Karlsruhe Institute of Technology

ETP-KA/2022-14

A Fast AI-Based Track Reconstruction on FPGA for the PANDA Experiment

Greta Heine

Master Thesis

at the Department of Physics
Institute for Data Processing and Electronics (IPE)
and
Institute of Experimental Particle Physics (ETP)

Advisor: Prof. Dr. Torben Ferber
Co-Advisor: Dr. Michele Caselle

15th March 2022 – 15th September 2022

Karlsruher Institut für Technologie
Fakultät für Physik
D-76128 Karlsruhe

Karlsruher Institut für Technologie

ETP-KA/2022-14

Eine Schnelle AI-Basierte Track-Rekonstruktion auf FPGA für das PANDA-Experiment

Greta Heine

Masterthesis

an der Fakultät für Physik
Institut für Prozessdatenverarbeitung und Elektronik (IPE)
und
Institut für Experimentelle Teilchenphysik (ETP)

Referent: Prof. Dr. Torben Ferber
Korreferent: Dr. Michele Caselle

15. März 2022 – 15. September 2022

Karlsruher Institut für Technologie
Fakultät für Physik
D-76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 15th September 2022

. . . . . .

(Greta Heine)

Abstract

Efficient reconstruction of charged particle trajectories is a crucial yet very difficult step in the analysis pipeline of high energy physics (HEP) experiments. Recent work has shown that graph neural networks (GNNs) are well suited for the pattern recognition task of track finding, where tracking detector hits can be naturally represented as nodes and particle track segments as edges. The interaction network (IN) GNN architecture provides computationally efficient edge classification of the high-dimensional and sparse tracker data, which is especially crucial for implementation in constrained computing environments such as field programmable gate arrays (FPGAs). This work describes the overall workflow for implementing and systematically analyzing an IN-based classification of track segments on FPGAs for the anti-**P**roton **An**nihilation at **DA**rmstadt (PANDA) forward tracking system (FTS). This workflow includes data preprocessing, graph building, GNN-based edge classification and a series of FPGA implementation design studies concerning latency, resource utilization, and classification quality using the high level synthesis for machine learning (`hls4ml`) compiler. The presented final implementation of the GNN-based track segment classifier on a Xilinx Zynq® UltraScale+™MPSoC FPGA provides an overall inference latency of about 0.99 µs using about 34 % of available digital signal processors (DSPs) and 85 % of available lookup tables (LUTs). This work enables the acceleration of charged particle tracking on heterogeneous computational resources toward real-time track reconstruction for the PANDA experiment. The discussed methods and studies could be easily adapted and used in other HEP experiments for accelerated charged particle tracking.

Zusammenfassung

Die effiziente Rekonstruktion von Trajektorien geladener Teilchen ist ein entscheidender, aber sehr schwieriger Schritt in der Analyse-Pipeline von Hochenergiephysik Experimenten (HEP). Studien der vergangenen Jahre haben gezeigt, dass sich graphbasierte neuronale Netze (GNNs) gut für die Mustererkennungsaufgabe der Spurensuche eignen, bei welcher die Treffer in den Spurendetektoren als Knoten und die Segmente der Teilchenspuren als Kanten dargestellt werden können. Die IN-Architektur bietet eine effiziente Kantenklassifizierung der hochdimensionalen Tracker-Daten bei geringer Trefferdichte, was insbesondere für die Implementierung in eingeschränkten Rechenumgebungen wie bei FPGAs entscheidend ist. Diese Arbeit beschreibt den gesamten Arbeitsablauf für die Implementierung und systematische Analyse einer IN-basierten Klassifizierung von Spur-Segmenten auf FPGAs für PANDA Vorwärtsspurdetektor Daten. Dieser Arbeitsablauf umfasst die Datenvorverarbeitung, die Graphenerstellung, die GNN-basierte Kantenklassifizierung und eine Reihe von Design Studien zur FPGA-Implementierung im Hinblick auf Latenz und Durchsatz, Ressourcennutzung, sowie Klassifizierungsqualität unter Verwendung des `hls4ml`-Compilers. Die vorgestellte finale Implementierung des GNN-basierten Spur-Segment Klassifizierungs-Algorithmus auf einem Xilinx Zynq® UltraScale+™MPSoC FPGA bietet eine Gesamtinferenzlatenz von etwa 0.99 µs unter Verwendung von etwa 34 % der verfügbaren DSPs und 85 % der verfügbaren LUTs. Diese Arbeit ermöglicht die Beschleunigung der Spur-Rekonstruktion geladener Teilchen auf heterogenen Rechenressourcen hin zu einer Echtzeit-Spur-Rekonstruktion für das PANDA-Experiment. Die diskutierten Methoden und Studien könnten leicht angepasst und für andere HEP-Experimente zur beschleunigten Rekonstruktion geladener Teilchen adaptiert werden.

Disclaimer

The GNN-based tracking and FPGA implementation studies presented in this thesis were proposed to me by my supervisor Dr. Michele Caselle (KIT, IPE), inspired by the work of Dr. Waleed Esmail (GSI). The setup of the `PandaRoot` framework and FTS data generation was done with the help of Dr. Tobias Stockmanns (Forschungszentrum Jülich). The event display plots Fig. 6.3, Fig. 6.7, Fig. 7.5, and Fig. 7.6 are inspired and based on the plots of Dr. Waleed Esmail. The preprocessing of the data and the graph building algorithm presented in Section 6.2 are based on the code of DeZoort et al. [1] and Dr. Waleed Esmail, adapted to the PANDA FTS data. The IN architecture used in this work, described in Section 7.1 and shown in Fig. 7.1, was introduced by DeZoort et al. [1]. The tracklet finding algorithm is based on code by Dr. Waleed Esmail. The `hls4ml` compiler extension for the GNN conversion was provided by Elabd et al. [2]. All of the analyses reported in this thesis are performed by me, all results are formulated by me, and all figures are created by me unless otherwise noted.

Contents

1. Introduction

Physics forms the basis for a large part of our current standard of living. Science helps to ensure technological advancement, prosperity, and environmental protection. Everything is based on physics and fundamental research, from radio, television, and the Internet, over GPS, lasers, and smartphones, to Mars probes and climate models. Therefore, fundamental mechanisms and processes must be studied and understood to enable innovations, progress, and the increase of the overall knowledge of humankind.
The last 150 years have seen immense breakthroughs in understanding the particle nature of the world: from the discovery of the electron in 1897 through the formulation of the Standard Model to the discovery of the Higgs particle in 2012. However, many particles and mechanisms that shape our world, such as dark matter, neutrino mass, or matter-antimatter asymmetry, to name but a few, are still largely unknown. Physicists are therefore striving to gain new insights into the various fields of particle physics, using numerous technical methods and conceptual approaches.
One key method of exploring the structure of matter and finding out "what holds the world together at its core" is the construction of large particle accelerators in which particles collide at high energies, commonly subsumed under the term HEP. Particles and their decay products are created at the collision points and measured by surrounding high-resolution detectors. The correct and accurate reconstruction of the kinematics of charged particles produced in collision events is a crucial element for precision measurements, detection of novel particles, and observation of new physical phenomena in detectors. The reconstruction of trajectories of charged particles, also called tracks, is based on so-called hits that particles cause in the detector units as they pass through the detector providing position measurements along the particle trajectories. Accurate track reconstruction is one of the first and, therefore, essential steps in the analysis pipeline and the basis for many other reconstruction tasks, such as vertex reconstruction [3, 4], particle reconstruction [5], and jet flavor tagging [6, 7].
To study the open questions in particle physics and to enable the creation of new and heavier particles, ever more powerful experiments with ever higher collision energies, event rates, and detector granularitiesp are being built. This is leading to a dramatic increase in the amount of data generated in the form of detector hits, requiring ever more complex and rapid data acquisition systems (DAQs). However, state-of-the-art track reconstruction algorithms [4, 8] based on the combinatorial Kalman filter scale computationally worse than quadratically in the number of detector hits [9–13]. The corresponding necessary computational time required on central processing units (CPUs) for track reconstruction is expected to increase even faster with an increasing number of hits than the evolution of computational resources [14, 15]. Therefore, with ever-increasing resolutions and data volumes, alternative pattern recognition algorithms with better computational scaling, lower latency, and high data throughput must be developed to fully exploit the potential

of future HEP experiments.
In recent years, analytical methods using machine learning (ML) techniques have attracted great interest in the HEP community, leading to a wide range of applications [16], such as the observation of jet substructures [17], the search for exotic particles [18] or hadronic B-tagging [19]. Also, for tracking, various ML architectures have been tested and applied, such as image-based convolutional neural networks (CNNs) and recurrent neural networks (RNNs) [20]. However, a major problem with these types of methods is the high dimensionality and sparsity of the underlying data structure. Solutions to this problem can be provided by methods based on geometric deep learning (GDL). GDL is a growing subfield of ML that attempts to generalize deep learning (DL) models to non-Euclidean domains such as graphs, sets, and manifolds [21]. A central method within GDL are GNNs [22], which operate on graphical representations of data: Sets of elements and their pairwise relationships. Particle tracking data can naturally be represented as graphs in terms of nodes (hits) and edges connecting pairs of hits (track segments). Therefore, the GNN must correctly classify edges belonging to track segments by considering the relationships between pairs of hits. Recent efforts following the tracking machine learning challenge (TrackML) in 2018 [23] have shown that GNNs are well suited for the task of particle tracking [1, 24].
In addition, particle tracking can be accelerated with highly parallel heterogeneous computing resources such as graphics processing units (GPUs) and FPGAs, providing a significant speedup of $O(100)$ over CPU-based execution [25]. FPGAs can be employed as low-power, low-cost co-processors in conjunction with CPUs to significantly speed up computation at similar performance [13]. The implementation of GNN-based track finding on FPGAs allows for use in real-time applications, e.g., at the trigger level [2, 26] with strict low-microsecond latency requirements. In this latency range, CPU or GPU solutions are not practical to meet the timing requirements.
The PANDA experiment is a novel collider experiment addressing fundamental questions of hadron and nuclear physics in interactions of antiprotons with protons and heavier nuclei currently under construction at the **G**esellschaft für **S**chwer**i**onenforschung (GSI) facility in Darmstadt [27]. The universal PANDA detector will be a state-of-the-art fixed-target detector at the **F**acility for **A**ntiproton and **I**on **R**esearch (FAIR) facility, providing new insights into hadron spectroscopy, hyperon physics, proton structure, and hadrons in nuclei. Each PANDA subdetector system runs autonomously in a self-triggering mode [28]. Therefore, the first-level tracking and reconstruction algorithms must operate fast and consume few resources to process all incoming data. The PANDA DAQ system aims to reduce the initial raw data rate of 10 GB/s by at least 2 orders. For example, for the FTS, a total of up to 370 kHits/s is expected, resulting in a total bandwidth of 2.914 GB/s. Therefore, first-level event filtering algorithms implemented on FPGAs are required to provide high tracking efficiency with low latency.

The main goal of this work is to address two key challenges: The first challenge is applying a novel GNN-based track finding algorithm to PANDA FTS data, inspired by the work of Esmail [29]. The second challenge concerns hardware acceleration of the tracking algorithm through the implementation on FPGAs.
The overall structure of this thesis is composed of ten chapters, including this introductory

chapter. It begins with a review of relevant related work in Chapter 2, emphasizing the importance of DL-based track reconstruction and the implementation of DL-based algorithms on FPGA technology. In Chapter 3, the theoretical foundations of hadron physics are described, including elementary particles and fundamental interactions based on the standard model of particle physics (SM), as well as the fundamentals of hadron physics that are important for understanding the physics program of the PANDA experiment described in Chapter 4. Chapter 4 also describes the PANDA detector in the context of the FAIR research facility, focusing on the FTS detector and the `PandaRoot` framework. In addition, the fundamentals of charged particle tracking, including track finding and track fitting, and state-of-the-art tracking methods are described, also in the context of the PANDA experiment. Chapter 5 gives a brief introduction to ML, in particular to GNNs, and describes the performance metrics used in this work.
Chapter 6 then addresses the generation of the training data via the PANDA-specific `PandaRoot` framework, which includes several steps of data preprocessing and graph construction. The implementation of a GNN-based tracking algorithm is described and discussed in Chapter 7, which is applied to two a full graph building and a segmented graph building approach, and includes track segment finding and tracklet finding.
Chapter 8 gives an overview of FPGAs in general, the `Vivado HLS` and `hls4ml` compiler, and the principles of design optimizations using these compilers. Chapter 9 focuses on design parameter studies to enable the implementation of the discussed GNN algorithm on FPGAs with optimal design. Finally, Chapter 10 summarizes the results of this work and provides an outlook for further research.

2. Related Work

In recent years, analytical methods based on ML techniques, and in particular GNNs, have attracted considerable interest in HEP and led to a variety of applications [22, 30], including the observation of jet substructures [17], the search for exotic particles [18], and the reconstruction of particle flows [31, 32].
In 2018, the tracking ML competition TrackML [23] on the Kaggle platform, which challenged computer scientists to find novel algorithms and approaches to the tracking problem, drew much attention to applying ML-based solutions to tracking problems. Several architectures have been tested, such as image-based CNNs and RNNs [20], and collaborations like the Exa.TrkX project [24] have emerged from this challenge. In particular, architectures based on GNNs have been intensively studied recently, such as the IN architecture proposed by [1]. This architecture allows for a significant reduction in graph sizes compared to other previously studied architectures, which is critical for implementation in accelerated hardware. For the PANDA experiment, recent studies have been conducted on ML and GNN applications on tracking data from the `PANDARoot` analysis framework [29, 33].
Also, hardware acceleration of GNN-based tracking algorithms is an active area of research where `hls4ml` has been used to study GNN applications on FPGAs to determine resource utilization, latency, and performance behavior for different hyperparameter configurations [2, 34–36].

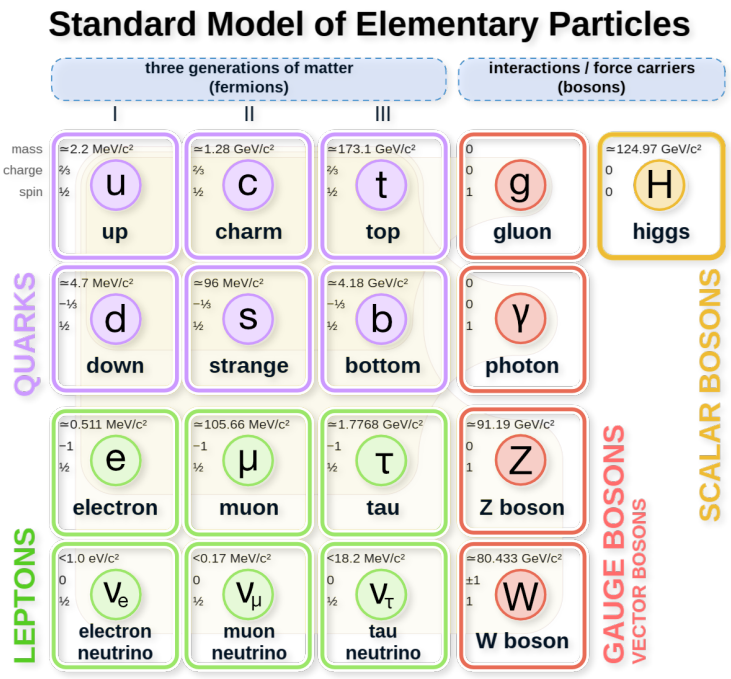


Figure 3.1.: Overview of the elementary SM particles which include quarks, leptons, gauge bosons, and the scalar Higgs boson. Source: [38].

3. Basic principles of Hadron Physics

This chapter briefly introduces the fundamentals of hadrons which are systems composed of elementary particles. Therefore, this section begins with a brief overview of elementary particles in Section 3.1 and the fundamental forces in Section 3.2 on which particle physics is based. Lastely, the basics of hadron physics are described in Section 3.3, in particular, focusing on the topics that are important for understanding the physics of the anti-**P**roton **An**nihilation at **DA**rmstadt (PANDA) experiment.

3.1. Elementary Particles

Elementary particles and fundamental forces, as well as the corresponding structure of all known matter, are currently described by the standard model of particle physics (SM). The SM is a very successful theory, which has been experimentally confirmed in various ways in the last decades [37]. Fig. 3.1 shows a schematic overview of the SM.
The SM comprises three groups of elementary particles: quarks, leptons, and gauge bosons. Atomic nuclei consist of protons and neutrons, which are not elementary particles

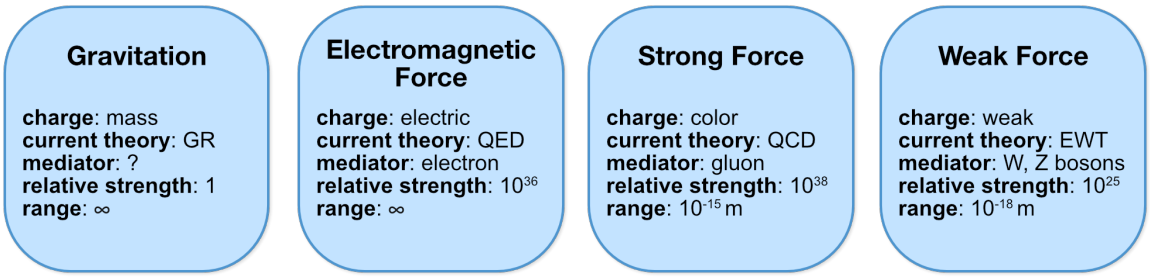


Figure 3.2.: Overview of the fundamental forces: gravitational, electromagnetic, strong and weak force described by the charge they couple to, the current describing theory, the mediator particle, the relative strength associated with the coupling constant and the range of the associated potential [39].

themselves but are composed of smaller constituents, namely the *quarks* and *gluons*. According to our present knowledge, quarks do not possess any further substructure and are, therefore, elementary particles, just like the negatively charged particles in the atomic shell, the electrons, which belong to the group of leptons. In the case of quarks and leptons, there are six different particles each, distinguished primarily by their mass and electric charge. The elementary spin $s = \pm 1/2$ particles (quarks and leptons), called fermions, are divided into three generations of different masses and are otherwise distinguished by charge and spin. In the case of quarks, the six different particles with increasing mass are up-, down-, charm-, strange-, top-, and bottom-quark. Up, charm, and top quark are electromagnetically charged with $q = +\frac{2}{3}e$ and down, strange and bottom quark with $q = -\frac{1}{3}e$, respectively, where e is the elementary charge. In addition, for each particle there is a corresponding antiparticle with a similar mass but with an opposite electric charge. The second group comprises leptons like the electron, the muon, and the tau, which are particles similar to the electron except for the mass, as well as the corresponding neutrinos. The third group covers the particles that mediate the fundamental interactions between particles, the gauge bosons (vector bosons) and the last group contains the (scalar) Higgs boson, experimentally discovered in 2012, which was the last missing particle predicted by the SM.

3.2. Fundamental Interactions

Currently, four fundamental interactions are known: the electromagnetic, the weak, the strong, and the gravitational force. The latter is currently described by general relativity (GR) theory but not included in the SM and has not yet been described in a singular theory among the other three forces which are described by the SM. Fig. 3.2 displays a short overview of the fundamental forces and their basic characteristics. In general, fundamental interactions are transmitted by exchanging mediator particles, called gauge bosons, which couple to the corresponding charges of the elementary particles. For example, the electromagnetic interaction between electromagnetically charged elementary particles is mediated by the photon described by the quantum electrodynamics (QED)

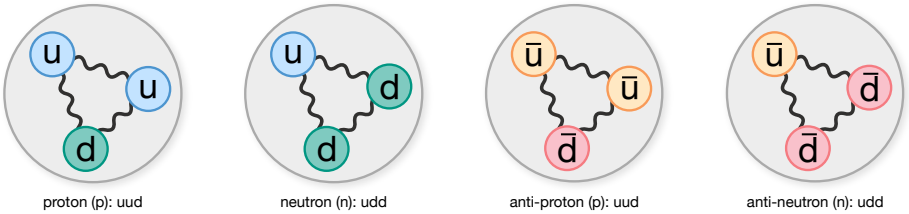


Figure 3.3.: Baryons: Protons and neutrons, as well as their corresponding antiparticles: antiprotons, and antineutrons, are composed of quarks. For example, the proton comprises two up quarks (u) and one down quark (d).

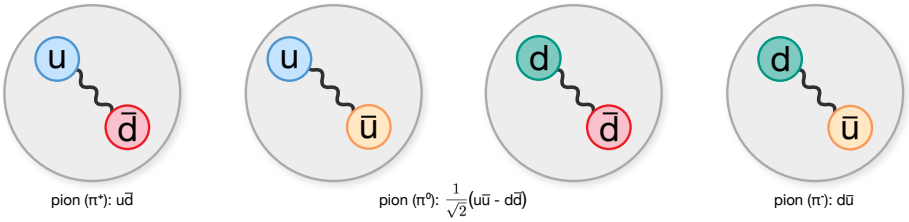


Figure 3.4.: Mesons: There are a neutral pion π^0, and two charged pions π^+ and π^-, which are composed of a particle-antiparticle pair each. All three pions are unstable and decay via weak or electromagnetic force.

theory. The electromagnetic force is, among other effects, responsible for the molecules' cohesion and the electron shell binding to the atomic nucleus. Particle decays such as radioactive β decays are driven by the weak interaction mediated by intermediate heavy W and Z bosons, which couple to the weak charges of particles. The weak force is described within the so-called electroweak theory, which unifies the electromagnetic and weak force. The third fundamental interaction is the strong interaction, which couples to the color charge of particles and is mediated by the exchange of gluons. The strong interaction is theoretically described by the quantum field theory quantum chromo dynamics (QCD). It is the strongest of all four forces on the nuclear scale and is therefore responsible for the internal and external cohesion of the nuclear building blocks of an atom, even though the positively charged protons within a nucleus electromagnetically repel each other. Particles with color charge cannot exist in isolation but only occur in bound states such as mesons or baryons. In attempts to separate quarks at high energies, new quark-antiquark pairs are spontaneously created. This imprisonment of quarks and gluons is called *confinement*. A challenge of modern high energy physics (HEP) is to increase understanding of confinement in the theory of the strong force.

3.3. Hadron Physics

Hadron physics is concerned with the study and precise measurement of all systems composed of quarks, called *hadrons*, and based on this, the study of the associated fundamental particles and forces. There are two types of hadrons: baryons and mesons. The characteristic properties of *baryons* are given by three quarks, such as protons and

neutrons, as shown in Fig. 3.3. For example, on the one hand, a proton consists of two up quarks and one down quark, which gives a total charge of $+e$. On the other hand, an antiproton consists of the corresponding antiparticles: two antidown quarks and one antiup quark, giving a total charge of $-e$.
Mesons, such as pions, consist of a quark-antiquark pair as indicated in Fig. 3.4. Pions are built from different configurations between up and down quarks and oppositely charged counterparts. For example, the neutral pion π^0 is a quantum mechanical superposition state of the two quarkonia $u\bar{u}$ and $d\bar{d}$. Other important forms of mesons include *quarkonia* that are bound states of a quark and its antiquark. A further example is the quarkonium $c\bar{c}$, also called *charmonium*. Charm quarks have a relatively large mass compared to up and down quarks, leading to a smaller distance between quark and antiquark and lower kinetic energy, such that the spectrum can be described by non-relativistic potential models such as EFT [40] and LQCD [41]. Although all 8 charmonium states below the open charm threshold are known, their parameters and decays have yet to be accurately measured. Beyond that, almost nothing is known above the threshold.
Another important group of mesons in this context are the D-mesons which are mesons composed of a charm quark as the heaviest particle. For example, D^0 is composed of a charm and an antiup quark, and D_s^+ is a combination of a charm and an antistrange quark.
The composite baryons and mesons are held together by strong interaction. Compared to mesons, the internal structure of baryons is more complex since they contain not only the two or three quarks mentioned above, the so-called valence quarks that define the quantum numbers of the corresponding hadrons, but also additional virtual particles. These virtual particles appear as excitations of the QCD vacuum in the strong interaction force field. The excitations include gluons and virtual quark-antiquark pairs (sea-quarks) that are created for a short time and annihilate again (vacuum fluctuation). This interaction leads to the observed rest mass of hadrons built from light quarks, such as up and down quarks, being significantly larger than the sum of the rest masses of the valence quarks. For example, the three valence quarks of the proton together have a rest mass on the order of $10\,\mathrm{MeV}/c^2$, however, the entire proton has a rest mass of $938\,\mathrm{MeV}/c^2$. In contrast, the rest masses of the heavy quarks (charm, top, and bottom) are so high that they are not significantly affected by the sea quarks. Considering this significant contribution to the overall properties of hadrons, they are composed of constituent quarks, quasi-elementary particles combining valence quarks with their respective fraction of virtual particles.
An important form of baryons for the PANDA experiment are *hyperons*, which include baryons with at least one strange quark and up or down quarks as valence quarks. Hyperons are characterized by their negative number of strange quarks, called strangeness S. Examples are Λ (uds), Σ (xxs), Ξ (xss), and Ω (sss) hyperons, where x can be u or d. Nuclei containing hyperons are called *hypernuclei*. Equivalently, there are particles consisting of several charm quarks, defined by the flavor quantum number charmness C.
In addition to baryons and mesons, there are so-called exotic particles, which have been predicted by theoretical models and some of which have already been experimentally detected. Exotic particles include, for example, particles with gluonic excitations, i.e., hadrons with gluons as principal components contributing to the total quantum numbers. Since gluons carry color charges themselves, an additional self-interaction of these intermediate particles is possible, allowing the existence of particles consisting purely of gluons,

so-called *glueballs*. Possible gluonic excitations include not only glueballs, but also exotic forms such as *hybrids* between hadrons and excited gluons, in which both components contribute to the quantum numbers (quark-antiquark pairs with an additional gluonic excitation). Due to the additional degrees of freedom provided by gluons as principal components, states with combinations of quantum numbers different from ordinary hadrons are enabled. Therefore, gluonic excitations are relatively easy to identify experimentally. However, the existence of glueballs has not yet been unambiguously confirmed experimentally, but in recent years promising searches have been undertaken, such as the search for $J^{PC} = 0^{--}$ glueballs in $\Upsilon(1S)$ and $\Upsilon(2S)$ decays at the Belle experiment [42, 43], where J^{PC} describes the quantum numbers total angular momentum J, parity P, and charge-conjucation C.
Furthermore, QCD allows the existence of more complex structures called exotic hadrons. Besides glueballs and hybrids, these include tetraquarks and pentaquarks, which are built of four or five quarks. Great progress has been made in recent years, for example, with the unexpected discovery of the excited X, Y, and Z states [42], the nature of which must now be studied in detail.

Although the SM has shown huge success in predicting the existence of W and Z bosons, gluons, top and charm quarks, and many of their properties before they were even observed, there are still many open questions of particle physics, such as baryon asymmetry, dark matter, neutrino mass, or the hierarchy problem. In the following chapter, the PANDA experiment will be described, which will focus on investigations related to the weak and strong forces at medium energy, such as investigating the structure of the QCD vacuum, confinement, the origin of hadron masses, and exotic states of hadronic matter.

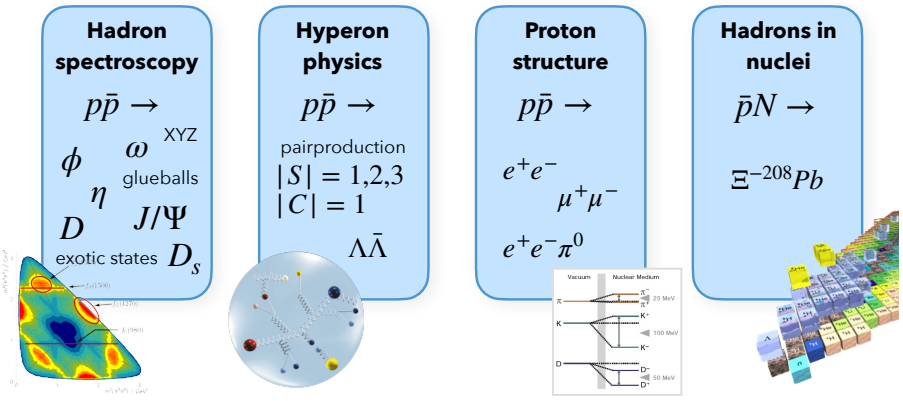


Figure 4.1.: Overview of the PANDA physics program. The research is based on four pillars: hadron and hyperon physics, proton structure, and interaction of hadrons in nuclei. Source: [45]

4. PANDA Experiment

The anti-**P**roton **An**nihilation at **DA**rmstadt (PANDA) experiment is one of the key experiments currently under construction at the **F**acility for **A**ntiproton and **I**on **R**esearch (FAIR) in Darmstadt, Germany [27, 44]. It is a fixed-target experiment using proton-antiproton annihilation to investigate many open questions in hadron physics in the charm and multi-strange hadron sectors, complementing existing facilities. The main aspects of the PANDA physics program include fundamental questions about weak and strong forces, exotic matter states, and hadron structure. In the following sections, the PANDA experiment with its extensive physics program, the FAIR facility in which it is located, the detector itself, and its software framework, `PandaRoot` will be presented. Lastly, a short introduction to the principles of particle track reconstruction, also in the context of PANDA, is given.

4.1. PANDA Physics Program

The PANDA experiment is expected to provide new insights in the field of quantum chromo dynamics (QCD). It will be operated complementary to existing facilities by using antiprotons as hadronic probes instead of primarily used electromagnetic probes or proton beams. Antiproton probes offer high production rates compared to electromagnetic probes and are unique in directly populating a large spectrum of spin parity states [46]. This enables high-resolution line-shape measurements and access to very high spin states, allowing detection and measurement of previously unobserved exotic particles predicted in the framework of strong interaction theory, such as glueballs and hybrids. The goal

of PANDA is to explore the dynamics of the weak and strong interactions by studying various composite quark systems and their production and decay mechanisms. Therefore, the PANDA physics program relies on four research pillars: hadron spectroscopy, hyperon physics, nucleon structure, and hadrons in nuclei depicted in Fig. 4.1 [27, 44]. In the following, the four pillars will be described in more detail.

4.1.1. Hadron Spectroscopy

The first pillar of the physics program focuses on precision hadron spectroscopy of light to charm quarks and gluons, the search for exotic particles, and the measurement of hadron properties.
One aspect of hadron physics is the search for glueballs and hybrids. Planned studies include the investigation of the glueball spectrum such as the glueball ground state candidate $f_0(1500)$ or states with the quantum numbers $J^{PC} = 2^{++}$ and 0^{-+}, the search for exotic forms of hybrids and meson-like or molecular states at expected luminosities of the benchmark channels at $2 \cdot 10^{32}/(\mathrm{cm}^2\,\mathrm{s})$.
Another prospect of hadron spectroscopy at PANDA is charmonium spectroscopy. PANDA is expected to be able to measure the properties of the newly discovered XYZ spectrum and to find the predicted D- and F-wave states by precision scans below and above the open charm threshold. Searches in the high spin region up to a total angular momentum of $J = 6$ and at larger masses up to $5.5\,\mathrm{GeV}/c$ are planned to complement existing findings and cover new decay channels.
Another important topic in hadron spectroscopy is the spectroscopy of D mesons, including the study of new open charm mesons recently discovered at the BaBar, Cleo, and Belle experiments [47, 48]. These open charm mesons deviate from the predictions of the quark model, unlike the previously known D states, and are therefore attracting considerable interest. One way to study these states is to look at the decay width of the D_s states, which allows distinguishing between different theoretical models. PANDA is expected to collect thousands of $c\bar{c}$ states per day which allows mass measurements with accuracies up to the order of 100 keV and widths to 10 % or better. The entire energy range below and above the open charm threshold at 3.73 GeV will be explored.

4.1.2. Hyperon Physics

The second pillar is dedicated to studying hyperon physics via the formation of mesons and baryons with open strangeness $|S|$ or open charmness $|C|$. Pair formation of baryons and mesons with $|S| = 1, 2, 3$ or $|C| = 1$ near their production threshold is expected to have large effective cross sections of two or even three orders of magnitude larger than those of photons. States that do not couple to photons or kaons will be studied at PANDA especially investigating the $|S| = 3$ sector, to which few experiments in the world have access.
The study of hyperons provides a direct test of the validity of few-body models and access to spin observables such as polarization and spin correlations through weakly decaying hyperons. Moreover, the fundamental question of matter-antimatter asymmetry can be addressed by a model-independent test of CP violation in baryon decays via a

multidimensional analysis of the particle-antiparticle symmetric final state of hyperon decays. High-precision effective cross sections around 64 µb for the production of $\Lambda\bar{\Lambda}$ pairs are already expected at phase-one luminosity and 2 µb for $\bar{p}p \rightarrow \Xi\bar{\Xi}$.

4.1.3. Proton Structure

The third physical pillar of PANDA involves experiments on proton structure with dileptonic electromagnetic final states $\bar{p}p \rightarrow l^+l^-$. One goal are time-like electric and magnetic form factor studies of $|G_E|$ and $|G_M|$, respectively, in an expected q^2 range from threshold to $22\,\mathrm{GeV}^2/c^2$ and above to provide new insights into the understanding of the proton radius and to test the universality of leptons. Significant improvements on the results of previous experiments and complementary results to space-like studies on lepton scattering experiments are expected in terms of energy range and accuracy. In particular, it is planned to measure $|G_E|$ and $|G_M|$ separately, which has not been possible so far due to limited statistics. In addition, PANDA will be able to probe the off-shell region $4m_e^2 < q^2 < 4m_p^2$ through a final state including an additional pion.

4.1.4. Hadrons in Nuclei

The fourth and final pillar of PANDA physics is devoted to the properties of hadrons in nuclear medium. This study aims to understand better the origin of hadron masses in the context of QCD at finite nuclear densities. Mass shifts of hadrons in a nuclear environment reflect the real part of a nuclear potential and are mainly expected for hadrons at rest or with small momentum relative to the nuclear medium. In PANDA, hadrons produced in antiproton-nucleon collisions are expected at much lower momenta than in proton-induced reactions, giving access to promising momentum ranges in terms of significant medium effects. For example, the antihyperon nuclear potential will be investigated by producing hyperon pairs near the production threshold. The measurement of fundamental in-medium properties of charmed hadrons, such as mass and width in the nuclear medium, is also planned to provide new insights into the formation of hadrons. In addition, color transparency, which describes the phenomenon that the strong interaction in the nuclear medium becomes very small for some observables, will be studied by the response of antiprotons to a nuclear target. Other topics include the implantation of hyperon pairs in nuclei, the unique search for X-ray transitions of heavy hyperatoms such as Ξ^{-208}Pb, and later high-resolution γ spectroscopy of double-hypernuclei.

4.2. FAIR Research Facility

The **F**acility for **A**ntiproton and **I**on **R**esearch (FAIR) research facility is currently under construction at the **G**esellschaft für **S**chwer**i**onenforschung (GSI) site in Darmstadt, Germany. A recent construction site picture is shown in Fig. 4.2. FAIR will have extensive facilities for the production, storage, and acceleration of antiprotons and will house various experiments [50]. These include the PANDA experiment and CBM, APPA, and NuSTAR, with projects in atomic, plasma, and astrophysics. The new accelerator center, one of the

Figure 4.2.: Photograph of the FAIRconstruction site taken in 2021. Source: [49].

largest international research projects in the world, builds upon the experience and infrastructure of the existing GSI facility. The existing GSI accelerators UNILAC and SIS18 will serve as the first accelerator stage injectors for the new accelerator ring SIS100. FAIR will deliver unprecedented intensity and quality particle beams. In FAIR, protons, antiprotons, unstable nuclei, and ions of all natural elements of the periodic table can be produced and accelerated with high beam intensities. The centerpiece of FAIR is a ring accelerator SIS100 with a circumference of 1100 meters. A complex system of storage rings and experimental stations is connected to it, including the PANDA experiment. Fig. 4.3 shows the layout of the FAIR facility with the existing facilities (blue: GSI accelerator), the future components (red: FAIR accelerator) and the location of the various new experiments such as PANDA, which is located at high-energy storage ring (HESR).
The antiproton beam for the PANDA experiment is prepared as follows: Protons are accelerated in different stages (proton-LINAC, SIS18, and SIS100) to a final energy of 29 GeV/c and then collided with an antiproton target, generating antiprotons at a production rate of $2 \cdot 10^7$/s. The produced antiprotons are then collected and precooled in the collector ring (CR) and then injected into HESR at a rate of 10^8 antiprotons per 10 s until a total number of 10^{10} antiprotons is reached in the accelerator. The final momentum of the antiproton beam ranges from 1.5 GeV/c to 15 GeV/c. Fig. 4.4 shows the layout of HESR with the injection point from the CR, complex magnetic systems, and stochastic cooling systems. HESR is a storage ring with a circumference of 575 m and a magnetic bending force of 50 T m, that inherits the PANDA, KOALA, and SPARC experiments. The PANDA detector will be located in one of the straight sections of HESR.

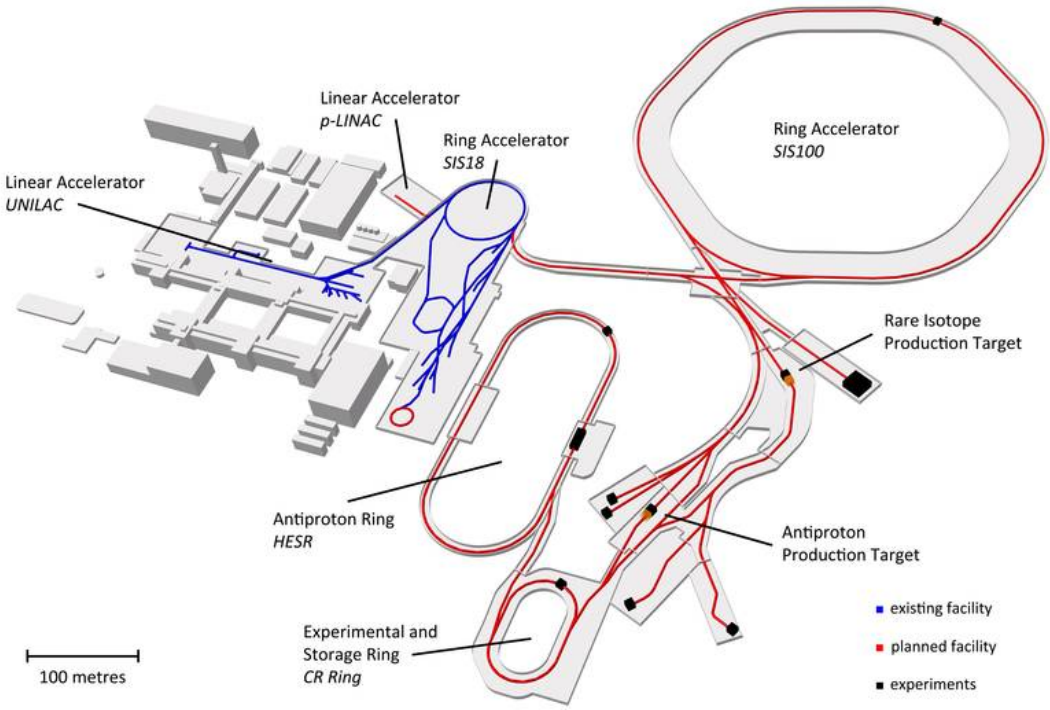


Figure 4.3.: Layout of the FAIR facility with the existing facilities (blue: GSI accelerators), the future components (red: novel FAIR systems) and the location of the various new experiments. Source: [51].

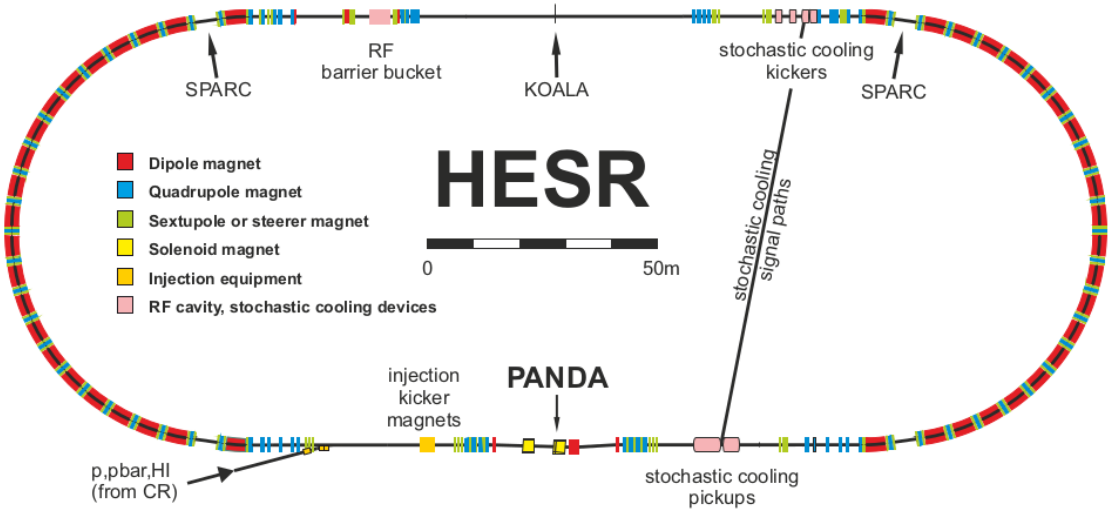


Figure 4.4.: Layout of HESR. Source: [52].

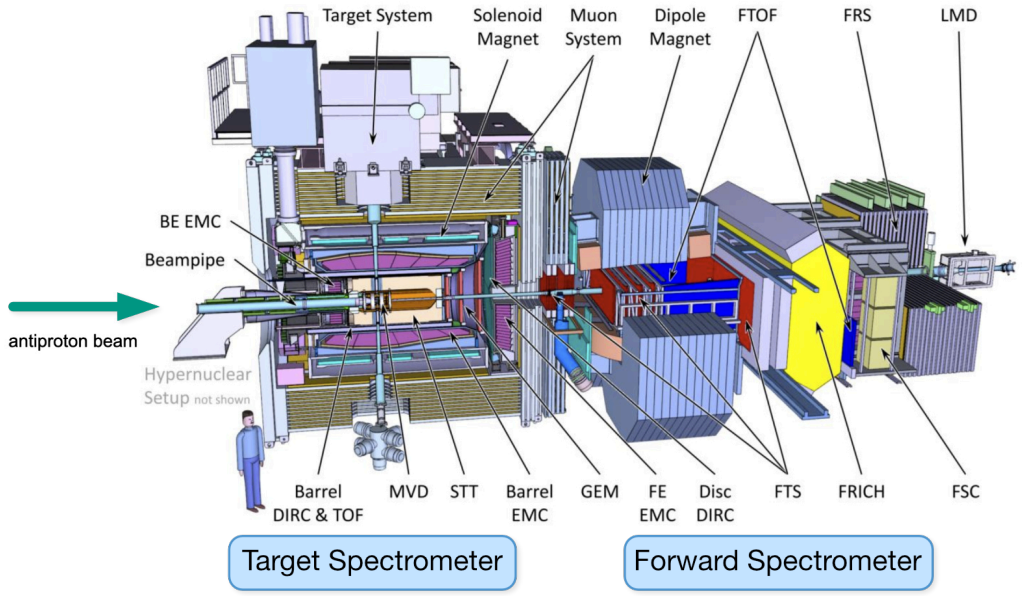
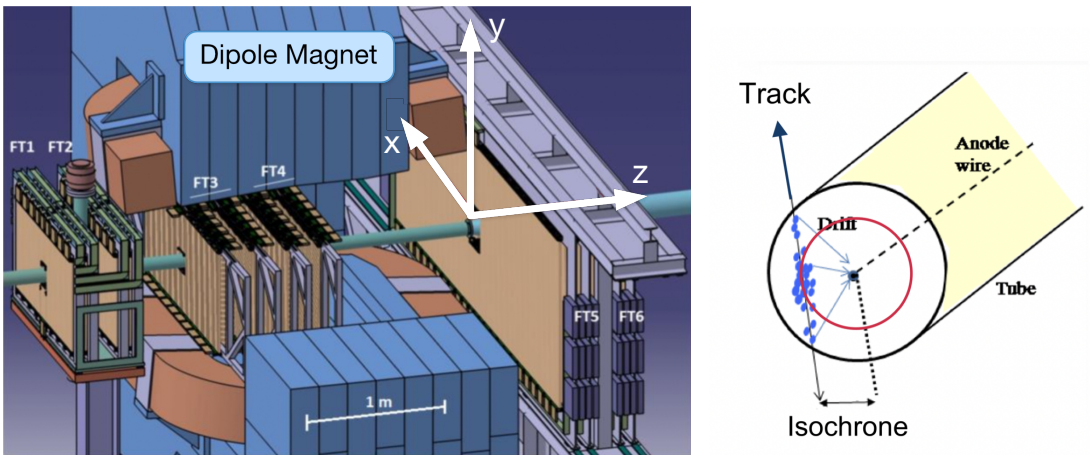


Figure 4.5.: PANDA detector overview with target and forward spectrometer composed of various detector systems. The antiproton beam enters from the left. Source: [53].

4.3. The PANDA Detector

The extensive physics program planned for the PANDA experiment requires a versatile detector with full solid angle coverage and capabilities for charged particle tracking, particle identification, calorimetry, and muon detection [28, 54]. As a fixed-target experiment, the PANDA antiproton beam collides with the quasi-resting target at the interaction point with a momentum range of 1.5 GeV/c to 15 GeV/c and a collision rate of up to 20 million particles per second at a target luminosity of $2 \cdot 10^{32}/(\mathrm{cm}^2\,\mathrm{s})$. Due to the conservation of momentum and energy, the particles generated at the interaction point are accelerated in the forward direction. The detector is therefore divided into two sections as shown in Fig. 4.5: the target spectrometer and the forward spectrometer. The target spectrometer covers the central region around the interaction point, providing nearly 4π coverage within a superconducting solenoid magnet with a magnetic field of up to 2 T with homogeneity of ±2%. The forward detector is constructed with a dipole magnet in the beam direction, providing a bending power of 2 T m for the detection and measurement of particles that are boosted in the beam direction at low polar angles. The forward detector measures forward-boosted particles at low polar angles θ of up to 5° in the vertical direction and 10° in the horizontal direction with respec tto the beam direction. Both spectrometers consist of several detector systems that enable precise particle track reconstruction, vertex determination, momentum and energy measurement, and particle identification. In total, the PANDA detector covers about 12 m along the beamline with a total height of about 6 m.

The PANDA detector operates on a so-called trigger-less readout. This means that without



(a) (b)

Figure 4.6.: (a) Schematic view of the PANDA forward tracking system (FTS) composed of 6 tracking systems FT1 to FT6, where FT3 and FT4 are located within a dipole magnet. Source: [55]. (b) Schematic view of a straw tube and particle tracking. The red circle indicates the isochrone radius. Source [55].

any hardware trigger each subdetector system runs autonomously in a self-triggering mode which is realized with self-triggered intelligent front-end electronics and online data processing algorithms. Up to 10 GB/s of raw data are expected to stream into the data acquisition system (DAQ) [28] that need to be reduced by at least two orders before storage on disk. The first level of event filtering is performed on field programmable gate arrays (FPGAs) attached to a graphics processing unit (GPU) server farm as a second level.

4.3.1. Forward Tracking System (FTS)

The forward tracking system (FTS) is particularly relevant for this thesis. It is used to determine the trajectories and particle momenta of charged particles that are boosted in the forward direction of the spectrometer. The FTS consists of three pairs of planar tracking detector stations arranged in planes perpendicular to the beam axis, depicted in Fig. 4.6(a). The overall coordinate system is oriented with z axis along the beam axis, x parallel to the ground, and y pointing upwards. The first pair (FT1, FT2) is located in front of the dipole magnet, the last pair (FT5, FT6) is located behind the dipole magnet, and a third pair (FT3, FT4) is located inside the magnet aperture. Each of the FTS tracking stations consists of four double layers: the first and last double layers are oriented vertically (in the xy-plane), while the second and third double layers, called skewed layers, are oriented with angles of 5° and -5° to the xy-plane, respectively. The skewed layers provide three-dimensional information in the y direction in addition to the x and z information obtained via the non-skewed layers. This overall arrangement of layers allows the independent reconstruction of tracks in each pair of tracking detector stations, even in the case of multiple tracks per event.
The FTS is a straw tube tracking system that comprises tubes of 10 mm in diameter,

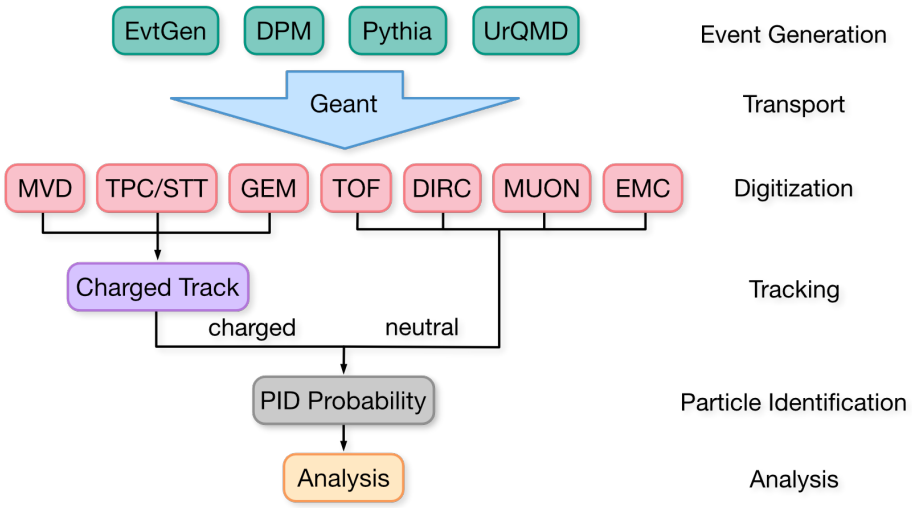


Figure 4.7.: Schematic view of the `PandaRoot` data flow. From event generation via different event generators (EvtGen, Dual Parton Model, Pythia, Ultra Relativistic Quantum Molecular Dynamics), transport through the detector via Geant, digitization in the different detector components (MVD, TPC/STT, GEM, TOF, DIRC, Muon, EMC) over charged particle tracking to particle identification and analysis.

providing a spatial resolution of $1/\sqrt{1210}$ mm. Straw tubes are multi-wire proportional chambers that operate in the proportional region containing a high-voltage anode wire surrounded by gas. In total, the FTS consists of 13065 tubes filled with a 90:10 argon-carbon dioxide mixture that has been found to be a gas proportion providing good properties, including reducing aging effects. Charged particles passing through the tubes ionize the gas atoms, creating electron-ion pairs. In the high electric field near the anode wires, the electrons are accelerated toward the anode wires, ionizing the gas atoms and again creating new electrons. This leads to an amplification of the current as a function of the applied voltage and the gas pressure, the so-called charge avalanche. The current is proportional to the initial charge and allows the determination of the particle energy loss and, in combination with the measured particle momentum, identification of the particle type. The drift time of the electrons to the anode wire can also be measured, which gives information about the distance of the charged particle track to the anode wire, the so-called isochron radius shown in Fig. 4.6(b). Integrating the isochrone radius, a full spatial resolution of 150 µm perpendicular to the tube axis can be achieved.
A total of up to 370 khits/s is expected, which results in a total bandwidth of 2.914 GB/s combining all six tracking stations.

4.4. PandaRoot

Since FAIR and the PANDA detector are still under construction, only simulation data for physical processes at PANDA are available so far. Simulation is performed using the

PandaRoot framework [56, 57], which is part of the more general FairRoot framework. PandaRoot is based on ROOT [58] and Geant [59] to simulate detector performances and evaluate different detector designs based on virtual Monte Carlo simulations [60]. PandaRoot is capable of simulating expected physical interactions and detector response, as well as full event reconstruction, selection, and analysis.
The structure of the PandaRoot framework is shown in Fig. 4.7: In the first stage, events are generated via different event generators (EvtGen, DPM, UrQMD, Pythia) for signal and background processes. The simulated particles are transported through the geometric volumes of the detector using Geant3/4, taking into account the magnetic field, particle decay, energy loss, scattering, and bremsstrahlung. The detector geometry can be described in ASCII format or as ROOT objects. CAD drawings of the detector components can be loaded and converted to ROOT geometry format. The next step is the detector response simulation as particles pass through the sensitive volumes, determining position, momentum, time, and energy loss. The detector response simulation is digitized to include detector effects such as segmentation, electronics emulation, noise, thresholds, and timing so that the data corresponds to a more realistic detector response to passing particles. Events can be displayed using an event display based on the event visualization environment (EVE) in ROOT. Finally, the detector response is used for hit reconstruction, cluster finding, local and global tracking, and particle identification for complete event reconstruction. In the final step, the reconstructed events can be used for various analysis tasks, including geometric and kinematic fits and angular distributions, to extract physics from the reconstructed 4-momentum events. The PandaRoot software is almost complete and is currently used for various physics analyses. However, improvements are still being made, such as including additional timing information for event building.

4.5. Track Reconstruction

In this section, charged particle tracking and the methods used for this important part of the data analysis chain in high energy physics (HEP) experiments are briefly described, based on the information in [61]. In HEP, track reconstruction, or short *tracking*, is the task of accurately reconstructing the basic kinematic parameters of charged particles such as position, direction, and momentum. Tracking detectors are placed around the interaction points of collider experiments to enable high-precision position measurements. Particles ionize the active materials of the detector, enabling position measurements along the particle trajectories. Tracking detectors are typically constructed of as little material as possible to minimize energy losses of passing particles. Furthermore, tracking detectors or parts of the detectors are typically located in a calibrated magnetic field that allows the determination of the particle momenta via the particle track curvatures in the magnetic fields.
In general, tracking is divided into two distinct steps: *track finding* and *track fitting*. Track finding is the task of pattern recognition to identify detector hits that are assumed to come from the same particle track. The goal of track fitting is to fit curves to the detector measurements of the track candidates found by the track finding from which particle momenta and charges can be determined.

Track reconstruction can be performed online and offline, where online means real-time analysis, usually based on simplified methods implemented via discrete electronic components, while offline analysis is performed after storing the data on disk. The main goal of offline reconstruction is maximum precision, while speed is the defining factor for online reconstruction.
Various tracking detector devices have been developed over the past decades based on different technologies. These include cloud chambers, nuclear emulsion plates, bubble chambers, spark chambers, multi-wire proportional chambers, drift chambers, and time projection chambers. In addition, there are also tracking detectors based on semiconductor technology, which enable high-precision and fast readout. Modern detectors often combine silicon and gaseous tracking detectors in three tracking systems: a high-precision detector near the primary interaction point, a central or inner tracker, and the muon tracking system in the outermost detector layers [62].

4.5.1. Track Finding

The aim of track finding is the correct assignment of detector hits to clusters representing tracks stemming from the same particle. The clusters are divided into a subset of interesting track candidates and a subset containing uninteresting measurements such as noise or low-energy curling particles.
Since tracking is performed in an early stage of the data analysis chain, track candidates discarded at this stage may not be recovered at a later stage. Therefore, track finding should be conservative and keep track candidates in doubt instead of discarding them. In general, tracking is a complex and time-consuming process. Computational speed is an critical aspect of track finding, especially for triggering applications. Therefore, simplified models must often be used.
Track finding can be divided into global and local methods. Global methods consider all detector measurements simultaneously. Examples are conformal mapping, the Hough transform, and the Legendre transform, while local methods such as track road and track-following methods process measurements sequentially.

4.5.2. Track Fitting

The goal of track fitting is to improve the track parameters generated by the previous track finding stage. While in track finding, usually simplified track models are used, track fitting aims to be as realistic as possible, taking into account all electromagnetic fields and realistic material distribution of the detector. The basic requirements for track fitting are that it should be numerically stable and robust to track finding errors. The final reconstructed tracks can then be used to verify the assignment of hits to a complete track candidate. Some hits may be misclassified and do not belong to a particular track, or sometimes a complete track may be a random collection of unrelated hits, so-called ghost tracks.

4.5.3. State-of-the-Art Tracking

Today, combinatorial algorithms based on the Kalman filter are widely used for particle tracking, combining track finding and track fitting simultaneously [9, 10]. The Kalman filter is a mathematical method for iteratively estimating parameters that describe system states based on noisy measurements. Algorithms based on the Kalman filter are robust to difficult operating conditions in tracking systems involving, for example, multiple scattering of particles in detectors. However, one problem with combinatorial Kalman filter algorithms is that they scale worse than linearly with an increasing number of hits, which will also increase the computational time required for tracking. In the future, the number of hits per event is expected to increase due to higher detector occupancies and luminosities. For example, the luminosity increase corresponding to the LHC upgrade to the HL-LHC [63] is expected to increase by order of magnitude, directly affecting the number of hits. The increase in computation time for track reconstruction is expected to be faster than the evolution of computational resources [15]. Therefore, current research is focused on developing new tracking algorithms, especially based on machine learning (ML) techniques. In addition, there is significant interest in accelerating tracking algorithms through parallelization on specialized hardware to enable online tracking at the trigger level, also using ML methods.

4.5.4. Tracking with `PandaRoot`

For the PANDA experiment, several reconstruction algorithms for tracking and particle identification are currently being developed and optimized to meet the performance requirements of the experiment. In the central tracker, the track is first fitted by a conformal map transformation based on a helix assumption. Then, the track is used as input to the Kalman filter based track-fitting toolkit `Genfit` [64] that is an experiment-independent toolkit combining fitting algorithms, track representations, and measurement geometries into a modular framework. Finally, the track is used in conjugation with the PID detectors (e.g., Cerenkov detectors, EM calorimeters, or muon chambers) to determine a global particle identification probability using Bayesian approaches or multivariate methods [57].

In this work, track finding is performed using FTS position and time measurements created by the interaction of charged particles with the FTS units. Recent work has demonstrated that ML algorithms, particularly graph neural networks (GNNs), are well suited for the pattern recognition task of track finding [1, 20, 23, 24, 29]. In the following chapter, the basics of ML and GNNs are described.

5. Basics of Graph Neural Networks

This chapter lays out machine learning fundamentals, specifically graph neural networks, as they are a key aspect of this thesis. Machine learning is based on artificial neural networks (NNs), i.e., computer algorithms inspired by biological neural networks. NNs are trained on task-specific data samples to recognize and learn certain features of the given data. A trained network can be used to perform data processing tasks such as regression and classification or even more complex tasks such as speech recognition. In a nutshell: NNs are stochastic minimization algorithms applied to a multidimensional input space to improve performance on a given task.
This chapter describes the general principles of artificial neural networks, the principles of over- and underfitting, and the implementation of machine learning with Pytorch. It also introduces graph neural networks (GNNs), a special class of NNs that operate on non-Euclidean data, and describes the performance metrics used in this work. The focus of this chapter is to provide a brief overview of the necessary information for this thesis. More detailed information on the theoretical aspects and applications of NNs can be found in [65, 66] and [67].

5.1. Neural Networks

In principle, NN consists of several mathematical building blocks, called modules, as shown in Fig. 5.1. The network itself consists of several layers that are interconnected. The first of these layers is called the input layer, and the last layer is called the output layer, representing the data input and output, respectively. There may be several *hidden layers* between the input and output layers that are not connected to the outside of the network. If the network contains more than one hidden layer, it is called a deep neural network.
Each network layer is associated with trainable parameters, *weights*, and *biases*. Input data x of arbitrary dimensions containing features of the training patterns are passed through the network layers, which perform multiple data transformations on the data x to obtain an output $\hat{y}$ as a prediction of the network for a particular task.
The layers of NNs contain several nodes or so-called *neurons*. The perceptron is a variant of artificial neurons and is one of the main building blocks of modern NNs. In principle, it is represented by a mathematical function of the following form as depicted in Fig. 5.2:

$$\hat{y}_i = f\left(\sum_{j=0}^{N} w_{ij}x_j + b_i\right) \qquad i \in \{1, \ldots, n_{\text{neurons}}\}, j \in \{1, \ldots, N\} \; . \tag{5.1}$$

The output of each perceptron $\hat{y}_i$ is computed by a weighted sum of the N inputs x_j plus a bias b_i enclosed by a nonlinear activation function f. Common activation functions

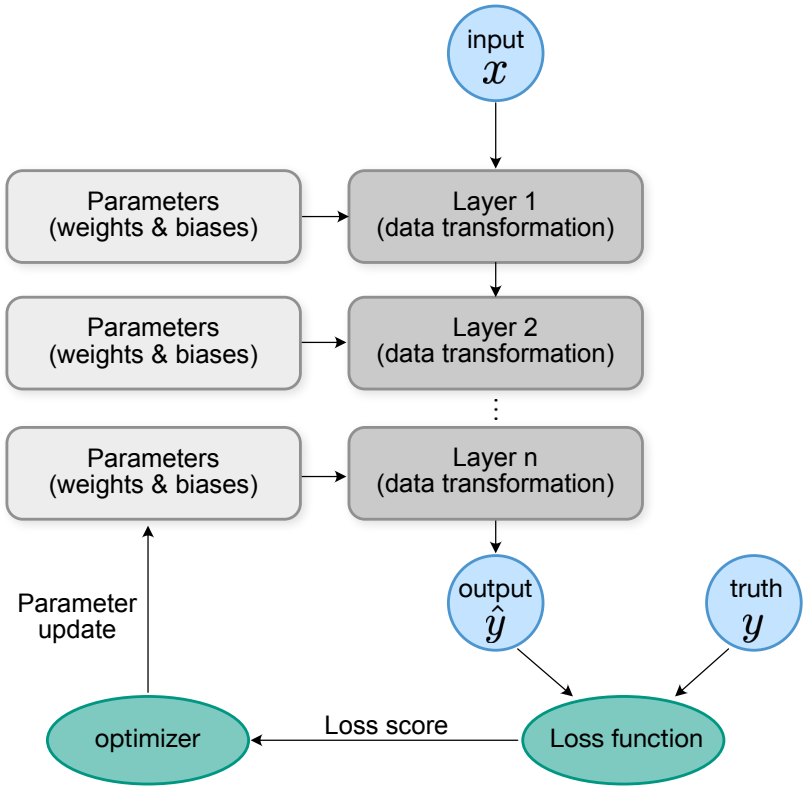


Figure 5.1.: Schematic NN model architecture (grey), training steps (green) and input/output data (blue).

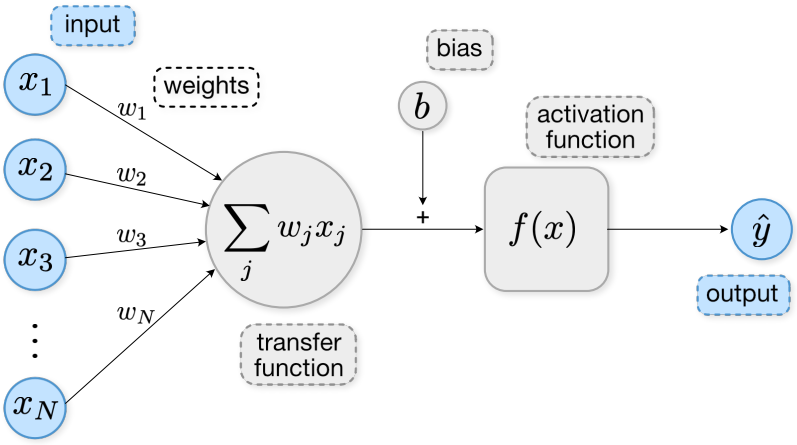


Figure 5.2.: Graphic representation of a perceptron as defined in Eq. (5.1). The weighted sum of the inputs plus a bias term is calculated and given to an activation function that calculates the output.

include a simple step function, the sigmoid function

$$f(x) = \frac{1}{1+\exp(-x)} = \frac{\exp(x)}{\exp(x)+1}\,, \tag{5.2}$$

the softmax function

$$f(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \quad \text{for } i = 1,\ldots,K \text{ and } \mathbf{x} = (x_1,\ldots,x_K) \in \mathbb{R}^K \tag{5.3}$$

with the number of classes K, and the rectified linear unit (ReLU) function

$$f(x) = \max(0,x)\,. \tag{5.4}$$

Multiple perceptrons (or artificial neurons) arranged in series form a multilayer perceptron (MLP), a multilayer feed-forward network. The layer is said to be fully connected if all neurons in one layer are connected to all other neurons in the adjacent layers. The final output layer is usually wrapped by a sigmoid or softmax function, which both ensure that all output values are in the range (0,1) and will sum to 1, thus constituting a valid probability distribution. The prediction is compared to a true target y associated with the input data x via a loss function L. Depending on the given problem, there are several types of loss functions. For example, in the case of classification problems, it is common to use the categorical cross entropy [68]. In this work, the two-dimensional special case of the categorical cross-entropy, the binary cross entropy,

$$L(y,\hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(\hat{y}_i) + (1-y_i)\cdot\log(1-\hat{y}_i)\right) \tag{5.5}$$

is applied. The discrepancy between the predicted ($\hat{y}$) and the true (y) vector is quantified by the output of the loss function, called the *loss* score, which must be minimized during the training process, to obtain the best match between $\hat{y}$ and y. The minimization is achieved by an optimization algorithm that operates on the loss score to minimize it in the next iteration. The loss score is propagated back from the output layer to the input layer and updates the network parameters depending on their influence on the loss score. This process is called *backpropagation*. The number of training iterations taken to improve the network predictions is called *epochs*, one of many hyperparameters that must be fixed before training and determine the network structure and training properties.

5.2. Overfitting and Underfitting

The goal of machine learning is to find models that best describe the features of the given data. Therefore, the networks must be trained on *training data* as close to the real data as possible. However, it is problematic when a network trains too long on training data; thus, it only memorizes the training data and fails to generalize. Such a trained network performs poorly on unseen data, the *test dataset*. This phenomenon is called *overfitting*, and techniques that counteract overfitting are called regularization methods. On the other

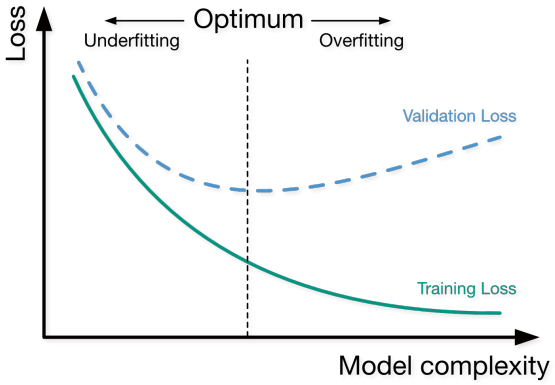


Figure 5.3.: Overfitting and underfitting of NN training in the context of model complexity and loss score.

hand, if a model is not fully trained, it has not learned all the features of the input data, resulting in poor predictions on new data, which is called *underfitting*.
A *validation dataset* is introduced in addition to the training and test datasets to monitor overfitting and underfitting. If the training error is significantly lower than the validation error, this indicates overfitting the network on the training data. Underfitting can be identified by a simultaneous decrease in the training and validation error compared to the previous epoch.
Both overfitting and underfitting depend strongly on the model complexity and the size of the data set, shown in Fig. 5.3. An optimal training condition between underfitting and overfitting is achieved with a minimal validation error.
There are several methods to avoid overfitting, such as introducing dropout layers or batch normalization [69, 70]. In PyTorch, both methods are enabled by default during training but disabled during the validation, evaluation, and prediction phases. As a result, PyTorch models may perform even better on validation and test data than on training data.

5.3. PyTorch Implementation

A basic machine learning (ML) workflow involves loading data, defining a model architecture, optimizing model parameters through training, and saving the trained model for further evaluation. In `PyTorch`, data for training and validation are loaded via `DataLoaders` [71], which wrap iterables over the dataset by processing the data in batches; in the following example with a batch size of 64. The batch size specifies the number of samples propagated simultaneously through the network, which affects training time, accuracy, and memory usage. `PyTorch DataLoaders` support automatic batch processing, adjusting the order of data loading, and loading data in one or more processes.

```
# Define DataLoaders
train_dataloader = DataLoader(training_data, batch_size=64)
val_dataloader = DataLoader(val_data, batch_size=64)
```

The network architecture is specified by a model definition and a forward pass that specifies how data is passed through the network. In this example, the `NeuralNetwork` class inherits from a `nn.Module` to create a simple sequential feed-forward network with a hidden layer with input dimension 100, hidden dimension 512, and output dimension 10. Data is passed through the network specified by the forward pass, which flattens the input data before passing the data through the model.

```
# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.simple_model = nn.Sequential(
            nn.Linear(100, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        return self.simple_model(x)

model = NeuralNetwork()
```

The model parameters are optimized based on a loss function and an optimizer, which are in this case the binary cross entropy (BCE) and the `adam` optimizer with a learning rate of 0.01 [72]:

```
# Define training configuration
loss = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
```

In a training loop, the model is put in train mode enabling `Dropout` layers and batch normalization. The model computes predictions based on the input data. A loss score between prediction and target vector is calculated, and the optimizer updates the model parameters by backpropagation.

```
#Define training loop
def train(train_dataloader, model, loss, optimizer):
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction error
        pred = model(X)
        loss = loss(pred, y)
```

```
        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

The validation loop is similar to the training loop with the difference that the model is put in evaluation mode, disabling `Dropout` layers, batch normalization, and backpropagation.

```
# Define validation loop
def validation(validation_dataloader, model, loss):
    model.eval()
    with torch.no_grad():
        for batch, (X, y) in enumerate(dataloader):
            pred = model(X)
            validation_loss = loss(pred, y)
```

Finally, the model is trained over several epochs, updating the model parameters with each epoch to obtain better predictions.

```
# Training
epochs = 20
for epoch in range(epochs):
    train(train_dataloader, model, loss, optimizer)
    test(validation_dataloader, model, loss)
```

5.4. Graph Neural Networks

An emerging subfield of deep learning (DL) is geometric deep learning (GDL), which generalizes deep neural models to non-Euclidean domains such as graphs, sets, and manifolds [21]. In particular, the field of GNNs as a central method of GDL has grown very rapidly in recent years, with applications in a variety of important open problems in particle physics [22]. Recent efforts have shown impressive performance of GNNs compared to classical DL methods, in particular in the field of charged particle tracking [1, 23, 24, 29]. Formally, a graph $G = (X, E, I)$ can be represented by a set of *objects* (O) called vertices or *nodes*

$$X = [\mathbf{x}_i] \in \mathbb{R}^{N_{\text{nodes}} \times D_{\text{nodes}}} \tag{5.6}$$

and another set of elements representing the *relations* (R) between two objects called *edges*

$$E = [\mathbf{e}_k] \in \mathbb{R}^{N_{\text{edges}} \times D_{\text{edges}}} \tag{5.7}$$

connected by an adjacency hit index pair list

$$I = [I_{r,k}, I_{s,k}] \in \mathbb{N}^{2 \times N_{\text{edges}}} \tag{5.8}$$

connecting receiving node r and sending node s by an edge k. The hit index pair list only consists of pairs of hit indices, and therefore its size only depends on the number of edges. Alternatively, the adjacencies can be encoded by two matrices of the form $\{0,1\}^{N_{\text{nodes}} \times N_{\text{edges}}}$, where one matrix encodes the incoming edges and the other matrix the outgoing edges, respectively. Ones indicate the incoming and outgoing edges, while all other matrix entries are zero. Hence, the matrices are filled sparsely with ones, which is highly inefficient in size, especially concerning implementation on heterogeneous hardware. Although the formulation based on a hit index pair list I via PyTorch Geometric (PyG) [73] is in theory equivalent to a matrix formulation defined via PyTorch [74], encoding the edge adjacency in I is much smaller and significantly reduces the resource consumption compared to the matrix formulation. Therefore, this implementation is significantly faster and more flexible than the matrix implementation [1].
A basic interaction network (IN) as proposed by [75] is defined as

$$IN(G) = \phi_A(m_2(\phi_O(a(G, \phi_R(m_1(G)))))) \tag{5.9}$$

with marshaling functions $m_{1,2}$ rearranging the nodes and edges into interaction terms, a relational model ϕ_R, aggregation function a, object model ϕ_O, and final output model ϕ_A. In general, an IN updates node and edge features iteratively by aggregating information derived from the node's neighborhood, also called messages.

5.5. Performance Metrics

Performance metrics are a crucial part of machine learning tasks to indicate the performance of an algorithm. In this thesis, only binary classification tasks are applied; therefore, the metrics discussed here refer exclusively to that case.
One important classification metric is the BCE loss between the target tensor y and the network output probabilities $\hat{y}$. The loss measuring the mismatch between target and network predictions with mean reduction and optional manual rescaling weight w_n is described as

$$\begin{aligned} l(\hat{y}, y) &= \text{mean}(L) = \text{mean}\left(\{l_1, \ldots, l_N\}^\top\right) \\ \text{with} \quad l_n &= -w_n \left[y_n \cdot \log \hat{y}_n + (1 - y_n) \cdot \log(1 - \hat{y}_n)\right] \end{aligned} \tag{5.10}$$

where the log functions are clamped to a minimum value of -100 to ensure finite loss values. Here, a low loss score indicates good classification performance.
Another essential metric is classification *accuracy* that describes the fraction of correct predictions, formally described as

$$\text{accuracy} = \frac{1}{N} \sum_{i}^{N} 1(y_n = \hat{y}_n) \tag{5.11}$$

where a high accuracy score indicates an overall good training performance. In addition to loss and accuracy, another important metric for binary classification is the confusion matrix, also called truth matrix, as shown in Fig. 5.4. The confusion matrix is used to compare true class values with classifier prediction values based on four cases:

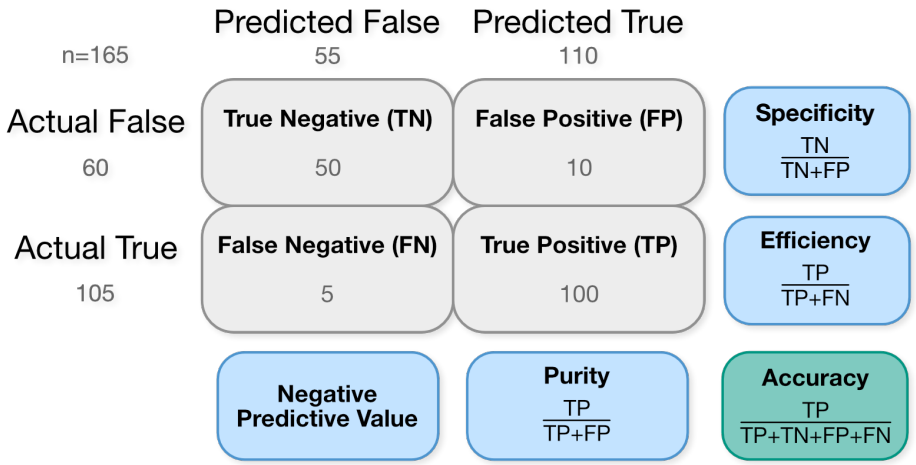


Figure 5.4.: Examplaric confusion matrix display with true negative (TN), true positive (TP), false negative (FN), and false positive (FP) based on different configurations of actual class and predicted label. In addition, different performance metrics such as purity, efficiency, specificity, negative predictive value, and accuracy are shown.

- true positive (TP): actual and predicted class indicate true
- false negative (FN): the actual class is true, but the classifier indicates false
- false positive (FP): the actual class is false, but the classifier indicates true
- true negative (TN): both actual and classifier indicate false

In a classification task the first and the last case frequencies TP and TN need to be maximised while minimising FN and FP.
The various relative frequencies of the confusion matrix are often used to determine additional metrics for the evaluation of the classifier. Purity and efficiency are two important metrics commonly used in high energy physics (HEP).
Purity, also called positive predictive value (PPV) or precision, indicates the proportion of correctly classified positive/true objects in the totality of results classified as positive. It is defined as

$$\text{purity} = \frac{\text{TP}}{\text{TP+FP}} \,. \tag{5.12}$$

Efficiency is the probability that a positive/true object is correctly classified as positive (signal (S)). In literature, efficiency is often termed true positive rate, sensitivity, recall, or hit rate. It is defined as the true positive rate (TPR)

$$\text{efficiency} = \text{TPR} = \frac{\text{TP}}{\text{P}} = \frac{\text{TP}}{\text{TP+FN}} \tag{5.13}$$

with positive count P = TP + FN.
Other two important measures are the true negative rate (TNR) (also termed specificity or

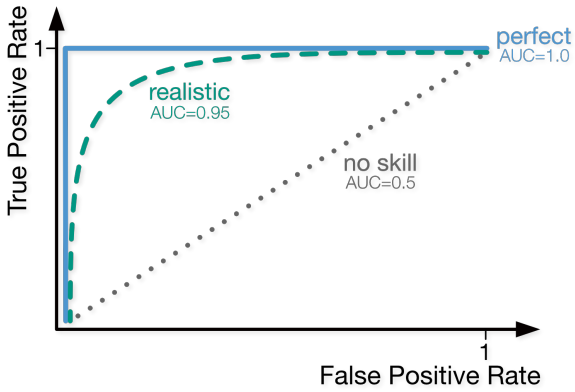


Figure 5.5.: Receiver operating characteristics (ROC) curve and area under the curve (AUC) score based on true positive rate (TPR) and false positive rate (FPR).

selectivity) and false negative rate (FNR) (also termed miss rate) defined as

$$\mathrm{TNR} = \frac{\mathrm{TN}}{\mathrm{N}} = \frac{\mathrm{TN}}{\mathrm{TN} + \mathrm{FP}} = 1 - \mathrm{FPR} \tag{5.14}$$

$$\mathrm{FNR} = \frac{\mathrm{FN}}{\mathrm{P}} = \frac{\mathrm{FN}}{\mathrm{TP} + \mathrm{FN}} = 1 - \text{efficiency} \tag{5.15}$$

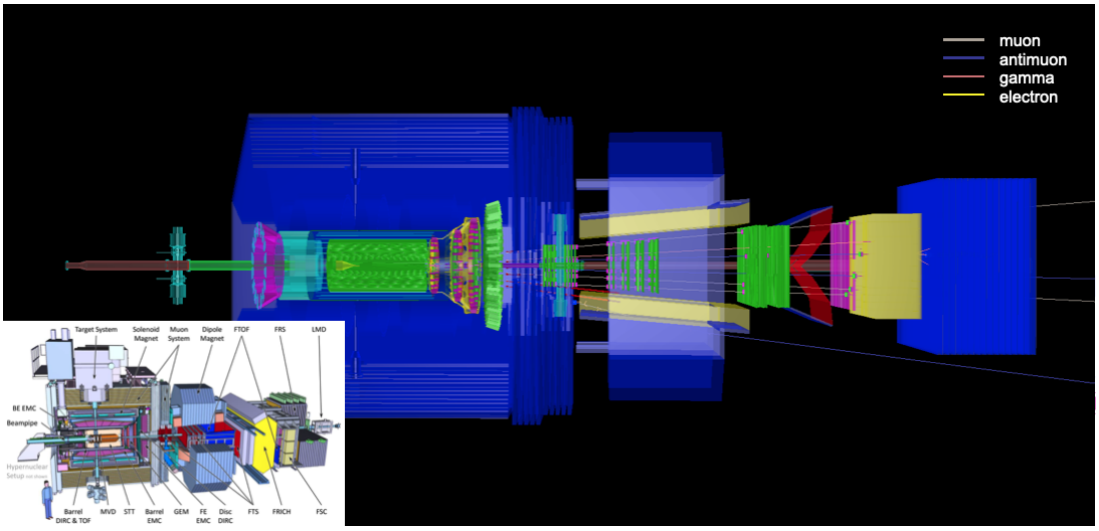
with false positive rate (FPR).
The last metrics covered in this section are the receiver operating characteristics (ROC) curve and the corresponding area under the curve (AUC) value displayed in Fig. 5.5. The ROC curve is based on the TPR and FPR. A perfect ROC curve initially rises vertically to a TPR of 1 at FPR=1 and remains at the level of TPR=1 by increasing FPR. The AUC is the area under the ROC curve and is ideally equal to 1.0. A ROC curve near the diagonal indicates that there is no significant learning process compared to a random process. The corresponding AUC is 0.5. A realistic, well-functioning network achieves AUC values close to 1. An AUC value below 0.5 indicates a misinterpretation of the data and indicates that the NN has not learned the real underlying features of the given data. The key difference and advantage of the ROC and AUC score compared to, for example, purity or efficiency is that no threshold parameter is required to distinguish between correct and incorrect prediction (usually in a range between 0 and 1) and the metric provides a performance measure for the entire threshold parameter space. In contrast, purity, efficiency, TNR or FNR can only be calculated for a specific parameter value.
Evaluating algorithms using complementary metrics is recommended since each metric focuses on different aspects of data and model predictions. In the following chapters, these performance metrics will be used to evaluate and optimize the data filtering and GNN-based edge classification.

6. PANDA FTS Training Data

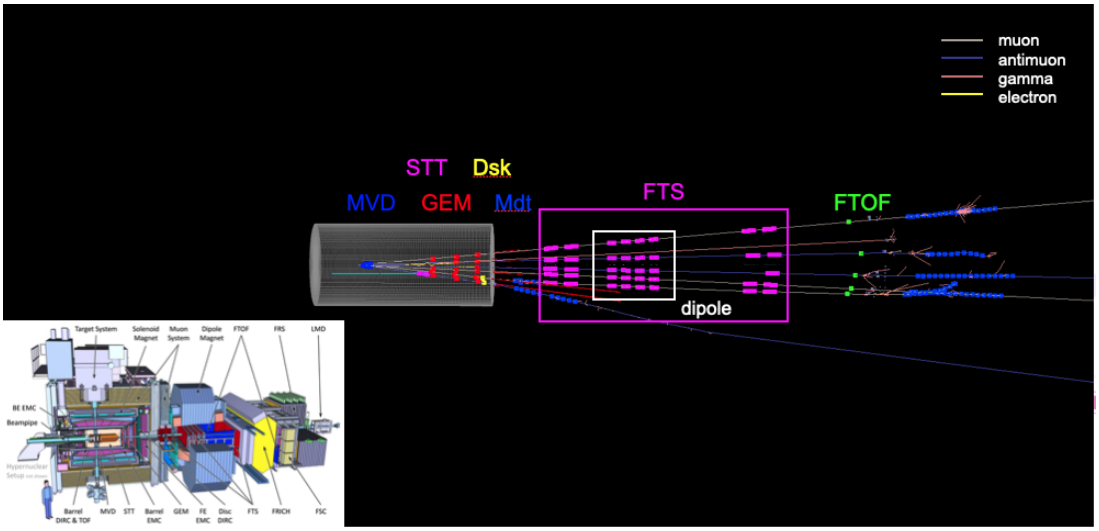
The aim of track finding, as explained in Section 4.5, is the correct identification of particle tracks traversing the detector by connecting detector hits via track segments and finally connecting the track segments to form full particle tracks. In Chapter 7, a graph neural network (GNN) based track finding algorithm is discussed. The preparatory work required for the GNN implementation is described in this chapter. It describes and discusses the methods used to generate, filter, and preprocess the anti-**P**roton **An**nihilation at **DA**rmstadt (PANDA) forward tracking system (FTS) data in Section 6.1 and 6.2. In Section 6.3, the hit graphs are created and evaluated as preliminary work for the GNN-based track finding in Chapter 7. Finally, Section 6.4 gives a summary of the applied data preprocessing and graph building filters and thresholds.

6.1. Data Simulation via PandaRoot

As described in Section 4.4, there is no real PANDA detector data yet, and simulations of the experiment are generated using the `PandaRoot` framework. For this study, a total of 30 000 events, each with three muons and antimuons, were generated using the `PandaRoot` simulation and digitization packages. Muons were selected as primary particles because muons generate few secondary particles as minimum ionizing particles (MIPs), providing clean and simple tracks for this study. In addition, antimuons were selected because they bend in the opposite direction of the oppositely charged muons in the magnetic field of the forward detector, providing crossing tracks. The 30 000 events with three muons and antimuons were generated with the Monte Carlo (MC) particle gun box event generator of `PandaRoot` in the forward direction of the detector in a full azimuthal angular range $\varphi = [0.2\pi]$ and a small polar angle $\theta = [0.5°, 10°]$, at a momentum in the range of $1\,\mathrm{GeV}/c$ to $10\,\mathrm{GeV}/c$ with uniform distribution within the ranges, respectively. A corresponding `PandaRoot` event display can be seen in Fig. 6.1. In Fig. 6.1(a), the event is shown with a full detector display to give an impression of the detector setup described in Section 4.3. A small image of the full detector is included for comparison. A large version of this image is displayed in Fig. 4.5 at a higher resolution. In Fig. 6.1(b), the same event as in Fig. 6.1(a) is displayed at the same size but without the detector, components to fully visualize the tracks of the traversing particles. The simulated interaction point between antiproton and target is on the left side, covered by the micro vertex detector (MVD). After the collision, the generated muons and antimuons are boosted in the forward direction of the detector (right) and pass through the forward detector of PANDA, which is shown by gray and blue lines, respectively. It can be seen that only a few secondary particles, such as photons (red) or electrons (yellow), are produced. The interaction of the particles with the detector components is represented by different colored squares corresponding to



(a) with detector display



(b) without detector display

Figure 6.1.: PANDA event display with three muons and three anti-muons in the forward direction with (a) and without (b) detector display. The full detector overview can be seen in Fig. 4.5.

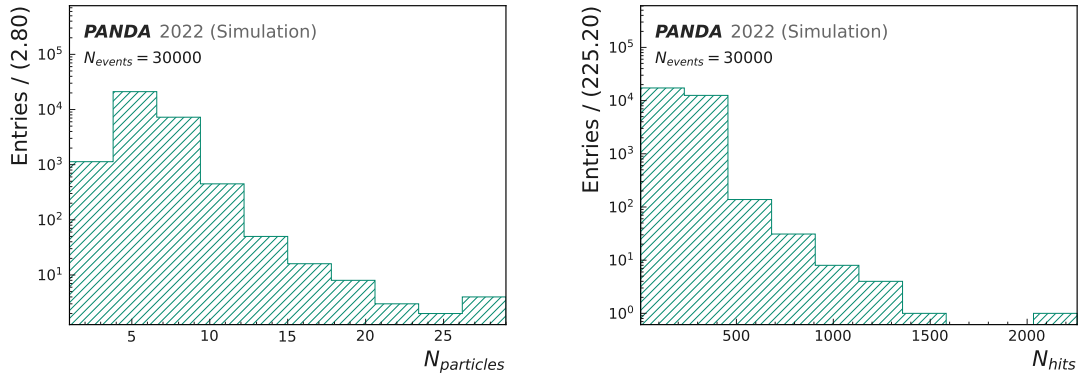
the different detector components. Each square represents a measurement point and the corresponding data that can be extracted. For example, the magenta squares correspond to hits in the drift chambers of the FTS. Parts of the FTS are inside a dipole magnetic field (white frame) to allow reconstruction of the track momentum based on the curvature of the particle track in the magnetic field.

6.2. Data Preprocessing

The following sections describe data import, exploration, and filtering on the way to graph encoding of the dataset.

6.2.1. Handling of ROOT Data Input

The input data is generated in the `ROOT` data structure [58], which is the main framework for data analysis in particle physics written in `C++` [76]. To access the data with `Python` [77], which is commonly used for machine learning tasks, `uproot` [78] is used. `Uproot` is a library designed to read `ROOT` files quickly and efficiently. It does not depend on `C++ ROOT` itself, but only requires `NumPy`, the most popular `Python` package for array handling [79].
The `ROOT` file trees contain information about the simulated particles and the digitized detector response. Only relevant features of the data are extracted and collected in a Pandas data frame [80], as the graphs are constructed with as little information as possible to keep the graph sizes small. These features include post-digitization information such as hit coordinates of anode wire positions, isochron radii, detector module IDs, and information on whether or not the detector plane is skewed (tilted by ±5°). In addition, the MC truth coordinates, 3-moments, and track IDs are also stored in the data frame. The dataset is based on a right-handed Cartesian coordinate system, with the z axis aligned along the beam axis and the x and y axes defining the transverse plane: the x axis is oriented horizontally, and the y axis points upward.
The digitized features only contain information about the x and z coordinates but not about the y coordinates since straw tube detectors can only provide two-dimensional information about their wire positions. Skewed layers solve this problem by adding a degree of freedom to the data, which can then be used to calculate the missing dimension. For simplicity, only detector hits in the vertical layers are used in this work. In future projects, the information from the skewed layers can be used to reconstruct 3D track candidates as described by Esmail [29]. The MC truth information is the underlying event generation information and is only available in the simulation but not in real experiments. The MC-truth track ID information is used as the truth label on which the neural network trains. The MC-truth coordinates, especially in y direction, and the moments can be used to study the performance of a full track finding and reconstruction at a later stage and are therefore kept for future studies.



(a) (b)

Figure 6.2.: Distribution of the number of particles (left) and the number of detector hits per event (right).

6.2.2. Data Exploration and Filtering

The next step is to analyze and preprocess the generated data before the graph building as input for training. Figure 6.2 shows the number of particles and hit distributions of the unprocessed data with $N_{\text{events}} = 30\,000$ events. The raw `PandaRoot` data contain an average of 5.8 particles and tracks interacting with the detector per event giving an average total number of 221 hits per event in the FTS. More particles are producedO than just the three primary muons and antimuons, as they produce secondary particles by interacting with the detector parts. There is also a fraction of events with fewer than six particles or tracks, which is smaller than the number of primary particles produced. In some events, fewer particles than six particles can be measured since not all generated particles must be detected in the FTS, and only particles interacting with the detector units are considered. A corresponding event display of the unprocessed data in the xz-plane is shown in Fig. 6.3. The particle hits are represented by points surrounded by circles representing the isochron radii by the circle size scaled by a factor of 200 for better visibility. In the x direction, there are 48 detector layers, 8 for each detector chamber. The layers are arranged in closely spaced double layers so that two closely spaced layers appear as one layer in the event display. The arbitrary event ID 3 was chosen because in this event, in addition to the primary particles, a secondary particle with low momentum, an electron (magenta), is detected, leaving a looping track in the detector called a *curler*.
In the event shown, all six primary particles (MC particles 0 to 5) traverse the entire FTS detector, leaving hits in all six chambers. The MC particle IDs here do not correspond to the commonly used particle data group (PDG) particle codes but represent a sequential numbering scheme of the particles involved. The curler, which is probably an electron, has the assigned MC particle ID 566. This ID does not appear to be consecutive concerning the other particles. One possible explanation is that many secondary particles do not interact with the detector units and therefore are not part of the dataset. Thus, at least 560 undetected secondary particles are involved in this example event.

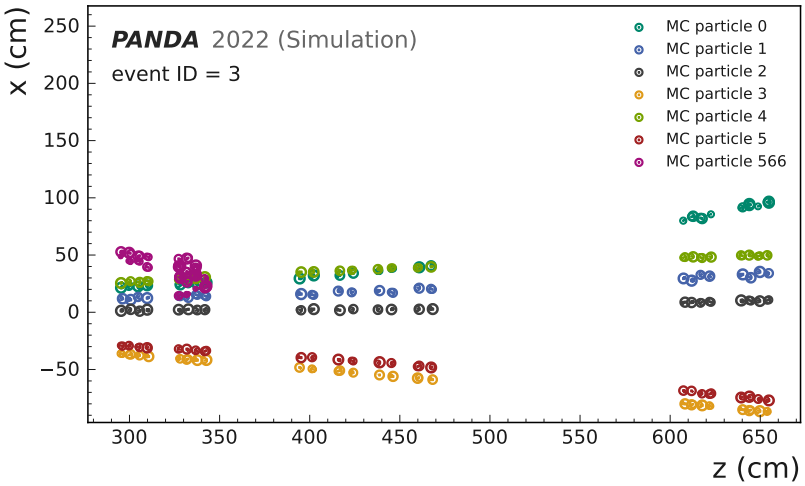
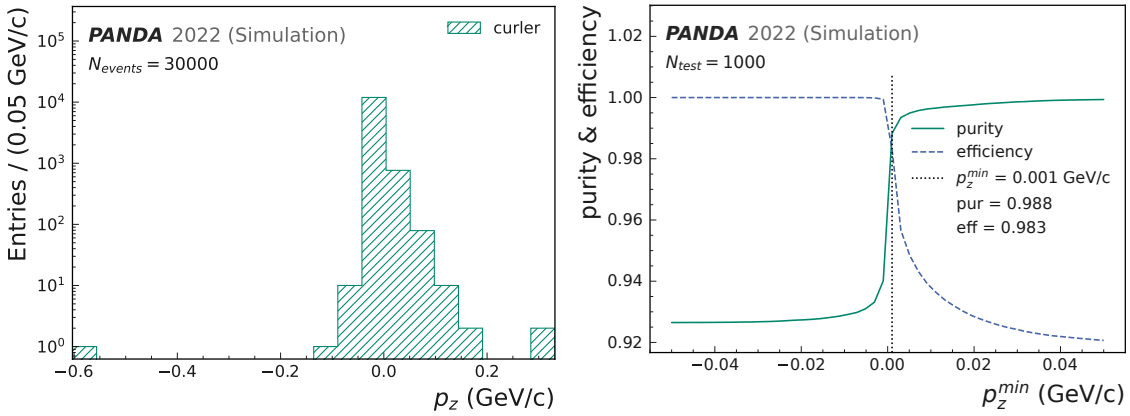


Figure 6.3.: Event display in xz-plane with six primary particles (MC particle 1-5), i.e., muons and antimuons and one secondary particle MC particle 566 (curler).



(a) (b)

Figure 6.4.: Application of minimum p_z threshold on the dataset. (a) Average forward momentum p_z per particle distribution of curling tracks at low momenta. (b) Purity and efficiency estimates after application of minimum p_z threshold. At the intersection point between purity and efficiency $p_z^{\min} = 0.001\,\mathrm{GeV}/c$ the purity reaches 0.988 and efficiency 0.983.

The raw data must be preprocessed to obtain a clean data sample for further steps. Therefore, a set of selection criteria is applied to the data set. The following filters can be considered as idealized hit filtering, modulating the number of hits per graph to make the data more feasible for the following application on GNN-based track segment classification. In the future, these filters will need to be reduced or removed to provide more realistic hit filtering for real-world applications.
Since only the vertical 2D layer information is used at this point, the first step is to remove

all hits from skewed layers from the dataset and revert the remaining non-consecutive layer IDs to consecutive layer IDs.
In the next step, curlers must also be removed from the dataset since curler tracks are difficult to classify by the neural network, and, in principle, only hits belonging to primary particles with higher momentum are of interest for track reconstruction. A natural discriminating variable to remove curlers from the dataset is the momentum in the forward direction p_z, which is extracted from the MC-truth information. In Fig. 6.4(a), the distribution of the average particle momentum p_z in the forward direction of curling tracks is presented on a logarithmic scale. Here, particles that leave at least one hit in a lower layer than the previous hit layer ID are called curlers. In this case, curlers make up only a small fraction of 7 % of the entire data set. It can be seen that the secondary particles are generated with a low momentum below 1 GeV/c or even with a negative momentum, while the primary particles have been generated in `PandaRoot` with a uniformly distributed momentum in the range of 1 GeV/c to 10 GeV/c.
Curler removal can now be achieved by applying a threshold to p_z that rejects particle tracks with an average momentum p_z below a minimum threshold p_z^{min}. Both purity and efficiency are determined with respect to the threshold for a test sample of $N_{\text{test}} = 1\,000$ displayed in Fig. 6.4(b). Here, purity and efficiency are defined as

$$\text{purity} = \frac{N_{\text{no-curler>threshold}}}{N_{\text{all tracks>threshold}}}, \tag{6.1}$$

$$\text{efficiency} = \frac{N_{\text{no-curler>threshold}}}{N_{\text{no-curler}}} \tag{6.2}$$

with the total number of non-curling tracks $N_{\text{no-curler}}$, the fraction of non-curling tracks greater than the threshold value $N_{\text{no-curler>threshold}}$, and the total number of tracks passing the threshold $N_{\text{all tracks>threshold}}$. It is easy to see that there is a trade-off between removing curlers and maintaining non-curling tracks: Increasing the threshold significantly improves purity and decreases efficiency, and vice versa. A natural discrimination choice between the two is the intersection of purity and efficiency, which in this case corresponds to applying a threshold of $p_z = 0.001\,\text{GeV}/c$. At this threshold, total purity of 0.988 and efficiency of 0.983 are achieved. This corresponds to the removal of 85.4 % of curlers (true negative rate (TNR)) at a loss of only 1.7 % of non-curlers (false negative rate (FNR)). In addition to the threshold p_z^{min}, a same-layer filter is applied to remove duplicate hits of a particle that leave multiple hits in the same layer.
One important thing to note is that the MC generation yields particle hits in a fixed order along the MC particle tracks. For example, the first 24 hits of the dataset belong to particle 0, the next 24 to particle 1, etc., depending on the number of hits each particle generates in the simulated detector. If this effect is not considered, this order will be maintained during graph building, which causes the network to learn this order instead of the actual graph features, resulting in significant overfitting. This problem can be avoided by assigning new random values to the hit IDs. Previous work did not account for this fact [29].

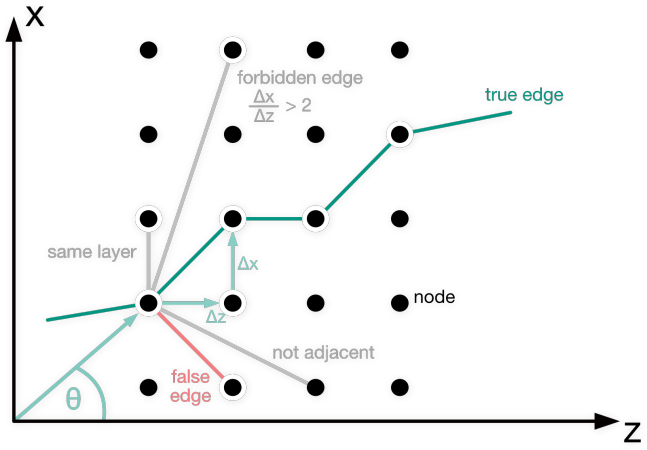
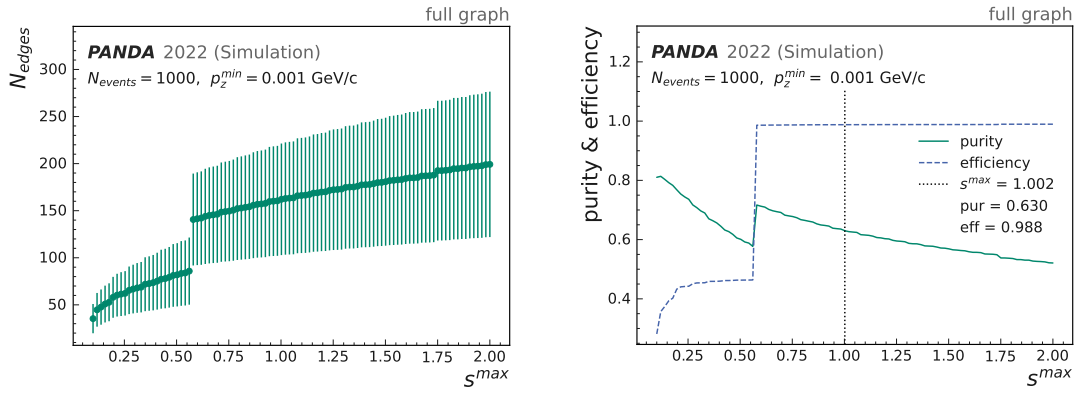


Figure 6.5.: Simple graph illustration with nodes connected by true, false, and forbidden edges. A true edge corresponds to a true particle track segment, whereas a false edge does not. A forbidden edge does not fulfill all geometric filter conditions: same-layer filter, only adjacent layer edges, or only connecting two nodes at a smaller slope than a slope threshold allows for, in this case, $s^{\max} = 2$.

6.3. Graph Building

After the initial data filtering, the hits of the dataset can be connected into *hit graphs* by a graph building algorithm based on geometric constraints to obtain a representation that provides the most likely representations of the true track segments. In the following, hits are referred to as *nodes* and connections between pairs of nodes as *edges*. The edge construction algorithm creates edges between all possible pairs of nodes through edge attributes $e_k = [\Delta x_k, \Delta z_k]$ and a hit index matrix $I = [I_{r,k}, I_{s,k}]$ that links a pair of nodes consisting of a receiver r and a sender s node with corresponding edge attributes, see Chapter 5 for more details. In this work, node attributes $X = [x, z]$ and edge attributes $e_k = [\Delta x_k, \Delta z_k]$ contain only x and z features. For the studied problem in this work, training even based on these two PANDA FTS features leads to similar results as training with more features such as the polar angle $\theta = \arctan \frac{x}{z}$, the isochron radius, and the radius $r = \sqrt{x^2 + z^2}$. A simple schematic representation of the graph structure is shown in Fig. 6.5. The nodes are connected by edges, where real edges correspond to real track segments while false edges do not. Several geometric constraints reduce the number of allowed edges. The filter conditions include the same-layer filter, connecting only adjacent layer edges or only connecting two nodes at a smaller slope than a slope threshold allows for, in this case, $s^{\max} = 2$. The graph building conditions are explained in more detail in the following.

A fully connected graph contains a total of $n_{\text{edges}} = \frac{1}{2} n_{\text{nodes}}(n_{\text{nodes}} - 1)$, which would result in a size of over 24000 edges for 220 nodes. This size is far too large for the implementation on field programmable gate array (FPGA). Furthermore, this large size can also lead to extremely inefficient GNN training due to a large imbalance between real and fake edges. In principle, there is a fundamental trade-off between minimizing the number of edges



(a) (b)

Figure 6.6.: (a) Edges per event for different slope thresholds $s^{\max}$. (b) Analysis of edge purity and efficiency over an upper slope threshold $s^{\max}$ at the graph building level. For example, at a threshold value of $s^{\max} = 1.002$ edge purity reaches a value of 0.630 and efficiency a value of 0.988, respectively.

and keeping as many real edges as possible.
Only nodes of adjacent layers are connected by edges, and connections between identical layers are omitted to keep the number of edges manageable. In addition to restricting the graph structure to only connecting adjacent layer nodes, other geometric constraints must be applied. A possible choice is the dimensionless variable slope $s = \frac{\Delta x}{\Delta z}$ in the non-equidistant node lattice, which describes the steepness of edges concerning the beam direction. Applying a threshold to the slope that excludes edges above an upper threshold $s^{\max}$ affects the number of graph edges.
In Fig. 6.6(a), the average number of edges at different slope thresholds $s^{\max}$ is plotted for $N_{\text{test}} = 1\,000$ test sample events generated at a minimum threshold $p_z^{\min} = 0.001\,\text{GeV}/c$. The number of edges varies greatly for different events as the number of connectable nodes also varies greatly. Hence, the large error bars indicate the broad distributions. As the threshold value increases, the number of edges also increases. For example, for a slope threshold of $s^{\max} = 0.6$, the average number of edges per event is 140, while for a threshold of $s^{\max} = 2.0$ it is 197. It is also interesting to note the region around $s^{\max} = 0.6$, where the number of edges increases dramatically. In this work, the underlying `PandaRoot` data is generated uniformly concerning the polar angle θ. At $s^{\max} = 0.6$, connections are created between hits originating from different particle tracks, while below that, mostly only hits from the same particle are connected. Therefore, enabling connections between hits originating from different particles highly increases the number of possible connections. The slope threshold $s^{\max}$ is a promising parameter to reduce the number of edges and will be further analyzed in the following. In Fig. 6.6(b) graph construction efficiency and purity are calculated for applying slope thresholds between 0.1 and 4 using $N_{\text{events}} = 1\,000$

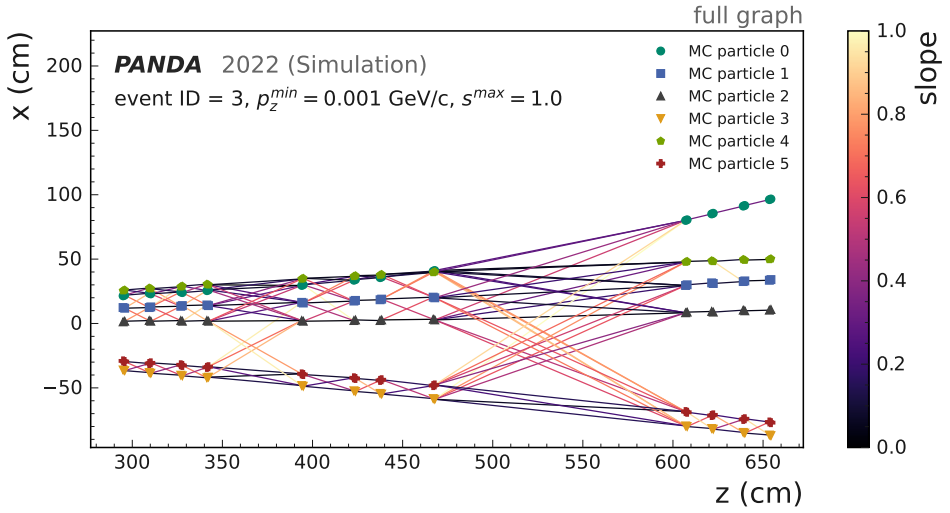


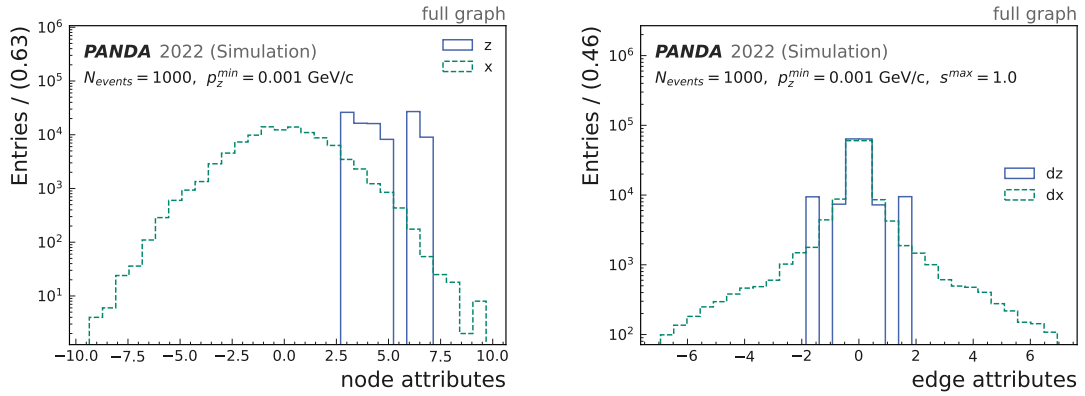
Figure 6.7.: Example of a hit graph as GNN input. Colored points represent detector hits (nodes) of corresponding particles distinguished by the different particle IDs, and the lines represent all generated edges after filtering. The edge color represents the associated slope feature up to $s^{\max} = 1.002$. A true track connects hits associated with the same particle.

random graph samples from the dataset. Here, purity and efficiency are defined as

$$\text{purity} = \frac{N_{\text{true edges<threshold}}}{N_{\text{edges<threshold}}}, \tag{6.3}$$

$$\text{efficiency} = \frac{N_{\text{true edges<threshold}}}{N_{\text{true edges}}}. \tag{6.4}$$

with the total number of true edges $N_{\text{true edges}}$, the fraction of true edges smaller than the threshold value $N_{\text{true edges<threshold}}$ and the total number of edges passing the threshold $N_{\text{edges<threshold}}$. Again, there is a trade-off between purity and efficiency: as purity increases, efficiency decreases, and vice versa. An interesting area in the graph is the large step at $s^{\max} = 0.6$. At this point, the efficiency reaches its maximum value, i.e., no more true edges are added to the graphs, but the purity increases as more false edges are included at higher thresholds. For training, it is useful to consider also the connections between different particles and not only the edges between the same particles. Therefore, rather than a threshold of 0.6, a threshold of $s^{\max} > 0.6$ is chosen. For example, with a threshold of $s^{\max} = 1.002$, edge purity reaches a value of 0.630 and efficiency reaches a value of 0.988, which corresponds to removing 98.8 % of false edges (TNR) while rejecting only 1.2 % of true edges (FNR). The threshold $s^{\max} = 1.002$ provides a good balance between the proportions of true and false edges since the interaction network (IN) architecture benefits from less-pure graphs, which corresponds to higher edge connectivity.
The final graphs consist of a 2-dimensional node attribute matrix $X = (x_i, z_i) \in \mathbb{R}^{N_{\text{nodes}} \times 2}$, a 2-dimensional edge attribute matrix $E = (\Delta x_{k,ij}, \Delta z_{k,ij}) \in \mathbb{R}^{N_{\text{edges}} \times 2}$, an adjacency node index pair list $I = [I_{i,k}, I_{j,k}] \in \mathbb{N}^{2 \times N_{\text{edges}}}$ and a target vector $y \in \{0, 1\}^{N_{\text{edges}}}$ that contains true



(a) loss (b) accuracy

Figure 6.8.: Dimensionless distribution of rescaled node (a) and edge attribute (b) dimensions with a logarithmic scale. All features are in the same order of magnitude.

or false edges with a proportion of 59.6 % belonging to true edges and 40.6 % belonging to false edges. An example graphical representation of a complete graph is shown in Fig. 6.7 for the same event ID as in Fig. 6.3. Edges connecting the different particles according to the filter conditions described above are shown. It can also be seen that the curling particle originally present in the dataset has been removed from the graph. The slope value of each edge, ranging from 0.0 to 1.0, is colored using the color map `magma`: Black represents a minimum slope of 0.0, yellow represents a maximum slope of 1.0, and purple to orange represents all values in between. It can be seen that edges connecting nodes of the same particles usually have a small slope, while connections between different particles are built with a larger slope.

A final adjustment to the node and edge features must be made to prepare the graph data for training. The node and edge features are of different value ranges compared to each other. Machine learning classification algorithms do not work properly without data normalization. Therefore, the features are normalized using the factors 20 for x and 100 for z to rescale the data to the same order of magnitude. This is a rough data normalization method but sufficient in the context of machine learning (ML). The rescaled node and edge attributes that meet the discussed criteria are plotted in Fig. 6.8 with a logarithmic scale. The x node features are symmetrically distributed between -10 and 10, and the z features are distributed between 2.7 and 7.1. The point (0,0) corresponds to the interaction point of the proton-antiproton collision on the beam axis covered by other detector units. The gap at 6.0 corresponds to the gap between chambers FT4 inside the dipole field and FT5 after the dipole field. The boundary attributes for both Δx and Δz are symmetrically distributed about the origin and cover a range from -7 to 7.

Table 6.1.: Summary of preprocessing and graph building filters, thresholds, and graph encoding.

Filter/ Preprocessing	
Skewed Layer Removal Same-Layer Filter Hit Reordering	
Thresholds	**Threshold Value**
$p_z^{\min}$	$0.001\,\mathrm{GeV}/c$
$s^{\max}$	1.002
Graph Encoding	**Tensors**
Node Attributes	$X = (x_i, z_i) \in \mathbb{R}^{N_{\text{nodes}} \times 2}$
Edge Attributes	$E = (\Delta x_{k,ij}, \Delta z_{k,ij}) \in \mathbb{R}^{N_{\text{edges}} \times 2}$
Adjacency Matrix	$I = [I_{i,k}, I_{j,k}] \in \mathbb{N}^{2 \times N_{\text{edges}}}$
Target Vector	$y \in \{0, 1\}^{N_{\text{edges}}}$

6.4. Preprocessing and Graph Building Summary

This chapter described and discussed the methods for generating, filtering, and preprocessing the PANDA FTS data and the generation of the hit graphs. A summary of the preprocessing steps and filters applied and the structure of the graph features are given in Table 6.1. In Chapter 7, the graphs generated and evaluated in this chapter are used for a GNN-based track segment classification.

7. Graph Neural Network Based Track Finding

This chapter describes and discusses a graph neural network (GNN)-based track finding algorithm using the graphs constructed from anti-**P**roton **An**nihilation at **DA**rmstadt (PANDA) forward tracking system (FTS) data described in Chapter 6. This chapter is structured as follows. Section 7.1 describes the GNN network architecture that is implemented in Section 7.2 for the track segment classification together with its performance evaluation. Section 7.3 focuses on the segmentation of the graph into multiple sub-graphs, which provides several advantages. Then, the GNN outputs for both the full graph and segmented graph approaches are used for a tracklet finding analysis in Section 7.4. Finally, Section 7.5 summarizes the GNN-based track segment and tracklet finding results on PANDA FTS data.

7.1. Interaction Network Architecture

The interaction network (IN) used in this work as proposed by [2] will be described in more detail in the following. It takes the form of three basic blocks: an edge block, computing edge feature updates, a node block updating the node features, and a final output edge block. A graphic representation of the complete corresponding forward pass is depicted in Fig. 7.1.
The first edge (relational) block $\phi_{R,1}$ computes edge feature updates $\tilde{e}_{rs}$, also called messages

$$\tilde{e}_{rs} = \phi_{R,1}(m_1(G)) = \phi_{R,1}(x_r, x_s, e_{rs}) \tag{7.1}$$

based on a message function m_1 concatenating the graph inputs where x_r and x_s are the input features of a receiver and sender node, respectively, e.g., coordinates of the corresponding hit and e_{rs} is the set of input edge features containing relational quantities between receiver and sender node, e.g., distances between two hits. This edge block is based on a relational model consisting of a simple sequential neural network (NN) model with two hidden layers, each with rectified linear unit (ReLU) activation function.
The generated messages are subsequently aggregated to map relational edge information to node-level outputs. There are multiple options for aggregation operations, including summation, element-wise mean, maximum, and minimum, which must be invariant to permutations of inputs to be generally applicable to unordered graph data. Here, the most simple case, a summation over all corresponding connected nodes, $\mathcal{N}(s)$ of node s was chosen:

$$\bar{e}_r = \sum_{s \in \mathcal{N}(r)} \tilde{e}_{rs} \ . \tag{7.2}$$

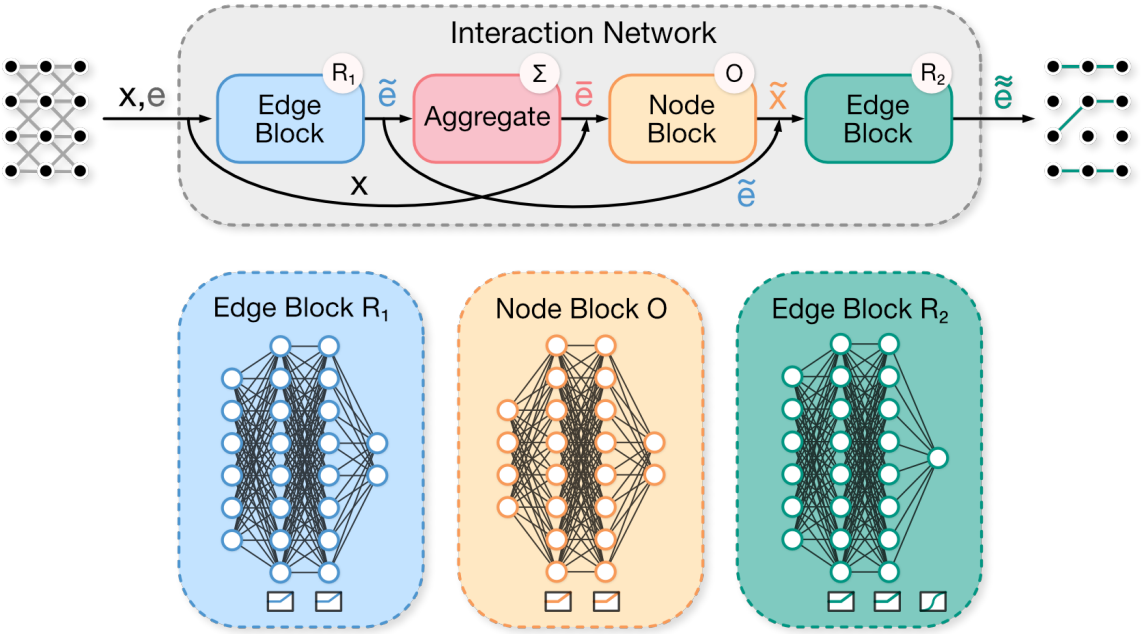


Figure 7.1.: The complete IN forward pass with edge blocks $R_{1,2}$, edge-aggregation Σ, and node block O. The graph node features x and edge features e are passed through the IN blocks R_1, O and R_2 with intermediate edge-aggregation by sum to compute one-dimensional edge weight predictions.

These two operations, edge block and aggregation, are known as the message-passing step. The output of the message-passing step $\bar{e}_r$ as well as the corresponding node features x_r are used by a node block in order to compute an update for each object (node), taking into account previous node features and one iteration of message passing among neighboring nodes

$$\tilde{x}_r = \phi_O(x_r, \bar{e}_r) \ . \tag{7.3}$$

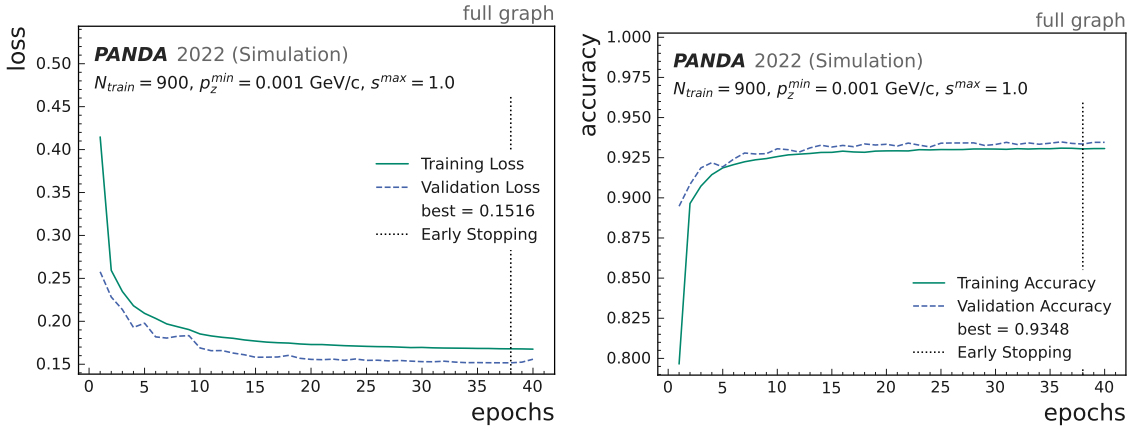
The edge block based on an object model is a sequential model with one hidden layer and a ReLU activation function. In principle, the steps described by Eq. (7.1) to Eq. (7.3) can be applied repeatedly in sequence.
The output block is a second edge block R_2 similar to the first edge block, which is applied on the re-embedded graph with updated node and edge features concatenated by a marshall function m_2

$$\tilde{\tilde{e}}_{rs} = \phi_{R,2}(m_2(G')) = \phi_{R,2}(\tilde{x}_r, \tilde{x}_s, \tilde{e}_{rs}) \tag{7.4}$$

computing a final one-dimensional edge weight output using a sigmoid as the final activation function such that the resulting edge weights $\tilde{\tilde{e}}_{rs}$ can be interpreted as probabilities that a specific edge is a track segment. The corresponding training target is the vector $y \in \{0,1\}^{n_{\text{edges}}}$ where 1 equals an edge corresponding to a true particle track segment and 0 otherwise.

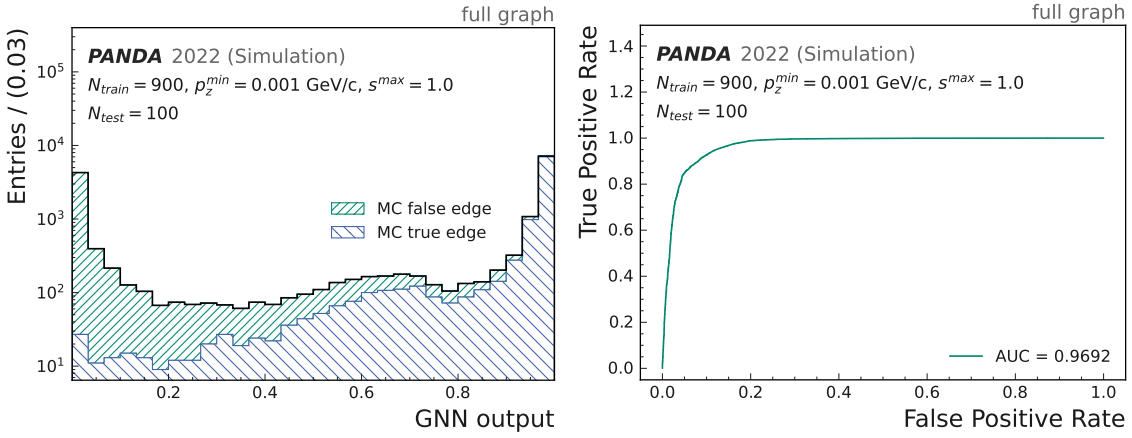
7.2. GNN Edge Classification

After discussing data preprocessing and graph construction, this section addresses the application of a GNN to the generated graph data that classifies track segments based on their associated detector hits. The network architecture used is an IN, as described in Section 7.1.
In the following studies, models are trained on full graphs created with $p_z^{\text{min}} = 0.001\,\text{GeV}/c$ and $s^{\text{max}} = 1.0$. As discussed by [1], the classification accuracy deteriorates when p_T^{min} is smaller than $1\,\text{GeV}/c$, which corresponds to p_z^{min} in this work. However, also low p_z tracks are included to cover a maximum of all true particle tracks. Hence, the associated lower classification accuracy must be taken into account. From the total graph dataset of approximately 30 000 events, 1000 random samples corresponding to a total number of about 200 000 edges are drawn and randomly divided into 80 % training, 10 % validation, and 10 % test datasets. Thus, a total of $N_{\text{train}} = 900$ events within GNN training and $N_{\text{test}} = 100$ events are used to analyze network performance on test data.
For stochastic optimization, the `Adam` optimizer is used, a commonly used first-order gradient-based optimizer based on adaptive estimates of lower-order moments [72]. The model is trained for 40 epochs with a batch size of 1 since each event contains an average of 160 trainable edges. In addition, an early stopping condition is applied to stop training if the validation loss does not decrease over 10 epochs. The GNN is trained to reduce the binary cross entropy (BCE) loss between the truth targets $y_k = 0, 1$ and the edge weights $w_k \in (0, 1)$. The hyperparameters learning rate lr, learning rate decay γ and period of learning rate decay (step size) are optimized using the next generation hyperparameter optimization framework `optuna` [81]. The optimal configuration identified is $lr = 6.55 \cdot 10^{-4}$, $\gamma = 0.86$, and step size = 3.
The average epoch loss and average epoch accuracy, as defined in Section 5.5, are monitored during training and plotted in Fig. 7.2. The loss and accuracy curves show that the model trains smoothly on the dataset without overfitting and reaches early convergence after 37 epochs. The best average batch loss is 0.1516, and the best accuracy is 0.9348. After training, the model with the lowest loss is saved and used for further steps.
In Fig. 7.3, two additional performance graphs are shown that evaluate the trained GNN on the test dataset of $N_{\text{test}} = 100$ samples. Fig. 7.3(a) shows the results of the GNN training: the stacked distribution of edge weight prediction distinguishing between true and false edges with a logarithmic scale. In a perfect classification, all true edges would be at 1, and all false edges would be at 0. As expected, the number of false edges decreases at higher output values while the number of true edges increases. A logarithmic scale was chosen because the range between 0 and 1 would otherwise be difficult to distinguish from 0 by eye. In Fig. 7.3(b), the receiver operating characteristics (ROC) curve is drawn based on the true positive rate (TPR) and false positive rate (FPR) of the classification. An ideal ROC curve falls in the upper left corner of the graph with TPR=1 and FPR=0 and a corresponding ideal area under the curve (AUC) of 1. Here the value is 0.9692.
Furthermore, edge classification purity and efficiency estimates based on different edge weight thresholds δ are displayed in Fig. 7.4(a). Here, edge classification purity and



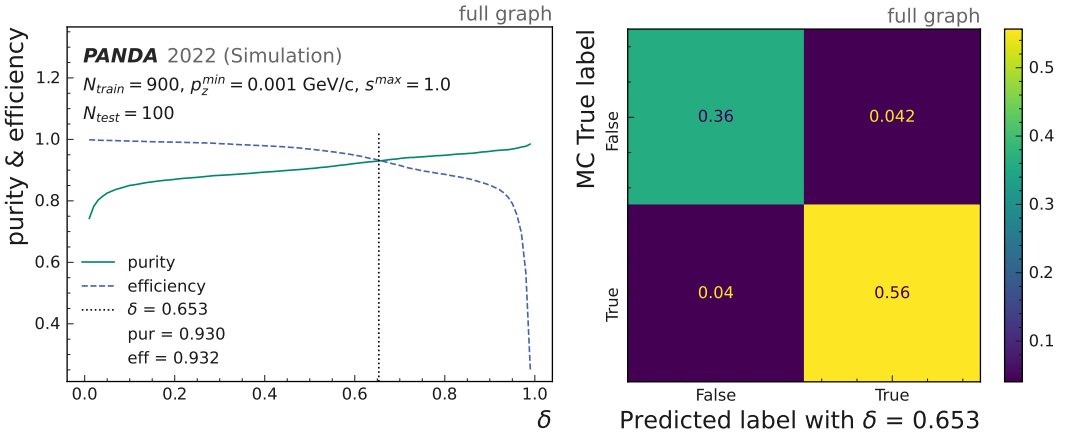
(a) loss (b) accuracy

Figure 7.2.: Average loss (BCE) and classification accuracy as a function of training epochs calculated on a training and validation dataset.



(a) GNN output (b) ROC curve

Figure 7.3.: (a) GNN edge weight output for true and false edges. (b) ROC curve evaluated on the test dataset with AUC value 0.9692.



(a) purity and efficiency (b) confusion matrix

Figure 7.4.: (a) Purity and efficiency estimates measuring the edge classification performance of a GNN trained on $N_{\text{train}} = 900$ full graph events and tested on $N_{\text{test}} = 100$ events for different classification threshold values δ. The graphs are built with the thresholds $p_z^{\text{min}} = 0.001\,\text{GeV}/c$ and $s^{\text{max}} = 1.0$. (b) Corresponding confusion matrix to (a) displaying true positive (TP), true negative (TN), false positive (FP), and false negative (FN) values based on a classification threshold $\delta = 0.653$. The total misclassification is 0.082.

efficiency are defined as

$$\text{purity} = \frac{N_{\text{true edges}>\delta}}{N_{\text{edges}>\delta}}, \tag{7.5}$$

$$\text{efficiency} = \frac{N_{\text{true edges}>\delta}}{N_{\text{true edges}}}. \tag{7.6}$$

with the total number of true edge segments $N_{\text{true edges}}$, the fraction of true edges greater than the threshold value $N_{\text{true edges}>\delta}$ and the total number of edges passing the threshold $N_{\text{edges}>\delta}$. Purity and efficiency show relatively high values above 0.8, almost over the entire threshold range. They intersect at an edge weight threshold of $\delta = 0.653$, corresponding to purity of 0.930 and efficiency of 0.932, respectively. The intersection at which purity equals efficiency seems to be a suitable selection criterion to distinguish false from true edges. Applying this threshold yields the metric estimates, which are presented as a confusion matrix in Fig. 7.4(b). Here, 36 % of all edges are correctly labeled as false track segments and 56 % as true track segments. Only 4 % of the false edges are misclassified as track segments, and 4.2 % misclassified vice versa.
Finally, Fig. 7.5 shows a complete example output graph corresponding to the input graph shown in Fig. 6.3 and Fig. 6.7. True tracks are represented by edges between hits associated with the same Monte Carlo (MC) particles. The color of the edges using the `viridis` color map shows the GNN output between 0 (false track segment), represented by yellow, and 1 (true track segment), represented by dark blue. Most false edges between hits from different particles are correctly classified with values close to 0.0 (yellow), while the connections

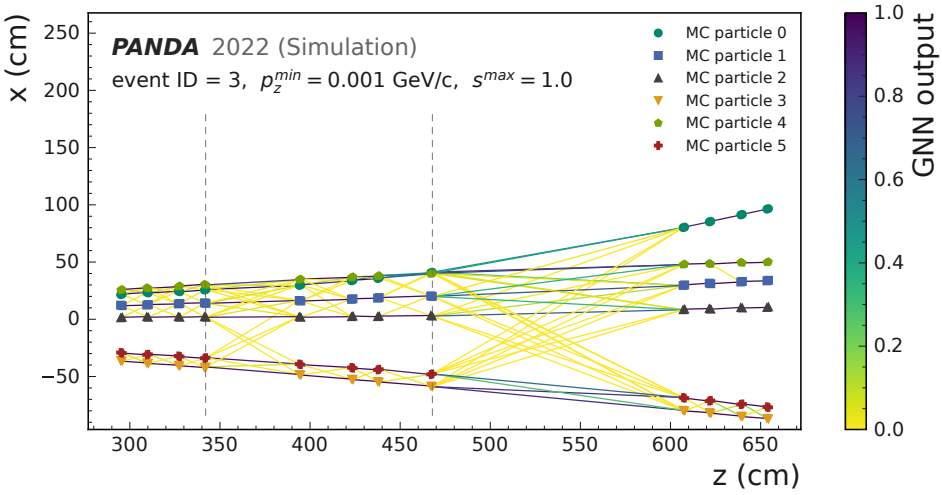


Figure 7.5.: GNN training output graph corresponding to the input graph shown in Fig. 6.7. Colored points represent detector hits (nodes) of corresponding particles distinguished by the different particle IDs. The connecting lines represent the estimated output edge weights in `viridis` colormap, where yellow represents low track segment probability and dark blue high probability, respectively. The graph segmentation discussed in Section 7.3 is indicated by gray dashed lines.

between hits originating from the same particle track are mostly classified with values close to 1.0 (dark blue). Only the long connections between the last hits of FT4 and the first hits of FT5 are classified with low confidence, indicated by green connections. Overall, edge classification seems to be successful in distinguishing between true and false track segments. The next step will repeat the graph building and edge classification steps for a segmented version of the full event graph.

7.3. Graph segmentation

The main goal of this thesis is to implement the GNN-based classification of track segments on field programmable gate arrays (FPGAs). Despite the promising performance of the GNN-based classification of track segments discussed in the last section, the implementation on FPGAs discussed in Chapter 9 must take into account that the computational resources on FPGAs are limited. The graph size considered in the previous section does not satisfy the resource constraints for a throughput-optimized FPGA implementation. To limit the graph size to a manageable level for resource-constrained environments such as FPGAs, the full event graphs discussed in Section 6.2 and Section 7.2 are segmented as discussed in [2], and all previously described steps are repeated for the segmented graphs. For the segmentation of the complete graph, it is convenient to divide the graph into three parts: The first part includes the eight detector layers corresponding to FT1 and FT2 in front of the dipole field, and the second part includes the nodes and edges inside the dipole field (FT3 and FT4) including the long edges between FT2 and FT3. The third part includes

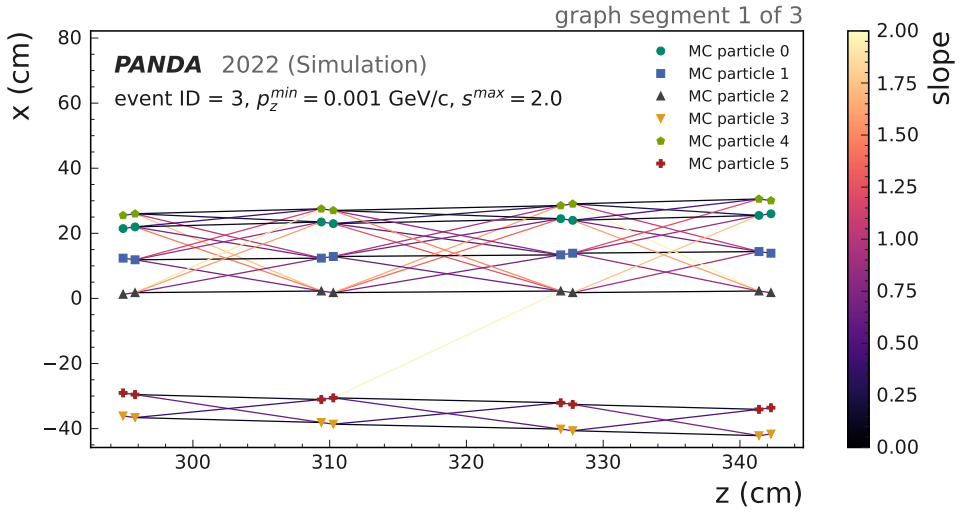


Figure 7.6.: Hit graph display of the first third of the segmented graph. Colored points represent detector hits (nodes) of corresponding particles distinguished by the different particle IDs, and the connecting lines represent all generated edges after filtering. A true track connects hits associated with the same particle. The edge slope is displayed by color from 0 (black) to 2. (yellow). The graph is constructed as described in Section 6.2 and Section 7.2 with $p_z^{\min} = 0.001\ \text{GeV}/c$ and $s^{\max} = 2.0$.

the nodes and edges corresponding to FT5 and FT6 behind the dipole field, including the long edges between FT4 and FT5.
Segmenting the graph into these three parts is useful for several reasons; it efficiently reduces the size of the graph based on the number of nodes and edges, directly reducing the occupancy of FPGA and allowing faster compilation of `hls4ml`, which will be important in the next chapter. In addition, this segmentation is a natural choice since the detector units are also segmented into these three units. This segmentation allows for different treatment of the regions inside and outside the magnetic field since the particle tracks, especially the slopes, behave differently depending on the magnetic field. Another important aspect is that the classification accuracy of the long edges between the three segments, particularly between FT4 and FT5, is reduced compared to the classification within the detector chambers. Therefore, treating the long links separately can improve classification performance. Furthermore, the method is similar to the tracklet construction approach described by [33], where three tracklets corresponding to the three regions described here are constructed in the first step and connected to complete track candidates in a second step by a second machine learning (ML) application. Therefore, [33]'s methods for full track discovery could also be applied here to find full tracks.
The first segment of the segmented graph is shown in Fig. 7.6. The figures for the other two graph segments can be found in the appendix. Unlike before, the double layer hits mentioned in Section 6.2 can be easily distinguished by eye. For simplicity, all three segments are generated with the same graph building configuration. Alternatively, individual graph

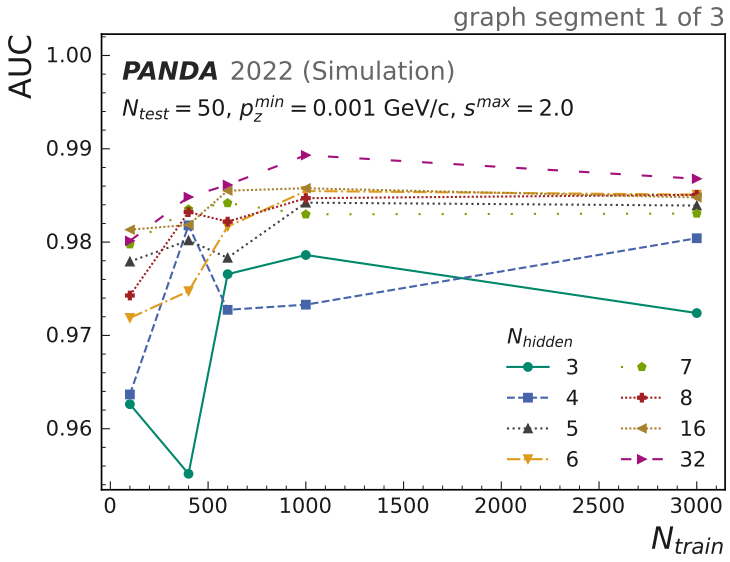


Figure 7.7.: Display of area under the curve (AUC) scores for different sets of number of training events N_{train} and number of hidden nodes N_{hidden} per hidden GNN layer performed on N_{test}=50 test events based on the first third of segmented graphs built with $p_z^{\text{min}} = 0.001\,\text{GeV}/c$ and $s^{\text{max}} = 2.0$.

building constraints could be applied to each segment. A maximum slope of $s^{\text{max}} = 2.0$ is used for graph building for all three parts, as opposed to the previous full graph building, to meet the purity and efficiency requirements of all three segments. This slope threshold allows for a fairly balanced ratio of true and false edges in the dataset, e.g., in the first segment, the proportion of true edges to all edges is 45.8 %.
The three segments are trained separately with a hyperparameter search using `optuna`. The hyperparameter search is performed for the learning rate lr, the learning rate decay γ, and the period of the learning rate decay (step size). The number of training events chosen for hyperparameter search ranges from 100 to 3000. As described in Chapter 5, large data sets are needed for deep learning (DL) applications to avoid overfitting. However, each event, and thus each hit graph, contains a subset of several hundred edges. Therefore, even a small number of events can provide a sufficiently large sample. The training performance is monitored for different numbers of hidden nodes N_{hidden} between 3 and 32 for the different IN blocks. In Fig. 7.7, the AUC values are plotted for the different training sample sizes N_{train}. In Fig. A.2, the classification performances for the other two segments can be found. AUC was chosen as the measure of classification performance on the test data (50 events in this case) because it encodes classification performance in a single number and allows efficient comparison between the different configurations. A value of 1.0 corresponds to perfect classification, while a value of 0.5 or below corresponds to an untrained network. The data points corresponding to different N_{hidden} values are connected by lines to guide the eye and do not represent data fits. The data points correspond to the best configurations found by hyperparameter search for each set of N_{train} and N_{hidden}. Each set is trained with 20 different hyperparameter configurations, and each configuration is

tested three times to find the best-performing models. The AUC scores are computed using the average of 100 trials with a test sample of $N_{\text{test}} = 50$ random events corresponding to the order of $1 \cdot 10^4$ edges. No error bars are plotted to increase the readability of the plot. However, to gain complete control over the statistics of this problem, a full bootstrap analysis [82] must be performed.
Despite the lack of statistics, two basic trends can be identified: A higher N_{hidden} corresponds to a higher AUC value, and a higher N_{train} in the range between 100 and 1000 also corresponds to better results. Overall, most performance results are between 0.97 and 0.99, and the highest AUC value of $\text{AUC}_{1/3} = 0.989$ for the first third of the graph segments is achieved at $N_{\text{events}} = 1000$ and $N_{\text{hidden}} = 32$. The best AUC results for segments two and three are $\text{AUC}_{2/3} = 0.989$ and $\text{AUC}_{3/3} = 0.984$, respectively. As expected, the classification performance for the segmented graphs is better than the full graph training result $\text{AUC}_{\text{full}} = 0.969$.

7.4. Tracklet Finding

In the next step, the classified track segments will be used to build tracklets. Tracklets are defined here as a connection of consecutive track segments to form track candidates for each of the defined segments (FT1+FT2, FT3+FT4, FT5+FT6) independently. Therefore, the maximum tracklet length is 8 hits for the first segment and 9 hits for the second and third segments, inlcuding the long interconnections. The tracklets are computed in the form of lists, which include all connected track segments using the network analysis `Python` package `NetworkX` [83].
The performance of tracklet finding is determined by a set of metrics, including the tracklet purity, the MC coverage, the rate of fully found tracklets, and the ghost rate proposed by [29]. These metrics will be defined in the following. The *tracklet purity* measures the fraction of hits in the reconstructed tracklet stemming from the majority particle, where the majority particle is defined as the MC particle that induces the majority of hits in the reconstructed tracklet. The tracklet purity is defined as

$$\text{tracklet purity} = \frac{1}{N_{\text{tracklets}}} \sum_{N_{\text{tracklets}}} \frac{N_{\text{majority particle}}}{N_{\text{tracklet hits}}} \tag{7.7}$$

with the number of hits by the majority particle $N_{\text{majority particle}}$, the total number of hits in the reconstructed tracklet $N_{\text{tracklet hits}}$, and the total number of reconstructible tracklets $N_{\text{tracklets}}$. A high tracklet purity indicates that most of the hits in the tracklet are induced by the same particle.
The *MC coverage* is closely related to the tracklet purity. It describes the fraction of $N_{\text{majority particle}}$, and the number of hits of the corresponding MC tracklets N_{MC} averaged over all reconstructible tracklets

$$\text{MC coverage} = \frac{1}{N_{\text{tracklets}}} \sum_{N_{\text{tracklets}}} \frac{N_{\text{majority particle}}}{N_{\text{MC}}} \,. \tag{7.8}$$

The ideal coverage is 100 %, where all hits generated by a MC particle are reconstructed in the same tracklet.

A tracklet is considered reconstructible if it comprises a minimum of three hits. Fully reconstructed tracklets are defined as tracklets with tracklet purity = 100 % and MC coverage 100 %. The rate of *fully found* tracklets is therefore defined as

$$\text{fully found rate} = \frac{N_{\text{full}}}{N_{\text{tracklets}}}\,, \tag{7.9}$$

where N_{full} is the number of fully reconstructed tracklets.
Tracklets with purity< 80 % are considered *ghost* or fake tracks. The ghost rate is defined as

$$\text{ghost rate} = \frac{N_{\text{ghost}}}{N_{\text{tracklets}}} \tag{7.10}$$

with the number of ghost tracks N_{ghost}. Fig. 7.8 displays the described tracklet finding metrics computed on the classification results of the full graph and the segmented graph approaches for each of the three-segment regions. The computation is performed on N_{test}=1000 test events for different thresholds δ, where only edges with classification scores higher than δ are included in the tracklet building. Some basic trends can be identified in the case of all three segments, and both classification approaches: The MC coverage starts at a value of nearly 1.0 at small δ and decreases with increasing δ. The opposite is true for the tracklet purity, which increases with δ. The rate of fully reconstructed tracklets is highest for medium values of δ and drops for both extremes of low and high δ. The ghost rate decreases with increasing δ. There are only small differences between the performance of the tracklet finding for segments 1 and 2, but a significant difference between those two and segment 3. The varying performance may be explained by the long connections between FT4 and FT5, which are included in segment 3, which are much longer and, therefore, more difficult to classify than all other track segments described before. The results for all metrics at a threshold of $\delta = 0.55$ are summarized in Table 7.1. As expected, the overall tracklet finding performance of the segmented approach is slightly better than the performance of the full graph approach. The full graph approach performs slightly better than the segmented version only in the case of the MC coverage. For both approaches, the overall purity and MC coverage averaged over all three segments is higher than 80 %, and the fully found rate is about 60 %. The found tracklets can now be used as input to full track finding methods such as the RNN-based approach by [33].

7.5. GNN-based Track Finding Summary

Overall, the last two chapters described and discussed a complete workflow of the GNN application for classifying track segments based on PANDA FTS simulation data: from data simulation using `PandaRoot`, through data preprocessing and graph building, to edge classification using a IN based full graph and segmented graph implementation. The final track segment classification results of $\text{AUC}_{\text{full}} = 0.969$ for the full graph approach and $\text{AUC}_{1/3} = 0.989$ for the first and second segments of the segmented graph approach are excellent and used for a final tracklet finding analysis. The tracklet finding based on the two approaches provides results with both overall purity and MC coverage higher

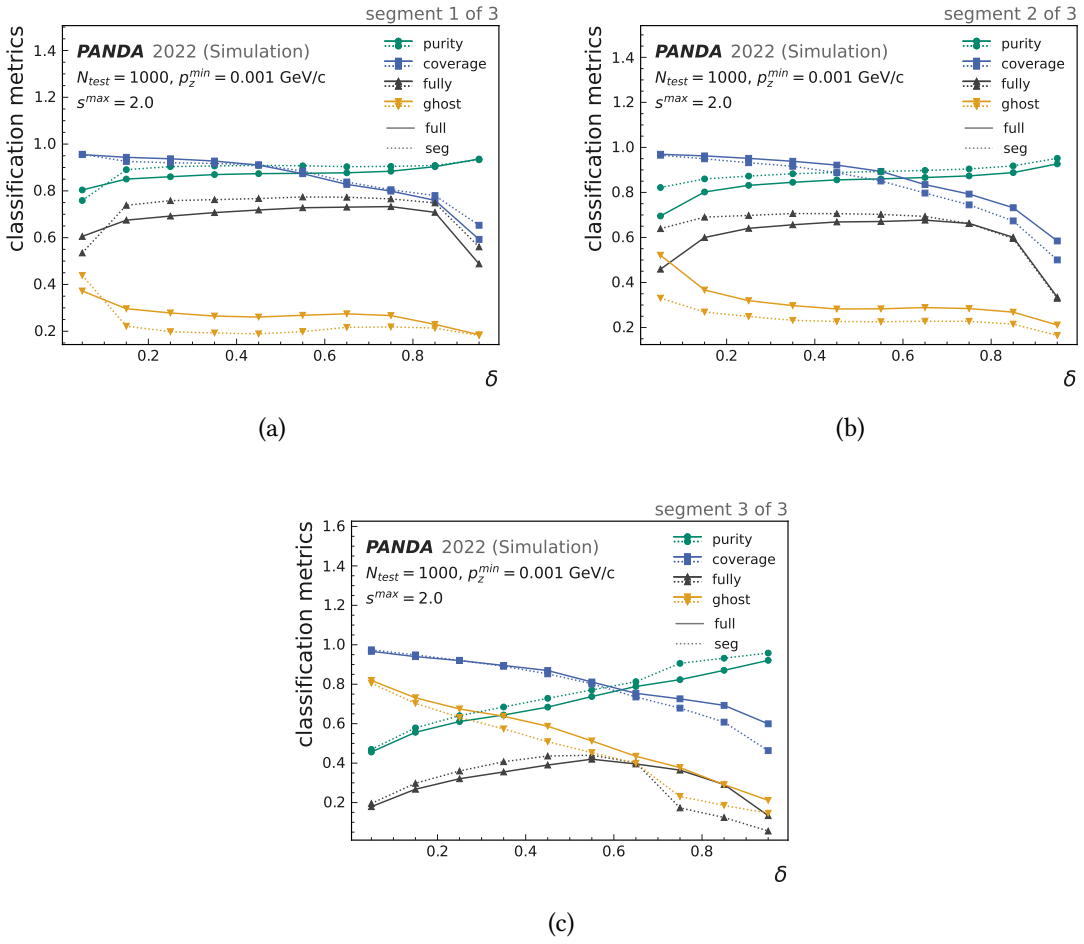


Figure 7.8.: Tracklet finding performance metrics including tracklet purity, MC coverage, fully found rate, and ghost rate applied on both classifier approaches: full and segmented graph classifiers. The metrics are computed for segment 1 including FT1 and FT2 hits (a), segment 2 including the FT1 last layer, FT2, and FT3 hits (b), and segment 3 including the FT4 last layer, FT5, and FT6 hits (c).

Table 7.1.: Tracklet finding performance metrics including tracklet purity, MC coverage, fully found rate, and ghost rate applied on different approaches and graph segments.

Approach	Segment	Purity	MC Coverage	Fully Found Rate	Ghost Rate
Full	1	0.881	0.881	0.729	0.256
Segmented	1	0.914	0.890	0.777	0.185
Full	2	0.860	0.898	0.673	0.280
Segmented	2	0.894	0.856	0.706	0.222
Full	3	0.721	0.800	0.394	0.543
Segmented	3	0.758	0.788	0.414	0.483
Full	Mean	0.821	0.860	0.599	0.360
Segmented	Mean	0.855	0.845	0.632	0.230

than 80 % and a fully found rate of about 60 %. These tracklets build the basement for further steps in the tracking working flow. For example, an interesting next step would be to extend this work by forming full tracks from the predicted tracklets and again comparing the full track finding results of the full graph approach with the results of the segmented graph approach. Also, additional optimizations need to be applied to enable the implementation of FPGA, which will be discussed in the following chapters.

8. FPGA Technology

Field programmable gate arrays (FPGAs)s are programmable digital integrated circuits. Unlike traditional computer or microcontroller programming, FPGAs can be programmed at the physical circuit structure level. Programming FPGAs is done in hardware description languages (HDLs) such as VHDL or Verilog, which specify how a given function block should behave. The actual physical elements are decided in the implementation, with additional general blocks inferred during synthesis. The configuration of the internal elements enables the implementation of various circuits and functions with FPGAs, from simple synchronous counters to highly complex circuits such as microprocessors or fast readout of high energy physics (HEP) detectors [84].
Machine learning (ML) applications can be implemented on accelerated hardware such as FPGAs using the high level synthesis for machine learning (`hls4ml`) package. `Hls4ml` enables the automated process of synthesizing FPGAs from trained neural network (NN) models written in `PyTorch`, `Keras`, `TensorFlow`, or `ONNX` leveraging `Vivado HLS` and its compilation functions. `Hls4ml` provides various evaluation, optimization, and customization tools to assist the user in developing application-specific optimal designs.
A major challenge in creating an optimal FPGA implementation is the tradeoff between resource utilization of the FPGA on the one hand and latency and throughput requirements on the other hand.
This section provides a brief introduction to FPGAs, including the key advantages of FPGA technology and FPGA structure and design. It also introduces the `Vivado HLS` development environment and the `hls4ml` framework for automatically translating neural network code from high level languages such as `PyTorch` or `Tensorflow` [85] into machine languages. Finally, several design optimization methods and their implementation in `Vivado HLS` are discussed.

8.1. Prospects of FPGA technology

Modern HEP experiments deal with ever-increasing detector granularities and data volumes. The main challenges of today's data acquisition systems (DAQs) are, therefore, high data throughput with low latency, increasing complexity of data processing in real-time applications, and at the same time, high design flexibility.
FPGAs can provide solutions for all these requirements. FPGA technology supports high data throughput and low latency data processing while providing high flexibility in dynamic development processes. The main advantage of FPGAs over general-purpose application processors such as central processing units (CPUs) and graphics processing units (GPUs) is its high parallelization capability combined with pipelining: independent operations can run fully in parallel, enabling trillions of operations per second with

relatively low power consumption compared to traditional CPU and GPU architectures [35, 86].
In addition, FPGAs provide high flexibility through the ability to reprogram FPGAs at the physical circuit level. The flexibility enables more efficient data throughput for specific applications than conventional processors. Higher data throughputs can only be achieved with application-specific integrated circuits (ASICs) custom manufactured for specific tasks. The main disadvantages of ASICs are the high investment costs and, in any case, the long development times, while FPGAs allows a very low cost and flexible prototype design at development costs that are much lower compared to ASICs. However, FPGAs are expensive on a large scale while ASICs are far cheaper in mass production. In addition, a major advantage of FPGAs over ASICs is its reconfigurability after manufacturing. Reconfigurability is a key advantage in ML applications in DAQs, as the neural network architecture and its parameters can be dynamically updated using FPGAs. The advantages of FPGAs over ASICs and GPUs are shown in Fig. 8.1.
Since FPGAs allow very fast signal processing and flexible circuit modification, they are used in almost all areas of digital technology and are especially common in the DAQ of HEP experiments.

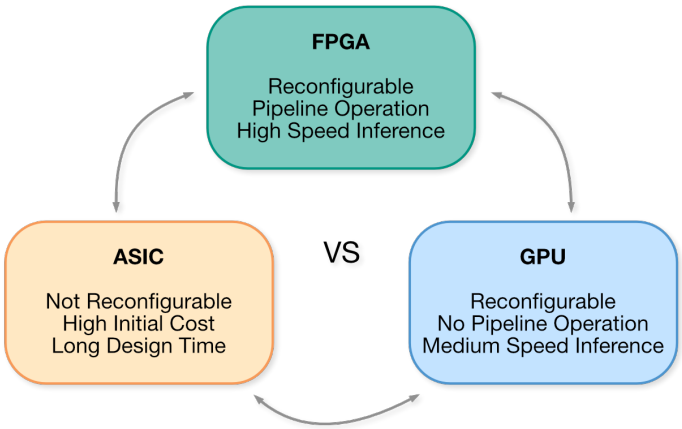


Figure 8.1.: Advantages of field programmable gate array (FPGA) over application-specific integrated circuit (ASIC) and graphics processing unit (GPU).

8.2. Structure and Design

A basic FPGA architecture consists of a set of configurable logic blocks (CLBs), input/output (I/O) blocks, and switching blocks (SBs), as seen in Fig. 8.2. A simple CLB consists of a programmable lookup table (LUT), a flip-flop (FF), and a multiplexer (MUX). A lookup table (LUT) replaces the computation of functions with a simpler array indexing, where the desired function output is obtained by reading the memory address of the defining truth table in the static random-access memory (SRAM) cells of the LUT corresponding to the given inputs. The LUT depends on the number of available inputs, e.g., for four

binary inputs, a 4-bit LUT is used to implement any 4-digit binary function. The output of the LUT is then passed to the flip-flop (FF), which is usually a D flip-flop (DFF). A DFF is used to store the incoming signal until it is forwarded synchronously with the next positive clock (clk) edge. The multiplexer (MUX) allows the FF to be included or bypassed, providing very fast local signal forwarding and increasing the degrees of freedom.
The CLBs are interconnected by a network of numerous bus structures to which the I/O blocks can be used to connect inputs and outputs for communication with the outside world. The interconnections are determined by programmable SBs in the grid's intersections, allowing signal distribution over the entire chip. Since implementing multiplications on FPGAs is very resource expensive, in addition to normal CLBs, most modern FPGAs are also equipped with digital signal processors (DSPs). DSPs are specialized processors optimized for real-time execution providing the ideal balance between high-performance and efficient implementation. DSPs allow real-time multiplication of two given numbers and are therefore required for many signal processing tasks that rely on multiplication, such as digital filters or fast Fourier transform (FFT). Along with FFs, LUTs and DSPs, one of four commonly identified FPGA components are block random access memorys (BRAMs), which are used to store large amounts of data onboard.
Today, the term FPGA is often used interchangeably with systems referred to as system-on-a-chip (SoC). Such complete systems integrate multiple embedded computer systems into a single device, including CPUs, bus systems, RAM, ROM, and more.

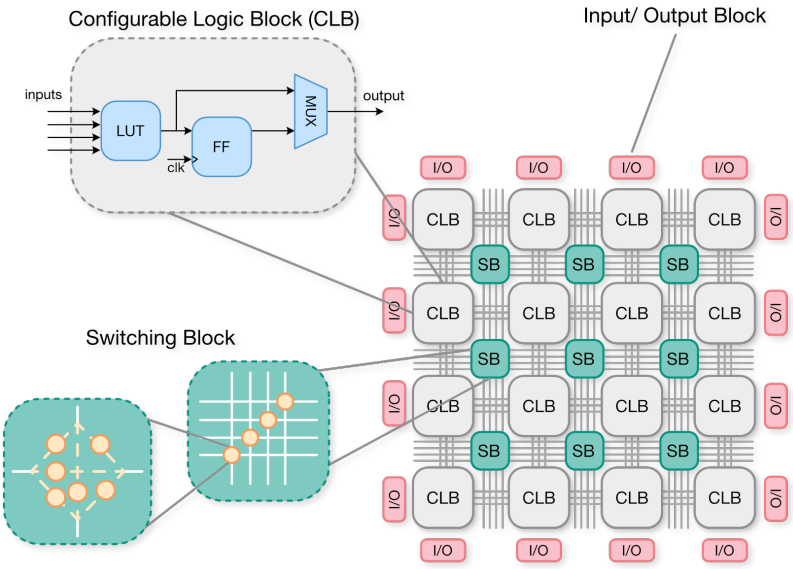


Figure 8.2.: Simplified illustration of an FPGA consisting of In- and Output Blocks, Switching Blocks and CLBs consisting of a lookup table (LUT), a flip-flop (FF), and a multiplexer (MUX).

8.3. Xilinx Vivado and Vivado HLS

The Vivado Design Suite is a integrated design environment (IDE) for the design, integration, and implementation of systems using Xilinx devices [87]. It is designed to improve the productivity of synthesis and analysis of HDL designs, including SoC development and high-level synthesis (HLS). The Vivado HLS compiler [88] enables conversion of code written in high-level languages such as C, C++, and SystemC to HDLs such as Verilog and very high speed integrated circuit hardware description language (VHDL) for register-transfer level (RTL) implementation. HLS is an alternative to processor-specific programming languages such as HDLs like VHDL or Verilog for FPGA or compute unified device architecture (CUDA) for GPU. Even though the performance of HLS-based approaches is less optimal compared to RTL-based designs, the simulation time is much lower, debugging is easier, design analysis provides various options, and the overall development process is generally less complicated [89].
Vivado's basic HLS design workflow, as shown in Fig. 8.3, is divided into the following

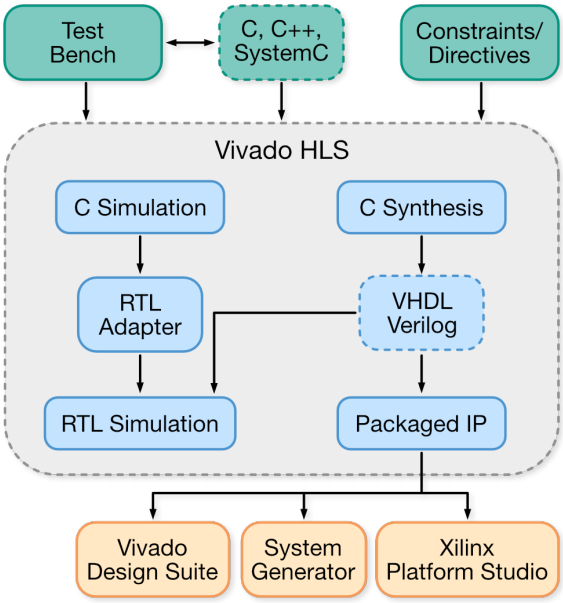


Figure 8.3.: Schematic overview of the Vivado HLS design workflow including C simulation and synthesis to VHDL or Verilog based on constraints/directives, RTL Simulation and packaged intellectual property (IP) export enabling the integration in other Xilinx hardware design tools.

steps: First, the C functions can be validated by a test bench to verify the required functionality. This step is called C validation or C simulation. In the next step, the C/C++ algorithm is compiled, which is called C synthesis. The synthesis outputs synthesized RTL files for Verilog or VHDL and a synthesis summary report on latencies and resource utilization. At this design stage, various additional analysis and optimization options can be applied to the HLS project. These include various viewers and the addition design directives. Again, the model's functionality can be validated on a C test bench to ensure that the RTL is

functionally identical to the C source code. This step is called C/RTL Co-Simulation or RTL verification. The final step is to package the RTL design and export it into a form other tools in the Xilinx design flow can use. A common choice is an export as a `Vivado` IP block.
Benefits of HLS include improved system performance for software developers, enabling the use of FPGA, and increased productivity for hardware developers by working at a higher level of abstraction. The higher level of abstraction resulting from the high-level C/C++ language and automatic translation in HDLs shortens validation phases, simplifies design space exploration through optimization directives, and overall reduces the time required to develop an optimal design implementation.
`Vivado HLS` offers several design analysis and optimization tools, such as performance estimate reports on latency and resource usage. However, the `Vivado HLS` performance estimates may vary from synthesis in Vivado after IP block export, especially regarding DSP and LUT usage. Vivado aims to optimize latency by using more DSP blocks instead of LUTs [35]. Hence, it can be expected that LUT utilization decreases while DSP utilization increases for implementation in Vivado. In general, the estimates based on Vivado HLS synthesis are found to be conservative compared to RTL implementation by Duarte et al. [34] which might be explained by additional optimizations by RTL synthesis.

8.4. Highlevel Synthesis for Machine Learning (hls4ml)

`Hls4ml` is a machine learning implementation package based on HLS that provides fast and efficient translation of ML models based on high-level languages such as `Python` and C/C++ to HDLs [34]. A schematic overview of a typical `hls4ml` design flow is shown in Fig. 8.4. At first, a usual neural network design workflow, shown with PyG, including training and compression steps, is required to generate a final trained model file. This model is then converted into an internal representation by `hls4ml`, the `HLSModel`. The `HLSModel` is then exported into an HLS project via an internal project writer by utilizing `Vivado HLS`. The compilation is based on several configuration parameters like precision, resource reuse factor, pipelining strategies, and optimizers for merging and cloning arrays. The exported project can be elaborated by `Vivado HLS` to perform synthesis and simulation steps such as C validation, C synthesis, and C/RTL verification. Finally, the HLS project can be exported as an IP block to integrate it into a larger hardware design in `Vivado`. Again, various synthesis and optimization steps, including implementation, placing, and routing, can be applied to implement the project on FPGA. In general, the time needed for model design by automated model translation by `hls4ml` is much shorter than designing model architectures directly for FPGA implementation that enables fast prototyping [34]. In `hls4ml` each layer or activation function is implemented as an independent configurable module predefined as C/C++ templates in the `NNET` library in `hls4ml`. During `hls4ml` conversion, these separate modules are arranged and connected in the correct order to reconstruct a full ML model by a top-level C/C++ function. In this work, a new `hls4ml` PyG converter, provided by Elabd et al. [2], is employed, which enables automatic translation of PyG models and, in particular, translation of the interaction network (IN) architecture discussed in this work. This improved version adds `NNET` modules for graph neural network

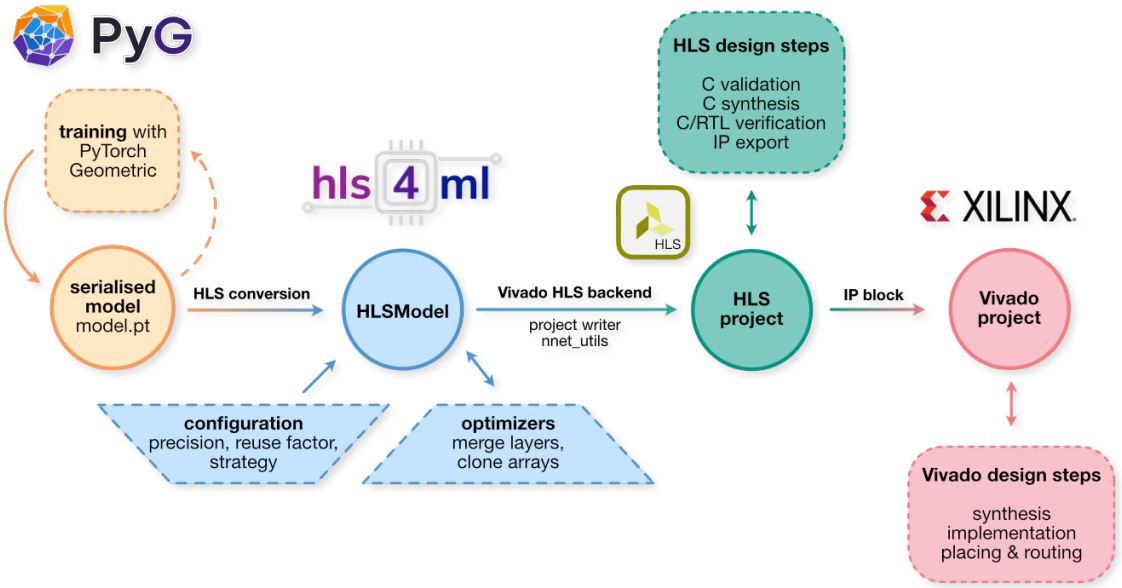


Figure 8.4.: Schematic overview of workflow from ML represented by a PyTorch Geometric (PyG) Model to an FPGA implementation using `hls4ml`. The yellow boxes (left) describe the training steps with a ML framework such as PyG generating a model file that gets converted with `hls4ml` to an `HLSModel` defined by configuration and optimizer settings. An `hls4ml` internal backend writer exports the model into an HLS project. This project can be used to perform several synthesis and simulation steps and can be exported as an IP block for a `Vivado` project block design to integrate the HLS project into a larger hardware design, e.g., on Xilinx FPGA

.

(GNN) implementation including the edge block, node block and edge-aggregation block. `hls4ml` provides several configurable parameters to enable simple design customization and exploration regarding performance, latency, and resource usage.
One key feature of `hls4ml` is the bit-accurate emulation of the `HLSModel` representation in a `Python` environment. The bit-accurate emulation provides an alternative to the `Vivado HLS` C simulation on a C testbench. The emulation is directly performed on `NUMPY` array data in `Python`, which enables running inference directly on the original NN test data facilitating and accelerating the design workflow by simplifying debugging and hyperparameter optimization at an early stage.

8.5. Design Optimization

FPGA designs most likely need optimization to meet performance, latency, and resource utilization requirements. The resource utilization, called *area*, is defined as the allocation of the different FPGA CLBs, namely: BRAM, DSPs, LUTs, and FFs. *Latency* is defined as the total time taken by a single iteration of the algorithm, usually expressed in units of clock cycles.

This section will discuss the different optimization methods for NN implementations on FPGAs using `hls4ml` especially focusing on the IN architecture that is used in this work. The `hls4ml` tool provides several configurable parameters to adjust and explore latency and resource utilization to find a task-specific optimal design in an automated design iteration.
Efficient optimization of resource utilization can be achieved through various techniques such as *quantization*, *compression*, and *parallelization* including *pipelining*. Compression or pruning aims to reduce the number of neurons without sacrificing performance since parts of the network neurons may be redundant. In the context of HLS, the number of hidden nodes is equated to the number of neurons. Furthermore, quantization aims to reduce the floating-point encoding precision of the model parameters to a minimal bit encoding that still provides good performance. Finally, parallelization can also be adjusted by configuring the degree of parallelization as part of the tradeoff between latency and resource consumption.

8.5.1. Quantization

The FPGA resource utilization depends on various model parameters. One of them is the number of bits required to represent the NN model to the level of accuracy required by the current application. The number of bits can be reduced by quantization. Unlike native C/C++ datatypes quantized in 8-bit bounds (8, 16, 32, 64 bits), which can lead to inadequate hardware implementation, HLS is based on arbitrary precision (AP) datatypes. The AP datatypes support arbitrary data lengths, i.e., enabling the usage of variables with smaller bit widths. As a result of smaller bit-widths, the logic allocates fewer FPGA resources and can be executed at higher clock frequencies with the same accuracy [86].
The fixed-point data format is denoted by `ap_fixed<W,I>` where W is the total number of bits or word length and I is the number of bits used to represent the signed integer bits above the binary point. The difference between W and I is the number of decimal place bits B after the decimal point. In addition, a quantization and an overflow mode can be specified, which will not be discussed here.
The standard precision of HLS calculations is based on the 32-bit floating point type. Quantization can be used to reduce precision without sacrificing performance. In `hls4ml`, the bit precision of inputs, network parameters, and network outputs can be set to exact bit sizes by AP data types before synthesis. The new converter added by Elabd et al. [2] additionally allows different precisions for the hit index adjacency matrices on the one hand and all other network parameters on the other hand, since the adjacency matrices can be encoded in integers. In contrast, all other network parameters require fixed-point data types.
The main limiting resource in many applications on FPGAs is the number of DSPs, mainly used for multiplications. For example, the DSPs blocks available on the part `xczu11eg-ffvc1760-2-e` combine an 18-bit by 27-bit signed multiplier with a 48-bit adder and 27-bit pre-adder [90]. This means that, e.g., a 27-bit by 18-bit signed multiplication can be accomplished by a single DSP whereas a 27-bit by 19-bit signed multiplication may require two DSPs. However, Vivado might reuse the same DSP in the next clock cycle or use some external FPGA logic to mimic the missing bits. In the case of bit widths of 10

and below `Vivado HLS` implements multiplications using LUTs instead of DSPs.
The bit widths of the variables used directly affect resource usage. For example, if a variable that only requires 18 bits is specified as a variable with 32 bits, this leads to the use of larger and slower operators such as DSPs. This drastically increases not only the area but also the number of operations that can be performed per clock cycle, and thus also, the throughput and latency. Therefore, finding suitable precision values for the data types is important to avoid oversized elements that waste valuable hardware resources.

8.5.2. Compression

An additional way of network size optimization is reducing the number of model parameters, called network compression or pruning. The number of model parameters is defined by the weights and biases of the underlying multilayer perceptrons (MLPs). Both strongly depend on the number of node and edge features, D_{node} and D_{edge} respectively, but also on the number of hidden nodes N_{hidden} defining the hidden layer dimensions of the edge and node block MLPs. In the case of MLPs with two hidden layers as used in this work, the number of parameters for a node or edge block N_b as the sum of weights N_w and biases N_b is given by

$$N_b = N_w + N_b = (D_{\text{in}} + D_{\text{hidden}} + D_{\text{out}}) \cdot D_{\text{hidden}} + (2D_{\text{hidden}} + D_{\text{out}}) \tag{8.1}$$

where a MLP block $b(D_{\text{in}}, D_{\text{out}}, N_{\text{hidden}})$ is defined by its input D_{in}, output D_{out} and hidden dimension N_{hidden}. For an IN described in Section 7.1 with a relational model (R) edge block

$$R_1(2 \cdot D_{\text{node}} + D_{\text{edge}}, D_{\text{edge}}, N_{\text{hidden}}) \,, \tag{8.2}$$

an object model (O) node block

$$O(D_{\text{node}} + D_{\text{edge}}, D_{\text{node}}, N_{\text{hidden}}) \,, \tag{8.3}$$

and a second relational model (R) output block

$$R_2(2 \cdot D_{\text{node}} + D_{\text{edge}}, 1, N_{\text{hidden}}) \tag{8.4}$$

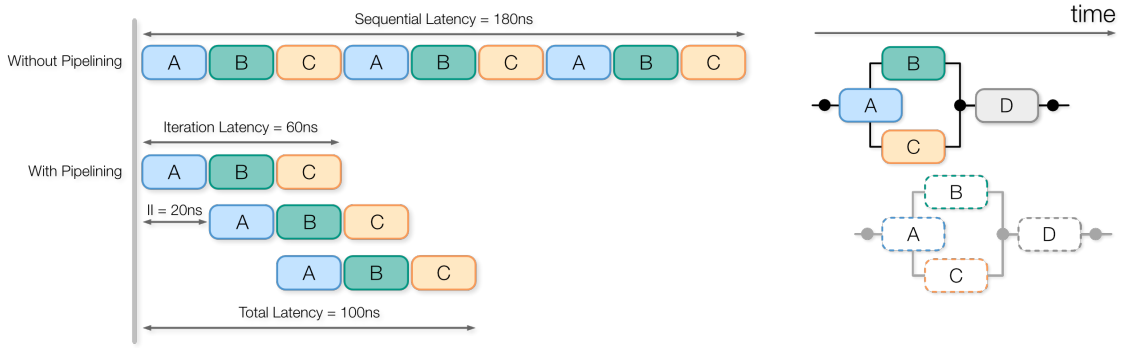
the total number of parameters equals

$$\begin{aligned} &N_{\text{IN}} = N_w + N_b \\ &N_w = D_{\text{hidden}} \cdot (6D_{\text{node}} + 4D_{\text{edge}} + 3D_{\text{hidden}} + 1), \quad N_b = 6D_{\text{hidden}} + D_{\text{edge}} + D_{\text{node}} + 1 \,. \end{aligned} \tag{8.5}$$

For example, with 2 node and edge features each and 6 hidden nodes for each hidden layer, the total number of model parameters is $N_{\text{IN}} = 275$, while with $D_{\text{node}} = 4$, $D_{\text{edge}} = 4$, and $N_{\text{hidden}} = 32$ the total number of parameters equals $N_{\text{IN}} = 4585$. In the following, the number of hidden nodes will be referred to as the number of hidden neurons. Each model weight corresponds to a multiplication, implemented on FPGAs by a DSP unit. Equation (8.5) demonstrates that the number of weights strongly depends on the number

of hidden neurons. Hence, the number of hidden neurons directly affects the number of utilized DSPs.
There are several methods to compress networks including parameter pruning, low-rank factorization, transferred/ compact convolutional filters, L_1 regularization, or knowledge distillation as described by Duarte et al. [34]. For simplicity, in this work, the resource usage and, later on, classification performance will be only analyzed concerning the number of hidden neurons. However, it would be interesting to analyze the performance of the other more sophisticated compression techniques in future studies.

8.5.3. Pipelining



(a) (b)

Figure 8.5.: Process Pipelining. (a) Working principles of pipelining based on three tasks A, B, and C. A comparison of a sequential execution without pipelining and parallelization with pipelining. (b) A more complex example of pipelining includes task parallelism and pipelining within a run, pipelining of runs, and pipelining within a task.

One key advantage of FPGA applications is throughput acceleration by parallelization, where throughput is the number of operations performed per unit of time. The most important process parallelization concept on FPGA is pipelining [86]. In Fig. 8.5(a), the basic concept of pipelining is displayed using an example workflow based on three tasks A, B, and C, where each task takes 20 ns to finish. The pipelined process is displayed in comparison to a sequential execution with a total completion time of 180 ns for three iterations of all three tasks, called runs. The time taken to finish the first run is called iteration latency; in this example, 60 ns. The next run starts 20 ns after the first run and additionally only takes 20 ns after the end of the first run to finish. This delay is known as the initiation interval (II) of the pipeline. The total time for all three runs to finish is the total latency of the pipeline, which can be described by

$$\text{total latency} = \text{iteration latency} + \text{II} \cdot (\text{number of functions} - 1)\ . \tag{8.6}$$

Here, the total latency equals 180 ns. The total latency can be improved by minimizing II. *Pipelining* is a classical architectural optimization that repetitively executes tasks by

reusing the same resources. It can be applied to multiple levels of abstraction, such as operators, loops, and functions. An efficient process design regarding throughput and usage of computational resources combines pipelining on multiple levels.
A more complex example is shown in Fig. 8.5(b) to explain pipelining and its potential, simultaneously combining task parallelism, pipelining within a run, pipelining of runs, and pipelining within a task. In this example, B and C are independent and can be executed in parallel, called task parallelism within a run. This is the only form of parallelization that can be run on a multithreaded CPU using shared memory. Task D depends on B and C, where the final outputs of B and C are synchronized only after all data is produced. The data synchronization is implemented by a parallel-in-parallel-out shift registers (PIPO) buffer, displayed by black dots. Pipelining of runs is added by executing, e.g., the second invocation of A in parallel to the first invocations of B and C, where A reuses the same computation resources as for the previous run as in the previous example. Furthermore, tasks B and C are executed pipelined concerning A, called pipelining within a run. The pipelining is enabled by synchronization between A, B, and C in an element-wise manner with first-in-first-out shift registers (FIFO) instead of PIPOs. A FIFO buffer enables the consumer task (here D) to start accessing the data inside the buffer as soon as the producer tasks (here B and C) fill in the data into the buffer. The FIFO buffers are represented by lines without circles. Lastly, pipelining within a task is added by pipelining task A, where the second invocation overlaps with the first invocation of A.
In `hls4ml`, block-level pipelining at the IN node, edge, aggregation, and output block levels is tunable by reuse factor (RF), which controls the II of each of these blocks, i.e., configures the number of times a multiplier is reused in the computation [34]. By design, II should be equal to RF. However, due to `Vivado` internal HLS optimizations, it may be smaller. For example, with a RF of one, the computation is performed fully in parallel, and with a RF of R, the computation is performed with a factor of $1/R$ of multipliers. In other words, the latency is proportional to the RF. The total latency of the GNN implementation on FPGAs using the IN architecture is composed of the combined latencies of the two IN edge blocks, the node block, the edge aggregation, and the final activation function, all of which can be pipelined. The latency of a single network-layer computation, L_m, is approximately

$$L_m = L_{\text{mult}} + (R-1) \cdot II_{\text{mult}} + L_{\text{activ}} \,, \tag{8.7}$$

with multiplier latency L_{mult}, multiplier II II_{mult}, and activation function latency L_{activ}.

8.5.4. HLS Design Directives

`Vivado HLS` offers various optimization directives, called *pragmas*, which enable specification of high-level design choices. In the following, the most important pragmas for this work are described.
By default, small functions are inlined using the INLINE directive. Inlining removes the function hierarchy, which enables logic optimization across function boundaries, improving the latency.
Arrays are generally implemented as BRAM, with a maximum of two available data ports. This limited array access can limit the overall throughput. Therefore, a common

practice is to increase the bandwidth by splitting the array into multiple smaller arrays using the ARRAY_PARTITION directive. There are three types of array partitioning: block partitioning (splitting into equally sized blocks of consecutive elements), cyclic partitioning (splitting into equally sized blocks interleaving the elements), and complete partitioning (splitting into the individual elements). For block and cyclic partitioning, a `factor` option enables the specification of the number of new arrays.
The ARRAY_RESHAPE pragma combines ARRAY_PARTITION with vertical ARRAY_MAP to reduce the number of BRAMs while keeping the parallel data access. Vertical array mapping concatenates multiple small arrays creating a single large array which reduces the number of BRAMs to the cost of higher bit-widths.
In `Vivado HLS`, for-loops can be (partially) unrolled using the UNROLL pragma to increase throughput at the cost of increased area. In the fully unrolled loop version, each loop iteration is executed as a separate copy of the loop-body to fully perform the complete loop operation in parallel in a single clock cycle. This implementation requires that the used arrays are partitioned to gain access.
Function and loop pipelining using the PIPELINE pragma reduces the II and increases the throughput. All loops in the hierarchy below the specified pipeline region are automatically unrolled; sub-functions must be pipelined manually. The II of pipelining is set to 1 by default but can be set as optional option, e.g. in `hls4ml` choosing the RF.
Another important optimization is the task level parallelism via the DATAFLOW pragma, which allows the overlap of sequential task executions (e.g., for functions or loops), improving design throughput and latency. The dataflow parallelism requires additional FIFO or BRAM registers, so the throughput is only limited by input and output access.

In the following chapter, the discussed methods and techniques will be used to apply and optimize a GNN-based track segment classifier on FPGA technology.

9. FPGA Implementation

Chapter 7 has shown promising performance of a graph neural network (GNN)-based track segment classification on anti-**P**roton **An**nihilation at **DA**rmstadt (PANDA) forward tracking system (FTS) data. This chapter will focus on field programmable gate array (FPGA) implementation of this classifier using the transpiler high level synthesis for machine learning (`hls4ml`) and methods discussed in Chapter 8, while taking into account limited resources on FPGA. This chapter begins by describing the working environment in Section 9.1 and the benchmark configurations in Section 9.2. Then, scans on the different optimization parameters described in Section 8.5 are performed investigating their effect on latency and resource usage in Section 9.3 and on the classification performance in Section 9.4. Lastly, these parameter scans are used to identify an optimal design solution in Section 9.5.

9.1. Working Environment

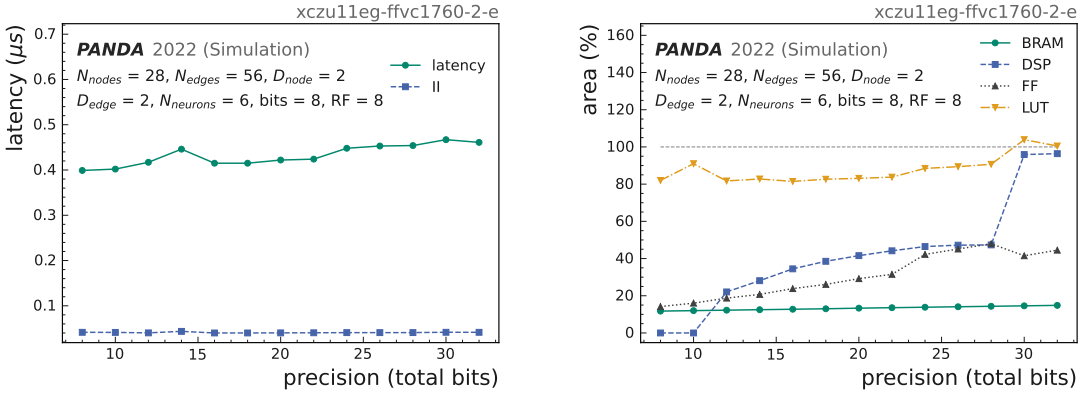
This chapter describes a GNN implementation on FPGA using the transpiler `hls4ml` targeting a Xilinx Zynq® UltraScale+™MPSoC FPGA (part number `xczu11eg-ffvc1760-2-e`). This device integrates programmable logic, two multi-core Arm® Cortex processing systems (quad-core A53 and dual-core R5F), 256 kB on-chip memory, multiport external memory interfaces, and a variety of interfaces for peripheral connection in a single device [90]. The total of 653 100 system logic cells include 597 120 flip-flops (FFs), 298 560 lookup tables (LUTs), 21.1 MB of block random access memory (BRAM) and 2 928 digital signal processor (DSP) slices. The maximal clock frequency of DSPs and BRAMs on Ultrascale+ devices is 738 MHz that corresponds to a clock period of1.35 ns.
The network implementation is applied through a `hls4ml` version provided by Elabd et al. [2] that allows GNN conversions. The `hls4ml` transpiler is based on the `Vivado HLS` version 2020.1 since `hls4ml` does not support newer versions of `Vivado HLS` or `Vitis HLS`.
Compilations are performed on an AMD Ryzen 9 5950X 16-core processor with 32 central processing units (CPUs) and 125 GB system memory. The system memory has in principle no influence on the compilation time but is crucial for whether the compilation is successful or not. The resulting latencies and areas for the precision scan are based on high-level synthesis (HLS) performance estimates of the Vivado HLS C Synthesis of the described benchmark network.

9.2. Benchmark Network, Graphs and Design

The effects of various `hls4ml` configuration parameters on HLS performance results are quantified using the FPGA latency and resource utilization, also called areas, of the

trained GNN models described in Section 7.3. The interaction network (IN) architecture described in Section 7.1 consists of edge, node, aggregation, and final output blocks. The HLS implementation of the corresponding PyTorch Geometric (PyG) models are compiled for the `xczu11eg-ffvc1760-2-e` part using a benchmark model with the following configurations: number of nodes N_{nodes} = 28, number of edges N_{edges} = 56, and reuse factor (RF) = 8, which are upper bounds on graph sizes that can be synthesized by Elabd et al. [2]. Furthermore, an edge index precision of `ap_uint<16>` and a precision for all other parameters of `ap_fixed<16.8>` are chosen. Moreover, according to the network architecture and dimensions described in Chapter 7, the node dimension D_{node}= 2, the edge dimension D_{edge}= 2, the number of hidden neurons N_{neurons}= 6, and the sigmoid activation function (see Section 7.1) are chosen. Simple activation functions such as the rectified linear unit (ReLU) function can be implemented in programmable logic, but more complex activation functions, e.g., based on exponential functions such as sigmoid, softmax, or hyperbolic tangent, must be computed in advance for the range of input values and stored in BRAMs for quick access.
The segmented graphs discussed in Section 7.3 are employed as test data for the implementation of FPGA. These graphs contain on average 32.8 ± 8.6 nodes and 53.4 ± 23.3 edges. Since hardware resources cannot accept variable-size input data, the graphs must be resized to a uniform size. The method used by Elabd et al. [2], which is also used in this work, is to truncate the data using the 95 % rule. In this method, a maximum graph size is set where 95 % of all graphs are smaller than this maximum size. The largest 5 % are truncated to the maximum graph size. All graphs smaller than the maximum graph size are padded with zero nodes and edges. All zero nodes are located at the origin of the coordinate system and zero edges are constructed by a series of loop connections of the last node of a graph with itself. Here, the 95th percentile for the number of nodes N_{nodes} and edges N_{edges} corresponds to a graph with 49 nodes and 98 edges.
The default design in this work is based on the throughput-optimized design of Elabd et al. [2] operating on a 5 ns clock period where pipelining is performed at the level of the IN edge block, the node block, and the edge aggregation block. The level of pipelining over the *initiation interval (II)* can be configured by the `hls4ml` *RF* parameter, where II is the time needed until new data can be accepted by a pipelined function. In the case of a non-pipelined function, the II equals the total latency. To realize the full potential of pipelining, all loops are fully unrolled and all arrays are fully partitioned. The above design directives are automatically implemented at the `Vivado HLS` level via the PIPELINE, UNROLL, and ARRAY_PARTITION pragmas.

Given the described working environment and benchmark model, the following sections examine the implementation of the discussed network architecture for various `hls4ml` parameters monitoring latency and area estimates. Then, the classification performance of different network configurations in `hls4ml` is investigated with respect to the truncated and zero-filled graph data.



(a) (b)

Figure 9.1.: `hls4ml` latency and hardware usage estimates depending on the arbitrary fixed-point precision where the number of total bits corresponds to the word length W in `ap_fixed<W,I>`, and I is half of the word length. The used benchmark network configuration is: N_{nodes}=28, N_{edges}=56, D_{node}=2, D_{edge}=2, RF=8, N_{neurons}=6. (a) Latency estimates in µs. (b) Hardware area utilization in percent of available board resources, namely: BRAM blocks, DSPs, FFs, and LUTs. Area usage corresponding to 100 % is indicated by a thin grey line.

9.3. Design Parameter Studies

This section describes the implementation of the design optimization methods described in Section 8.5 applied to the GNN-based edge classification model described in Section 7.3. The effects of various `hls4ml` configuration parameters on resource utilization and latency on FPGA are discussed.

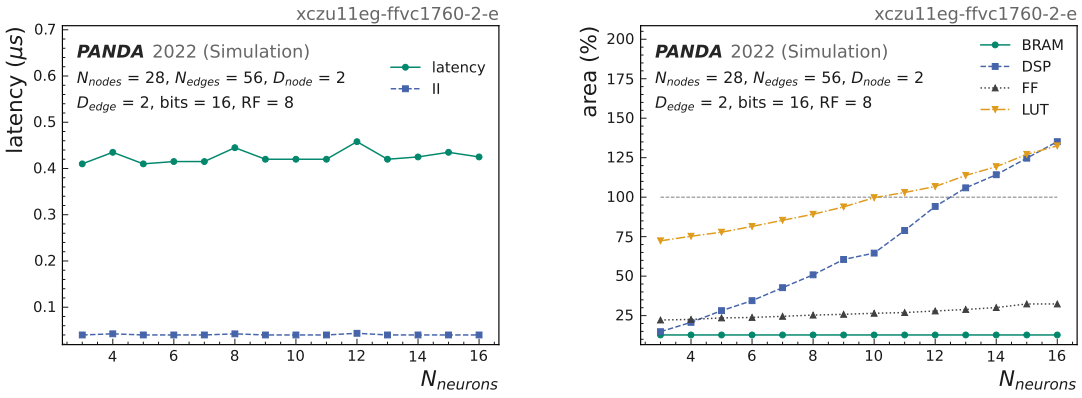
9.3.1. Quantization

In Section 8.5 is discussed how the fixed-point *precision* that is characterized by its total width and integer bits has a significant impact on the area. To reduce the bit widths of the variables, the neural network (NN) model parameters (weights and biases) of each layer are encoded in arbitrary fixed-point format. Fig. 9.1 shows the latency and areas as a function of total precision bit width for the described benchmark network as results of the `hls4ml` synthesis performance estimates. For simplicity, precision is represented in terms of the total bit width W of the fixed-point data type, where both the number of integers $I = W/2$ and decimal bits are equal to half the total bit width. For example, a precision of 16 total bits corresponds to `ap_fixed<16,8>` with 8 integers and 8 decimal bits. An integer bit width of $I = W/2$ is chosen since this relationship has been found to be most performing over the entire range of values.
For readability, the data points are connected by lines. Each point is computed only once because compilation time increases dramatically as model complexity increases. Therefore, statistical variations are to be expected. Compilation times for different configurations

can be found in the appendix Appendix A.3. In this case, the total latency to output the results, given in Fig. 9.2(a), does not seem to depend strongly on accuracy and, apart from statistical fluctuations, increases relatively weak from a value of 0.399 µs at $W = 8$ bits to 0.461 µs at $W = 32$ bits. Therefore, the throughput is slowed down at higher precisions needing more clock cycles to finish the operations. The II after which the next set of inputs is accepted appears to be completely independent of the precision at a relatively constant value of 0.041 µs. The II is significantly lower than the overall latency, indicating that pipelining is applied, and initialized by the RF. The aspect of pipelining by RF is discussed in more detail in Section 9.3.3.
More interesting metrics for the effect of precision on performance estimates are the areas. In Fig. 9.1(b), the resource usage of BRAM blocks, DSPs, FFs, and LUTs is shown as a percentage of available resources on `xczu11eg-ffvc1760-2-e`. While the effect of precision on the use of BRAM appears to be small, as expected, increasing the overall precision bit width increases the areas, especially for FFs and DSPs. The number of LUTs increases roughly in proportion to the precision. As expected, the number of DSPs equals zero below 10 bits and increases slightly after W=10 bits with a larger step after W=28 bits from 1386 to 2808 units used. As explained earlier, for precision sizes greater than 27 signed bits, corresponding to a total of 28 bits, two DSPs are needed to compute a single multiplication in the same clock cycle, rather than one DSP unit below that value. Near the accuracy limit of DSPs, an additional slight increase in FFs is observed. However, even at the highest accuracy, the number of DSPs does not exceed the available resources, and the number of LUTs does not exceed the available LUTs until 30 bits. The impact on the use of BRAM is vanishingly small since BRAM blocks are only used to store the precomputed activation function values.
Finally, it should be mentioned that there are additional, more sophisticated methods for further quantization of the network, such as quantization-aware training. It may be interesting to explore additional quantization techniques.

9.3.2. Compression

In Fig. 9.2, the latencies and areas are shown for a number of neurons N_{neurons} between 3 and 16. The estimates are obtained using the same configuration as for the network quantization, except that a fixed precision of `ap_fixed<16,8>` is given by the total number of bits = 16. There appears to be no effect of N_{neurons} on the latency values. This indicates that the computations for the individual neurons are performed completely in parallel. This also affects the consumption of DSP and LUT, which seems to increase approximately proportionally with N_{neurons}. The increase in LUTs, however, may in fact also correspond to a higher degree polynomial with a weak slope. For the given configuration, the number of LUTs already exceeds the available resources at N_{neurons}=11, and the number of BRAM blocks exceeds the resources at N_{neurons}=13. The impact on the usage of BRAM and FF is vanishingly small.



(a) (b)

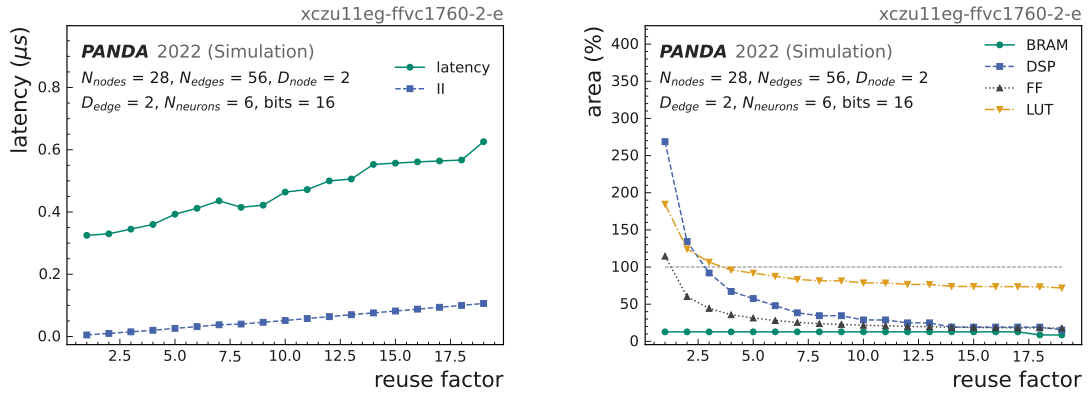
Figure 9.2.: `hls4ml` latency and hardware usage estimates on the part `xczu11eg-ffvc1760-2-e` depending on the number of neurons N_{neurons}. With benchmark network configuration: N_{nodes}=28, N_{edges}=56, D_{node}=2, D_{edge}=2, RF=8, and arbitrary precision fixed-point with word length of 16 bits (8 places above and 8 places below the decimal point). (a) Latency estimates in µs. (b) Hardware area utilization in percent of available board resources, namely: BRAM blocks, DSPs, FFs, and LUTs. Area usage corresponding to 100 % is indicated by a thin grey line.

9.3.3. Pipelining

Fig. 9.3 shows how the design scales as a function of the RF ranging from 1 to 19. By design, the II should equal the RF in clock cycles except for internal `Vivado HLS` optimizations. In general, increasing the RF reduces the area at the cost of increasing latency and II. The total latency and II increase proportionally with the RF by a small rate of increase, as seen in Fig. 9.3(a). At higher RF, the computation is less parallelized, which directly affects the latency. As expected, the II equals the RF in clock cycles of 5 ns. The degree of parallelization can also be observed by the decrease in the usage of DSP, FF and LUT at higher RF. In the case of the benchmark model used, at least a RF of 4 must be applied in order to avoid exceeding the available resources. Particularly noteworthy is the significant area reduction at a relatively small increase in latency with increasing RF, especially for low RF values.

9.3.4. Graph Dimensions

Fig. 9.4 shows the latency and area estimates over the number of nodes N_{nodes} from 5 to 50. The number of edges is set to $N_{\text{edges}} = 2N_{\text{nodes}}$, which equals the edge-to-node ratio of the graph data used. Separate investigations of N_{nodes} and N_{edges} independently from each other can be found in the appendix. With increasing graph dimensions, the total latency as well as the area usage increases proportionally. An interesting effect appears at N_{nodes}=40 and above. Below N_{nodes}=40, the II is independent from N_{nodes} at a constant

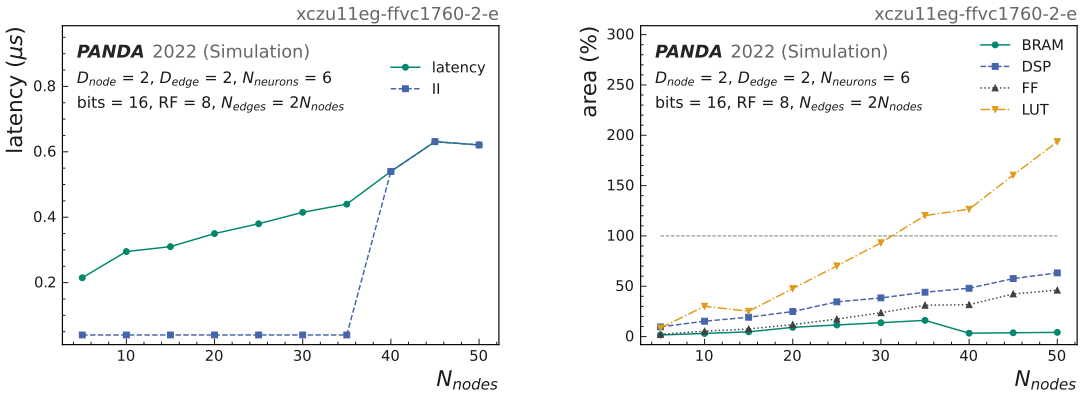


(a) (b)

Figure 9.3.: `hls4ml` latency and hardware usage estimates on the part `xczu11eg-ffvc1760-2-e` depending on the reuse factor. With benchmark network configuration: N_{nodes}=28, N_{edges}=56, D_{node}=2, D_{edge}=2, N_{neurons}=6, and arbitrary precision fixed-point with word length of 16 bits (8 places above and 8 places below the decimal point). (a) Latency estimates in µs. (b) Hardware area utilization in percent of available board resources, namely: BRAM blocks, DSPs, FFs, and LUTs. Area usage corresponding to 100 % is indicated by a thin grey line.

value of 8 clock cycles which is 40 ns. But at N_{nodes}=40, the II jumps to the same values as the total latency, e.g. at N_{nodes}=40 this is 108 clock cycles or 0.54 µs. This effect is also apparent in the area use. Here, the BRAM usage drops from 193 to 45 used blocks. Also, a slight decrease in the LUT usage increase is to be seen. The log file states that `Vivado HLS` is unable to satisfy the pipeline directive at N_{nodes}=40 and higher due to a too complicated edge aggregation control-flow. Therefore, the functions are executed sequentially.
It is also worth mentioning that the compilation time grows exponentially with increasing graph size. Already with N_{nodes} = 45 the compilation takes over 11 hours, see also the compilation time estimates in the appendix chapter.
Another important aspect is that in the case of too large graphs, loop unrolling to process the data in parallel can exceed the memory limit, causing Vivado to kill the process, which is discussed by Fuad and Vallecorsa [35]. For synthesizing large graphs, different approaches must be adopted. They propose a limitation of the unrolling by imposing a unroll factor.

In this section, the `hls4ml` implementation was scanned for different parameters to determine the impact of each of these parameters on the latencies and area usages. In the next section, the physics performance of the `hls4ml` implementation with different configurations is evaluated.



(a) (b)

Figure 9.4.: `hls4ml` latency and hardware usage estimates on the part `xczu11eg-ffvc1760-2-e` depending on the number of graph nodes N_{nodes} and edges N_{edges} where $N_{\text{edges}} = 2N_{\text{nodes}}$. With benchmark network configuration: D_{node}=2, D_{edge}=2, N_{neurons}=6, and arbitrary precision fixed-point with a word length of 16 bits (8 places above and 8 places below the decimal point). (a) Latency estimates in µs. (b) Hardware area utilization in percent of available board resources, namely: BRAM blocks, DSPs, FFs, and LUTs. Area usage corresponding to 100 % is indicated by a thin grey line.

9.4. Classification Performance

In this section, the classification performance of the `hls4ml` model for different configurations is investigated in comparison with the corresponding `PyTorch` models. The classification performance of the converted trained network models is evaluated using the AUC score represented in Fig. 9.5. Performance is measured by scanning the `hls4ml` inferences for different numbers of neurons N_{neurons} and precisions, where the scans are repeated for different numbers of training events N_{train}. The inference is determined by applying the models to N_{test}=100 events with graphs that satisfy the 95th percentile rule explained in Section 9.2. The 95th percentile rule corresponds to graphs with fixed graph sizes of N_{nodes}=49 and N_{edges}=98. As before, performance estimation is investigated for a range of the total number of fixed-point bits W according to `ap_fixed<W,W/2>` with $I = W/2$ integer bits (including signs). For better readability, the evaluated measurement points are connected by lines. The maximum AUC values that can be obtained are those corresponding to the trained `PyTorch` models. The scatter between data points is quite high, which is probably due to the lack of statistics in the evaluations. However, the basic trend is that as the accuracy increases, the maximum power can be reconstructed with a higher agreement. It is obvious that the accuracy of 10 bits and below is too low to reconstruct the classification performance of `PyTorch` models. In some cases, 12 bits is sufficient to reproduce the performance, but to ensure adequate performance with negligible loss, the accuracy of at least 16 bits or higher should be chosen. In general, no significant difference between the performance scans for different sizes of N_{train} can be observed.

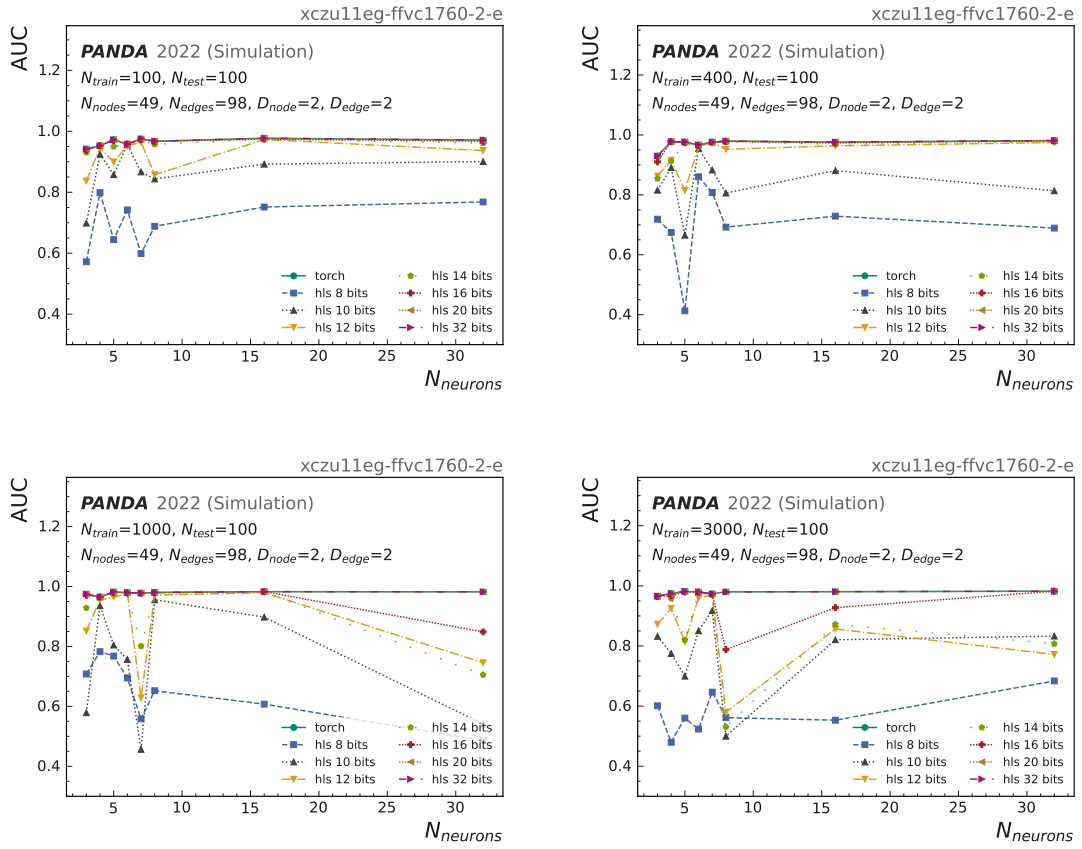


Figure 9.5.: HLS model classification performance estimations in the form of area under the curve (AUC) scores as functions of the number of neurons N_{neurons} for different numbers of training samples N_{train}. The AUC scores are computed for different precisions from 8 bits to 32 bits in comparison to the original `PyTorch` model performances.

The overall best AUC values are obtained for the networks trained on 1000 events. To provide a more detailed look at the reconstruction of classification performance from a different perspective, the ROC curves are shown with the corresponding AUC values for N_{train}=1000, different numbers of hidden neurons N_{neurons}, and accuracies in Fig. 9.6. With increasing N_{neurons}, a slight increase in the `PyTorch` AUC values can be observed. Otherwise, no significant difference in `hls4ml` reconstruction is seen for different N_{neurons}. Again, it can be clearly seen that both 8 and 10 bits are insufficient to reproduce the `PyTorch` results and at least 16 bits should be used.

9.5. Design Optimization

In this section, the information obtained from the studies in Section 9.3 is combined to find a design that satisfies the latency and area requirements, as well as the classification requirements. The size of the graph is chosen according to the 95th percentile rule and

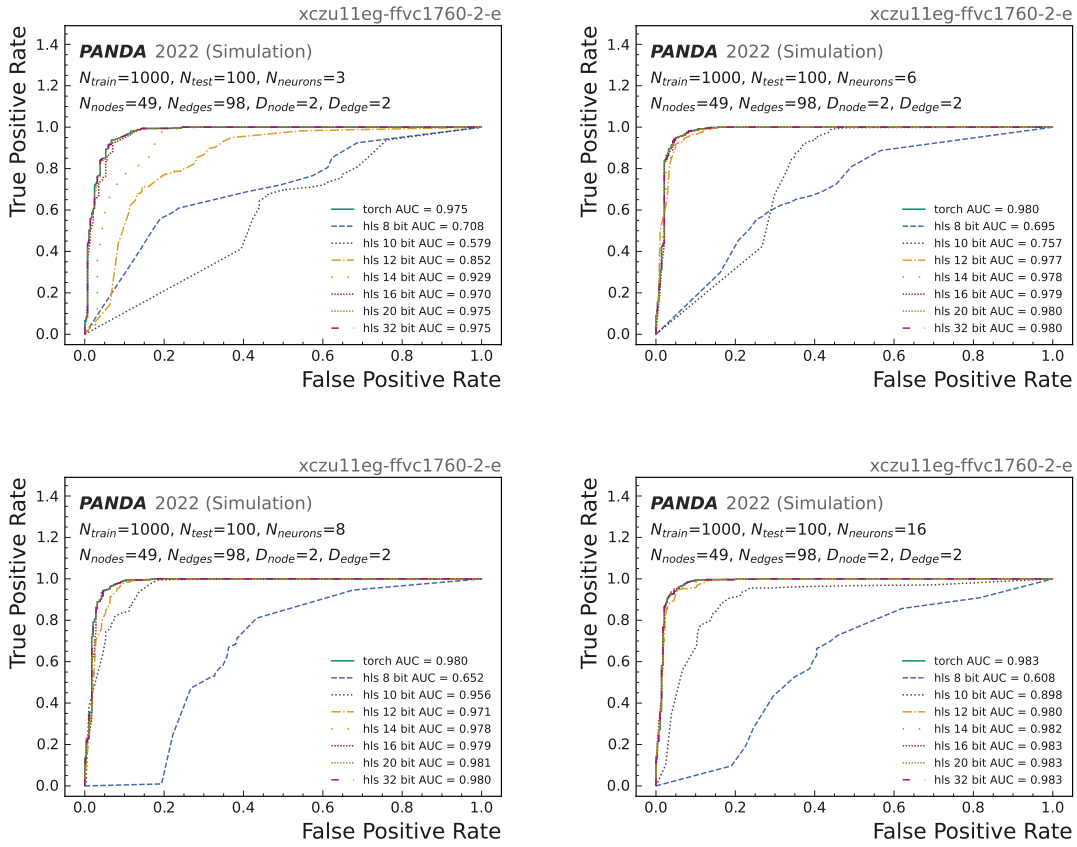


Figure 9.6.: HLS model classification performance estimations in the form of receiver operating characteristics (ROC) curves for different numbers hidden neurons N_{neurons}. The AUC scores are computed for different precisions from 8 bits to 32 bits in comparison to the original `PyTorch` model performances.

corresponds to N_{nodes}=49 and N_{edges}=98. Based on the results of Section 9.4, the number of precision bits is set to 16 bits and the number of neurons N_{neurons} is set to 6, which is the configuration that provides full classification performance with minimal area. The previous optimization studies were performed using the throughput-optimized design described above. This design is well suited for a small number of nodes and edges, but exceeds the device LUT capacity for the required graph sizes. Increasing RF has only a small and far from required effect on LUT utilization. Therefore, other designs must be investigated. The different design approaches are summarized in Table 9.1.

Solutions from 1 to 4 include applications of the throughput-optimized design for various RFs from 1 to 32. The resulting latency and area estimates are shown in Table 9.2. It can be seen that the area difference between an RF of 1 and 8 is significant, while the effect of further increasing the RF is significantly smaller, especially in the case of LUT usage.
Most of the LUT usage is due to the edge aggregation block, which alone occupies 94 % of all available LUT resources in the case of a throughput-optimized design at RF=32, and even more for smaller RFs. Solutions from 5 to 7 aim to reduce the area of the edge

Table 9.1.: Overview of design solutions based on different design directives.

Solution	RF	Description
1	1	throughput-optimized design
2	8	throughput-optimized design
3	16	throughput-optimized design
4	32	throughput-optimized design
5	16	edge-aggregation without array partitioning
6	16	edge-aggregation with array partitioning factor of 2
7	16	dataflow design & edge-aggregation without array partitioning
8	16	loop unrolling with factor parallelization factor (PF)=16
9	16	similar to solution 6 with 2 ns clock period

Table 9.2.: Latency and Area estimates for the different design solutions presented in Table 9.1.

Solution	Latency [cyc.]	II [cyc.]	DSP [%]	LUT [%]	FF [%]	BRAM [%]
1	103	103	546	386	88	2
2	124	124	63	193	46	4
3	139	139	34	175	42	4
4	173	173	19	169	40	4
5	384	247	34	85	17	4
6	311	247	34	85	17	4
7	290	247	34	104	18	113
8	85	85	34	169	20	4
9	345	247	34	87	23	4

aggregation block by removing the array partitioning. Solution 5 removes the array partitioning completely, while solution 6 applies block partitioning with a factor of 2. Solution 7 is based on a dataflow design instead of pipelining. The area reduction due to removing the edge aggregation partitioning is promising in the reduction of LUT usage from 175 % to 85 % while increasing the total latency from 139 to 384 cycles and the II from 139 to 247 cycles. Applying array block partitioning by a factor of 2 to the edge aggregation block reduces the latency to 311 cycles, while having negligible impact on the area and II. Replacing the overall pipelining directive with the DATAFLOW pragma to enable task-level pipelining (solution 7) dramatically increases BRAM usage to 113 % and LUT usage to 104 %, while reducing the overall latency by only 21 cycles compared to solution 6.
Considering the resource constraints, classification performance, and latency requirements, solution 6 can be determined as optimal solution for the given problem compared to the other solutions. Solution 9 repeats solution 6 at a lower clock period of 2 ns which has a vanishingly small effect on the resource usage but reduces the total latency from 1.71 µs 991 ns, even though the number of clock cycles increases slightly. The design of solution 9 fits very well on the used device and provides classification performance up to an AUC value of 0.979 with an overall latency of 991 ns. This latency is sufficient for online tracking and it is a significant improvement over CPU-based processing, which has a latency of about 646 ms using the described architecture.

10. Conclusion and Outlook

The goal of the work presented in this thesis is the development of a track finding algorithm based on machine learning (ML) for the anti-**P**roton **An**nihilation at **DA**rmstadt (PANDA) forward tracker and implementation of this algorithm on field programmable gate array (FPGA) technology. In this thesis, it has been shown that the physics-motivated interaction network (IN) architecture, which is a special type of graph neural networks (GNNs), can be successfully applied to the task of charged particle tracking on PANDA forward tracking system (FTS) data. The performance of the IN was evaluated based on various data preprocessing steps and graph construction conditions in the context of a track segment classification pipeline. The used detector hit filtering preselection steps by, e.g., applying a p_z threshold, p_z^{min} reduce the number of detector hits and remove curling particles. The used graph building algorithm creates nodes, which contain the hit coordinates, and edges connecting pairs of nodes based on geometric constraints including only allowing the connection of adjacent layer nodes and a slope threshold, s^{max}. The graphs were built with two different approaches: a full graph and a segmented graph approach. The full graph approach comprises hits of all FTS chambers, while the segmented versions subdivide the full graphs into three regions, corresponding to the three tracking station pairs before, within, and after the magnetic field. Evaluation studies have shown that graphs using only the x and z coordinate informations for nodes and edges built with $p_z^{\text{min}} = 0.001\,\text{GeV}/c$ and $s^{\text{max}} = 1.002$ perform best in the trade-off between purity, efficiency and data size for the full graph approach. In particular, for this configuration, a true edge efficiency of 98.9 % at a true edge purity of 63.0 % was achieved.
In this work, the IN architecture for edge classification is used since it has been shown to provide good classification performance in constrained computing environments. In particular, the encoding of the edge adjacency information is substantially smaller and significantly reduces resource usage compared to other approaches, such as a matrix formulation encoding in- and outgoing edges. Therefore, this implementation is significantly faster and more flexible than the matrix implementation used in previous works, such as the work by Esmail [29]. Here, the used architecture comprises only 275 parameters for hidden network layers composed of six hidden nodes each. The GNN-based track finding for PANDA FTS data was evaluated for the full graph and segmented approaches. In both cases, high classification scores could be obtained up to AUC=0.989, while the segmented versions performed slightly better than the full graph approach. Tracklet building evaluations performed on these edge-weighted graphs demonstrated high tracklet finding power by correctly identifying 100 % of hits associated with a tracklet in up to 77 % of all tracklets, where tracklets are segments of the full tracks divided into the three regions.
Moreover, this thesis has discussed the application of the GNN-based track segment classification in heterogeneous hardware with FPGAs using the deep neural network

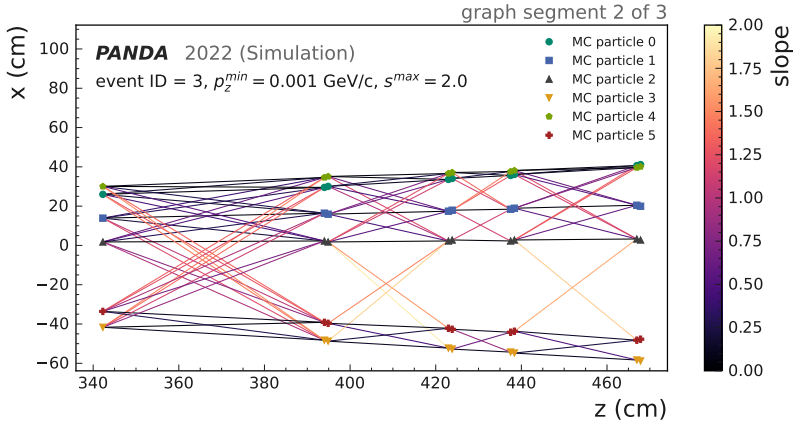
compiler high level synthesis for machine learning (`hls4ml`) based on Vivado `hls`. Generic design optimization techniques such as network quantization, compression, and pipelining are discussed. The effect of these optimization methods on the design latencies, resource usages, and classification performance was scanned using the `hls4ml` configuration parameters. Applying the investigated optimization techniques it was possible to implement the GNN-based track segment classifier in a Xilinx Zynq® UltraScale+™MPSoC FPGA using roughly 34 % of the available digital signal processors (DSPs) and 85 % of the available lookup tables (LUTs). The total latency of the inference is approximately 0.99 µs with a clock frequency of 2 ns. Real-time event reconstruction and filtering require pipelined inference with latencies on the scale of a few microseconds.

The work discussed in this thesis has several important limitations that need to be considered. First, the graph building did not include hits of skewed detector layers. Therefore, no y-information is included in the graphs. Furthermore, the geometric constraints applied are simple and do not include, e.g., next-to-next layer edges or same-layer edges. Background hits and more complex events must be included for a more realistic graph building. Moreover, this thesis only includes track segment classification and simple tracklet finding. Further work needs to investigate full track finding techniques based on the presented results, such as the methods discussed by [1] including a union-find algorithm [91] or DBSCAN [92].
Although this work has found a possible implementation of the discussed track segment classifier on the given FPGA device, several further improvements can be applied regarding resource usage and latency. Each of the design optimization techniques has been implemented most simply. It would be interesting to analyze the performance of other more sophisticated optimization techniques in future studies. For example, quantization-aware training (QAT) can significantly reduce the number of required bits [2]. Additionally, the high-level synthesis (HLS) performance estimates need to be validated by implementation in Vivado and on real FPGAs.
For the implementation of the algorithm in the frame of a real-time trigger in real experimental settings, much work still needs to be done. In particular, accelerated graph building on FPGAs is an important step in the track finding pipeline on heterogeneous resources, to which ongoing efforts are dedicated. Finally, the whole algorithm must be implemented in the `PandaRoot` framework as an official part of the track finding in the PANDA experiment.

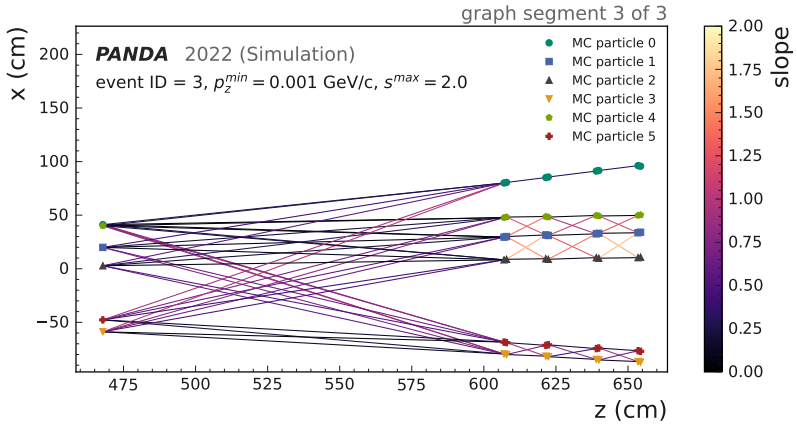
The presented track finding algorithm provides high tracking efficiency with low latency for real-time event filtering for the PANDA experiment. Efficient data reduction in PANDA data acquisition system (DAQ) helps to identify and store the most interesting events for further precise processing to answer many open quantum chromo dynamics (QCD) questions in the charm and multi-strange hadron sector. In addition, the techniques presented contribute to the growing research field of ML-based real-time tracking on FPGAs. The tracking methods used in this project can also be applied to other HEP experiments, and this research can serve as a foundation for future studies in real-time tracking.

A. Appendix

A.1. Segmented Graph Displays



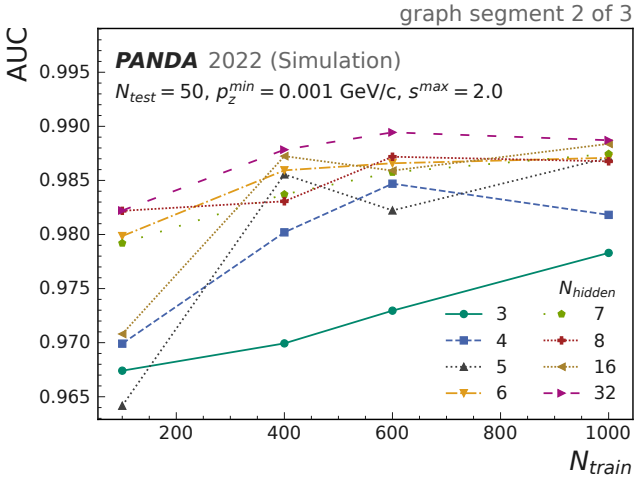
(a)



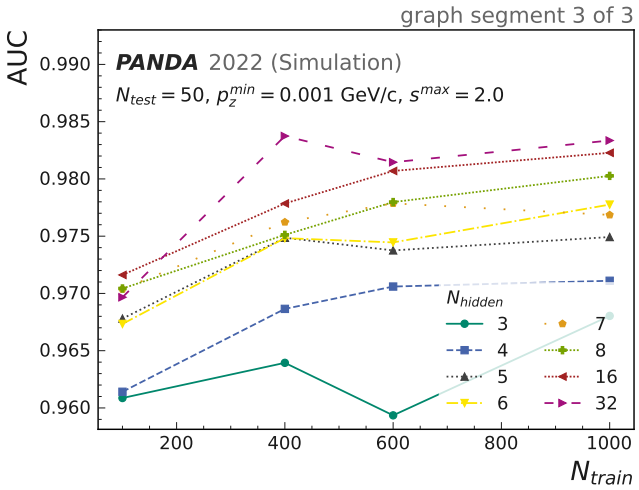
(b)

Figure A.1.: Hit graph display of the second (a) and third (b) segmented graph thirds. Colored points represent detector hits (nodes) of corresponding particles distinguished by the different particle IDs and the connecting lines represent all generated edges after filtering. The edge slope is displayed by color from 0 (black) to 2. (yellow). The graphs are constructed with $p_z^{\min} = 0.001\,\mathrm{GeV}/c$ and $s^{\max} = 2.0$.

A.2. GNN Classification Performances for Segmented Graphs



(a)



(b)

Figure A.2.: Display of area under the curve (AUC) scores for different sets of number of training events N_{train} and number of hidden nodes N_{hidden} per hidden graph neural network (GNN) layer performed on N_{test}=50 test events based on the second (a) and third (b) segmented graph thirds built with $p_z^{\text{min}} = 0.001\,\text{GeV}/c$ and $s^{\text{max}} = 2.0$.

A.3. hls4ml Compilation Times

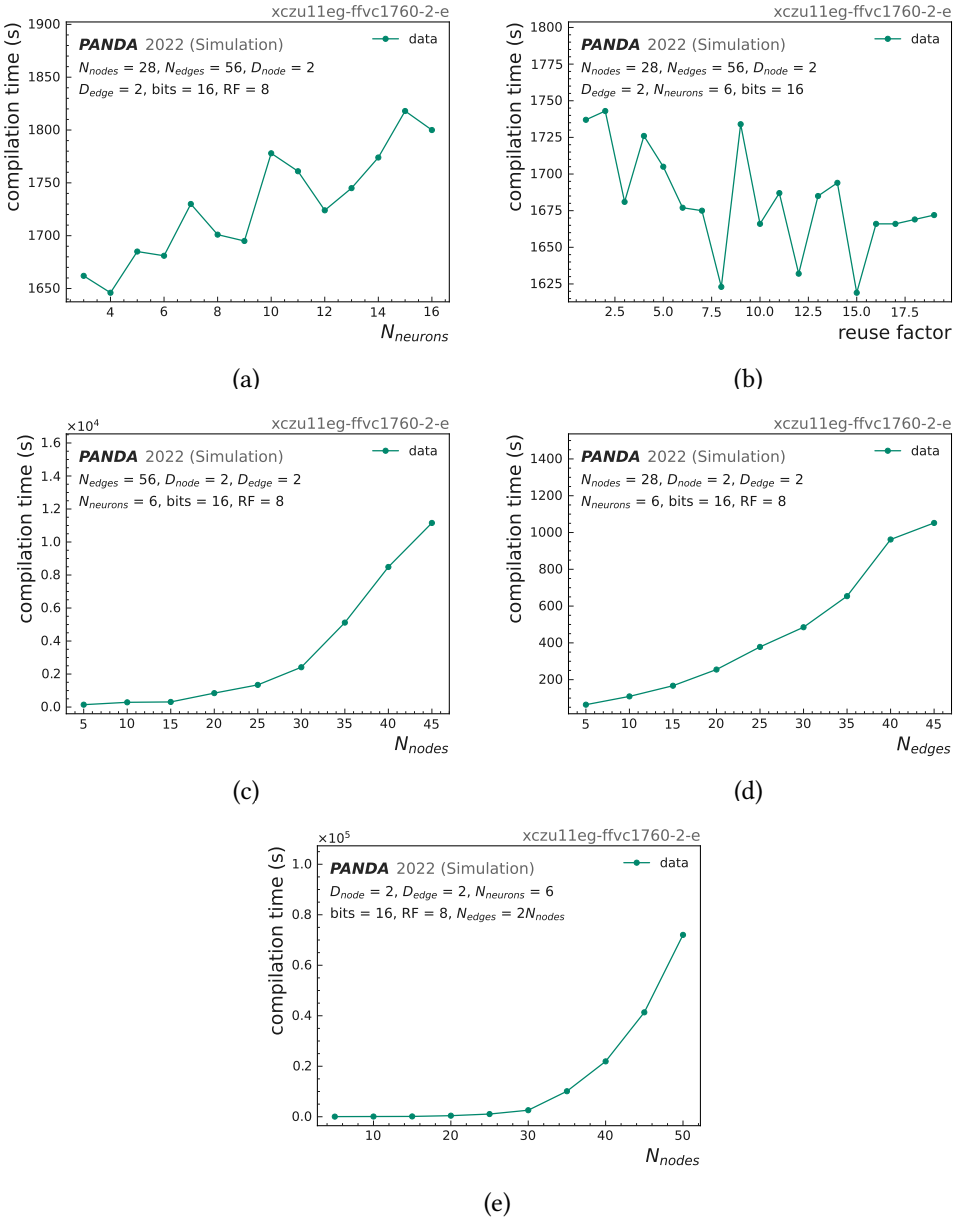


Figure A.3.: hls4ml compilation times performed on an AMD Ryzen 9 5950X 16-core processor with 32 CPUs. Scans as functions of N_{neurons} (a), the RF (b),N_{nodes} (c), N_{edges} (d), and N_{nodes} at fixed ratio N_{edges}=2N_{nodes} (e).

A.4. Design Graph Dimension Studies

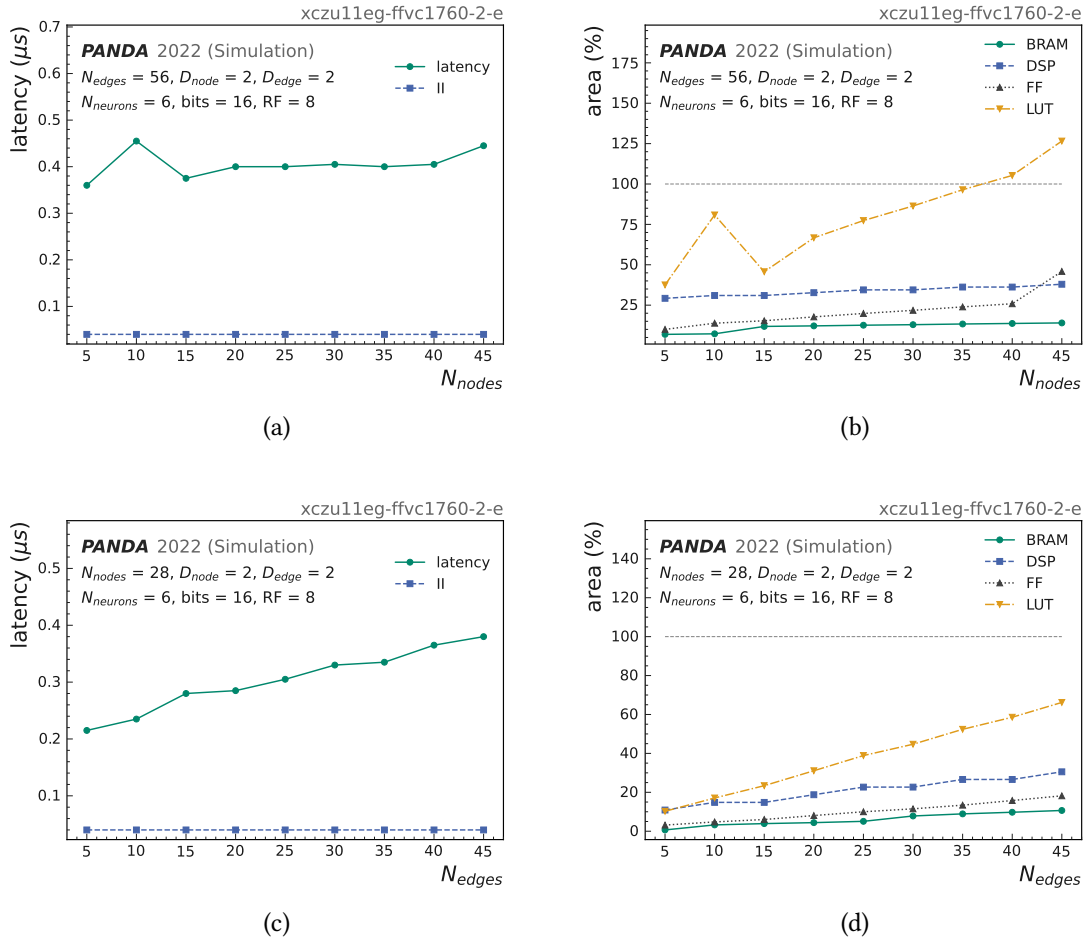


Figure A.4.: high level synthesis for machine learning (`hls4ml`) latency and hardware usage estimates on the part `xczu11eg-ffvc1760-2-e` with respect to the number of graph nodes N_{nodes} and number of graph edges N_{edges}. The benchmark network configuration is as described in Section 9.2. Latency estimates in µs (a) and hardware area utilization in percent of available board resources (b) as function of N_{nodes}. Latency estimates (c) and area (d) as function of N_{edges}. The area usage corresponding to 100 % is indicated by a thin grey line.

List of Acronyms

hls4ml high level synthesis for machine learning. i, iii, v, 3, 5, 59, 63–65, 68, 71–73, 75–78, 84, 87, 88, 94

AP arbitrary precision. 65

ASIC application-specific integrated circuit. 60

AUC area under the curve. 33, 49, 50, 54, 55, 77–79, 81, 83, 86, 93, 94

BCE binary cross entropy. 29, 31, 49, 50

BRAM block random access memory. 61, 64, 68, 69, 71–77, 81

CLB configurable logic block. 60, 61, 64

clk clock. 61

CNN convolutional neural network. 2, 5

CPU central processing unit. 1, 2, 59, 60, 68, 71, 81

CR collector ring. 16

CUDA compute unified device architecture. 62

DAQ data acquisition system. 1, 2, 19, 59, 60, 84

DFF D flip-flop. 61

DL deep learning. 2, 3, 30, 54

DSP digital signal processor. i, iii, 61, 63–67, 71, 73–77, 84

FAIR **F**acility for **A**ntiproton and **I**on **R**esearch. 2, 3, 13, 15–17, 20, 93

FF flip-flop. 60, 61, 64, 71, 73–77

FFT fast Fourier transform. 61

FIFO first-in-first-out shift registers. 68, 69

FN false negative. 32, 51

FNR false negative rate. 33, 40, 43

FP false positive. 32, 51

FPGA field programmable gate array. i, iii, v, 2, 3, 5, 19, 41, 52, 53, 58–69, 71–73, 83, 84, 93

FPR false positive rate. 33, 49

FTS forward tracking system. i, v, 2, 3, 19, 20, 23, 35, 37, 38, 41, 45, 47, 56, 71, 83, 93

GDL geometric deep learning. 2, 30

GNN graph neural network. i, iii, v, 2, 3, 5, 23, 25, 30, 33, 35, 39, 41, 43, 45, 47, 49–52, 54, 56, 63, 68, 69, 71–73, 83, 84, 86, 93

GPU graphics processing unit. 2, 19, 59, 60, 62

GR general relativity. 8

GSI **G**esellschaft für **S**chwer**i**onenforschung. 2, 15, 16

HDL hardware description language. 59, 62, 63

HEP high energy physics. i, iii, 1, 2, 5, 9, 21, 32, 59, 60, 84

HESR high-energy storage ring. 16, 17, 93

HLS high-level synthesis. 62–65, 68, 71, 72, 84

I/O input/output. 60, 61

IDE integrated design environment. 62

II initiation interval. 67–69, 72, 74–76, 81

IN interaction network. i, iii, v, 5, 31, 43, 47–49, 54, 56, 63, 65, 66, 68, 72, 83, 93

IP intellectual property. 62–64

LUT lookup table. i, iii, 60, 61, 63, 64, 66, 71, 73–77, 79, 81, 84

MC Monte Carlo. 35, 37, 38, 40, 51, 55–58

MIP minimum ionizing particle. 35

ML machine learning. 2, 3, 5, 23, 28, 44, 53, 59, 60, 63, 64, 83, 84

MLP multilayer perceptron. 27, 66

MUX multiplexer. 60, 61

MVD micro vertex detector. 35

NN neural network. 25, 26, 28, 33, 47, 59, 64, 65, 73, 93

PANDA anti-**P**roton **An**nihilation at **DA**rmstadt. i, iii, v, 2, 3, 5, 7, 10, 11, 13–16, 18–20, 23, 35, 41, 45, 47, 56, 71, 83, 84, 93

PDG particle data group. 38

PF parallelization factor. 80

PIPO parallel-in-parallel-out shift registers. 68

PPV positive predictive value. 32

PyG PyTorch Geometric. 31, 63, 64, 72

QCD quantum chromo dynamics. 9–11, 13, 15, 84

QED quantum electrodynamics. 8

ReLU rectified linear unit. 27, 47, 48, 72

RF reuse factor. 68, 69, 72–75, 79, 87

RNN recurrent neural network. 2, 5, 56

ROC receiver operating characteristics. 33, 49, 50, 78, 79, 93

RTL register-transfer level. 62, 63

S signal. 32

SB switching block. 60, 61

SM standard model of particle physics. 3, 7, 8, 11, 93

SoC system-on-a-chip. 61, 62

SRAM static random-access memory. 60

TN true negative. 32, 51

TNR true negative rate. 32, 33, 40, 43

TP true positive. 32, 51

TPR true positive rate. 32, 33, 49

TrackML tracking machine learning challenge. 2

VHDL very high speed integrated circuit hardware description language. 62

List of Figures

List of Tables

Bibliography

[1] Gage DeZoort et al. “Charged Particle Tracking via Edge-Classifying Interaction Networks”. In: *Comput. Softw. Big Sci.* 5.1 (2021), p. 26. DOI: 10.1007/s41781-021-00073-z. arXiv: 2103.16701 [hep-ex].

[2] Abdelrahman Elabd et al. “Graph Neural Networks for Charged Particle Tracking on FPGAs”. In: *Front. Big Data* 5 (2022), p. 828666. DOI: 10.3389/fdata.2022.828666. arXiv: 2112.02048 [physics.ins-det]. URL: https://doi.org/10.3389%5C%2Ffdata.2022.828666.

[3] ATLAS Collaboration. “Search for events with a pair of displaced vertices from long-lived neutral particles decaying into hadronic jets in the ATLAS muon spectrometer in pp collisions at $\sqrt{s} = 13$ TeV”. In: (2022). arXiv: 2203.00587 [hep-ex].

[4] CMS Collaboration. “Description and performance of track and primary-vertex reconstruction with the CMS tracker”. In: *Journal of Instrumentation* 9.10 (Oct. 2014), P10009–P10009. DOI: 10.1088/1748-0221/9/10/p10009. URL: https://doi.org/10.1088%5C%2F1748-0221%5C%2F9%5C%2F10%5C%2Fp10009.

[5] CMS Collaboration. “Particle-flow reconstruction and global event description with the CMS detector”. In: *Journal of Instrumentation* 12.10 (Oct. 2017), P10003–P10003. DOI: 10.1088/1748-0221/12/10/p10003. URL: https://doi.org/10.1088%5C%2F1748-0221%5C%2F12%5C%2F10%5C%2Fp10003.

[6] Andrew J. Larkoski, Ian Moult, and Benjamin Nachman. “Jet substructure at the Large Hadron Collider: A review of recent advances in theory and machine learning”. In: *Physics Reports* 841 (Jan. 2020), pp. 1–63. DOI: 10.1016/j.physrep.2019.11.001. URL: https://doi.org/10.1016%5C%2Fj.physrep.2019.11.001.

[7] CMS Collaboration. “Identification of heavy-flavour jets with the CMS detector in pp collisions at 13 TeV”. In: *Journal of Instrumentation* 13.05 (May 2018), P05011–P05011. DOI: 10.1088/1748-0221/13/05/p05011. URL: https://doi.org/10.1088%5C%2F1748-0221%5C%2F13%5C%2F05%5C%2Fp05011.

[8] ATLAS Collaboration. “Performance of the ATLAS track reconstruction algorithms in dense environments in LHC Run 2”. In: *The European Physical Journal C* 77.10 (Oct. 2017). DOI: 10.1140/epjc/s10052-017-5225-7. URL: https://doi.org/10.1140%5C%2Fepjc%5C%2Fs10052-017-5225-7.

[9] Pierre Billoir. “Progressive track recognition with a Kalman-like fitting procedure”. In: *Computer Physics Communications* 57.1 (1989), pp. 390–394. ISSN: 0010-4655. DOI: https://doi.org/10.1016/0010-4655(89)90249-X. URL: https://www.sciencedirect.com/science/article/pii/001046558990249X.

[10] P. Billoir and S. Qian. "Simultaneous pattern recognition and track fitting by the Kalman filtering method". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 294.1 (1990), pp. 219–228. ISSN: 0168-9002. DOI: https://doi.org/10.1016/0168-9002(90)91835-Y. URL: https://www.sciencedirect.com/science/article/pii/016890029091835Y.

[11] Rainer Mankel. "A concurrent track evolution algorithm for pattern recognition in the HERA-B main tracking system". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 395.2 (1997), pp. 169–184. ISSN: 0168-9002. DOI: https://doi.org/10.1016/S0168-9002(97)00705-5. URL: https://www.sciencedirect.com/science/article/pii/S0168900297007055.

[12] R. Frühwirth. "Application of Kalman filtering to track and vertex fitting". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 262.2 (1987), pp. 444–450. ISSN: 0168-9002. DOI: https://doi.org/10.1016/0168-9002(87)90887-4. URL: https://www.sciencedirect.com/science/article/pii/0168900287908874.

[13] Javier Duarte et al. "FPGA-accelerated machine learning inference as a service for particle physics computing". In: *Comput. Softw. Big Sci.* 3.1 (2019), p. 13. DOI: 10.1007/s41781-019-0027-2. arXiv: 1904.08986 [physics.data-an].

[14] R.H. Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[15] Hadi Esmaeilzadeh et al. "Dark Silicon and the End of Multicore Scaling". In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 365–376. ISBN: 9781450304726. DOI: 10.1145/2000064.2000108. URL: https://doi.org/10.1145/2000064.2000108.

[16] Dan Guest, Kyle Cranmer, and Daniel Whiteson. "Deep Learning and its Application to LHC Physics". In: *Ann. Rev. Nucl. Part. Sci.* 68 (2018), pp. 161–181. DOI: 10.1146/annurev-nucl-101917-021019. arXiv: 1806.11484 [hep-ex].

[17] Andrew J. Larkoski, Ian Moult, and Benjamin Nachman. "Jet Substructure at the Large Hadron Collider: A Review of Recent Advances in Theory and Machine Learning". In: *Phys. Rept.* 841 (2020), pp. 1–63. DOI: 10.1016/j.physrep.2019.11.001. arXiv: 1709.04464 [hep-ph].

[18] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. "Searching for Exotic Particles in High-Energy Physics with Deep Learning". In: *Nature Commun.* 5 (2014), p. 4308. DOI: 10.1038/ncomms5308. arXiv: 1402.4735 [hep-ph].

[19] T. Keck et al. "The Full Event Interpretation: An Exclusive Tagging Algorithm for the Belle II Experiment". In: *Comput. Softw. Big Sci.* 3.1 (2019), p. 6. DOI: 10.1007/s41781-019-0021-8. arXiv: 1807.08680 [hep-ex].

[20] Steven Farrell et al. *Novel deep learning methods for track reconstruction*. 2018. DOI: 10.48550/ARXIV.1810.06111. URL: https://arxiv.org/abs/1810.06111.

[21] Michael M. Bronstein et al. "Geometric Deep Learning: Going beyond Euclidean data". In: *IEEE Sig. Proc. Mag.* 34.4 (2017), pp. 18–42. DOI: 10.1109/MSP.2017.2693418. arXiv: 1611.08097 [cs.CV].

[22] Jonathan Shlomi, Peter Battaglia, and Jean-Roch Vlimant. "Graph Neural Networks in Particle Physics". In: (July 2020). DOI: 10.1088/2632-2153/abbf9a. arXiv: 2007.13681 [hep-ex].

[23] Sabrina Amrouche et al. "The Tracking Machine Learning challenge : Accuracy phase". In: *The NeurIPS '18 Competition: From Machine Learning to Intelligent Conversations*. Apr. 2019. DOI: 10.1007/978-3-030-29135-8_9. arXiv: 1904.06778 [hep-ex].

[24] Xiangyang Ju et al. "Performance of a geometric deep learning pipeline for HL-LHC particle tracking". In: *The European Physical Journal C* 81.10 (Oct. 6, 2021), p. 876. ISSN: 1434-6052. DOI: 10.1140/epjc/s10052-021-09675-8. URL: https://doi.org/10.1140/epjc/s10052-021-09675-8 (visited on 08/21/2022).

[25] Song Han et al. *EIE: Efficient Inference Engine on Compressed Deep Neural Network*. 2016. DOI: 10.48550/ARXIV.1602.01528. URL: https://arxiv.org/abs/1602.01528.

[26] Aneesh Heintz et al. "Accelerated Charged Particle Tracking with Graph Neural Networks on FPGAs". In: *34th Conference on Neural Information Processing Systems*. Nov. 2020. DOI: 10.48550/ARXIV.2012.01563. arXiv: 2012.01563 [physics.ins-det]. URL: https://arxiv.org/abs/2012.01563.

[27] M. F. M. Lutz et al. "Physics Performance Report for PANDA: Strong Interaction Studies with Antiprotons". In: (Mar. 2009). arXiv: 0903.3905 [hep-ex].

[28] The PANDA Collaboration. "Technical Design Report for the: PANDA Data Acquisition and Event Filtering". In: (Aug. 2020). URL: https://indico.scc.kit.edu/category/124/attachments/4444/6702/tdr-daqt_v2.pdf.

[29] Waleed Ahmed Mohammed Esmail. "Deep learning for track finding and the reconstruction of excited hyperons in proton induced reactions". PhD thesis. Ruhr-Universität Bochum, Fakultät für Physik und Astronomie, Bochum, Germany, Ruhr U., Bochum, 2022. DOI: 10.13154/294-8563.

[30] Javier Duarte and Jean-Roch Vlimant. "Graph Neural Networks for Particle Tracking and Reconstruction". In: *Artificial Intelligence for High Energy Physics*. Chap. Chapter 12, pp. 387–436. DOI: 10.1142/9789811234033_0012. eprint: https://www.worldscientific.com/doi/pdf/10.1142/9789811234033_0012. URL: https://www.worldscientific.com/doi/abs/10.1142/9789811234033_0012.

[31] Jan Kieseler. "Object condensation: one-stage grid-free multi-object reconstruction in physics detectors, graph, and image data". In: *The European Physical Journal C* 80.9 (2020), p. 886. DOI: 10.1140/epjc/s10052-020-08461-2. URL: https://doi.org/10.1140/epjc/s10052-020-08461-2.

[32] Joosep Pata et al. "MLPF: efficient machine-learned particle-flow reconstruction using graph neural networks". In: *The European Physical Journal C* 81.5 (2021), p. 381. DOI: 10.1140/epjc/s10052-021-09158-w. URL: https://doi.org/10.1140/epjc/s10052-021-09158-w.

[33] W. Esmail, T. Stockmanns, and J. Ritman. *Machine Learning for Track Finding at PANDA*. 2019. DOI: 10.48550/ARXIV.1910.07191. URL: https://arxiv.org/abs/1910.07191.

[34] J. Duarte et al. *Fast inference of deep neural networks in FPGAs for particle physics*. July 2018. DOI: 10.1088/1748-0221/13/07/p07027. URL: https://doi.org/10.1088%5C%2F1748-0221%5C%2F13%5C%2F07%5C%2Fp07027.

[35] Kazi Ahmed Asif Fuad and Sofia Vallecorsa. "Graph Neural Network Inference on FPGA". In: (Apr. 2020). DOI: 10.5281/zenodo.3764836.

[36] Stefan Abi-Karam et al. *GenGNN: A Generic FPGA Framework for Graph Neural Network Acceleration*. 2022. DOI: 10.48550/ARXIV.2201.08475. URL: https://arxiv.org/abs/2201.08475.

[37] Bogdan Povh et al. *Particles and Nuclei*. Jan. 2015. ISBN: 978-3-662-46320-8. DOI: 10.1007/978-3-662-46321-5.

[38] the free encyclopedia: MissMJ Wikipedia. *Standard model of elementary particles*. [Online; accessed July 5, 2022]. 2021. URL: https://en.wikipedia.org/wiki/File:Standard_Model_of_Elementary_Particles.svg.

[39] Abdus Salam. "Gauge unification of fundamental forces". In: *Rev. Mod. Phys.* 52 (3 July 1980), pp. 525–538. DOI: 10.1103/RevModPhys.52.525. URL: https://link.aps.org/doi/10.1103/RevModPhys.52.525.

[40] C. P. Burgess. *Introduction to Effective Field Theory* -. Cambridge: Cambridge University Press, 2020. ISBN: 978-0-521-19547-8.

[41] C. T. H. Davies et al. "High-Precision Lattice QCD Confronts Experiment". In: *Physical Review Letters* 92.2 (Jan. 2004). DOI: 10.1103/physrevlett.92.022001. URL: https://doi.org/10.1103%5C%2Fphysrevlett.92.022001.

[42] Nora Brambilla et al. "The XYZ states: experimental and theoretical status and perspectives". In: *Phys. Rept.* 873 (2020), pp. 1–154. DOI: 10.1016/j.physrep.2020.05.001. arXiv: 1907.07583 [hep-ex].

[43] S. Jia et al. "Search for the 0^{--} Glueball in $\Upsilon(1S)$ and $\Upsilon(2S)$ decays". In: *Physical Review D* 95.1 (Jan. 2017). DOI: 10.1103/physrevd.95.012001. URL: https://doi.org/10.1103%5C%2Fphysrevd.95.012001.

[44] G. Barucca et al. "PANDA Phase One". In: *The European Physical Journal A* 57.6 (June 2021). DOI: 10.1140/epja/s10050-021-00475-y. URL: https://doi.org/10.1140%5C%2Fepja%5C%2Fs10050-021-00475-y.

[45] PANDA Collaboration. *Panda website*. [Online; accessed August 12, 2022]. 2022. URL: https://panda.gsi.de/article/panda-physics.

[46] Tord Johansson. “Antihyperon-Hyperon Production in Antiproton-Proton Collisions”. In: *AIP Conf. Proc.* 796 (2005), pp. 95–101. DOI: 10.1063/1.2130143. URL: https://cds.cern.ch/record/936215.

[47] Hua-Xing Chen et al. “A review of the open charm and open bottom mesons”. In: (Sept. 2016).

[48] Valentina Zhukova. “Open charm studies at Belle”. In: *EPJ Web of Conferences* 212 (Jan. 2019), p. 09003. DOI: 10.1051/epjconf/201921209003.

[49] M. Konradt/GSI/FAIR. *Ring accelerator SIS100, June 2021.* [Online; accessed July 1, 2022]. 2021. URL: https://fair-center.eu/overview/construction/media.

[50] P. Spiller and G. Franchetti. “The FAIR accelerator project at GSI”. In: *Nucl. Instrum. Meth. A* 561 (2006), pp. 305–309. DOI: 10.1016/j.nima.2006.01.043.

[51] *FAIR: The Accelerator facility.* [Online; accessed July 1, 2022]. 2021. URL: https://fair-center.eu/overview/accelerator.

[52] *High Energy Storage Ring HESR.* [Online; accessed July 1, 2022]. URL: https://www.ep1.rub.de/PandaBMBF/index.php/pandaphysik/forschunsprogramm.

[53] *PANDA Detector overview.* [Online; accessed July 1, 2022]. URL: https://panda.gsi.de/panda.

[54] A. Belias. “Overview of the PANDA Detector design at FAIR”. In: (Aug. 2020). URL: https://panda.gsi.de/system/files/user_uploads/a.belias/PA-PRO-2021-002.pdf.

[55] W. Esmail, T. Stockmanns, and J. Ritman. *Machine Learning for Track Finding at PANDA*. 2019. DOI: 10.48550/ARXIV.1910.07191. URL: https://arxiv.org/abs/1910.07191.

[56] Tobias Stockmanns. “PandaRoot — the simulation and reconstruction framework of PANDA”. In: ICTP-SAIFR/FAIR Workshop on Mass Generation in QCD, Sao Paulo (Brazil), 25 Feb 2019 - 1 Mar 2019. Feb. 25, 2019. URL: https://juser.fz-juelich.de/record/873504.

[57] Stefano Spataro. “The PandaRoot framework for simulation, reconstruction and analysis”. In: *Journal of Physics: Conference Series* 331.3 (Dec. 2011), p. 032031. DOI: 10.1088/1742-6596/331/3/032031. URL: https://doi.org/10.1088/1742-6596/331/3/032031.

[58] Rene Brun and Fons Rademakers. “ROOT - An Object Oriented Data Analysis Framework”. In: *AIHENP’96 Workshop, Lausane*. Vol. 389. 1996, pp. 81–86.

[59] S. Agostinelli et al. “Geant4—a simulation toolkit”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 506.3 (2003), pp. 250–303. ISSN: 0168-9002. DOI: https://doi.org/10.1016/S0168-9002(03)01368-8. URL: https://www.sciencedirect.com/science/article/pii/S0168900203013688.

[60] I. Hrivnacova et al. “The Virtual Monte Carlo”. In: *eConf* C0303241 (2003), THJT006. arXiv: cs/0306005.

[61] Are Strandlie and Rudolf Frühwirth. "Track and vertex reconstruction: From classical to adaptive methods". In: *Rev. Mod. Phys.* 82 (2 May 2010), pp. 1419–1458. DOI: 10.1103/RevModPhys.82.1419. URL: https://link.aps.org/doi/10.1103/RevModPhys.82.1419.

[62] Rudolf Frühwirth and Are Strandlie. "Tracking Detectors". In: *Pattern Recognition, Tracking and Vertex Reconstruction in Particle Detectors*. Cham: Springer International Publishing, 2021, pp. 3–21. ISBN: 978-3-030-65771-0. DOI: 10.1007/978-3-030-65771-0_1. URL: https://doi.org/10.1007/978-3-030-65771-0_1.

[63] "High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1". In: 4/2017 (2017). Ed. by G. Apollinari et al. DOI: 10.23731/CYRM-2017-004.

[64] Johannes Rauch and Tobias Schlüter. "GENFIT — a Generic Track-Fitting Toolkit". In: *Journal of Physics: Conference Series* 608 (May 2015), p. 012042. DOI: 10.1088/1742-6596/608/1/012042. URL: https://doi.org/10.1088%2F1742-6596%2F608%2F1%2F012042.

[65] François Chollet. *Deep Learning with Python*. Manning, Nov. 2017. ISBN: 9781617294433.

[66] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. ISBN: 9780262035613. URL: https://books.google.co.in/books?id=Np9SDQAAQBAJ.

[67] Aston Zhang et al. *Dive into Deep Learning*. http://www.d2l.ai. 2019.

[68] Shie Mannor, Dori Peleg, and Reuven Rubinstein. "The Cross Entropy Method for Classification". In: *Proceedings of the 22nd International Conference on Machine Learning*. ICML '05. Bonn, Germany: Association for Computing Machinery, 2005, pp. 561–568. ISBN: 1595931805. DOI: 10.1145/1102351.1102422. URL: https://doi.org/10.1145/1102351.1102422.

[69] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435.

[70] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. DOI: 10.48550/ARXIV.1502.03167. URL: https://arxiv.org/abs/1502.03167.

[71] PyTorch Contributors. *Quickstart - PyTorch tutorials + Documentation*. [Online; accessed August 12, 2022]. 2022. URL: https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html.

[72] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[73] Matthias Fey and Jan Eric Lenssen. *Fast Graph Representation Learning with PyTorch Geometric*. 2019. DOI: 10.48550/ARXIV.1903.02428. URL: https://arxiv.org/abs/1903.02428.

[74] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. DOI: 10.48550/ARXIV.1912.01703. URL: https://arxiv.org/abs/1912.01703.

[75] Peter W. Battaglia et al. “Interaction Networks for Learning about Objects, Relations and Physics”. In: (Dec. 2016). arXiv: 1612.00222 [cs.AI].

[76] ISO. *ISO/IEC 14882:1998: Programming languages — C++*. Available in electronic form for online purchase at http://webstore.ansi.org/ and http://www.cssinfo.com/. Sept. 1998, p. 732. URL: http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%5C%2FIEC+14882%5C%2D1998;%20http://webstore.ansi.org/ansidocstore/product.asp?sku=ISO%5C%2FIEC+14882%5C%3A1998;%20http://www.iso.ch/cate/d25845.html;%20https://webstore.ansi.org/.

[77] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[78] Jim Pivarski et al. *scikit-hep/uproot: 3.12.0*. Version 3.12.0. July 2020. DOI: 10.5281/zenodo.3952728. URL: https://doi.org/10.5281/zenodo.3952728.

[79] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[80] Wes McKinney. “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[81] Takuya Akiba et al. *Optuna: A Next-generation Hyperparameter Optimization Framework*. 2019. DOI: 10.48550/ARXIV.1907.10902. URL: https://arxiv.org/abs/1907.10902.

[82] Michael R. Chernick et al. “Bootstrap Methods”. In: *International Encyclopedia of Statistical Science*. Ed. by Miodrag Lovric. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 169–174. ISBN: 978-3-642-04898-2. DOI: 10.1007/978-3-642-04898-2_150. URL: https://doi.org/10.1007/978-3-642-04898-2_150.

[83] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. “Exploring Network Structure, Dynamics, and Function using NetworkX”. In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[84] J. Alme et al. “The ALICE TPC, a large 3-dimensional tracking device with fast readout for ultra-high multiplicity events”. In: *Nucl. Instrum. Meth. A* 622 (2010), pp. 316–367. DOI: 10.1016/j.nima.2010.04.042. arXiv: 1001.1950 [physics.ins-det].

[85] TensorFlow Developers. *TensorFlow*. Version v2.8.2. Specific TensorFlow versions can be found in the "Versions" list on the right side of this page.
See the full list of authors on GitHub. May 2022. DOI: 10.5281/zenodo.6574269. URL: https://doi.org/10.5281/zenodo.6574269.

[86] Inc. Xilinx. *Vitis High-Level Synthesis User Guide (UG1399)*. [Online; accessed August 24, 2022]. June 2022. URL: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls.

[87] Inc. Xilinx. *Vivado Design Suite User Guide: Getting Started*. [Online; accessed August 11, 2022]. 2021. URL: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug910-vivado-getting-started.pdf.

[88] Inc. Xilinx. *Vivado Design Suite User Guide: High Level Synthesis*. [Online; accessed August 11, 2022]. 2021. URL: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2020_2/ug902-vivado-high-level-synthesis.pdf#nameddest=xApplyingOptimizationDirectives.

[89] Mostafa W. Numan et al. "Towards Automatic High-Level Code Deployment on Reconfigurable Platforms: A Survey of High-Level Synthesis Tools and Toolchains". In: *IEEE Access* 8 (2020), pp. 174692–174722. DOI: 10.1109/ACCESS.2020.3024098.

[90] Inc. Xilinx. *Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891)*. [Online; accessed August 24, 2022]. May 2021. URL: https://www.mouser.com/datasheet/2/903/ds891_zynq_ultrascale_plus_overview-1662253.pdf.

[91] Catherine Biscarat et al. "Towards a realistic track reconstruction algorithm based on graph neural networks for the HL-LHC". In: *EPJ Web of Conferences* 251 (2021). Ed. by C. Biscarat et al., p. 03047. DOI: 10.1051/epjconf/202125103047. URL: https://doi.org/10.1051%5C%2Fepjconf%5C%2F202125103047.

[92] Martin Ester et al. "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise". In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, pp. 226–231.