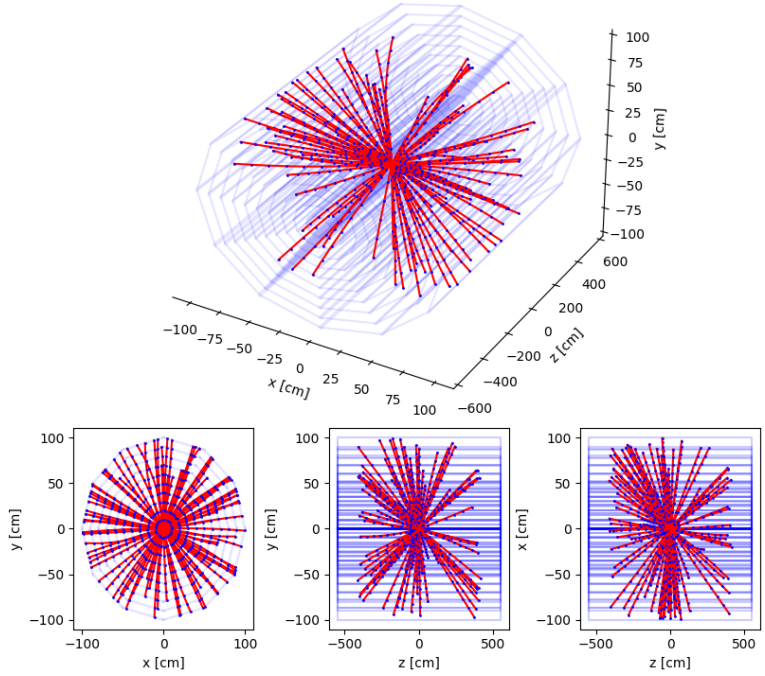

Figure 3.4: 100 tracks generated randomly with the stabilised constraints. A 3D view as well as all the 2D associated planes are appreciated

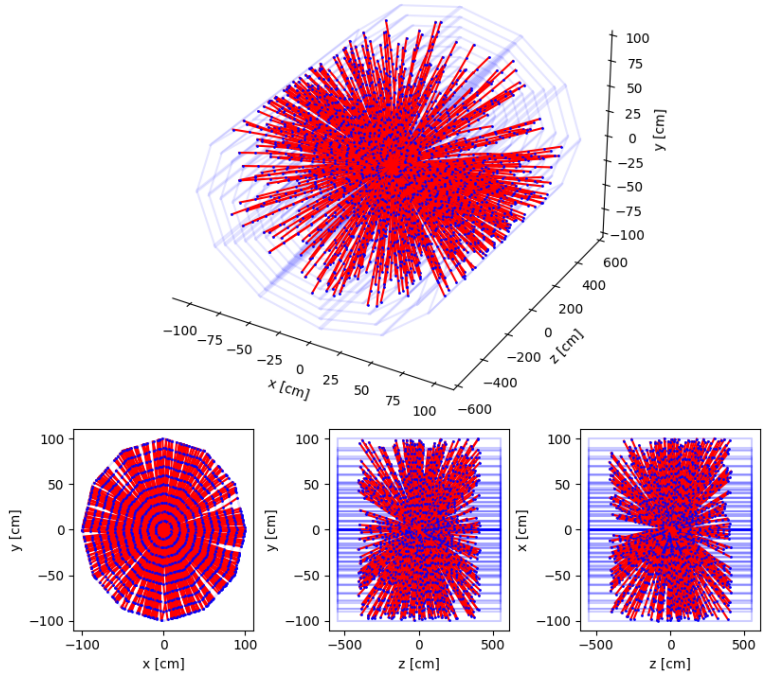

Figure 3.5: 500 tracks generated randomly with the stabilised constraints. A 3D view as well as all the 2D associated planes are appreciated

Chapter 4

Graph Neural Networks

This chapter describes the basic ideas required to understand graphs and graph neural networks (GNNs), then a short introduction to the library used to perform the analysis and finally the details of the model employed in this work.

4.1 Graph notions

4.1.1 Definition

A graph can be understood as an ordered pair of finite sets (V,E) where pairwise relations are modeled. V represents the set of vertices or nodes, and E the set of edges, meaning the relation or link there could be between nodes [13]. A graph is said to be weighted if the edges map to a set of real values [14].Figure 4.1 shows a simple representation of a weighted graphs indicating nodes and edges.

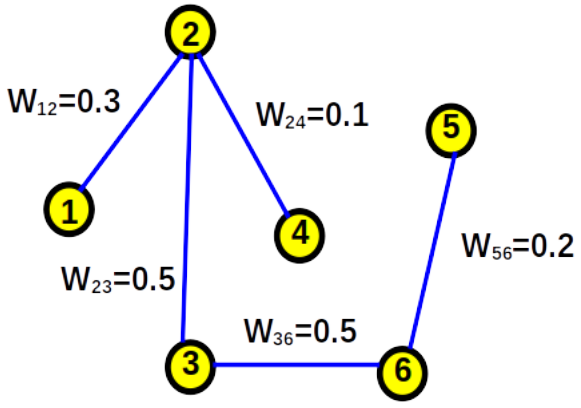

Figure 4.1: A simple weighted graph with 6 nodes and 5 edges.

The number of vertices in a graph defines its order $|G|$. This order is the same as the cardinality of the vertex set $|V|$. The number of edges in a graph is its size m, denoted as the cardinality of the edge set |E|. A graph with n vertices may also be referred to as an n-graph, denoted G_n [13].

A subgraph of a graph G = V, E is a graph H = W, F such that W is a subset of the vertices V $W \subset V$ and F is a subset of the edges E, $F \subset E$.

Usually, the edges of a graph have no direction, they are called *undirected*, meaning that none of the nodes is considered the source or the end (xy = yx). However, on the other hand there are *directed* graphs. Where the last equality does not hold, and the origin and the end of the connection has to be specified. This way xy will be a source vertex x and an end vertex y. An edge yx that joins y to x is called the inverse edge of xy.[13]

4.2 Graph Neural Networks

Deep learning is used to understand or "learn" complicated concepts, patterns; starting from simple ones in a multi layer manner. The most popular examples of such structure are artificial neural networks (ANN) [15].

Deep neural networks (DNN) are successful because they can take advantage of the data statistical properties of the input datasets. These networks can be applied to a large variety of different data formats. For example, when the input is a collection of images, the concept of convolution is used. Convolutions are geometrical filters that combine the information of a set of adjacent pixels in the image. Two effects come by using convolutions: one is to allow to extract shared local features, greatly reducing the number of parameters compared to other generic architectures; the other, it imposes initial insight about the data.

It can be said that geometric deep learning is an attempt to generalize deep neural models to structures like for instance graphs or manifolds instead of images. Thus, a graph neural network (GNN) is a deep learning model that will learn about a graph, rather than an image or speech. If a convolutional layer is present it will then be a convolutional graph neural network (GCN).

In particular, a GNN can learn and predict tasks associated with operations in a graph. For example, it can classify nodes or edges; predict the presence of edges between nodes; or even extract features of the whole graph [6]. This results very interesting in many problems of real life. Track reconstruction is one of these problems, since tracks can be visualized as connected points in a particular pattern. A graph model that takes as input a collection of hits, considered as nodes, and predicts which ones are connected to make a track can be an alternative method to the track reconstruction process. Interestingly enough, this process would not use any seeding procedure and would be therefore particularly suitable to reconstruct tracks not emerging from the center of the detector.

4.2.1 The Message Passing Mechanism

Conventional ANN are based on neurons or nodes, organized in layers with a given direction. The input information is provided to the neurons in the first layer. Each of the neurons multiplies the input by a given weight and adds up the total. A non-linear function such a sigmoid, RELU, tanh, etc is then applied on the previous quantity resulting in the neuron outcome, which is propagated to the next layer. The neurons in the last layer produce the output of the ANN. In the training process, input examples are provided, and

the output is contrasted with the ground truth, through a Loss function that measures their compatibility. This function is minimized with respect to the weights of the neurons by estimating the Loss function gradient through the back-propagation algorithm.

The message passing mechanism refers to the aggregation of information from a node's neighborhood [16]. In a GNN the input is a graph with a set of features associated to either nodes, edges or both and with a given connectivity that the GNN must not alter. Nonetheless, what the network can modify is the output features, called embeddings. The aggregation, i.e. the gathering of information, can be done by means of sums, mean or max calculation, so that the message summarizes the neighborhood details. The term node's neighborhood refers to all the connections a node has. Therefore, even if the edge and edge attributes are not explicitly mentioned, they are present.

For example, [17] uses the following equation to explain the message passing of node features:

$$x_i^{(k)} = \gamma_k(x_i^{k-1}, \bigotimes \phi^k(x_i^{k-1}, x_j^{k-1}, e_{ji})) \tag{4.1}$$

Where, x_i^{k-1} denotes node features of node-i, in layer $k-1$, e_{ji} denote optional features from node j to i. And $\bigotimes$ is a differentiable invariant function such as *sum*, *mean* or *max*, γ and ϕ are differentiable functions as MLP (Multi Layer Perceptrons).

4.2.2 GNN variants

Two different types of GNN's will be shortly described.

Graph Convolutional Networks

Graph Convolutional Networks (GCNs) perform in a similar way to convolutional neural networks CNN, in that learning about features is achieved by analyzing neighborhoods of nodes, by multiplication of the inputs with weights or filters. They are one of the basic variants and are applied to solve many problems as classification or forecasting for example.

A GCN will work following this order. First, each node is associated to a feature vector. Then, convolution is applied, information of a node's neighborhood is extracted, this can be done with weighted sum of feature vectors. Next is the weighted aggregation, the extracted information is scaled and accumulated. Then, these aggregated features go through an activation function, like ReLu, this process lets the GNN adapt to the desired task, before the output is transformed according to the desired task many layers can be stacked where the output of one layer is the input for the next.

$$Z = \sigma(A' \quad F \quad W + b) \tag{4.2}$$

This equation represent the output of a layer Z, dependence of the non linear activation function σ applied to the normalized adjacency matrix A', node features F and the weighted matrix W and a bias vector b.

Graph Attention Networks

Graph Attention Network (GAT/GAN) are variations of GCN that incorporates the called "attention mechanism". It allows nodes to focus on characteristics of its neighbors without much computational expense and assign different weights.

A GAT will work in this order. First the nodes are analyzed, associating them with a feature vector. Here the attention mechanism is introduced, where each node is allowed to focus on different neighborhoods when extracting information and assign different coefficients to neighbor nodes. Next is the weighted aggregation for all neighboring nodes. Then, is often used multiple attention heads to compute attentions core and accumulate aggregation. Similar to GCN , many layers can be stacked and the final output will depend on the desired task.

$$h_i^{l+1} = \sigma(\sum_{j \epsilon N} \frac{1}{C_{i,j}} W^l h_j^l) \tag{4.3}$$

in this case the output h_i of a particular node is the result of a non linear activation function σ applied on the sum one hop neighbor of the weight matrix W, the normalized vector C and the previous node h.

To summarize, graph neural networks are an extension of deep learning models but applied on different complex structures. In order to build a GNN special caution must be taken, first a task must be selected, then data must be processed to fit a specific graph structure and type, its nodes and edge must be clearly specified as well as their features. In this way a class of GNN can be selected and an architecture built.

4.3 PyTorch Geometric

PyTorch Geometric is a library built upon PyTorch to easily write and train Graph Neural Networks (GNN) for a wide range of applications related to structured data [17]. Its use for this work is justified not only because it has a connection to python, the language chosen to develop the simulation, but also because it is one of the most popular libraries to develop GNN-based applications.

PyTorch Geometric consists of various methods for deep learning on graphs and other irregular structures. This category of Machine Learning is usually known as geometric deep learning and it is supported by a variety of papers. PyTorch contains a large number of libraries covering several functionalities: easy to use mini batch loaders for operating on many small and single giant graphs, multi GPU support, torch compile support, DataPipe support, a large number of common benchmark datasets and helpful transforms, both for learning on arbitrary graphs as well as on 3D meshes or point clouds [17].

A single graph in PyG is described by an instance of a *Data* object, which could have the following attributes:

- *Data.x*. It is the node feature matrix, containing the list of features for every node of the graph. Its dimension is the number of nodes times the number of features.

- *Data.edge_index*. It is the graph connectivity in COO format: it represents all the edges by the source and end nodes. Its dimension is therefore 2 times the number of edges.
- *Data.edge_attr*. It is the edge feature matrix, it contains the list of features for every edge of the graph. Its dimension is the number of edges times the number of features.
- *Data.y*. It is a matrix that contains the ground truth associated to the nodes. Its dimension is 1 times the number of nodes.

This work will address the task of predicting proper links between nodes. The mandatory attributes of the graph object will then be, the x matrix of node features, the *edge index* to denote connections within the graph and the *edge features matrix*, where it has been relabeled as edge_label and only one feature is being considered, which assigns a value to each edge of the graph.

4.4 Graph generation from a collision event

Once the details of the library used have been described, a full description of the procedure to generate graphs is provided. This includes the simulation of the tracks and the graph generation.

4.4.1 Point Generation

Following the description of the simulation framework in chapter 3, the point generation consists in the simulation of a collision event composed by a number N of tracks that are detected by a tracker made of plane detectors. The set of the intersections of the tracks and the detectors is the set of the nodes. They have have several attributes, such as the position (x,y,z) and the track parameters, and three labels: the track label, to refer to which track the point belongs; the layer label, to refer in which layer was the point detected; and finally, the node label, a unique identification for each point, starting from 0. This information is stored in a .csv file with an structure as the one shown in figure 4.2.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	x	y	z	N_side	N_layer	t_label	phi	eta	q	pt	d0	z0	n_label
2	-9.0524	3.5894	-23.1666	5	1	T0	2.763	-1.584	1	33.5065	0.0268	-0.4308	0
3	-18.0827	7.2341	-45.9023	5	2	T0	2.763	-1.584	1	33.5065	0.0268	-0.4308	1
4	-27.1005	10.9094	-68.6381	5	3	T0	2.763	-1.584	1	33.5065	0.0268	-0.4308	2
5	-36.1059	14.6151	-91.3739	5	4	T0	2.763	-1.584	1	33.5065	0.0268	-0.4308	3
6	-45.0987	18.3513	-114.1096	5	5	T0	2.763	-1.584	1	33.5065	0.0268	-0.4308	4
7	-54.0789	22.1178	-136.8454	5	6	T0	2.763	-1.584	1	33.5065	0.0268	-0.4308	5

Figure 4.2: Structure of a csv file containing the generated points and their attributes.

4.4.2 Edge generation

In this step the connections between the nodes (points) are generated. Though every node in a graph can be connected to the other ones; in this case, certain conditions must be met in order to establish a link between two nodes.

First of all, the points in the same layer are by definition not connected. Indeed, only edges between nodes in consecutive layers are considered. This choice is justified by the fact that high-momentum particles in the detector will always have an in-out direction. From a technical point of view, this is achieved by grouping the points table using the layer label.

A second restriction is applied in the generation of the edges due to the track topology. Indeed, high momentum tracks do not bend very large angles. To illustrate this effect, it is possible to think that for a track pointing towards the upper part of the detector, it does not make sense to consider points in the lower part of the detector.

Then, a space region is extended to constraint even further the edge connection. This can be divided in two ways. Extending an angle for a given point with reference to the origin and looking if other points lay inside this new region, so this way impossible connections are excluded, as shown in figure 4.3. And lastly, after the second layer the progressing z path must be coherent, no jumps from negative z to positive z are allowed and vice versa.

As for the connection attribute value only two values are assigned as a "weight" label. "1" if they correspond to a known track, checking if they share the same track label. Otherwise "0" is applied to denote possible connections.

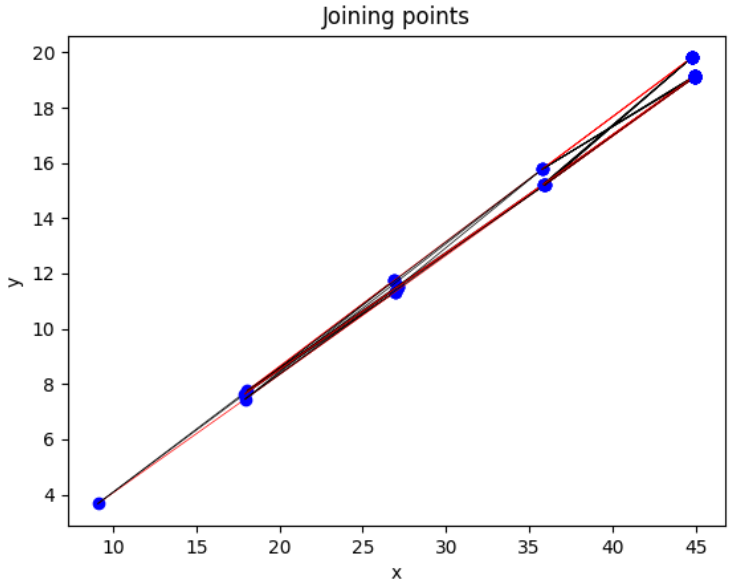

Figure 4.3: Example of how points are joined, red lines mean simulated tracks, black lines are possible connections

4.4.3 PyG Graph creation

The chosen object type is Hetero Data, which is a data object describing a heterogeneous graph, holding multiple node and/or edge types in disjoint storage objects. In the following lines, the key elements of the code are provided just to show the notation and power of the PyTorch Geometric framework.

Graph object creation

The first step in order to convert the list of nodes and edges into a PyTorch Graph consists in the creation of an empty object of the class *HeteroData*:

data=HeteroData ()

Node addition

Two types of nodes will be added to the heterogeneous graph: *source* and *target*. It should be noted that, there is no physical difference among them, but it will be useful to take advantage of the graph type. This can be easily expressed in the framework as:

data['source'].node_id = torch.tensor(node_s, dtype=torch.long)

data['target'].node_id = torch.tensor(node_t, dtype=torch.long)

where node_s and *node*_t are just vectors containing the the unique node label of each point. They are the same, just assigned to a different node type.

Then node attributes are added. For the final graph, only the point positions are considered. Though track attributes can also be added in this part, it was decided no to, because in a real scenario the only available information for points will be their position.

data['source'].x = Tensor(dataframe[['x','y','z']])

data['target'].x = Tensor(dataframe[['x','y','z']])

Edges addition

Edges are added establishing a relation between the two types of nodes and a name for this relation, in this case *'weight'*. These edges come from the file described in the edge generation section:

edge_index=torch.tensor([df_edge['source'] ,df_edge['target']], dtype=torch.long)

Then, the weight attribute is also added:

data['source','weight','target'].edge_label= weight_values

Direction

Once the edges have been defined in one direction, the connections are reversed, to add also the reciprocal connection in the HeteroGraph. PyG takes care of this operation by applying the corresponding transformation.

data = T.ToUndirected()(data)

Validation

Finally, PyG has a function to validate the graph object, checking the coherence of connections and tensor dimensions. If the edge index does not contain different nodes than the ones added, if the attributes correspond to the nodes length.

```
data.validate ( raise_on_error = True )
```

```
True

HeteroData(
  source={
    node_id=[4929],
    x=[4929, 3],
  },
  target={
    node_id=[4929],
    x=[4929, 3],
  },
  (source, weight, target)={
    edge_index=[2, 20517],
    edge_label=[20517],
  },
  (target, rev_weight, source)={ edge_index=[2, 20517] }
)
```

Figure 4.4: Output of running the object construction code, validation is correct and nodes and their connections are shown

4.5 Model Definition

So far the type of graph to be used and the task is defined. With those considerations in mind the GNN model architecture can be established. To recall, the objects in question are heterogeneous undirected graphs, with node and edge features; and the task is to make edge predictions.

The designed network is a Graph Convolutional Network (GCN). This architecture comprises of two primary sections: an encoder and a decoder. The encoder leverages convolutional layers of the SAGE (Sample and AggregatE) type to capture and encode the graph's structural and feature information. The decoder, on the other hand, uses a predictor to perform the desired task.

The encoder is crucial for effectively capturing the complex interconnections within the graph. Using SAGE convolutional layers[18], the encoder iteratively samples and aggregates information from neighboring nodes. This approach allows the network to learn robust representations of each node by considering both its features and the features of its neighbors. The SAGE method is particularly advantageous as it improves scalability and can handle graphs with varying neighborhood sizes, making it suitable for large and heterogeneous graphs. Two layers of convolutions are present with the first being followed by a non lineal activation function ReLu.

In the decoder section, the encoded embeddings are used to make predictions about the presence or absence of edges between nodes. This is achieved through linear transformations resembling a Multi-Layer Perceptron (MLP) predictor [19]. This way an easy implementation and computational efficiency is assured. The output will return a score of the edge attribute for two nodes between the two values used to build the graphs, 0 and 1.

It is thought that combining the strengths of SAGE convolutional layers for encoding and the linear transformation for decoding, this GCN network is well-suited for the link prediction task, and in turn it can reconstruct tracks without going through the whole process described previously in chapter 2. Figure 4.5 shows the PyG output after defining the model. Again, the encoder consists of convolutional layers and the decoder of two linear transformations that will return the task specific output.

```
Model(
  (encoder): GraphModule(
    (conv1): ModuleDict(
      (source__weight__target): SAGEConv((-1, -1), 32, aggr=mean)
      (target__rev_weight__source): SAGEConv((-1, -1), 32, aggr=mean)
    )
    (conv2): ModuleDict(
      (source__weight__target): SAGEConv((-1, -1), 32, aggr=mean)
      (target__rev_weight__source): SAGEConv((-1, -1), 32, aggr=mean)
    )
  )
  (decoder): EdgeDecoder(
    (lin1): Linear(in_features=64, out_features=32, bias=True)
    (lin2): Linear(in_features=32, out_features=1, bias=True)
  )
)
```

Figure 4.5: Output of the code defining the model's architecture. The encoder and decoder can be noted, just like the convolutional SAGE layers

4.6 Model Training

After generating the data and adapting it for the case study purpose, and having defined the architecture for a graph convolutional network (GCN), the next stage is to train the model. But before doing so, several parameters must be fixed. These include, the optimizer, the learning rate and the number of epochs the model will be trained. Also, the sets for training and validation have to be defined. Here hyperparameter tuning must be performed. That is trying different configurations of the learning rate, epochs, etc., with the goal to ensure a proper learning process and avoid overfitting or underfitting.

4.6.1 Loss function

The loss functions are a way to measure the performance of the GCN as it learns through iterations. There are many option to choose from the torch library.

The loss functions used in this work is the MSE loss. It creates a criterion that measures the Mean Squared Error between each element in the input and target. [20]

4.6.2 Optimizer

Similar to the loss function there are many options for the optimizer, Adam, Stochastic Gradient Descent or its variants are available to use. In this work, the used optimizer is Adam with a learning rate of 0.005.

4.6.3 Training and validation sets

Contrary to what is common to perform in model training, the training and validation sets are not split from one data graph. Instead, a whole graph of 1000 tracks is used to train the model and others with a reduced number of tracks than the training graph are used for validation.

4.7 Model Evaluation

4.7.1 Post prediction filter

The defined GNN has the task to predict weighted connections between nodes. After training the model it will return a predicted edge with a value as attribute. Nonetheless, this value is not quite the same as the one from input, the values 1 or 0 defined in the graph building section, but in turn it returns a value in the continuous interval [0,1]. And also, there is the possibility that a node has many edges, so a selection is needed so the track can be built. Therefore, to properly test the model with generated graphs as explained before, a post training process must be implemented.

For the multiple connection case each node is analyzed. If it has multiple connections the one with the highest prediction value is selected. However, as the graph is undirected and heterogeneous, two searches are performed one looking from the "source" type node and the other the other way around, looking from the "target" type node. And after each pass the non selected edges are discarded.

To address the self imposed classification problem a threshold is set. This way it reduces to a simple comparison of the predicted value with the threshold value and if it is greater then change the prediction to 1 or if it is less change the prediction to 0. So the usual graph structure feature is recovered and comparison can be made. The value of the threshold is a matter of tuning until proper behavior is observed.

4.7.2 Evaluation metrics

Now that that the output data has been adapted to correctly make comparison the next task is to evaluate its performance against not known graphs for the model. The model will receive as an input a graph, that is an *HeteroData* object with two types of nodes, 'source' and 'target', with each node type having their spatial positions (x,y,z) as attributes. And it will predict edges between nodes with a certain value from 0 to 1, this has been called the weight of the connection.

The accuracy is the quantity used to evaluate the model performance. Though, it could be as simple as comparing the predicted edges with the corresponding ground truth, this strategy is poor since this kind of graph is very unbalanced. This means that there is a larger amount of edges not connecting points than connecting points. For this reason, two accuracies are studied in parallel, the accuracy to predict active and negative links. The definition is the following:

$$\begin{aligned} \text{Total Accuracy} &= \frac{\text{Correct Edge Predictions}}{\text{Total Edges}} \\ \text{Connected Accuracy} &= \frac{\text{Correct Connected Prediction}}{\text{Connected Edges}} \\ \text{Disconnected Accuracy} &= \frac{\text{Correct Disconnected Prediction}}{\text{Disconnected Edges}} \end{aligned}$$

Equation 4.4 expresses the idea behind accuracy. A number of selected events are counted and then divided by the total giving the proportion regarding the entire data.

Even if the edge counting is more interesting from the point of view of the network evaluation, from a physical point of view, it is more interesting to define an accuracy associated to the tracks. To to this, tracks from the testing graph are compared to the predicted tracks by counting the correct edge predictions and dividing by the sum of correct predictions, missing predictions and incorrect predictions. Equation 4.4 describes this track accuracy.

$$\text{Track Accuracy} = \frac{\text{Good Edge}}{\text{Good Edge} + \text{Fake Edge} + \text{Missing Edge}} \tag{4.4}$$

These metrics will be used in the results chapter in order to describe the performance of the graph model.

Chapter 5

Results

This chapter describes the results obtained by applying the GNN described before. Two different cases have been considered: results with an opening window (see previous section) of 20 degrees, and results with an opening window of 45 degrees. The use of a narrow opening window is justified by the fact that high momentum tracks are almost straight and they do not bend significantly in the tracker volume. For this reason, the baseline result is the one obtained with an opening window of 20 degrees. On the other hand, it was interesting how much degradation was observed in the results when opening the window, increasing the complexity of the problem, due to the higher number of edges involved in the graphs.

In both cases, a total of 9 graphs have been produced with a different number of tracks: 10, 20, 30, 50, 100, 200, 300, 500 and 1000. The higher track multiplicity case (1000 tracks) has been used as the training sample, and the others have been utilized to benchmark the performance of the method.

5.1 Results using an opening window of 20 degrees

5.1.1 Model training

As mentioned before, the training has been performed using the graph with 1000 tracks. The learning rate has been fixed to 0.005. Variations of this hyper-parameter were done although no large improvements were observed. The model was trained for a total of 3000 epochs. The total running time was of the order of 10 minutes running on a conventional laptop.

Figure 5.1 shows the evolution of the loss curve for the model along the 3000 training epochs. Both the train and validation curves are displayed. In this case the validation curve is based on a graph with 100 tracks. The two curves rapidly diminish the value of the loss function, although this happens much faster for the training sample than for the validation. The curves exhibit curious, quasi-periodic fluctuations, whose origin has not been completely understood.

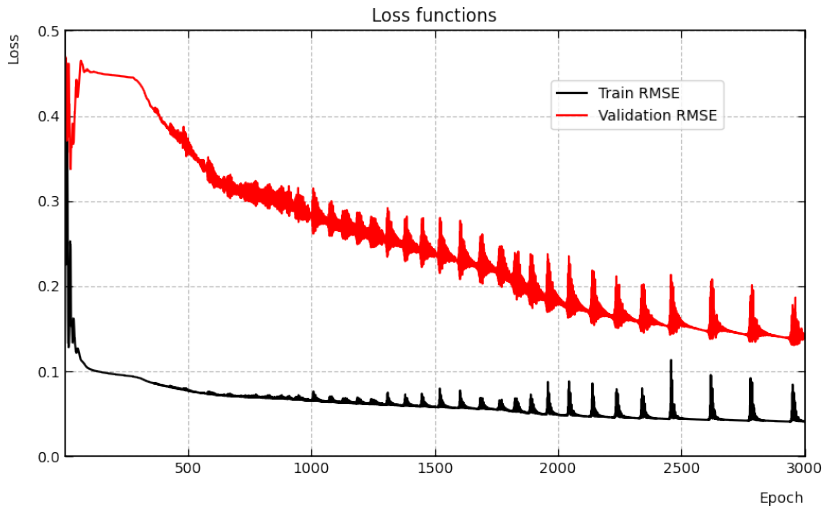

Figure 5.1: Evolution of the loss function (square root of MSE) for the training dataset (graph of 1000 tracks) and the validation dataset (graph of 100 tracks), and an opening angle of 20 degrees.

5.1.2 Accuracy studies

Table 5.1 shows the accuracy obtained for the different size graphs and a training based on the 1000 tracks graph and an opening angle of 20 degrees. The results show the overall accuracy, the connected accuracy, the disconnected accuracy and the track accuracy as stated in chapter 4.

N° Tracks	10	20	30	50	100	200	300	500
Overall Accuracy	0.90	0.98	0.92	0.94	0.93	0.94	0.94	0.95
Connected Accuracy	0.90	0.98	0.92	0.94	0.91	0.90	0.85	0.81
Disconnected Accuracy	1.0	1.0	0.91	0.96	0.97	0.98	0.98	0.98
Track Accuracy	0.90	1.0	0.93	0.96	0.96	0.96	0.86	0.80

Table 5.1: Overall, connected, disconnected and track accuracies are shown compared to the number of tracks.

Results of the table 5.1 are visually represented in figure 5.2. As the number of tracks increases the connected accuracy starts to decrease. However, all accuracies are over the 0.8 mark, meaning that at least 80% of the time the model will make a correct prediction for graph with a reduced number of tracks compared to the graph training set, 1000 in this case. It has to be noted that for the smallest sets there is a fluctuation, the reason behind this might be that as it contains a small number of tracks, each individual connections exerts a disproportional large influence on the accuracies measures.

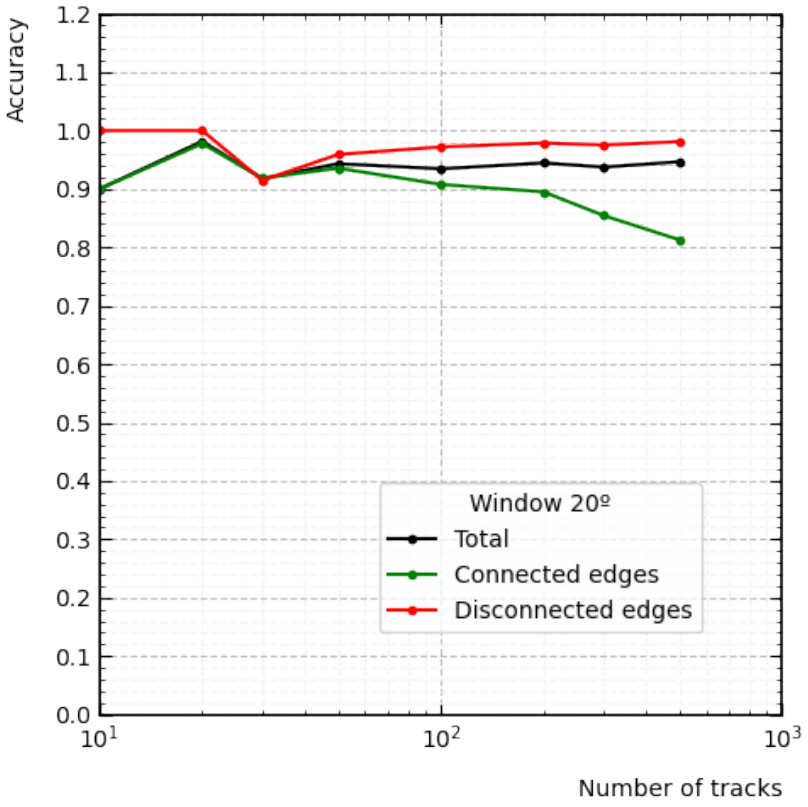

Figure 5.2: Plot of the overall, connected and disconnected accuracies shown in table 5.1

Nevertheless, it is best if the results other than the selected metrics are displayed in a plot. This way not only the accuracy values will be confirmed but also the tracks would be noted, which are the subject of study of the work. Figure 5.3 show the two end stages for a 100 and 500 tracks graph, up and down respectively. The input stage, just a cloud of points before turning into a graph and the output stage, after passing through the GNN model and post processing it, building tracks.

Figure 5.3 is remarkable. Following the post prediction filter process practically all tracks look properly connected. Though, there is at least one notable missing link in those figures it has to be remembered that their that their respective accuracies are over 0.9 and 0.8. It also has to be noted that although as the number of tracks increases the image looks blurred, no strange patterns are distinguishable.

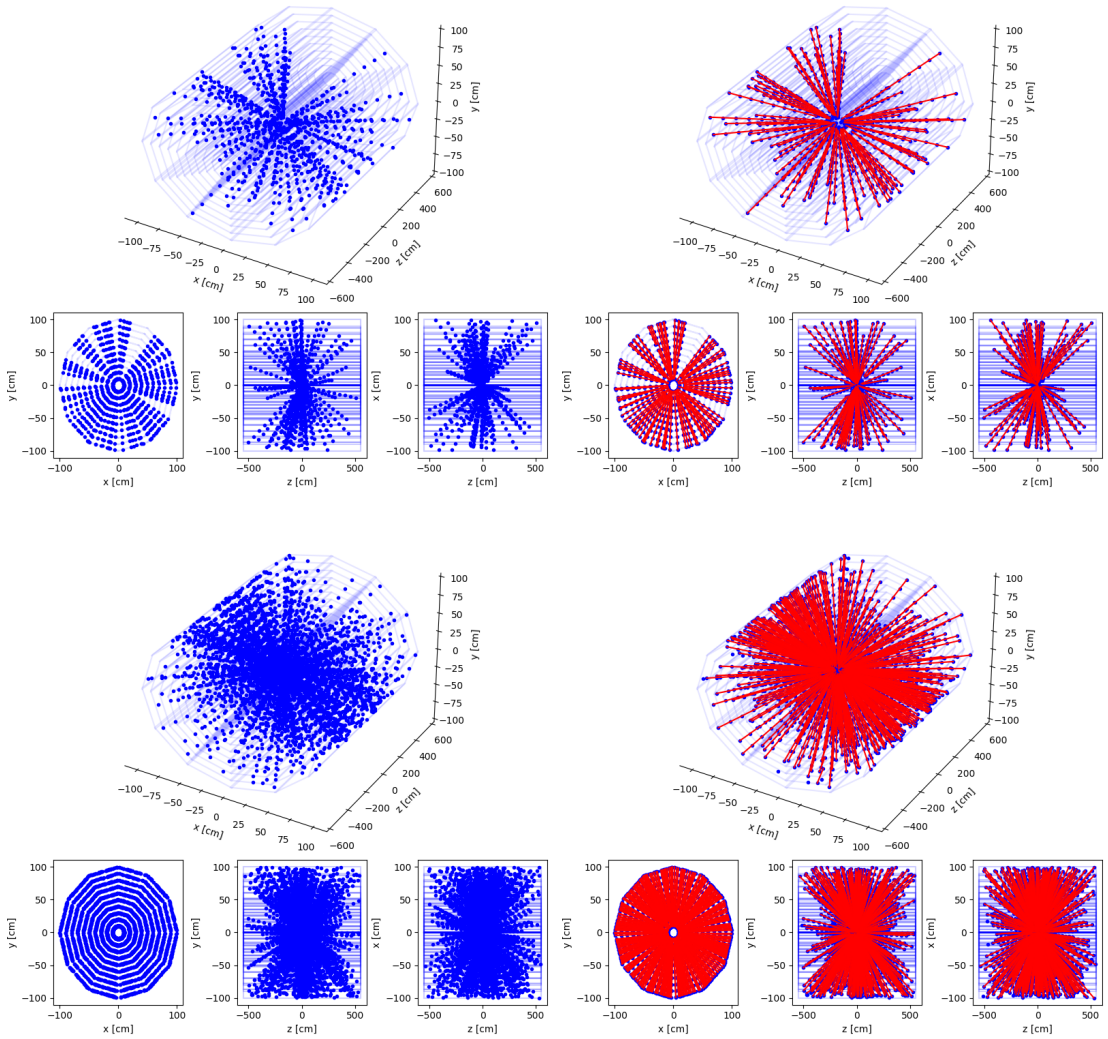

Figure 5.3: Left: cloud of hits for a case with 100 tracks (top) and 500 tracks (bottom). Right: Track reconstruction provided by the GNN for the two cases on the left.

5.2 Results obtained using an opening window of 45 degrees

5.2.1 Model training

The training conditions in this case are the same as for the 45 degrees case. Figure 5.4 shows the loss curve for the model along the 3000 training epochs. Both the train and validation curves are displayed.

A quick comparison between this curve and the previous case, indicates that the training is not as effective for the wide opening window. Indeed, there is a large bias between the validation and the training loss. This may indicate that a longer training, or a graph with more than 1000 tracks, could be needed in order to achieve better results.

Also, the observed difference between training and validation loss, which is large and

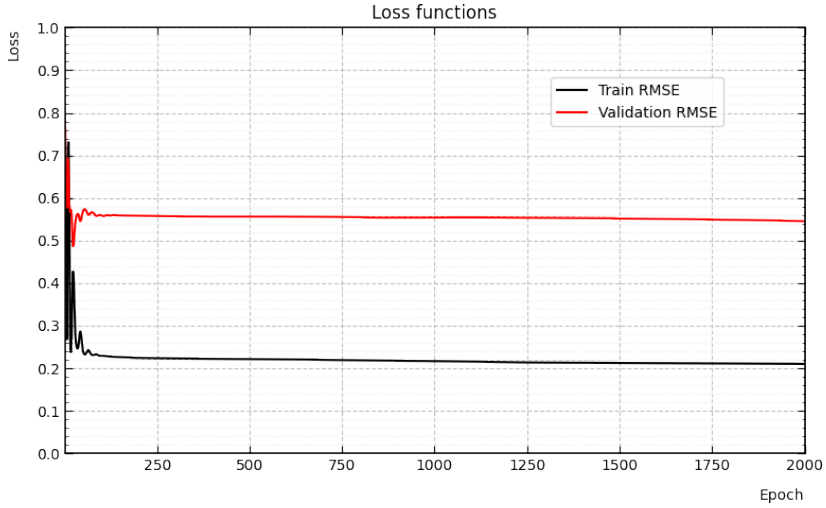

Figure 5.4: Evolution of the loss function (square root of MSE) for the training dataset (graph of 1000 tracks) and the validation dataset (graph of 100 tracks), and an opening angle of 45 degrees.

maintained for over 2000 epochs, leads to the possibility that we may have overfitting. One possible explanation is that the model is learning too well the training graph structure, therefore not being able to discern track characteristics. This would also explain why the validation curve does not decrease.

5.2.2 Accuracy

Table 5.2 show the results of testing the model, trained with a graph made of 1000 tracks and an opening angle of 45 degrees. The results show the overall accuracy, the connected accuracy, the disconnected accuracy and the track accuracy as stated in Chapter 4

N° Tracks	10	20	30	50	100	200	300	500
Overall	0.49	0.67	0.75	0.77	0.80	0.89	0.90	0.93
Connected	0.48	0.56	0.61	0.55	0.50	0.57	0.50	0.41
Disconnected	1.0	1.0	0.99	0.97	0.98	0.98	0.98	0.98
Track Accuracy	0.50	0.60	0.63	0.58	0.48	0.50	0.43	0.24

Table 5.2: Overall, connected, disconnected and track accuracies are shown compared to the number of tracks.

Results of table 5.2 are visually displayed in Figure 5.5. Even though the disconnected accuracy gets closer to the maximum value of 1; the connected accuracy, the interest of this work, barely surpasses the 0.5 mark. This biased behavior may indicate that by broadening the angle window the model is learning more about the structure of the non connected edges than the connected. And it is reasonable, because by using a wider angle window the number of non connected edges is larger than the connected ones. This size discrepancy can account for the increase of the total accuracy but the almost constant connected and non connected accuracies. Thus, resulting that the model trained with an

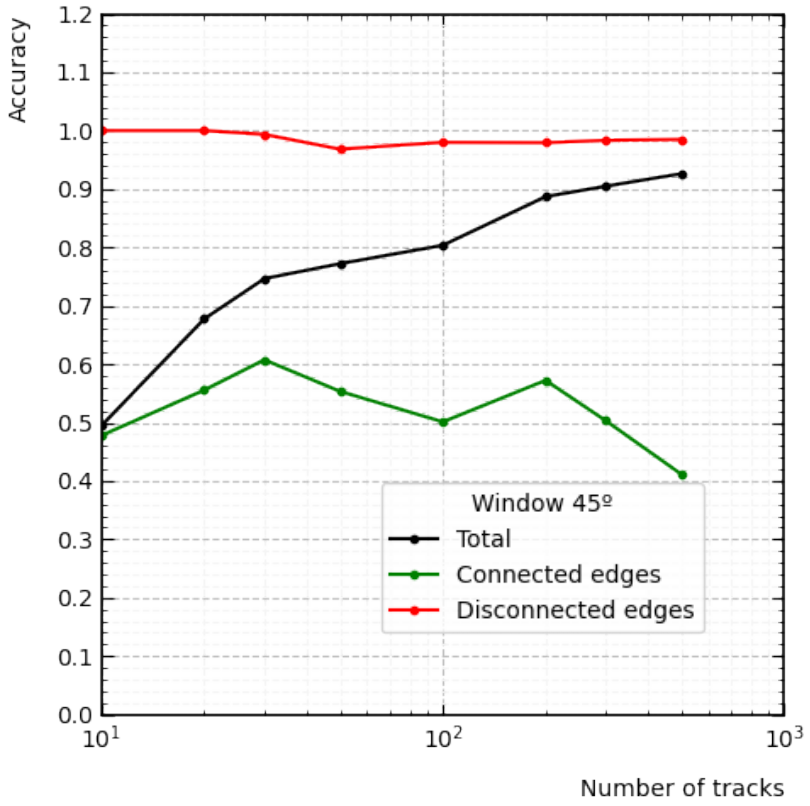

Figure 5.5: Overall, connected, disconnected and track accuracies are shown compared to the number of tracks

unbalanced set, and it is able to make high accurate predictions of only one of the edge types in study.

5.3 Track accuracy comparison

Finally, both angle windows are compared in terms of track accuracy. Figure 5.6 shows the track accuracies of tables 5.1 and 5.2 as a function of the number of tracks in the graph. That is, how well the model behaves in terms of recognizing the original tracks by changing the opening window to build the graph.

The figure clearly indicates that results are very good for the 20 degrees opening window, and certainly better than for the 45 degrees case. For both cases, the efficiency degrades with the number of tracks, which is expected, but it degrades quicker for the 45 degrees case. This plot, together with the loss evolution, are most likely indicating that more tracks are needed for training in the case of 45 degrees. It should be stressed in any case, that an angle of 20 degrees is considered realistic for high momentum tracks and therefore results are encouraging.

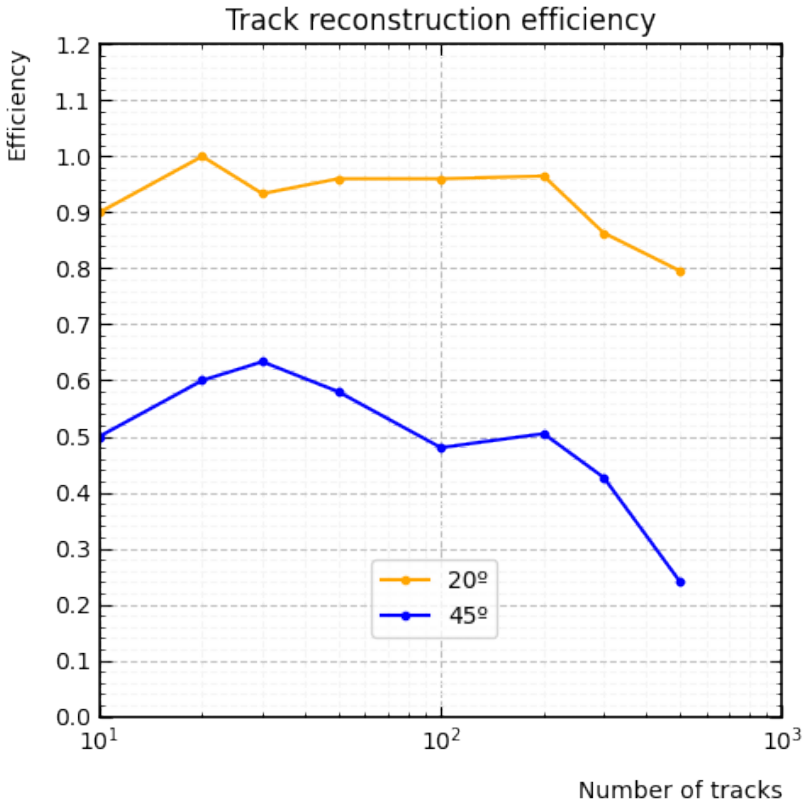

Figure 5.6: Track accuracy comparison between the two opening angle choices with reference to the number of tracks a graph contains.

Chapter 6

Conclusions

A model based on a Graph Neural Network has been implemented in order to identify tracks in a simplified tracker. The architecture of this model was explained in detail in chapter 4, although it can be summarized as model that returns edge predictions based on graphs that contain two types of nodes, with each having the spatial position of hits as arguments.

The performance of the model has been studied under several conditions: the opening window angle and the number of tracks to be reconstructed.

Concerning the opening window, two angles were studied, 20 and 45 degrees. The study clearly shows that a smaller opening window yields better results. The results for the 20 degree window show not only convergent loss curves but also high accuracies, even for the 500 tracks scenario ($> 80\%$), compared to the 45 degree window case. It has to be mentioned as well that the loss curves for the 20 degree window 5.1 exhibit oscillatory fluctuations. Whether this comes from noise in the training data or an inappropriate learning rate was not determined.

The fact that a GNN model can reconstruct tracks so successfully without the need of the iterative process of the Kalman filter shows there are alternative ways for track reconstruction that need to be studied. Also, because this process does not require seeding, could be a promising way to study displaced vertex coming from particles that do not decay near the interaction point.

The approach taken in this work relies on simplicity and a controlled test environment. Further work could consider adding energy loss and interactions in the track propagation section, different shapes or number of layer for the barrel detectors employer and different choices for track parameters $p_T, \eta, etc.$ Also, it would be interesting to study the case of increasing track number, larger than the 1000 tracks used.

Bibliography

[1]CERN, *Cms*, Accessed on june, 2024.

[2]CERN, *The large hadron collider*, Accessed on june, 2024.

[3]R. Frühwirth and A. Strandlie, *Pattern recognition, tracking and vertex reconstruction in particle detectors* (2021).

[4]D. Curtin and R. Sundrum, "Hidden worlds of fundamental particles", Physics Today **70**, 46–52 (2017).

[5]CERN, *High-luminosity lhc*, Accessed on june, 2024.

[6]S. Farrell, P. Calafiura, M. Mudigonda, Prabhat, D. Anderson, J.-R. Vlimant, S. Zheng, J. Bendavid, M. Spiropulu, G. Cerati, L. Gray, J. Kowalkowski, P. Spentzouris, and A. Tsaris, *Novel deep learning methods for track reconstruction*, 2018.

[7]CERN, *Cms detector design*, Accessed on june, 2024.

[8]T. C. Collaboration, "Description and performance of track and primary-vertex reconstruction with the cms tracker", Journal of Instrumentation **9**, P10009 (2014).

[9]G. Van Rossum and F. L. Drake, *Python 3 reference manual* (CreateSpace, ScottsValley,CA, 2009).

[10]C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy", Nature **585**, 357–362 (2020).

[11]J. D. Hunter, "Matplotlib: a 2d graphics environment", Computing in Science & Engineering **9**, 90–95 (2007).

[12]P. D. G. Live, μ, Accessed on june, 2024.

[13]S. Georgousis, M. P. Kenning, and X. Xie, "Graph deep learning: state of the art and challenges", IEEE Access **9**, 22106–22140 (2021).

[14]M. Fey and J. E. Lenssen, *Fast graph representation learning with pytorch geometric*, 2019.

[15]M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data", IEEE Signal Processing Magazine **34**, 18–42 (2017).

[16]B. Khemani, S. Patil, K. Kotecha, and S. Tanwar, "A review of graph neural networks: concepts, architectures, techniques, challenges, datasets, applications, and future directions", Journal of Big Data **11**, 10.1186/s40537-023-00876-4 (2024).

[17]P. Team, *Pyg documentation*, Accessed on june, 2024.

[18]W. L. Hamilton, R. Ying, and J. Leskovec, *Inductive representation learning on large graphs*, 2018.

[19]S. Sheikh, *Link prediction in gnns made easy- deep graph library (dgl)*, Accessed on july, 2024.
[20]L. Foundation, *Pytorch documentation*, Accessed on july, 2024.